Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors

Onur Mutlu Thomas Moscibroda

Microsoft Research
{onur,moscitho}@microsoft.com

Abstract

DRAM memory is a major resource shared among cores in a chip multiprocessor (CMP) system. Memory requests from different threads can interfere with each other. Existing memory access scheduling techniques try to optimize the overall data throughput obtained from the DRAM and thus do not take into account inter-thread interference. Therefore, different threads running together on the same chip can experience extremely different memory system performance: one thread can experience a severe slowdown or starvation while another is unfairly prioritized by the memory scheduler.

This paper proposes a new memory access scheduler, called the Stall-Time Fair Memory scheduler (STFM)*, that provides quality of service to different threads sharing the DRAM memory system. The goal of the proposed scheduler is to "equalize" the DRAM-related slowdown experienced by each thread due to interference from other threads, without hurting overall system performance. As such, STFM takes into account inherent memory characteristics of each thread and does not unfairly penalize threads that use the DRAM system without interfering with other threads.*

We show that STFM significantly reduces the unfairness in the DRAM system while also improving system throughput (i.e., weighted speedup of threads) on a wide variety of workloads and systems. For example, averaged over 32 different workloads running on an 8-core CMP, the ratio between the highest DRAM-related slowdown and the lowest DRAM-related slowdown reduces from 5.26X to 1.4X, while the average system throughput improves by 7.6%. We qualitatively and quantitatively compare STFM to one new and three previously-proposed memory access scheduling algorithms, including network fair queueing. Our results show that STFM provides the best fairness, system throughput, and scalability.

1. Introduction

Chip multiprocessor (CMP) systems enable multiple threads to run simultaneously on a single chip. A CMP system consists of multiple independent processing cores that share parts of the memory subsystem. This chip organization has benefits in terms of power-efficiency, scalability, and system throughput compared to a single-core system. However, shared hardware resources pose a significant resource management problem in designing CMP systems. Different threads can interfere with each other while accessing the shared resources. If inter-thread interference is not controlled, some threads could be unfairly prioritized over others while other, perhaps higher priority, threads could be starved for long time periods waiting to access shared resources. There are at least four major problems with such unfair resource sharing in CMP systems:

- First, unfair resource sharing would render system software's (operating system or virtual machine) priority-based thread scheduling policies ineffective [5] and therefore cause significant discomfort to the end user who naturally expects threads with higher (equal) priorities to get higher (equal) shares of the performance provided by the computing system.
- Malicious programs that intentionally deny service to other threads can be devised by exploiting the unfairness in the resource sharing schemes [20]. This would result in significant productivity loss and degradation in system performance.
- Unfairness would reduce the performance predictability of applications since the performance of an application becomes much more dependent on the characteristics of other applications running on other cores. This would make it difficult to analyze and optimize system performance.
- In commercial *grid computing* systems (e.g. [30]), where users are charged for CPU hours, unfair resource sharing would result in very unfair billing procedures because the performance a user program experiences would not necessarily correlate with CPU hours it takes as it is dependent on other programs running on the CMP.

As processor designers put more processing cores on chip, the pressure on the shared hardware resources will increase and inter-thread interference in shared resources will become an even more severe problem. Therefore, techniques to provide *quality of service (or fairness)* to threads sharing CMP resources are necessary.

The DRAM memory subsystem is a major resource shared between the processing cores in a CMP system. Unfortunately, conventional high-performance DRAM memory controller designs do not take into account interference between different threads when making scheduling decisions. Instead, they try to maximize the data throughput obtained from the DRAM using a first-ready first-come-first-serve (FR-FCFS) policy [25, 24]. FR-FCFS prioritizes memory requests that hit in the row-buffers of DRAM banks over other requests, including older ones. If no request is a row-buffer hit, then FR-FCFS prioritizes older requests over younger ones. This scheduling algorithm is thread-unaware. Therefore, different threads running together on the same chip can experience extremely different memory system performance: one thread (e.g. one with a very low row-buffer hit rate) can experience a severe slowdown or starvation while another (e.g. one with a very high row-buffer hit rate) is unfairly prioritized by the memory scheduler.

Figure 1 illustrates the problem by showing the memory-related slowdowns of different threads on a 4-core and an 8-core CMP system. A thread's memory-related slowdown is the memory stall time (i.e. number of cycles in which a thread cannot commit instructions due to a memory access) the thread experiences when running simultaneously with other threads, divided by the memory stall time it experiences when running alone.[1] There is a very large variance between the threads' memory-related slowdowns in both systems. In the 4-core system, *omnetpp* experiences a slowdown of 7.74X whereas *libquantum* experiences almost no slowdown at all (1.04X). The problem becomes even worse in the 8-core system with *dealII* experiencing a slowdown of 11.35X while *libquantum* experiences only a 1.09X slowdown.[2] Clearly, trying to maximize DRAM throughput with FR-FCFS scheduling results in significant unfairness across threads in a CMP system.

In this paper we propose a new memory scheduling algorithm, called the *Stall-Time Fair Memory Scheduler (STFM)*, that provides fairness to different threads sharing the DRAM memory system. We define a memory scheduler to be *fair* if the memory-related slowdowns of equal-priority threads running together on the CMP system are the same. Hence, the *quality of service (QoS) goal* of the proposed scheduler is to "equalize" the memory-related slowdown each thread expe-

[1] The cores have private L2 caches, but they share the memory controller and the DRAM memory. Our methodology is described in detail in Section 6.

[2] *libquantum* is a memory-intensive streaming application that has a very high row-buffer locality (98.4% row-buffer hit rate). Other applications have significantly lower row-buffer hit rates. Since *libquantum* can generate its row-buffer-hit memory requests fast enough, its accesses are almost always unfairly prioritized over other threads' accesses by the FR-FCFS scheduling algorithm.

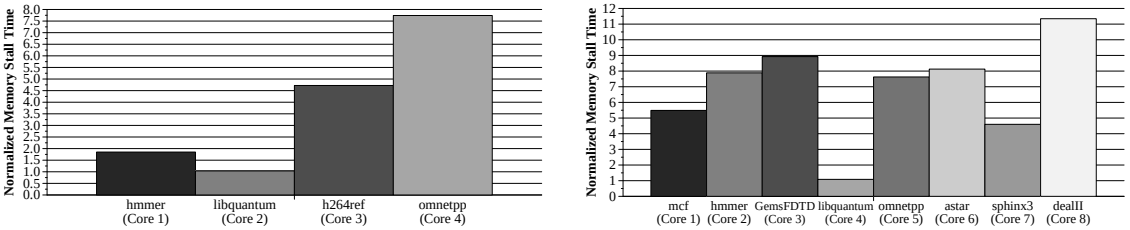

Figure 1. Memory slowdown (normalized memory stall time) of programs in two workloads run on 4-core (left) and 8-core (right) CMP systems

riences due to interference from other threads, without hurting overall system performance.

Basic idea: To achieve this QoS goal, the scheduler estimates two values for each thread: 1) T_{shared}: Memory stall time the thread experiences when running with others, 2) T_{alone}: Memory stall time the thread would have experienced had it been running alone. Based on these estimates, the scheduler computes the *memory-slowdown* S of each thread where $S = T_{shared}/T_{alone}$. If the ratio between the maximum slowdown value and the minimum slowdown value in the system exceeds a threshold (the threshold of maximum tolerable unfairness), the scheduler prioritizes memory requests from threads that are slowed down the most. Otherwise, the scheduler tries to maximize DRAM throughput by using the baseline FR-FCFS scheduling policy.

We explain how the memory-slowdown of threads can be estimated in hardware and describe the design and implementation of STFM. We also describe how STFM can be exposed to the system software to provide more flexibility and control to the system software in thread scheduling and resource management. We compare and contrast STFM to scheduling techniques that try to equalize the memory bandwidth utilized by different threads, such as in *network fair queueing* [23, 22]. In contrast to these approaches, STFM takes into account inherent memory-related performance of each thread. Therefore, it does not unfairly penalize threads that use the DRAM memory system without interfering with other threads (e.g. when other threads are not issuing memory requests).

Contributions: We make the following contributions in this paper:

- We provide a new definition of DRAM fairness that takes into account inherent memory characteristics of each thread executing on a CMP. We compare the merits of our definition with previously proposed definitions and provide insights into why *stall-time fairness* is a more meaningful fairness definition for DRAM systems.
- We introduce and describe the design and implementation of a new memory access scheduler, STFM, that provides quality of service to threads using the proposed definition of DRAM fairness.
- We qualitatively and quantitatively evaluate our new memory access scheduler with extensive comparisons to one new and three previously proposed schedulers (FR-FCFS, first-come-first-serve, and network fair queueing) in terms of fairness and system throughput. Our results show that STFM provides the best fairness, throughput, and scalability as the number of cores increases over a wide variety of workloads.
- We describe how the system software can utilize the flexible fairness substrate provided by STFM to enforce *thread weights* and to control the unfairness in the system.

2. Background on DRAM Memory Controllers

We provide the pertinent details of DRAM memory systems and controllers. Our description is based on DDR2 SDRAM systems, but it is generally applicable to other DRAM types that employ page-mode. More details can be found in [3, 25, 4, 24].

2.1. SDRAM Organization and Access Latencies

A modern SDRAM system as shown in Figure 2 consists of one or more DIMMs (dual in-line memory modules). A DIMM is comprised of multiple SDRAM chips put together and accessed in parallel. Since each SDRAM chip has a narrow data interface (e.g. 8 bits) due to packaging constraints, combining several of them in a DIMM widens the data interface (e.g. to 64 bits) to DRAM. An SDRAM chip consists of multiple independent memory banks such that memory requests to different banks can be serviced in parallel. Each bank is organized as a two-dimensional array of DRAM cells, consisting of multiple rows and columns. A location in the DRAM is thus accessed using a DRAM address consisting of bank, row, and column fields.

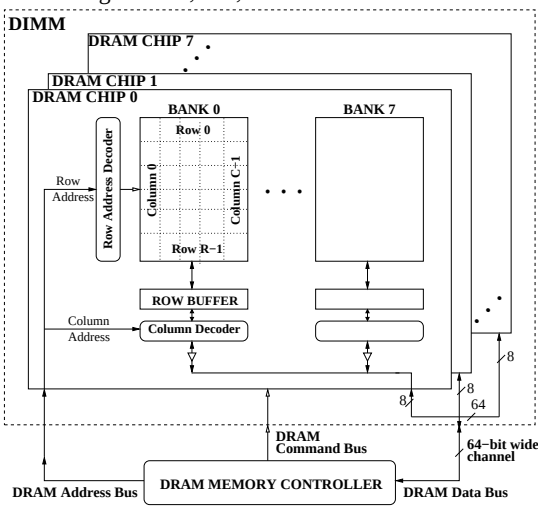

Figure 2. High-level organization of a DRAM system

Physically only one row in a bank can be accessed at any given time. This row is stored in the *row-buffer* (i.e. sense amplifiers) dedicated for that bank. The size of the row buffer is typically 1-2 KB in a DRAM chip, allowing it to hold tens of cache lines. To move a row from the memory array to the row buffer (i.e. to open a row), an *activate* command needs to be issued first. Once a row is in the row buffer, then *read* or *write* commands can be issued to read/write data from/into the memory addresses (columns) contained in the row. The latency of a memory request therefore depends on whether or not the requested row is in the row buffer of the bank. A memory request falls into three different categories:

- **Row hit**: The request is accessing the row currently in the row buffer. Only a *read* or a *write* command is needed. This case results in the lowest bank access latency (called t_{CL} in DRAM nomenclature, e.g. [18]) as only a column access is required.
- **Row closed**: There is no row in the row buffer. An *activate* command needs to be issued to open the row followed by a *read* or *write* command. The bank latency of this case is $t_{RCD} + t_{CL}$ as both a row access and a column access are required.
- **Row conflict**: The access is to a row different from the one currently in the row buffer. The contents of the row buffer first need to be written back into the memory array using the *precharge* command (opening a row destroys the row's contents in the memory array). The required row then needs to be opened and accessed using the *activate* and *read*/*write* commands. This results in the highest bank access latency $t_{RP} + t_{RCD} + t_{CL}$ (We later provide the values of these DRAM parameters in Table 2).

