Enhancing the Performance and Fairness of Shared DRAM Systems with Parallelism-Aware Batch Scheduling

Onur Mutlu Thomas Moscibroda
Microsoft Research

Abstract

In a chip-multiprocessor (CMP) system, the DRAM system is shared among cores. In a shared DRAM system, requests from a thread can not only delay requests from other threads by causing bank/bus/row-buffer conflicts but they can also destroy other threads' DRAM-bank-level parallelism. Requests whose latencies would otherwise have been overlapped could effectively become serialized. As a result both fairness and system throughput degrade, and some threads can starve for long time periods.

This paper proposes a fundamentally new approach to designing a shared DRAM controller that provides quality of service to threads, while also improving system throughput. Our parallelism-aware batch scheduler (PAR-BS) design is based on two key ideas. First, PAR-BS processes DRAM requests in batches to provide fairness and to avoid starvation of requests. Second, to optimize system throughput, PAR-BS employs a parallelism-aware DRAM scheduling policy that aims to process requests from a thread in parallel in the DRAM banks, thereby reducing the memory-related stall-time experienced by the thread. PAR-BS seamlessly incorporates support for system-level thread priorities and can provide different service levels, including purely opportunistic service, to threads with different priorities.

We evaluate the design trade-offs involved in PAR-BS and compare it to four previously proposed DRAM scheduler designs on 4-, 8-, and 16-core systems. Our evaluations show that, averaged over 100 4-core workloads, PAR-BS improves fairness by 1.11X and system throughput by 8.3% compared to the best previous scheduling technique, Stall-Time Fair Memory (STFM) scheduling. Based on simple request prioritization rules, PAR-BS is also simpler to implement than STFM.

1. Introduction

The DRAM memory system is a major shared resource among multiple processing cores in a chip multiprocessor (CMP) system. When accessing this shared resource, different threads running on different cores can delay each other because accesses from one thread can cause additional DRAM bank conflicts, row-buffer conflicts, and data/address bus conflicts to accesses from another thread. In addition, as we show in this paper, inter-thread interference can destroy the bank-level access parallelism of individual threads. Memory requests whose latencies would otherwise have been largely overlapped effectively become serialized, which can significantly degrade a thread's performance. Moreover, some threads can be unfairly prioritized, while other –perhaps more important– threads can be starved for long time periods.

Such negative effects of uncontrolled inter-thread interference in the DRAM memory system are crucial impediments to building viable, scalable, and controllable CMP systems as they can result in 1) low system performance and significant productivity loss, 2) unpredictable program performance, which renders performance analysis, optimization, and isolation extremely difficult [27, 22, 40, 25], 3) significant discomfort to the end user who naturally expects threads with higher (equal) priorities to get higher (equal) shares of the system performance. As the number of cores on a chip increases, the pressure on the DRAM system will also increase and both the performance and fairness provided by the DRAM system will become critical determinants of the performance of future CMP platforms. Therefore, to enable viable, scalable, and predictable CMP systems, fair and high-performance memory access scheduling techniques that control and minimize inter-thread interference are necessary [27, 22, 25].

In this paper, we propose a new approach to providing fair and high-performance DRAM scheduling. Our scheduling algorithm, called *parallelism-aware batch scheduling (PAR-BS)*, is based on two new key ideas: *request batching* and *parallelism-aware DRAM scheduling*. First, PAR-BS operates by grouping a limited number of DRAM requests into

batches based on their arrival time and requesting threads. The requests from the oldest batch are prioritized and therefore guaranteed to be serviced before other requests. As such, PAR-BS is *fair and starvation-free*: it prevents any thread from being starved in the DRAM system due to interference from other, potentially aggressive threads. Second, within a batch of requests, PAR-BS is *parallelism-aware*: it strives to preserve bank-level access parallelism (i.e., the degree to which a thread's DRAM requests are serviced in parallel in different DRAM banks) of each thread in the presence of interference from other threads' DRAM requests.[1] It does so by trying to group requests from a thread and service them concurrently (as long as they access different banks) using heuristic-based prioritization rules. As such, our approach reduces the serialization of a thread's requests that would otherwise have been serviced in parallel had the thread been running alone in the memory system.