In all cases, transferring an entire cache line from/to the DRAM bank over the DRAM data bus incurs additional latency. The cache line is transferred using *burst mode* and a programmable *burst length* (BL) determines how many cycles the transfer takes.

2.2. DRAM Controller Organization

The DRAM controller is the mediator between processors and the DRAM. Its job is to satisfy processors' memory requests while obeying the timing and resource constraints of the DRAM banks, chips, and address/data buses. To do so, it translates processor requests into DRAM commands. A DRAM controller consists of the following structures:

- **Request buffer** holds state associated with each memory request (e.g. the address, type, identifier, age of the request, readiness, completion status). It can be organized as a single unified buffer for all banks or multiple per-bank buffers.
- **Read/Write data buffers** hold the data that is read from/written to the DRAM. Each *memory read/write* request is allocated an entry in its respective buffer until the request is completely serviced.
- **DRAM access scheduler** decides which DRAM command to issue every DRAM clock cycle. It consists of the logic that keeps track of the state of the DRAM banks/bus and the timing constraints of the DRAM. It takes as input the state of the memory requests in the request buffer along with the state of the DRAM banks/buses, and decides which DRAM command should be issued based on the implemented scheduling and access prioritization policies (which usually try to optimize memory bandwidth and latency). The structure of our baseline DRAM controller is later depicted in Figure 4.

2.3. DRAM Access Schedulers

Modern high-performance DRAM schedulers are implemented (logically and sometimes physically) as two-level structures [25]. The first level consists of the *per-bank schedulers*. Each per-bank scheduler maintains a logical priority queue of the memory requests waiting to be serviced in the bank it is associated with.[3] It selects the highest-priority request from that queue and issues DRAM commands to service that request (while respecting the bank timing constraints). The second level is the *across-bank channel scheduler* that takes as input all the commands selected by the per-bank schedulers and chooses the highest-priority command (while respecting the timing constraints and scheduling conflicts in the DRAM address and data buses). The prioritization algorithms implemented in these two levels determine which memory requests are prioritized over others.

2.4. State-of-the-art Scheduling Algorithms

DRAM schedulers can employ a variety of algorithms to prioritize the memory requests in the request buffer. The FR-FCFS algorithm [25, 24] has been shown to be the best performing one overall in single-threaded systems and it is used as the baseline in this paper (however, we also evaluate other previously-proposed algorithms such as a simple FCFS algorithm). FR-FCFS is designed to optimize the throughput obtained from the DRAM. To do so, it prioritizes DRAM commands in the following order:

1. **Column-first:** Ready *column accesses* (i.e. *read* and *write* commands) are prioritized over ready *row accesses* (i.e. *activate* and *precharge* commands).[4] This policy improves throughput by maximizing the row-buffer hit rate.

2. **Oldest-first:** Ready DRAM commands from older requests (i.e. requests that arrived earlier in the memory controller) are prioritized over those from younger requests. Note that a simple FCFS algorithm uses only this rule to prioritize commands.

Thus, with FR-FCFS, the oldest row-hit request has the highest priority, whereas the youngest row-conflict request has the lowest.

2.5. Thread-Unfairness of FR-FCFS Scheduling

The DRAM command prioritization policies employed by the FR-FCFS algorithm are unfair to different threads due to two reasons. First, the *column-first* policy gives higher priority to threads that have high *row-buffer locality*: If a thread generates a stream of requests that access different columns in the same row, another thread that needs to access a different row in the same bank will not be serviced until the first thread's column accesses are complete. For example, assuming 2KB row-buffer size per DRAM chip, 8 DRAM chips per DIMM, and 64-byte cache lines, $2KB * 8/64B = 256$ row-hit requests from a streaming thread can be serviced before a row-closed/conflict request from another thread. Second, the *oldest-first* policy implicitly gives higher priority to threads that can generate memory requests at a faster rate than others. Requests from less memory-intensive threads are not serviced until all earlier-arriving requests from more memory-intensive threads are serviced. Therefore, less memory-intensive threads suffer relatively larger increases in memory-related stalls.[5]

Our goal: Based on these observations, our goal in this paper is to design a new memory access scheduler that is fair to threads executing on different cores without sacrificing system throughput.

3. Stall-Time Fair Memory Scheduling: Approach and Algorithm

3.1. Stall-Time Fairness in DRAM Systems

Defining fairness in DRAM systems is non-trivial. Simply dividing the DRAM bandwidth evenly across all threads is insufficient, for instance, because this would penalize threads with "good" row-buffer locality, high parallelism, or in general, threads that by virtue of their memory access behavior are able to achieve a higher throughput to the DRAM system than others.

A thread's performance degradation due to DRAM interference is primarily characterized by its *extra memory-related stall-time* that is caused due to contention with requests from other threads. Because DRAM banks have limited bandwidth, simultaneously executing multiple threads on different cores inevitably causes the memory-related stall-time of threads to increase. The goal of a fair DRAM scheduler is therefore to balance these extra stall-times across different threads such that all threads exhibit a similar slowdown. This intuition highlights the need for the following, fundamentally new definition of DRAM fairness:

A stall-time fair DRAM scheduler schedules requests in such a way that the extra memory-related slowdown (due to interference caused by other threads) is equalized across all threads.[6]

In order to achieve stall-time fairness, we propose a novel DRAM scheduler that is based on the following basic idea. For each thread, the scheduler maintains two values: T_{shared} and T_{alone}. T_{shared} captures the memory-related stall-time (in processor cycles) experienced by the

[3] The "logical" priority queue is adjusted every DRAM cycle to sort the requests to the bank based on their priorities. The physical structure of the priority queue and the scheduler depend very much on the implementation. Many implementations use multiple priority encoders and arbiters to implement the priority-based selection of requests. Alternatively, some implementations use hardware priority queues [19] that are sorted every DRAM cycle.

[4] A DRAM command is said to be ready if it can be issued without violating the timing constraints and without resulting in bank or bus conflicts. DRAM commands that are *not ready* are not considered by the scheduler.

[5] The FCFS algorithm is also thread-unfair due to this second reason.

[6] The limitations of other notions of fairness in the context of shared DRAM systems are discussed in Section 4.

thread in the shared DRAM system, when running alongside other threads. On the other hand, T_{alone} expresses the estimated memory-related stall-time the thread would have had if it had run alone (without any contending threads on other cores). Based on these two estimates, the scheduler can compute for each thread its *memory-slowdown* $S = T_{shared}/T_{alone}$. Intuitively, a thread has high memory-slowdown S if its experienced memory-related stall-time is high, whereas without the interference caused by other threads, its stall time would have been low. Conversely, a thread's memory-slowdown S is low if the thread's memory stall-time is similar to when it runs alone. Our scheduler achieves stall-time fairness by prioritizing requests from threads with very high memory-slowdown S, thereby equalizing the experienced memory slowdown across all threads.

3.2. STFM: Stall-Time Fair Memory Scheduler

We first describe STFM assuming that all threads are equally important. We provide a discussion of how to incorporate thread weights in Section 3.3. Section 5 shows STFM's hardware implementation.

3.2.1. STFM Scheduling Policy: STFM estimates the two values T_{shared} and T_{alone} for each thread. Obtaining accurate estimates for T_{shared} is simple. The processor increases a counter when it cannot commit instructions due to an L2-cache miss. This counter is communicated to the memory scheduler. Obtaining accurate estimates for T_{alone} is more difficult and we discuss our techniques in a separate subsection. Assuming for now that the STFM scheduler knows each thread's slowdown $S = T_{shared}/T_{alone}$, it uses the following policy to determine the next command to be scheduled:

1) **Determine Unfairness:** From among all threads that have at least one ready request in the request buffer, the scheduler determines the thread with highest slowdown (S_{max}) and the thread with lowest slowdown (S_{min}).

2a) **Apply FR-FCFS-Rule:** If the ratio $S_{max}/S_{min} \leq \alpha$, then the acceptable level of unfairness is not exceeded and—in order to optimize throughput—the next DRAM command is selected according to the FR-FCFS priority rules described in Section 2.4.

2b) **Apply Fairness-Rule:** If the ratio $S_{max}/S_{min} > \alpha$, then STFM decreases unfairness by prioritizing requests of thread T_{max} with largest slowdown S_{max}. In particular, DRAM commands are prioritized in the following order:

- 2b-1) T_{max}**-first:** Ready commands from requests issued by T_{max} over any command from requests issued by other threads.
- 2b-2) **Column-first:** Ready column accesses over ready row accesses.
- 2b-3) **Oldest-first:** Ready commands from older requests over those from younger requests.

In other words, STFM uses either the baseline FR-FCFS policy (if the level of unfairness across threads with ready requests is acceptable), or a fair FR-FCFS policy in which requests from the most slowed-down thread receive highest priority.

3.2.2. Maintaining T_{alone}: Estimating T_{alone} is challenging because STFM needs to determine how much memory stall-time a thread would have accrued if it had executed by itself. Since directly determining T_{alone} while the thread is running with other threads is difficult, we express T_{alone} as $T_{alone} = T_{shared} - T_{Interference}$ and estimate $T_{Interference}$ instead. $T_{Interference}$ is the *extra stall-time* the thread experiences because requests from other threads are serviced by the DRAM ahead of this thread's requests. In order to compute each thread's memory slowdown S, the STFM scheduler maintains an estimate of $T_{Interference}$.

Initially, each thread's $T_{Interference}$ value is zero. *$T_{Interference}$ of each thread is updated whenever the STFM scheduler schedules a request.* For instance, when a request is issued to a DRAM bank, the extra stall-time $T_{Interference}$ of all other threads that have a *ready request* (i.e. a request that can be scheduled by the controller without violating timing constraints) to the same bank increases. These ready requests could have been scheduled if the thread that generated them had run by itself, but they were delayed due to interference from other threads, thereby increasing the thread's extra stall-time. Hence, the scheduler needs to adjust its estimate of $T_{Interference}$ appropriately.

When a request R from thread C is scheduled, our mechanism updates $T_{Interference}$ values of all threads. STFM updates $T_{Interference}$ differently for the request's own thread C versus for other threads as we explain below:

1. Updating other threads' $T_{Interference}$ values: The extra stall time a scheduled request inflicts on another thread that has an outstanding *ready request* consists of two portions: extra stall time due to interference in the (a) DRAM bus and (b) DRAM bank.

a) Updating $T_{Interference}$ due to interference in DRAM bus: When a read/write command is sent over the bus to a DRAM bank, it keeps the DRAM data bus busy for t_{bus} cycles.[7] During this time, no other thread is able to schedule a read/write command even though the commands might otherwise be ready to be scheduled. Hence, the $T_{Interference}$ of each thread (except thread C) that has at least one ready read/write command in the request buffer increases by t_{bus}.

b) Updating $T_{Interference}$ due to interference in DRAM bank: Because thread C has issued a request, other threads with requests to the same bank have to wait for R to be serviced and therefore experience an increased stall-time. However, increasing $T_{Interference}$ of these threads by the service latency of R is too simplistic as it ignores memory-level parallelism [8, 2] of threads. This is best illustrated with an example. Assume two requests $R1$ and $R2$ are simultaneously being serviced in two different banks. Assume further that another thread C' has ready requests for both of these banks that are waiting in the memory request buffer. As C''s requests need to wait for $R1$ and $R2$ to be serviced first, C' accrues extra stall-time. However, it would be overly pessimistic to assume that the extra stall-time caused by $R1$ and $R2$ is the *sum of the latencies of $R1$ and $R2$*. Instead, because they are serviced in parallel, these two requests cause extra stall-time only in the order of *one* memory access latency. This example highlights that our update mechanism needs to take into account the parallelism inherent to each thread. Our heuristic is that *if a thread C' has ready requests waiting to be serviced in X different banks, then the extra latency thread C' incurs due to the scheduling of request R from another thread is amortized across those waiting requests*. Hence, the extra stall-time of thread C' due to request R is approximately R's service latency divided by X. We call the value X the $BankWaitingParallelism(C')$ of thread C'.[8]

Concretely, STFM estimates the extra stall-time caused by a request to other threads as follows: When STFM schedules a DRAM command R from thread C to bank B, it increases the $T_{Interference}$ of any thread $C' \neq C$ that has at least one ready command waiting to be scheduled to bank B. If $Latency(R)$ is the service latency of R, the new $T^{new}_{Interference}(C') =$

$$T^{old}_{Interference}(C') + \frac{Latency(R)}{\gamma * BankWaitingParallelism(C')}$$

[7] The value of t_{bus} depends on the DRAM type, command type and burst length. For a read or write command, $t_{bus} = BL/2$ for DDR2 SDRAM.

[8] This is an approximation of the extra stall-time that is actually incurred. Exactly determining the extra stall-time is very difficult because it requires the determination of how much impact the delayed request has on a thread's performance (and how much $BankWaitingParallelism$ matters). There are more elaborate ways of approximating the extra stall-time (such as by determining whether the delayed request is on the critical path of execution), but they are much more difficult to implement.

The constant γ is a parameter that determines how aggressively the scheduler should consider its (potentially inaccurate) estimate of $BankWaitingParallelism$ in its updates of $T_{Interference}$. We use γ as a scaling factor because the actual bank parallelism value is an estimate: some of the waiting requests estimated to be serviced in parallel might not actually be serviced in parallel in the future. We set $\gamma = \frac{1}{2}$, which makes the logic to scale with γ trivial to implement.[9]

2. Updating own thread's $T_{Interference}$ value: Even the thread whose own request is being scheduled may experience extra stall-time, i.e., may be delayed more than it would have been if it had run alone. Consider a thread that has two consecutive requests R_1 and R_2 to the same row in the same bank. If this thread was running alone, its second request would result in a *row-hit* (with latency t_{CL}). In a shared DRAM system, however, it is possible that other requests from other threads are serviced between R_1 and R_2 and, therefore, R_2 could result in a *row-conflict* with much higher latency ($t_{RP} + t_{RCD} + t_{CL}$).

To account for this potential extra stall time, STFM determines whether a scheduled request would have been a row-hit or a row-conflict had the thread run alone. Determining this is simple; we only need to maintain the address of the last accessed row by each thread in each bank. If the scheduled request is a row-conflict, but it would have been a row-hit had thread C run alone, then C's $T_{Interference}$ increases by the difference in latency between a row-conflict and a row-hit ($ExtraLatency = t_{RP} + t_{RCD}$)[10] divided by the *bank access parallelism* of C, i.e.,

$$T^{new}_{Interference}(C) = T^{old}_{Interference}(C) + \frac{ExtraLatency}{BankAccessParallelism(C)}$$

We do not add the full $ExtraLatency$ to $T_{Interference}$ because the whole $ExtraLatency$ might not manifest itself as extra stall time for thread C. If more than one of C's requests are being serviced in parallel in different DRAM banks, some of the $ExtraLatency$ will remain hidden because it will be amortized across those concurrent requests [8, 2]. Therefore, we divide $ExtraLatency$ by $BankAccessParallelism$. $BankAccessParallelism$ is the number of requests that are *currently being serviced* in DRAM banks by this thread. In other words, it is the number of banks that are kept busy due to Thread C's requests.

3.3. Support for System Software & Thread Weights

So far, we have assumed that fairness should be enforced by equalizing the threads' memory-related slowdowns. However, this may not always be desirable at the system level as the system software (i.e. operating system or virtual machine monitor) might:

1. not want fairness to be enforced by the hardware at all because that could possibly interact adversely with the system software's own high-level fairness mechanisms (such as fair thread scheduling).
2. not want each and every thread to be treated equally because some threads can be (and usually are) more/less important than others. In this case, some threads should be allowed to be slowed down more than others.

We adjust STFM to seamlessly incorporate enough flexibility to support the system software. First, the threshold α that denotes the maximum tolerable unfairness can be set by the system software via a privileged instruction in the instruction set architecture. If the system software does not need hardware-enforced fairness at the DRAM controller it can simply supply a very large α value.

[9] We determined γ empirically. Our simulations show that setting $\gamma = \frac{1}{2}$ captures the average degree of bank parallelism accurately.

[10] The row-closed case follows the same principle. For brevity, we explain only the row-conflict case. Moreover, it is possible that the interference between two threads is *positive* especially if the threads share code or data. In other words, a request might result in a row-hit in the shared system whereas it would have been a row-conflict had the thread run alone. In that case, $ExtraLatency$ would be negative; our scheme considers all possibilities.

Second, to support different treatment of threads based on their importance, we add the notion of *thread weights* to our mechanism. The system software conveys the *weight* of each thread to STFM. The smaller the weight, the less important the thread and the more tolerable its slowdown. Threads with equal weights should still be slowed down equally. To support this notion of thread weights and to prioritize threads with larger weights, STFM scales the slowdown value computed for the thread by the thread's non-negative weight such that the *weighted slowdown* is $S = 1 + (S - 1) * Weight$. That is, threads with higher weights are interpreted to be slowed down more and thus they are prioritized by STFM. For example, for a thread with weight 10, a measured slowdown of 1.1 is interpreted as a slowdown of 2 whereas the same measured slowdown is interpreted as 1.1 for a thread with weight 1. Note that even after this modification, it is the ratio S_{max}/S_{min} that determines whether or not the fairness-rule is applied. Measured slowdowns of equal-weight threads will be scaled equally and thus those threads will be treated equally by the scheduler.

4. Comparison with Existing DRAM Schedulers

We compare the notion of stall-time fairness and the corresponding STFM scheduler with other ways of defining and enforcing fairness in shared DRAM memory systems. The existing DRAM schedulers we are aware of suffer from the fact that they disregard the inherent memory performance (properties) of different threads, which leads to extremely variable behavior in terms of fairness for a given DRAM scheduler. We already showed how this limitation manifests itself in the FR-FCFS scheduling scheme. We now examine three other scheduling schemes that could provide better fairness than FR-FCFS.

FCFS: The simplest fairness mechanism is to use a first-come first-serve scheduling policy (among ready DRAM commands), disregarding the current state of the row-buffer. However, FCFS starves threads that do not issue a lot of memory accesses because their requests get backed up in the request buffer behind the large number of requests from memory-intensive threads. Since FCFS completely neglects the potential performance gain from exploiting row-buffer locality, the achieved DRAM throughput deteriorates significantly. As a result, the overall system performance degrades.

FR-FCFS with a Cap on Column-Over-Row Reordering (FR-FCFS+Cap): This is a new algorithm that addresses one major source of unfairness in FR-FCFS: the reordering of younger column (row-hit) accesses over older row (row-closed/conflict) accesses. The algorithm enforces a *cap* on the number of younger column accesses that can be serviced before an older row access to the same bank. When the cap is reached, the FCFS policy is applied. While such a cap alleviates the problem that threads with poor row-buffer locality are penalized, it does not solve the FCFS-inherent problem of penalizing non-memory-intensive threads.

Network fair queueing (NFQ): A more involved alternative fairness definition for shared DRAM memory systems has been proposed in the form of *network fair queueing* [23]. The objective of a memory scheduler based on network fair queueing is [22]: *A thread i that is allocated a fraction ϕ_i of the memory system bandwidth will run no slower than the same thread on a private memory system running at ϕ_i of the frequency of the shared physical memory system.*

One problem with this fairness definition is that it may be impossible to enforce it. DRAM scheduling, as opposed to packet scheduling on a wire, is not a pure bandwidth allocation problem. The fundamental observation is that sustained memory bandwidth does not correspond to observed performance when different threads interfere. If several threads access the shared memory system at the same time, they can destroy each other's row-buffer locality and bank parallelism (i.e. the number of requests serviced in parallel in different banks), and hence, all threads can be slowed down more than their "fair NFQ-share" suggests. Therefore, even if a thread is guaranteed "some"

amount of bandwidth as indicated by its fair NFQ-share, it is not guaranteed "some" amount of performance. For this very reason, the NFQ notion of fairness also cannot provide performance isolation in shared DRAM memory systems.[11] As such, NFQ-based fairness definitions are suited primarily for *stateless systems that do not have parallelism* (such as a network wire) as opposed to shared DRAM memory systems that have state (row buffers) and parallelism (multiple banks), which affect the performance impact of the scheduling decisions.

In practice, schedulers that enforce NFQ-fairness have typically made use of *earliest virtual deadline first* schemes, e.g. [13, 22]. A virtual deadline (or virtual finish-time) is the virtual time a thread's memory request will finish on the thread's private virtual time memory system. In the context of shared DRAM memory systems, Nesbit et al. [22] propose an NFQ-based memory scheduler that maintains such virtual deadlines for each thread in each bank as follows. When a request of a thread is serviced, the thread's virtual deadline in this bank is increased by the request's access latency times the number of threads in the system. The intuition is that if a thread shares the DRAM memory system with X other threads, it should get $1/X$th of the total DRAM bandwidth and hence, its requests can be slowed down by a factor of X. The NFQ-based memory scheduler prioritizes requests from threads that have an early deadline. The premise is that by doing so, every thread's requests will be scheduled within the time-limits set by the NFQ definition, and hence, the resulting DRAM performance would be fairly shared among all threads.