We show that the *request batching* component of PAR-BS is a general framework that provides fairness and starvation freedom in the presence of inter-thread interference. Within a batch of DRAM requests, any existing and future DRAM access scheduling algorithm (e.g., those proposed in [32, 31, 27, 25]) can be implemented. However, our results show that using our proposed *parallelism-aware scheduling* algorithm provides the best fairness as well as system throughput. We describe how PAR-BS operates within a batch and analyze the complex trade-offs involved in batching and parallelism-aware scheduling in terms of fairness, DRAM throughput, row-buffer locality exploitation, and individual threads' bank-level access parallelism. We also describe how the system software can control PAR-BS to enforce thread priorities and change the level of desired fairness in the DRAM system.

Our experiments compare PAR-BS qualitatively and quantitatively to four previously proposed DRAM scheduling techniques, including the recently-proposed QoS-aware Network Fair Queueing based [27] (NFQ) and Stall-Time Fair [25] (STFM) memory access schedulers, as well as Rixner's commonly used first-ready first-come-first-serve (FR-FCFS) scheduler [31]. None of these schedulers try to preserve individual threads' bank-level parallelism or strictly guarantee short-term starvation-freedom in the presence of inter-thread interference. Our results on a very wide variety of workloads and CMP configurations show that PAR-BS provides the best fairness and system throughput.

Contributions: We make the following contributions in this paper:

- We show that inter-thread interference can destroy bank-level parallelism of individual threads, thereby leading to significant degradation in system throughput. We introduce a novel parallelism-aware DRAM scheduling policy that maintains the bank-level parallelism of individual threads while also respecting row-buffer locality.
- We introduce the concept of *request batching* in shared DRAM schedulers as a general framework to provide fairness/QoS across threads and starvation freedom to DRAM requests. We show that request batching is orthogonal to and can be employed with existing DRAM access scheduling algorithms, but it is most beneficial when applied with parallelism-aware scheduling. We describe how the system software can control the flexible fairness substrate provided by request batching to enforce thread priorities and to control the unfairness in the DRAM system.
- We qualitatively and quantitatively compare our scheduler to four previously proposed fairness- or throughput-oriented schedulers and show that PAR-BS provides both the best fairness and the best system throughput. Our proposal is also simpler to implement than the best previously-proposed memory access scheduler, Stall-Time Fair

[1] In this paper, we refer to the bank-level parallelism of a thread as *intra-thread bank-level parallelism*. We use the terms bank-level parallelism and bank-parallelism interchangeably. A quantifiable definition of bank-parallelism is provided in Section 7.

Memory Scheduler [25], in that it does not require complex calculations, such as division.

2. Motivation

DRAM requests are very long latency operations that greatly impact the performance of modern processors. When a load instruction misses in the last-level on-chip cache and needs to access DRAM, the processor cannot commit that (and any subsequent) instruction because instructions are committed in program order to support precise exceptions [34]. The processor's instruction window becomes full a few cycles after a last-level cache miss [13, 24] and the processor stalls until the miss is serviced by DRAM. Current processors try to reduce the performance loss due to a DRAM access by servicing other DRAM accesses in parallel with it. Techniques like out-of-order execution [39], non-blocking caches [15], and runahead execution [5, 23] strive to overlap the latency of future DRAM accesses with the current access so that the processor does not need to stall (long) for future DRAM accesses. Instead, at an abstract level, the processor stalls once for all overlapped accesses rather than stalling once for each access in a serialized fashion [24]. The concept of generating and servicing multiple DRAM accesses in parallel is called *Memory Level Parallelism* (MLP) [9].

The effectiveness of the aforementioned latency tolerance techniques depends on whether or not the concurrent DRAM accesses are actually serviced in parallel by different DRAM banks (i.e., whether or not intra-thread bank-level parallelism is maintained). In a single-core system,[2] a thread has exclusive access to the DRAM banks, so its concurrent DRAM accesses are serviced in parallel as long as they are not to the same bank. This is illustrated in the simple, conceptual example in Figure 1.[3] Request1's (Req1) latency is hidden by the latency of Request0 (Req0), effectively exposing only a single bank access latency to the thread's processing core. Once Req0 is serviced, the core can commit Load 0 and thus enable the decode/execution of future instructions. When Load 1 becomes the oldest instruction in the window, its miss has already been serviced and therefore the processor can continue computation without stalling.