Unfortunately, in addition to the fact that NFQ-fairness may be theoretically unachievable, these NFQ-deadline-based strategies suffer from practical shortcomings as they fail to properly take into account the inherent memory access characteristics of individual threads. This problem manifests itself in several ways. For the sake of brevity, we discuss two most important ones.

NFQ: Idleness Problem: A severe problem inherent to NFQ-based approaches is the *idleness problem*. Consider the 4-core scenario illustrated in Figure 3. Thread 1 continuously issues memory requests, whereas the three remaining threads generate requests in bursts with idle periods in between. Until time $t1$, only Thread 1 issues requests. These requests are scheduled without interference; hence, Thread 1's virtual deadline advances. At time $t1$, Thread 1's virtual deadline is very large and Thread 2's virtual deadline is zero; hence, Thread 2's requests are prioritized in the interval $[t1, t2]$ and Thread 1 is starved. Similarly, in the interval $[t2, t3]$, Thread 3's requests are prioritized over Thread 1, and so on. Overall, the non-bursty Thread 1 suffers significant performance loss over its bursty competitors *even though it fairly utilized the shared DRAM resources in the interval* $[0, t1]$ *when no other thread needed them*. In fact, the problem can be quite dramatic because Thread 1 may be completely denied access to DRAM memory during certain intervals. In contrast, our stall-time fair scheduler treats all four threads fairly: At time $t1$, for instance, STFM realizes that neither Thread 1 nor Thread 2 has been slowed down and hence, they are treated equally during $[t1, t2]$. The example shows that NFQ-based schemes become unfair if some threads are bursty and remain idle for some time. When such threads resume issuing memory requests, they capture the DRAM bandwidth, starving threads that had run previously.

NFQ: Access Balance Problem: Another problem of NFQ-based approaches arises when some threads spread their accesses across many banks, while others mostly access a small number of banks

[11] NFQ does indeed provide performance isolation in networks where flows are scheduled over a single, memoryless channel. In such "stateless" systems, fair scheduling is purely a fair bandwidth allocation problem—since bandwidth directly correlates with performance (because what was scheduled previously does not affect the latency of what is scheduled next). In contrast, the existence of row-buffer state and multiple banks in the DRAM system eliminates the direct relationship between bandwidth and performance.

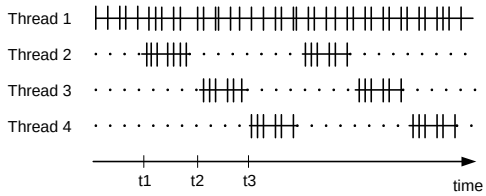

Figure 3. Idleness problem on a 4-core system. Each vertical line represents a DRAM memory request. Memory schedulers based on network fair queueing treat the non-bursty Thread 1 unfairly.

(good vs. poor bank access balance). Threads with unbalanced bank usage are penalized because in the few banks they typically access their virtual deadlines accrue much faster than deadlines of other threads. In these critical banks, unbalanced threads are therefore deprioritized compared to threads whose requests are well-balanced across all banks. This leads to starvation and high memory-related slowdown for threads with poor bank access balance. In a sense, such threads suffer from the idleness problem in an individual bank. STFM avoids this problem because it implicitly takes into account any performance slowdown incurred by a thread due to poor bank access balance.

In summary, all alternative memory access schedulers including the NFQ-based approach suffer from the fact that they do not sufficiently consider the inherent memory characteristics of different threads. As our evaluations in Section 7 show, problems such as the *idleness* or the *access balance* problem can result in significant unfairness and performance loss in real workloads.

5. Implementation

STFM is implemented by modifying the baseline FR-FCFS scheduler to incorporate an additional priority policy: prioritization of commands based on the slowdowns of the threads they belong to. The basic structure of the memory controller (as explained in Sections 2.2 and 2.3) is not changed. However, additional circuitry is added to (1) estimate the slowdown of each thread, (2) compute the unfairness in the system, and (3) prioritize commands based on the slowdowns of the threads they belong to.

Figure 4 shows the organization of the on-chip STFM memory controller. Additional logic required for implementing STFM is boxed. We briefly describe the key aspects of this logic. The logic of the STFM controller is very similar to that of the baseline controller except for the additional STFM logic which sits on the side and communicates with the baseline scheduling logic.

5.1. State Required to Estimate Unfairness

To estimate and store the memory-related slowdown S of each thread, the STFM scheduler maintains a set of registers per hardware thread. These per-thread registers are reset at every context switch and at regular intervals (every $IntervalLength$ cycles) to adapt to threads' time-varying phase behavior. Table 1 describes the set of registers that need to be maintained in one implementation of STFM (this is the implementation we evaluate in Section 7). Additionally, each entry in the memory request buffer stores the ID of the thread (thread-ID) that generated the memory request. With 8 threads, an $IntervalLength$ value of 2^{24}, 8 DRAM banks, 2^{14} rows per bank, and a 128-entry memory request buffer, the additional state required by STFM is 1808 bits.

T_{shared} for each thread is the only counter computed in the processor core and communicated to the DRAM scheduler periodically (in our implementation, with every memory request). The processor core increments T_{shared} if the thread cannot commit instructions because the oldest instruction is an L2 miss. $T_{interference}$ and $Slowdown$ registers are updated when a DRAM command is scheduled. $BankWaitingParallelism$ registers and $IntervalCounter$ are updated every DRAM cycle. $BankAccessParallelism$ register for a thread is incremented when a DRAM command for that

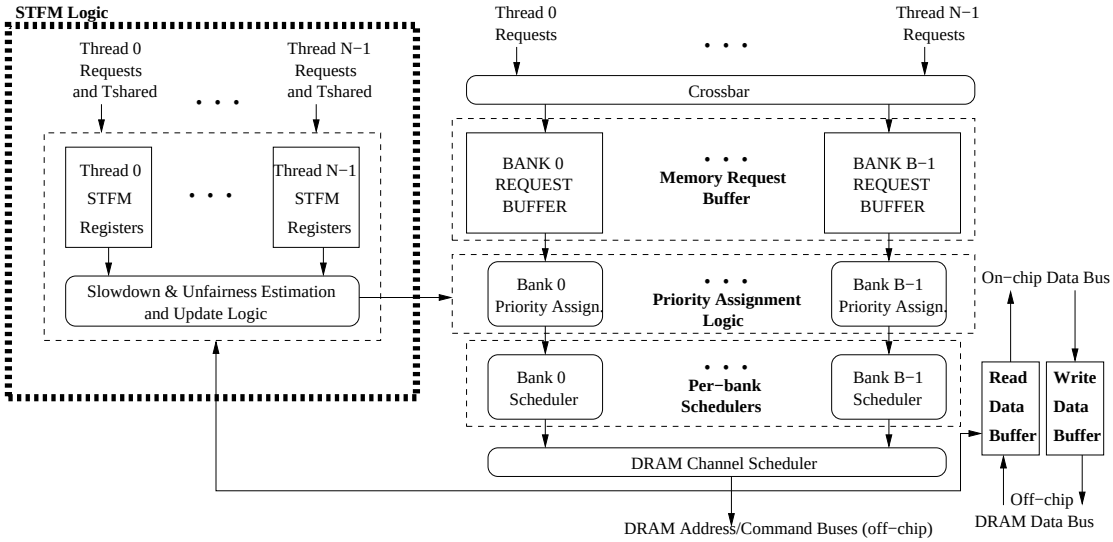

Figure 4. Organization of the on-chip STFM memory controller. Only major control and data paths are shown to keep the figure simple.

Register	Function	Size (bits)
Per-thread registers used to compute and store slowdowns		
T_{shared}	Number of cycles in which the thread cannot commit instructions due to L2 miss (supplied by the core)	$log_2 IntervalLength$ (24)
$T_{interference}$	Number of extra stall cycles due to interference from other threads (computed in the memory controller)	$log_2 IntervalLength$ (24)
$Slowdown$	T_{shared}/(T_{shared}-$T_{interference}$)	8 (fixed point)
$BankWaitingParallelism$	Number of banks that have at least one request that is waiting to be serviced for the thread	$log_2 NumBanks$ (3)
$BankAccessParallelism$	Number of banks that are currently servicing requests from the thread	$log_2 NumBanks$ (3)
Per-thread per-bank registers used in estimating the extra latency due to inter-thread interference		
$LastRowAddress$	The last row address accessed by thread i in bank b	$log_2 NumRowsInBank$ (14)
Per-request registers stored in the memory request buffer to perform prioritization		
$ThreadID$	The ID of the thread that generated the memory request	$log_2 NumThreads$ (3)
Individual registers		
$IntervalCounter$	Counter used to reset other registers when it reaches a maximum threshold value $IntervalLength$	$log_2 IntervalLength$ (24)
$Alpha$	Register used to store the α value (which can be set by system software)	8 (fixed point)

Table 1. Registers used in our STFM implementation

thread is scheduled and decremented when the command is completely serviced. When a thread initiates a row access in a bank, the $LastRowAddress$ register for that thread-bank pair is updated to store the address of the accessed row. Slowdowns are computed as described in Section 3.2. Logic required to update all the registers consists of adders/subtracters, muxes, and shifters (to approximate fixed-point division and multiplication). The update logic can be pipelined (if needed) and components can be shared by different threads. There is ample flexibility to accommodate these changes as the on-chip DRAM controller is not on the critical path of execution and it only needs to make a decision every DRAM cycle, which is significantly longer than the processor core's cycle time.

5.2. Prioritization and Scheduling Logic

Every DRAM cycle, the memory controller orders threads with at least one ready command based on their $Slowdown$ values. It also computes unfairness by dividing the maximum slowdown value by the minimum slowdown. If the unfairness computed in the previous DRAM cycle is greater than α, the controller prioritizes commands from threads with higher $Slowdown$ values. Otherwise, it prioritizes commands using the baseline FR-FCFS policy. Prioritization of commands can be implemented in several different ways. Our baseline FR-FCFS implementation assigns a single priority value to each ready command based on its type (column or row access) and arrival time. A priority encoder selects the command with the highest priority value. STFM adds only one more variable into the computation of the priority value of each ready command. If unfairness is greater than α at the beginning of a DRAM cycle, each ready command is assigned a priority value based on its Thread-ID (i.e. slowdown), type, and arrival time. Otherwise, baseline FR-FCFS priority assignments are used. This implementation changes only the priority assignment logic without affecting the structure of request buffers or priority encoders.

6. Experimental Evaluation Methodology

We evaluate STFM using a cycle-accurate x86 CMP simulator. The functional front-end of the simulator is based on Pin [14]. The DRAM memory system model and the performance model are loosely based on DRAMsim [31] and Intel Pentium M [9], respectively. Table 2 presents the major parameters for both DRAM and processors. We scale the number of DRAM channels with the number of cores so that configurations with more cores are not unfairly penalized in terms of DRAM bandwidth.