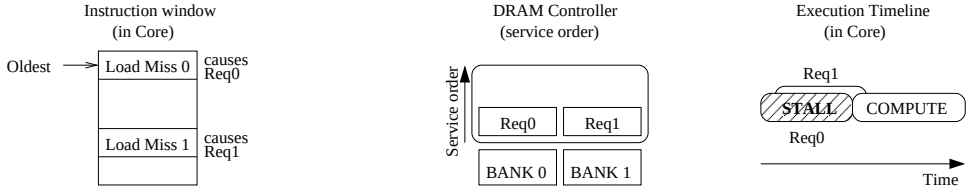


Figure 1. Example showing how latencies of two DRAM requests are overlapped in a single-core system

Unfortunately, if multiple threads are generating memory requests concurrently (e.g. in a CMP system), modern DRAM controllers schedule the outstanding requests in a way that completely ignores the inherent memory-level parallelism of threads. Instead, current DRAM controllers exclusively seek to maximize the DRAM data throughput, i.e., the number of DRAM requests serviced per second [32, 31]. As we show in this paper, blindly maximizing the DRAM data throughput does not minimize a thread's stall-time (which directly correlates with system throughput). Even though DRAM throughput may be maximized, some threads can be stalled overly long if the DRAM controller destroys their bank-level parallelism and serializes their memory accesses instead of servicing them in parallel.

The example in Figure 2 illustrates how parallelism-unawareness can result in suboptimal CMP system throughput and increased stall-times. We assume two cores, each running a single thread, Thread 0 (T0) and Thread 1 (T1). Each

[2]We assume, for simplicity and without loss of generality, that a core can execute one thread, and use the terms *thread* and *core* interchangeably. However, the ensuing discussion and our techniques are applicable to cores that can execute multiple threads as well.

[3]This and subsequent figures abstract away many details of the DRAM system, such as the DRAM bus and timing constraints. However, these are second order effects as bank access latency usually dominates the latency of DRAM requests [3, 4], especially with a wide DRAM data bus.

thread has two concurrent DRAM requests caused by consecutive independent load misses (Load 0 and Load 1), and the requests go to two different DRAM banks. The figure shows 1) (top) how a current DRAM scheduler may destroy intra-thread bank-parallelism, thereby increasing a thread's stall-time, and 2) (bottom) how a parallelism-aware scheduler can schedule the requests more efficiently.

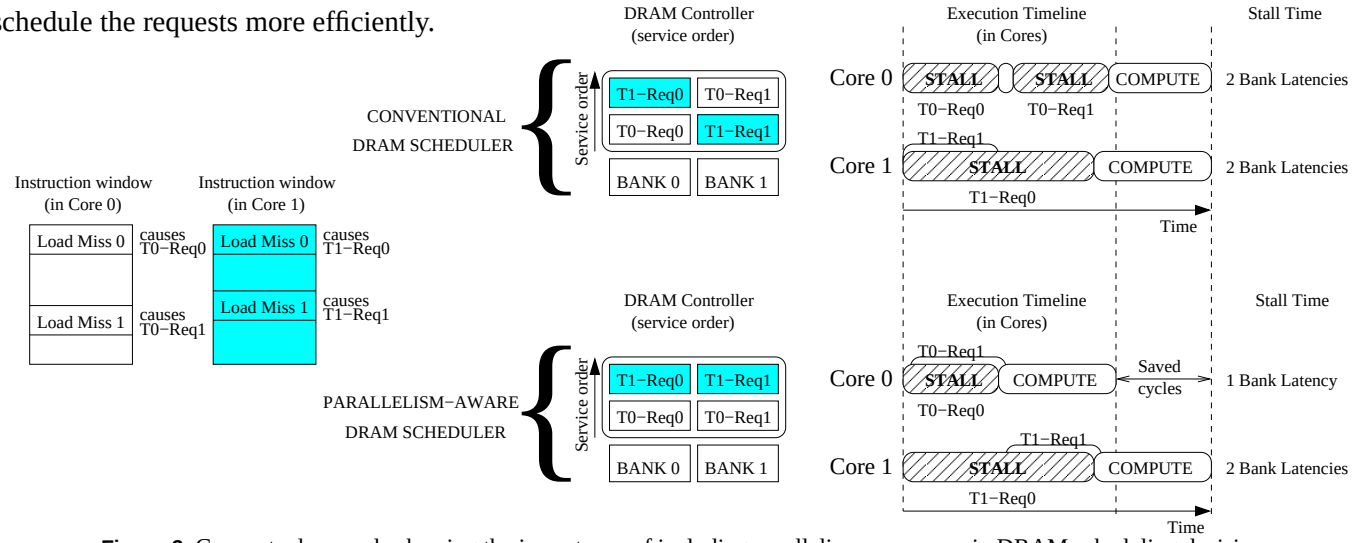


Figure 2. Conceptual example showing the importance of including parallelism-awareness in DRAM scheduling decisions

With a *conventional parallelism-unaware DRAM scheduler* (such as any previously proposed scheduler [32, 31, 27, 25]), the requests can be serviced in their arrival order shown in Figure 2(top). First, T0's request to Bank 0 is serviced in parallel with T1's request to Bank 1. Later, T1's request to Bank 0 is serviced in parallel with T0's request to Bank 1. This service order serializes each thread's concurrent requests and therefore exposes two bank access latencies to each core. As shown in the execution timeline (top right), instead of stalling once (i.e. for one bank access latency) for the two requests, both cores stall twice. Core 0 first stalls for Load 0, and shortly thereafter also for Load 1. Core 1 stalls for its Load 0 for two bank access latencies.

In contrast, a *parallelism-aware scheduler* services each thread's concurrent requests in parallel, resulting in the service order and execution timeline shown in Figure 2(bottom). The scheduler preserves bank-parallelism by first scheduling both requests from T0 in parallel, and then T1's requests. This enables Core 0 to execute faster (shown as "Saved cycles" in the figure) as it stalls for only one bank access latency. Core 1's stall time remains unchanged: although its second request (T1-Req1) is serviced later than with a conventional scheduler, T1-Req0 still hides T1-Req1's latency.

The crucial observation is that *parallelism-aware request scheduling improves overall system throughput because one core now executes much faster*: the average core stall time is 1.5 bank access latencies with the parallelism-aware scheduler (bottom) whereas it is 2 with the conventional scheduler (top).[4] While this example shows only two cores for simplicity, the destruction of intra-thread bank-parallelism becomes worse as more cores share the DRAM system.

Our goal: Our goal in this paper is to design a fair, QoS-aware memory scheduler that provides high system throughput. Based on the observation that inter-thread interference destroys the bank-level parallelism of the threads running concurrently on a CMP and therefore degrades system throughput, we incorporate parallelism-awareness into the design of our fair and high-performance memory access scheduler. To this end, we develop the key notions of *request batching*

[4]Notice that the system throughput improvement would be the same if the DRAM scheduler first serviced Core 1's requests in parallel, then Core 0's requests. In that case, Core 1 would only stall for a single bank access latency while Core 0's stall time would remain the same as with a conventional scheduler. Similarly, system throughput would also improve if T1-Req0 was to Bank 1 and T1-Req1 was to Bank 0.

and *parallelism-aware request prioritization*, which we describe in detail in Section 4.

3. Background on DRAM Memory Controllers

This section gives a brief description of how modern SDRAM systems and controllers operate. The DRAM system is presented at a level of abstraction that is sufficient to understand the terminology and key concepts of this paper. For a detailed description, we refer the reader to [32, 4, 25].