6.1. Benchmarks: Characteristics and Classification

We use the SPEC CPU2006 benchmarks as well as desktop applications (see Section 7.4) for evaluation.[12] We classify the benchmarks into four categories based on their memory intensiveness (low or high) and row-buffer locality (low or high). Table 3 shows the category and characteristics of the benchmarks when they run alone in the memory system. Each benchmark was compiled using gcc 4.1.2 with -O3 optimizations and run for 100 million instructions chosen from a phase as determined by the SimPoint tool [27]. Benchmarks are ordered based on their memory intensiveness in Table 3 and in all the figures.

We evaluate combinations of multiprogrammed workloads running on 2, 4, 8, and 16-core CMPs. Obviously, evaluating each combination of 4 benchmarks on a 4-core system requires an enormous amount of simulation time. Therefore, we have evaluated combinations of benchmarks from different categories. For 4-core simulations, we evaluated

[12] 410.bwaves, 416.gamess, and 434.zeusmp are not included because we were not able to collect representative traces for them.

Processor pipeline	4 GHz processor, 128-entry instruction window, 12-stage pipeline
Fetch/Exec/Commit width	3 instructions per cycle in each core; only 1 can be a memory operation
L1 Caches	32 K-byte per-core, 4-way set associative, 64-byte block size, 2-cycle latency
L2 Caches	512 K-byte per core, 8-way set associative, 64-byte block size, 12-cycle latency, 64 MSHRs
DRAM controller	on-chip; 128-entry req. buffer, FR-FCFS/open-page policy, 32-entry write data buffer, reads prioritized over writes, XOR-based addr-to-bank mapping [6, 32]
DRAM chip parameters	Micron DDR2-800 timing parameters (see [18]), t_{CL}=15ns, t_{RCD}=15ns, t_{RP}=15ns, $BL/2$=10ns; 8 banks, 2K-byte row-buffer per bank
DIMM configuration	single-rank, 8 DRAM chips put together on a DIMM (dual in-line memory module) to provide a 64-bit wide data interface to the DRAM controller
Round-trip L2 miss latency	For a 64-byte cache line, uncontended: row-buffer hit: 35ns (140 cycles), closed: 50ns (200 cycles), conflict: 70ns (280 cycles)
Cores and DRAM channels	Channels scaled with cores: 1, 1, 2, 4 parallel lock-step 64-bit wide channels for respectively 2, 4, 8, 16 cores (1 channel has 6.4 GB/s peak bandwidth)

Table 2. Baseline processor and DRAM system configuration

#	Benchmark	Type	MCPI	L2 MPKI	RB hit rate	Category	#	Benchmark	Type	MCPI	L2 MPKI	RB hit rate	Category
1	429.mcf	INT	10.02	101.06	41.9%	2	14	464.h264ref	INT	0.71	3.22	65.3%	1
2	462.libquantum	INT	9.10	50.00	98.4%	3	15	401.bzip2	INT	0.55	3.55	41.4%	0
3	437.leslie3d	FP	7.82	36.21	82.5%	3	16	435.gromacs	FP	0.37	1.26	41.0%	1
4	450.soplex	FP	7.48	45.66	63.9%	3	17	445.gobmk	INT	0.19	0.94	56.8%	1
5	433.milc	FP	6.74	51.05	91.77%	3	18	447.dealII	FP	0.16	0.86	90.2%	1
6	470.lbm	FP	6.44	43.46	54.6%	3	19	481.wrf	FP	0.14	0.77	76.9%	1
7	482.sphinx3	FP	5.49	24.97	57.8%	3	20	458.sjeng	INT	0.12	0.51	23.4%	0
8	459.GemsFDTD	FP	3.87	17.62	0.2%	2	21	444.namd	FP	0.11	0.54	72.6%	1
9	436.cactusADM	FP	3.53	14.66	2.0%	2	22	465.tonto	FP	0.07	0.39	34.5%	0
10	483.xalancbmk	INT	3.18	21.66	54.8%	3	23	403.gcc	INT	0.07	0.42	58.6%	1
11	473.astar	INT	2.02	9.25	44.8%	0	24	454.calculix	FP	0.05	0.29	71.8%	1
12	471.omnetpp	INT	1.78	13.83	21.9%	0	25	400.perlbench	INT	0.03	0.20	69.8%	1
13	456.hmmer	INT	1.52	5.82	32.7%	0	26	453.povray	FP	0.01	0.09	76.6%	1

Table 3. Characteristics of benchmarks. MCPI: Memory Cycles Per Instruction (i.e. cycles spent waiting for memory divided by number of instructions), L2 MPKI: L2 Misses per 1000 Instructions, RB Hit Rate: Row-buffer hit rate, **Categories:** 0 (Not-intensive,Low RB hit rate), 1 (Not-intensive,High RB hit rate), 2 (Intensive,Low RB hit rate), 3 (Intensive,High RB hit rate)

256 combinations; for 8-core, 32 combinations; and for 16-core, 3 combinations. Space limitations prevent us from enumerating all evaluated combinations, but Section 7 tries to show as many results with representative individual combinations as possible.[13]

6.2. Metrics for Fairness and System Throughput

Our fairness metric is the *unfairness index* of the system, which is the ratio between the maximum memory-related slowdown and the minimum memory-related slowdown among all threads sharing the DRAM system.[14] The unfairness index of a perfectly-fair system is 1 and a perfectly-unfair system is *infinity*. Memory related slowdown of each thread *i* is computed by dividing the memory stall time per instruction a thread experiences when running together with other threads with the memory stall time per instruction it experiences when running alone in the same memory system using the FR-FCFS policy.

$$MemSlowdown_i = \frac{MCPI_i^{shared}}{MCPI_i^{alone}}, Unfairness = \frac{\max_i MemSlowdown_i}{\min_i MemSlowdown_i}$$

We measure overall system throughput using the *weighted speedup* metric [28], defined as the sum of relative IPC performances of each thread in the evaluated workload: $Weighted\ Speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}}$. We also report results using the *hmean speedup* metric that balances fairness and throughput [15]. This is the harmonic mean of the relative IPC performance of each thread in the workload:

$$Hmean\ Speedup = NumThreads / \sum_i \frac{1}{IPC_i^{shared}/IPC_i^{alone}}$$

Finally, the *sum of IPCs* metric considers IPC throughput only, without taking into account fairness (or forward progress) at all and thus *should be interpreted with extreme caution*: $Sum\ of\ IPCs = \sum_i IPC_i^{shared}$. This metric should not be used to evaluate system throughput [28, 15] since even throughput-oriented realistic systems need to consider fairness and ensure forward progress of individual threads. We report results with this metric *only* to provide deeper insights into some results and to show that some scheduling algorithms unfairly speed up non-memory-intensive threads (thereby improving sum-of-IPCs).

[13]The evaluated combinations are shown in `http://research.microsoft.com/~onur/pub/stfm-workloads.txt`.

[14]This metric is the inverse of the fairness metric proposed in [7]. We use unfairness instead of fairness since we believe looking at unfairness is a more intuitive way of understanding the system. A large unfairness value immediately shows the ratio of maximum and minimum slowdowns without requiring any calculation.

6.3. Parameters Used in Evaluated Schemes

STFM: We set $\alpha = 1.10$ and $IntervalLength = 2^{24}$ in the proposed STFM scheme. Increasing α increases the amount of unfairness. We found that $IntervalLength$ value does not impact fairness or throughput unless it is less than 2^{18}, in which case STFM becomes less effective in enforcing fairness because its slowdown estimates become less reliable due to the short sampling intervals. We evaluate STFM in comparison to three other scheduling algorithms (FCFS, FR-FCFS+Cap, and NFQ) described in Section 4. **FR-FCFS+Cap:** In our implementation, the *cap* is set to 4 based on empirical evaluation. Hence, only 4 younger column accesses can bypass an older row access. **NFQ-based scheduling (NFQ):** We use Nesbit et al.'s best scheme (FQ-VFTF) presented in [22]. We use the priority inversion prevention optimization they propose in Section 3.3 of [22] with a threshold of t_{RAS} (the same threshold used in [22]). This optimization limits the amount of prioritization of younger column accesses over older row accesses.[15]

7. Experimental Results

7.1. STFM on Dual-core Systems

To evaluate STFM on 2-core systems, we run the mcf benchmark concurrently with every other benchmark. Figure 5(a) shows the memory slowdowns experienced by mcf and the concurrently running benchmark with the baseline FR-FCFS scheduler. Note the wide variance in the slowdowns of benchmarks. When mcf is run with dealII, mcf experiences a slowdown of only 1.05X whereas dealII slows down by 4.5X.[16] In contrast, when mcf is run with libquantum, libquantum's slowdown is negligible (1.04X) while mcf slows down by 5.28X, resulting in an unfairness of 5.08. This is due to the very high row-buffer locality in libquantum. FR-FCFS prioritizes libquantum's row-hit requests over mcf's row-conflict requests and starves mcf as libquantum is memory-intensive enough to continuously generate DRAM requests. Mcf's impact and thus unfairness is drastic especially on non-memory-intensive benchmarks (to the right of the figure): these benchmarks' slowdowns are almost always more than 2X because mcf is

[15]In fact, we found the behavior of NFQ (as implemented with the FR-VFTF scheme of [22]) is similar to FR-FCFS without this optimization.

[16]We found that dealII's DRAM accesses are heavily skewed to only two DRAM banks. As mcf generates DRAM accesses much more frequently than dealII, mcf's DRAM requests that go to those two banks are almost always prioritized over dealII's requests by the FCFS nature of the FR-FCFS scheduler.

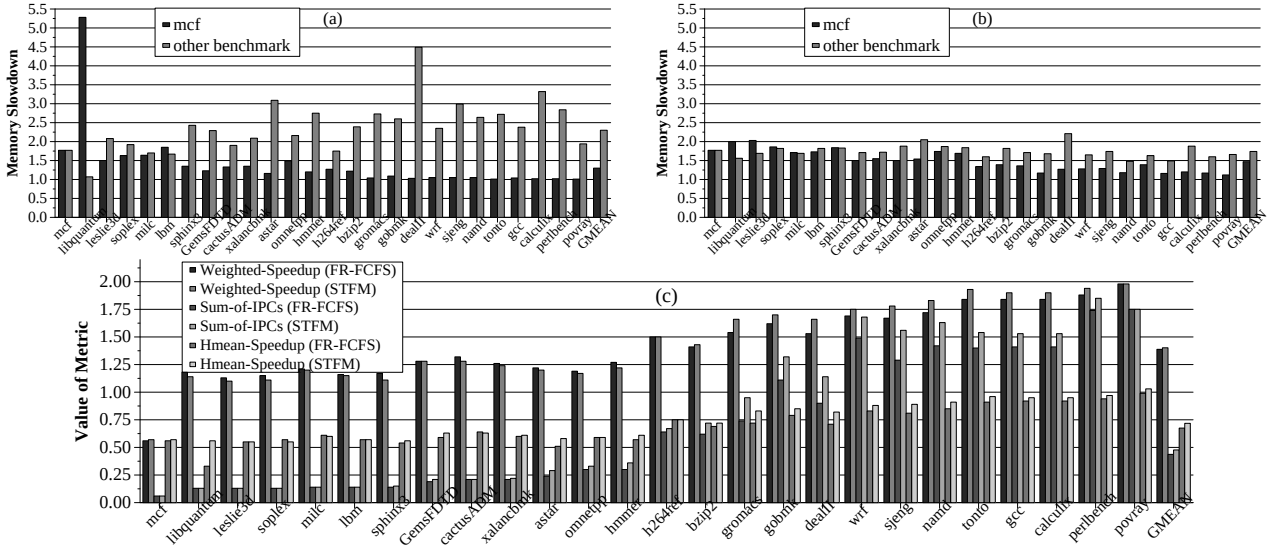

Figure 5. Memory-slowdown of each of the two threads when mcf is run with all others in a 2-core system using FR-FCFS scheduling (a) and STFM (b). Bottom figure compares FR-FCFS with STFM in terms of Weighted Speedup, Sum of IPCs, and Hmean Speedup.

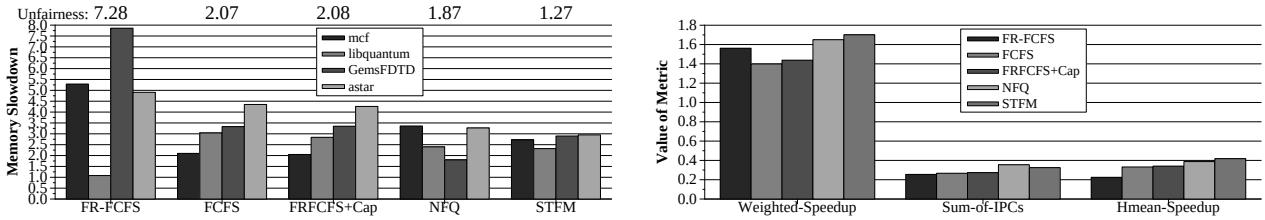

Figure 6. A memory intensive 4-core workload: memory slowdowns and unfairness (left), throughput metrics (right)

able to flood the memory system and get prioritized due to the first-come-first-serve nature of FR-FCFS scheduling.

Figure 5(b) shows the corresponding slowdowns when STFM is used. STFM brings the memory-slowdowns of the two threads to very close values. The maximum unfairness observed is 1.74 and the average (geometric mean) unfairness is reduced by 76%, from 2.02 to 1.24.[17] Figure 5(c) provides insight into the balance between throughput and fairness in STFM. STFM improves weighted speedup by 1% and hmean-speedup by 6.5%. Hence, STFM improves not only fairness but also performance. Performance improvement comes mainly from better system utilization: without fair memory scheduling, non-memory-intensive threads slow down too much and cannot make progress when run with memory-intensive ones (in this case mcf). STFM prevents the starvation of such threads' infrequent requests and thus allows them to make faster progress. This is supported by the data in Figure 5(c): STFM's throughput improvement is most salient when mcf is run with non-intensive benchmarks.

7.2. STFM vs. Other Scheduling Techniques: Case Studies and Results on 4-Core Systems

Fairness becomes a much larger problem as the number of cores sharing the DRAM system increases and as the threads diversify in terms of their memory behavior. We demonstrate this by comparing STFM to other scheduling techniques on diverse workloads running on 4, 8, and 16-core systems. We start with case studies of three typical workloads running on 4-core systems.

7.2.1. Case study I: Memory-intensive workload (3 intensive benchmarks run with 1 non-intensive benchmark) Figure 6(left) shows the memory slowdown of each thread when five different memory controllers are used. The unfairness of each controller is denoted on top of the figure. Figure 6(right) compares the effect of controllers on throughput using three different metrics. Several observations are in order:

- As seen before, FR-FCFS heavily prioritizes libquantum over other threads due to its high row-buffer locality and memory intensiveness. GemsFDTD is heavily penalized because its row-buffer hit rate is extremely low (0.2% as shown in Table 3).
- FCFS eliminates the unfairness due to row-buffer locality exploitation, but it unfairly prioritizes heavily memory-intensive threads (mcf and libquantum) over others (because older requests tend to be from memory-intensive threads). FCFS results in a higher slowdown for libquantum than mcf even though the L2 miss rate of these two threads are very similar. This is because the baseline memory performance of libquantum is much higher than that of mcf due to libquantum's high row-buffer locality. FCFS increases fairness but degrades system performance (in terms of weighted speedup) compared to FR-FCFS.
- FRFCFS+Cap improves throughput compared to FCFS because it is able to better exploit row buffer locality. For this reason, it slightly improves the slowdowns experienced by all threads except GemsFDTD, which has the lowest row-buffer hit rate.
- NFQ provides better fairness and throughput than both FCFS and FRFCFS+Cap for this workload. However, it penalizes (slows down) mcf significantly (by 3.4X). We found that mcf continuously generates memory requests whereas the other three benchmarks have bursty memory access patterns. This is exactly the *idleness problem* discussed in Section 4. Due to its design, NFQ prioritizes bursty threads over non-bursty ones. Astar also slows down significantly (by 3.3X) with NFQ, which is due to the *access balance problem* (Section 4). Astar's accesses are heavily concentrated in two DRAM banks whereas other threads have much

[17] Percentage reduction in unfairness is calculated relative to 1 since unfairness cannot take a value less than 1.

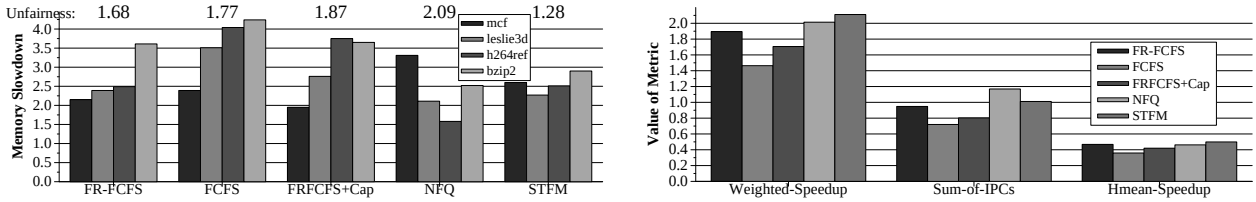

Figure 7. A mixed-behavior 4-core workload: memory slowdowns and unfairness (left), throughput metrics (right)

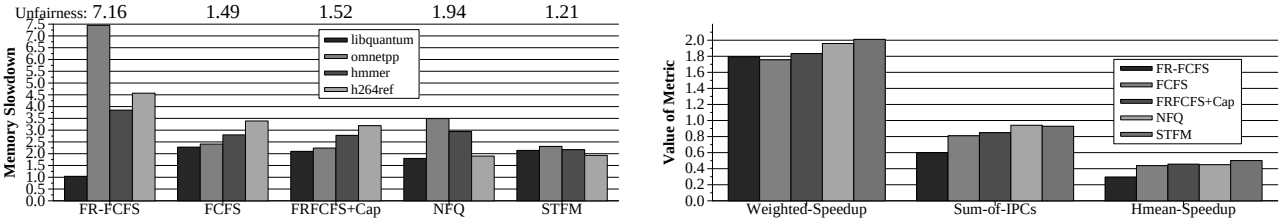

Figure 8. A non-memory-intensive 4-core workload: memory slowdowns and unfairness (left), throughput metrics (right)

more uniform access distribution to banks. Therefore, astar's virtual deadlines lag significantly on those two banks, which results in its deprioritization when other threads also access those banks.

- STFM provides the best fairness substrate, improving unfairness from 1.87 (NFQ) to 1.27. It also improves weighted speedup by 3% and hmean-speedup by 8% over NFQ. The sum-of-IPCs of NFQ is slightly better than that of STFM because NFQ unfairly prioritizes a less memory-intensive thread (GemsFDTD), which is able to execute its instructions much faster. We found that STFM is not able to achieve perfect fairness because it slightly underestimates the slowdown of libquantum in this case. Improving STFM's technique to estimate thread slowdowns could therefore result in even better fairness.

7.2.2. Case study II: Mixed workload (2 intensive benchmarks run with 2 non-intensive benchmarks) Figure 7 shows the unfairness and throughput comparison of different schedulers for a mixed-behavior workload that contains benchmarks from all four different categories. FR-FCFS is not as unfair for this workload as it was for the previous one because the variance between the row-buffer locality of these four benchmarks is relatively low (see Table 3). Therefore, eliminating or reducing row-buffer locality exploitation in the scheduler by implementing FCFS or FRFCFS+Cap actually *increases* unfairness while also reducing system performance! This shows that the unfairness behavior of FR-FCFS is very much dependent on the workload running on the system.

NFQ *increases* unfairness compared to FR-FCFS for this mixed workload because it implicitly prioritizes the bursty and non-intensive threads leslie3d and h264ref over the intensive thread mcf (idleness problem). This improves sum-of-IPCs but degrades hmean-speedup (i.e. it degrades the balance between fairness and throughput). STFM achieves the best fairness (1.28) while improving weighted-speedup by 4.8% and hmean-speedup by 8% over NFQ. As with the previous workload, STFM lags NFQ on sum-of-IPCs because it does not unfairly prioritize non-intensive threads. On the contrary, STFM tends to slightly favor intensive threads whose memory parallelism and thus slowdown is more difficult to estimate (e.g. leslie3d) over non-intensive ones.

7.2.3. Case study III: Non-memory-intensive Workload (1 intensive benchmark run with 3 non-intensive benchmarks) As shown in Figure 8, FCFS significantly reduces unfairness *and* improves sum-of-IPCs for this workload because it allows all three non-intensive threads to make much faster progress rather than being starved until libquantum's row-hit requests are serviced. FR-FCFS+Cap further improves throughput (all three metrics) over FCFS without significantly sacrificing fairness, as it enables faster progress in libquantum and h264ref, which have high row-buffer hit rates.

NFQ is less fair than FCFS in this workload, resulting in a memory slowdown of 3.47X for omnetpp. The reason is that in the banks accessed by both h264ref and omnetpp, NFQ prioritizes h264ref's accesses because (1) h264ref's accesses are bursty, (2) h264ref has better row-buffer locality which is exploited by NFQ up to some static threshold [22]. This unfair prioritization leads to a reduction in omnetpp's bank parallelism and thus serializes omnetpp's requests. Such serialization significantly degrades omnetpp's performance because the processor stalls for the latency of each miss rather than amortizing the memory latency by overlapping multiple misses. A similar behavior is also observed for hmmer.

In contrast, STFM results in the smallest unfairness value (1.21) while also providing the best performance, improving weighted speedup (2.7%) and hmean-speedup (11.3%) over NFQ.

Summary of Case Studies: These three case studies provide insight into why previously proposed memory access scheduling techniques cannot consistently provide sufficient fairness. As pointed out in Section 4, the major reason is that these previous techniques do not take into account the inherent memory behavior and performance of each thread. This inherent performance is dependent on many properties, including memory intensiveness, row-buffer locality, bank access balance, and memory parallelism. For this very reason, different scheduling policies (other than STFM) provide widely varying fairness values in different workloads. For example, the scheduling policy that provides the second-best fairness value is NFQ for the memory-intensive workload whereas it is FCFS for the two other workloads. STFM incorporates the effect of all these machine-dependent properties in its estimate of $Slowdown$ for each thread and therefore it is able to provide much better fairness than any of the other techniques.