A modern SDRAM chip consists of multiple DRAM banks to allow multiple outstanding memory accesses to proceed in parallel if they require data from different banks. Each DRAM bank is a two-dimensional array, consisting of columns and rows. Rows typically store data in consecutive memory locations and are of 1-2KB in size. The data in a bank can be accessed only from the *row-buffer*, which can contain at most one row. A bank contains a single row-buffer. The amount of time it takes to service a DRAM request depends on the status of the row buffer and falls into three categories:

- **Row hit:** The request is to the row that is currently open in the row buffer. The DRAM controller needs to issue only a *read* or *write* command to the DRAM bank, resulting in a bank access latency of t_{CL} (See Table 2).
- **Row closed:** There is no open row in the row buffer. The DRAM controller needs to first issue an *activate* command to open the required row, then a *read/write* command, resulting in a bank access latency of $t_{RCD} + t_{CL}$.
- **Row conflict:** The request is to a row different from the one currently in the row buffer. The DRAM controller needs to first close the row by issuing a *precharge* command, then open the required row (*activate*), and then issue a *read/write* command. These accesses incur the highest bank access latency of $t_{RP} + t_{RCD} + t_{CL}$.

A DRAM controller consists of a *memory request buffer* that buffers the memory requests (and their data) while they are waiting to be serviced and a (possibly two-level) scheduler that selects the next request to be serviced [32, 27, 25]. When selecting the next request to be serviced, the scheduler considers the state of the DRAM banks and the DRAM buses as well as the state of the request. A DRAM command for a request can be scheduled only if its scheduling does not cause any resource (bank and address/data/command bus) conflicts and does not violate any DRAM timing constraints. Such a DRAM command is said to be *ready*.

Because of the large disparity in the latency incurred by a row-hit access and a row-conflict/closed access, state-of-the-art DRAM controllers employ scheduling techniques that prioritize row-hit requests over other requests, including younger ones. A modern memory controller employs the FR-FCFS (first-ready first-come-first-serve) scheduling policy [32, 31], which prioritizes *ready* DRAM commands from 1) row-hit requests over others and 2) row-hit status being equal, older requests over younger ones. Such a scheduling policy aims to minimize the average service latency of DRAM requests and thus maximize the data throughput obtained from the DRAM. For single-threaded systems, the FR-FCFS policy was shown to provide the best average performance [32, 31], significantly better than the simpler FCFS policy, which simply schedules all requests according to their arrival order, regardless of the row-buffer state.

When multiple threads share the DRAM system, the FR-FCFS scheduling policy tends to unfairly prioritize threads with high *row-buffer locality* (i.e. row-buffer hit rate) over those with relatively low *row-buffer locality* due to the row-hit-first prioritization rule. It also tends to unfairly prioritize *memory-intensive* threads over *non-intensive* ones due to the oldest-first prioritization rule.[5] As a result, even though FR-FCFS achieves high DRAM data throughput, it may starve

[5] A thread is more memory-intensive than another if it spends more cycles per instruction waiting for DRAM requests. See Section 7 for more.

requests/threads for long time periods, causing unfairness and relatively low overall system throughput [27, 22, 25].

Previous research [27, 22, 25] experimentally demonstrated the unfairness of FR-FCFS and proposed new scheduling policies that are fairer and that provide QoS to different threads. Nesbit et al. [27] applied Network Fair-Queueing (NFQ) techniques to DRAM controllers in order to divide the DRAM bandwidth among multiple threads sharing the DRAM system. Mutlu and Moscibroda [25] proposed a stall-time fair memory scheduler (STFM) that equalizes the slowdowns experienced by threads as compared to when each one is run alone. None of these previous scheduling policies take into account intra-thread bank-parallelism, which—as seen in Section 2—can significantly degrade system performance when requests of different threads interfere in the DRAM system.