Average Results on 4-Core Systems: We conclude the section by showing the unfairness of different scheduling techniques on 10 other sample workloads along with the unfairness averaged across the 256 different combinations of benchmarks from the different categories in Figure 9(left). The average unfairness of FR-FCFS, FCFS, FR-FCFS+Cap, NFQ, and STFM techniques are respectively 3.42, 1.80, 1.65, 1.58, and 1.24. Hence, STFM provides the best fairness. As shown in Figure 9(right), STFM also provides the best system throughput: it improves weighted-speedup and hmean-speedup by respectively 5.8% and 10.8% compared to NFQ.

7.3. Scaling to 8-Core and 16-Core Systems

Unfairness in the memory system will increase as the number of cores sharing it increases. We evaluate how existing and new DRAM scheduling techniques scale to 8 and 16 cores. Figure 10 shows the comparison of different schedulers for a non-memory-intensive work-

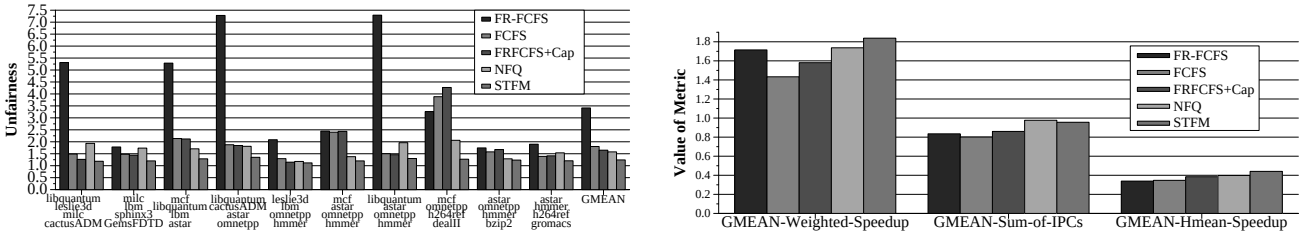

Figure 9. Unfairness (left) and throughput metrics (right) averaged (using geometric mean) over all 256 workloads run in the 4-core system

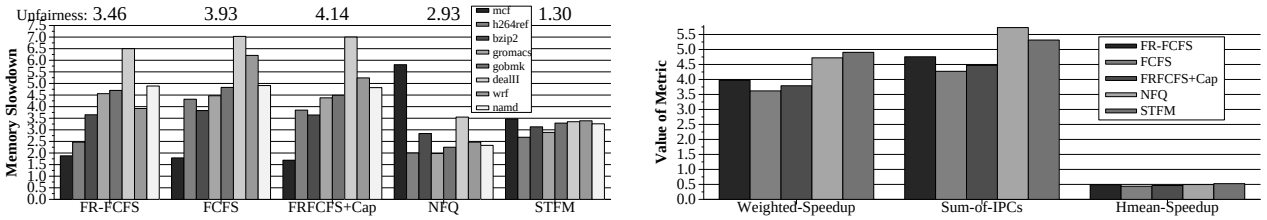

Figure 10. A non-memory-intensive 8-core workload: memory slowdowns and unfairness (left), throughput metrics (right)

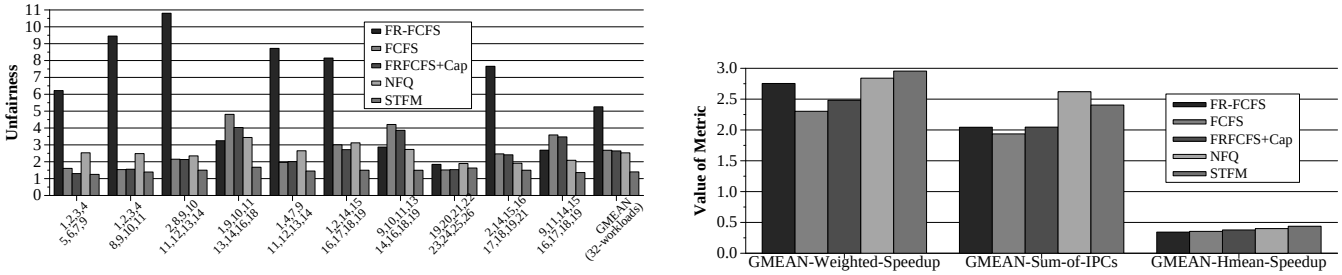

Figure 11. Unfairness (left) and throughput metrics (right) averaged (using geometric mean) over all 32 workloads run in the 8-core system

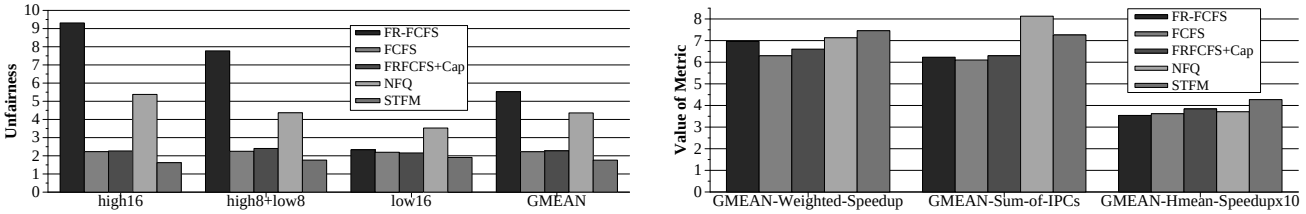

Figure 12. Unfairness (left) and throughput metrics (right) averaged (using geometric mean) over all workloads run in the 16-core system

load (1 intensive and 7 non-intensive benchmarks) running on an 8-core system. Even in this non-intensive workload, the unfairness of FR-FCFS is very high (3.46). NFQ reduces unfairness to 2.93. However, the memory-intensive thread, mcf, is very heavily penalized by NFQ because the non-intensive threads have bursty access patterns. Hence, NFQ's *idleness problem* becomes more severe as we move on to more cores. STFM reduces unfairness to 1.30 while also improving system throughput.

Figure 11(left) shows the unfairness of different scheduling techniques on 10 other sample workloads along with the average unfairness over 32 diverse combinations of benchmarks selected from different categories. FR-FCFS's average unfairness has increased significantly to 5.26 as compared to the 4-core system evaluated in Figure 9, supporting our claim that DRAM unfairness will become more significant in systems with more cores. Furthermore, the difference between the unfairness of STFM and other techniques widens as other techniques become increasingly ineffective at providing fairness: the average unfairness of FRFCFS+Cap and NFQ are respectively 2.64 and 2.53. In contrast, STFM's unfairness is only 1.40.

Figure 12(left) shows the unfairness of different scheduling techniques on 3 workloads run on the 16-core system. The workloads, from left to right consist of 1) the most memory intensive 16 benchmarks, 2) the most intensive 8 benchmarks run with the least intensive 8 benchmarks, and 3) the least intensive 16 benchmarks. NFQ becomes highly unfair in 16 cores due to two reasons. First, the idleness problem becomes much more severe because more threads with bursty access patterns can disrupt and unfairly get prioritized over a memory-intensive thread. Second, a thread with poor *bank access balance* becomes much more likely to be penalized because more threads with better access balance compete for the same banks that thread is heavily accessing. Therefore, both FCFS and FRFCFS+Cap, which do not suffer from these two problems, provide better fairness than NFQ.

In all 16-core workloads, STFM provides the best fairness, improving average unfairness from 2.23 (FCFS) to 1.75. STFM also provides the best system throughput, improving weighted-speedup and hmean-speedup respectively by 4.6% and 15% over NFQ. We conclude that *STFM scales better with the number of cores than the other DRAM scheduling techniques*.

7.4. Effect on Desktop Applications

We also evaluated STFM on Windows desktop workloads and present one case study.[18] Figure 13 considers a 4-core workload scenario with two background threads (XML parser searching a file database, and Matlab performing convolution on two images) and two foreground threads the user is focusing on (Internet Explorer and Instant Messenger). Table 4 shows the application characteristics.

The baseline FR-FCFS scheduler significantly penalizes the non-intensive threads because the background threads are very memory-

[18] We used iDNA [1] to trace the evaluated Windows applications.

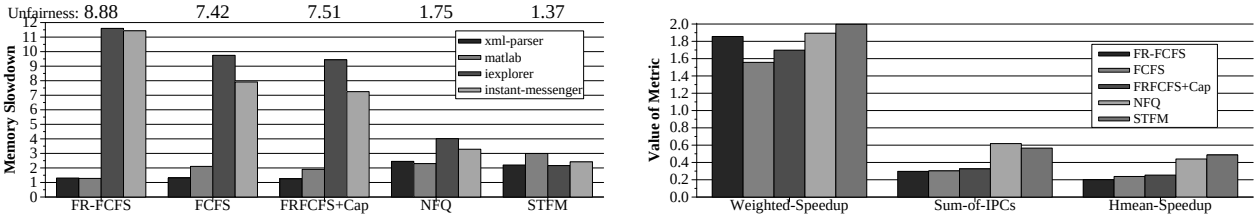

Figure 13. A mixed 4-core workload of desktop applications: memory slowdowns and unfairness (left), throughput metrics (right)

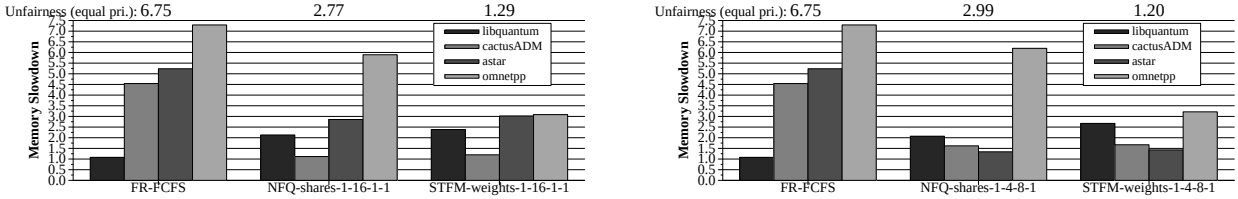

Figure 14. Evaluation of STFM and NFQ with different thread weights

Benchmark	Type	MCPI	L2 MPKI	RB hit rate	Category
matlab	INT	11.06	60.26	97.8%	3
instant-messenger	INT	1.56	7.72	22.8%	0
xml-parser	INT	8.56	53.46	95.8%	3
iexplorer	INT	0.55	3.55	41.4%	0

Table 4. Characteristics of the evaluated desktop applications

intensive and have very high row-buffer locality. NFQ reduces unfairness to 1.75, but still penalizes both iexplorer and messenger significantly because the accesses of these two applications are concentrated only on two and three banks, respectively. STFM improves unfairness to 1.37, while also improving system throughput: weighted-speedup by 5.4% and hmean-speedup by 10.7%.

7.5. Evaluation of Support for System Software

As explained in Section 3.3, STFM can be configured by the system software to assign different weights to different threads. We present one typical result to show the effectiveness of STFM's support for thread weights. Figure 14(left) shows the memory slowdowns of threads with different weights running on a 4-core system using FR-FCFS, NFQ, and STFM. Threads are assigned the following weights: libquantum (1), cactusADM (16), astar (1), omnetpp (1). FR-FCFS is thread-unaware and slows down the high-priority cactusADM by 4.5X. In contrast, NFQ takes into account thread weights by assigning each thread a share of the DRAM bandwidth that is proportional to the thread's weight [22].[19] NFQ succeeds in prioritizing the higher-priority cactusADM, but fails to treat equal-priority threads equally because, as shown in previous sections, equalizing DRAM throughput of threads does not necessarily correlate with equalizing performance slowdowns. STFM enforces thread weights more effectively, by prioritizing cactusADM such that its memory slowdown is only 1.2X while at the same time treating equal-priority threads more fairly.

Figure 14(right) shows memory slowdowns when threads are assigned the following weights: libquantum (1), cactusADM (4), astar (8), and omnetpp (1). Both NFQ and STFM manage to prioritize higher-priority threads over lower-priority ones. However, NFQ is again unable to ensure fairness across equal-priority threads, slowing down omnetpp (6.2X) more than libquantum (2.07X). In contrast, STFM preserves equal memory-slowdowns among equal-priority threads (unfairness: 1.20).

Figure 15 shows the effect of STFM's α parameter on unfairness and throughput. Recall that α is set by system software and it determines the maximum tolerable unfairness among threads. As α increases, STFM resembles FR-FCFS in terms of both unfairness and throughput. Hence, if the system software does not need the enforcement of fairness by the memory controller, it can set α to be very large (in this case, to a value of 20) to maximize the throughput obtained from the DRAM. Note that STFM provides better throughput when α is set to 1.1 rather than 1.0, without sacrificing much fairness. An α value of 1.0 causes the fairness-rule to be applied too often, thereby disabling the DRAM controller's ability to optimize for throughput most of the time. An α value of 1.05 results in similar, but better, behavior as STFM's slowdown estimates are not always accurate.

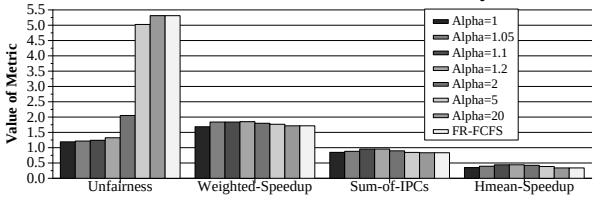

Figure 15. Effect of α on fairness and throughput

7.6. Sensitivity to DRAM Banks and Row-buffer Size

We have analyzed the sensitivity of STFM to the number of DRAM banks and row-buffer size. Table 5 shows the fairness and throughput comparison with FR-FCFS averaged over 32 diverse workloads on the 8-core system. Average unfairness of FR-FCFS decreases as the number of banks increases because the interference between threads becomes less prevalent due to reduced bank conflicts. On the other hand, average unfairness of FR-FCFS increases as row-buffer size increases because the probability of exploiting row-buffer locality (i.e. reordering younger column accesses over row accesses) increases. STFM significantly improves both unfairness and weighted-speedup for all row buffer sizes and DRAM bank counts. Furthermore, the fairness provided by STFM is independent of the number of DRAM banks and the size of the row-buffer. Even though these two parameters affect the slowdowns experienced by threads, STFM is able to balance those slowdowns without depending on these parameters.[20]

8. Related Work

Nesbit et al. [22] is the only work we are aware of that addresses fairness issues at the DRAM controller level. We have already provided extensive qualitative and quantitative comparisons to this work. Our previous work [20] describes how the unfairness in the DRAM subsystem can be exploited by malicious programs to perform denial of service against other programs sharing the memory system.

[19] This bandwidth partitioning is performed dynamically by adjusting a thread's virtual deadlines inverse-proportionally to its share [22]. In Figure 14(left) cactusADM's bandwidth share is 16/19, whereas the other three threads have equal bandwidth shares of 1/19.

[20] Also note that STFM's weighted-speedup improvement increases with the number of DRAM banks. This is because a larger number of banks allows more flexibility in scheduling DRAM commands and enables STFM to improve fairness while sacrificing less of DRAM throughput.

	DRAM banks						Row-buffer Size					
	4		8		16		1 KB		2 KB		4 KB	
	Unfairness	W. Speedup	Unfairness	W. Speedup	Unfairness	W. Speedup	Unfairness	W. Speedup	Unfairness	W. Speedup	Unfairness	W. Speedup
FR-FCFS	5.47	2.41	5.26	2.75	5.01	3.14	4.98	2.53	5.26	2.75	5.51	2.81
STFM	1.41	2.54	1.40	2.96	1.39	3.49	1.37	2.71	1.40	2.96	1.38	3.03
Improvement	3.88X	5.4%	3.78X	7.6%	3.60X	11.1%	3.64X	7.1%	3.78X	7.6%	3.99X	7.8%

Table 5. Sensitivity of fairness and throughput of STFM to DRAM banks and row-buffer size

DRAM Throughput Optimizations for Multithreaded Systems: Natarajan et al. [21] examine the effect of different memory controller policies on the performance of multiprocessor server systems. Zhu and Zhang [33] propose techniques to improve throughput in DRAM controllers used for simultaneous multithreading processors. Their techniques utilize SMT-specific information about threads, such as reorder buffer or issue queue occupancies, to improve throughput, but they do not consider fairness. Even though we evaluate STFM on CMP systems, it is trivially applicable to multithreaded systems as well.

DRAM Access Scheduling: Several works [25, 24, 10, 26] proposed and evaluated access scheduling algorithms to optimize throughput and latency in DRAM. McKee et al. [17] proposed access reordering to optimize bandwidth in streamed accesses. These techniques primarily optimize throughput for single-threaded systems; they do not try to provide fairness to accesses from different threads.

Real-Time Memory Schedulers: Several DRAM schedulers [12, 16] provide hard real-time guarantees to be used in real-time embedded systems. These schedulers provide QoS guarantees at the expense of DRAM throughput and flexibility. The throughput degradation of these schedulers is usually not acceptable in high-performance systems where such guarantees are not required.

Fairness Issues in Shared CMP Caches: Providing fair access to threads sharing CMP caches has recently received increasing attention. Several studies (e.g. [29, 11]) proposed techniques for fair cache partitioning. These are complementary to our work.

Fairness Issues in Multithreaded Systems: Although fairness issues have been studied in multithreaded systems especially at the processor level [28, 15, 7], the DRAM subsystem has received significantly less attention. Our STFM scheduler is applicable not only to CMP systems; it can be used in the general case of multiple threads sharing the DRAM subsystem.

9. Conclusion

We introduced, implemented, and evaluated the concept of *stall-time fair memory scheduling (STFM)*. STFM is a configurable substrate that provides fair DRAM access to different threads sharing the DRAM system. The key idea that makes STFM work is that equal-priority threads, when run together, should experience equal amounts of slowdown as compared to when they are run alone. We described the design and implementation of STFM. We also showed how STFM can be controlled by system software to control the unfairness in the system and to enforce thread priorities.

Our qualitative and quantitative evaluations demonstrate that STFM provides the best fairness and system throughput compared to one new and three previously-proposed state-of-the-art DRAM scheduling techniques, including FR-FCFS and Network Fair Queueing. We provide insights into why this is the case via detailed case studies of workloads. Previous techniques do not take into account the inherent memory performance of threads and therefore the fairness they provide is very much dependent on the workload mix, access patterns, and system configuration. In contrast, we demonstrate that STFM is able to consistently provide very high levels of fairness across a wide variety of workloads in 2-, 4-, 8-, and 16-core systems, while improving system throughput.

We conclude that STFM is a scalable, flexible, and high-throughput fairness substrate for the DRAM subsystem. An important area of future work is to research how STFM interacts with other fairness mechanisms at the shared caches or the operating system.

Acknowledgments

We would like to thank Burton Smith and Hyesoon Kim for valuable discussions and feedback. We also thank Brad Beckmann, John Davis, Jim Larus, Mark Oskin, Santhosh Srinath, and the anonymous reviewers for their comments on earlier drafts of this paper.

References

[1] S. Bhansali et al. Framework for instruction-level tracing and analysis of programs. In *VEE*, 2006.
[2] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA-31*, 2004.
[3] V. Cuppu, B. Jacob, B. T. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In *ISCA-26*, 1999.
[4] B. T. Davis. *Modern DRAM Architectures*. PhD thesis, University of Michigan, 2000.
[5] A. Fedorova, M. Seltzer, and M. D. Smith. Cache-fair thread scheduling for multi-core processors. Technical Report TR-17-06, Harvard University, Oct. 2006.
[6] J. M. Frailong, W. Jalby, and J. Lenfant. XOR-Schemes: A flexible data organization in parallel memories. In *ICPP*, 1985.
[7] R. Gabor, S. Weiss, and A. Mendelson. Fairness and throughput in switch on event multithreading. In *MICRO-39*, 2006.
[8] A. Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Idea Session '98*, Oct. 1998.
[9] S. Gochman et al. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2), May 2003.
[10] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *MICRO-37*, 2004.
[11] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. *PACT-13*, 2004.
[12] T.-C. Lin, K.-B. Lee, and C.-W. Jen. Quality-aware memory controller for multimedia platform soc. In *IEEE Workshop on Signal Processing Systems*, 2003.
[13] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
[14] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.
[15] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*, 2001.
[16] C. Macian, S. Dharmapurikar, and J. Lockwood. Beyond performance: secure and fair memory management for multiple systems on a chip. In *FPT*, 2003.
[17] S. A. McKee et al. Dynamic access ordering for streamed computations. *IEEE Transactions on Computers*, 49(11):1255–1271, Nov. 2000.
[18] Micron. *1Gb DDR2 SDRAM Component: MT47H128M8HQ-25*, May 2007. http://download.micron.com/pdf/datasheets/dram/ddr2/1GbDDR2.pdf.
[19] S.-W. Moon et al. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Transactions on Computers*, 49(11):1215–1227, 2000.
[20] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security Symposium*, 2007.
[21] C. Natarajan et al. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI*, 2004.
[22] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO-39*, 2006.
[23] A. K. Parekh. *A Generalized Processor Sharing Approach to Flow Control in Integrated Service Networks*. PhD thesis, MIT, 1992.
[24] S. Rixner. Memory controller optimizations for web servers. In *MICRO-37*, 2004.
[25] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000.
[26] J. Shao and B. T. Davis. A burst scheduling access reordering mechanism. In *HPCA-13*, 2007.
[27] T. Sherwood et al. Automatically characterizing large scale program behavior. In *ASPLOS-X*, 2002.
[28] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous mutlithreading processor. In *ASPLOS-IX*, 2000.
[29] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. *HPCA-8*, 2002.
[30] Sun Microsystems. *Sun Utility Computing*. `http://www.sun.com/service/sungrid/index.jsp`.
[31] D. Wang et al. DRAMsim: A memory system simulator. *Computer Architecture News*, 33(4):100–107, 2005.
[32] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *MICRO-33*, 2000.
[33] Z. Zhu and Z. Zhang. A performance comparison of DRAM memory system optimizations for SMT processors. In *HPCA-11*, 2005.