4. Parallelism-Aware Batch Scheduling Algorithm

Overview: Our proposed DRAM scheduling algorithm is designed to provide 1) a configurable substrate for fairness and QoS and 2) high CMP system throughput by incorporating parallelism-awareness into scheduling decisions. To achieve these goals, *Parallelism-Aware Batch Scheduling* (PAR-BS) consists of two components. The first component is a *request batching (BS)*, or simply *batching*, component that groups a number of outstanding DRAM requests into a batch and ensures that all requests belonging to the current batch are serviced before the next batch is formed. Batching not only ensures fairness but also provides a convenient granularity (i.e., a batch) within which possibly thread-unfair but high-performance DRAM command scheduling optimizations can be performed. The second component of our proposal, *parallelism-aware within-batch scheduling (PAR)* aims to reduce the average stall time of threads within a batch (and hence increase CMP throughput) by trying to service each thread's requests in parallel in DRAM banks.

After describing the two components separately, we discuss advantages/disadvantages of our proposal compared to existing DRAM schedulers and present possible alternative design choices in Sections 4.3 and 4.4, respectively.

4.1. Request Batching

The idea of batching is to consecutively group outstanding requests in the memory request buffer into larger units called *batches*. The DRAM scheduler avoids request re-ordering across batches by prioritizing requests belonging to the current batch over other requests. Once all requests of a batch are serviced (i.e., when the batch is finished), a new batch is formed consisting of outstanding requests in the memory request buffer that were not included in the last batch. By thus grouping requests into larger units according to their arrival time, batching—in contrast to FR-FCFS and other existing schemes—prevents request starvation at a very fine granularity and enforces steady and fair progress across all threads. At the same time, the formation of batches allows for the flexibility to re-order requests within a batch in order to maximally exploit row-buffer locality and bank-parallelism without significantly disturbing thread-fairness.

The batching component (BS) of PAR-BS works as follows. Each request in the memory request buffer has an associated bit indicating whether the request belongs to the current batch. If the request belongs to the current batch, this bit is set, and we call the request *marked*. BS forms batches using the following rules:

Rule 1 PAR-BS Scheduler: Batch Formation

1: **Forming a new batch:** A new batch is formed when there are no marked requests left in the memory request buffer, i.e., when all requests from the previous batch have been completely serviced.
2: **Marking:** When forming a new batch, BS marks up to `Marking-Cap` outstanding requests per bank for each thread; these requests form the new batch.

`Marking-Cap` is a system parameter that limits how many requests issued by a thread for a certain bank can be part of a batch. For instance, if `Marking-Cap` is 5 and a thread has 7 outstanding requests for a bank, PAR-BS marks only the 5 oldest among them. If no `Marking-Cap` is set, all outstanding requests are marked when a new batch is formed.

PAR-BS always prioritizes marked requests (i.e., requests belonging to the current batch) over non-marked requests in a given bank. On the other hand, PAR-BS neither wastes bandwidth nor unnecessarily delays requests: if there are no marked requests to a given bank, outstanding non-marked requests are scheduled to that bank. To select among two marked or two non-marked requests, any existing or new DRAM scheduling algorithm (e.g., FR-FCFS) can be employed. In PAR-BS, this "within-batch" scheduling component is PAR, which we describe next.

4.2. Parallelism-Aware Within-Batch Scheduling (PAR)

Batching naturally provides a convenient granularity (i.e., the batch) within which PAR can optimize scheduling decisions to obtain high performance. There are two main objectives that this optimization should strive for. It should simultaneously maximize 1) *row-buffer locality* and 2) intra-thread *bank-parallelism* within a batch. The first objective is important because if a high row-hit rate is maintained within a batch, bank accesses incur smaller latencies on average, which increases the throughput of the DRAM system. The second objective is similarly important because scheduling multiple requests from a thread to different banks in parallel effectively reduces that thread's experienced stall-time. Unfortunately, it is generally difficult to simultaneously achieve these objectives—e.g. FR-FCFS sacrifices parallelism in lieu of row-buffer locality.[6]

Our scheduling algorithm uses the request prioritization rules shown in Rule 2 to exploit both row-buffer locality and bank parallelism. Within a batch, row-hit requests are prioritized. This increases row buffer locality and ensures that any rows that were left open by the previous batch's requests are made the best possible use of in the next batch. Second, requests from threads with higher *rank* are prioritized over those from threads with lower rank to increase bank-level parallelism, as explained in detail below. Finally, all else being equal, an older request is prioritized over a younger one.

Rule 2 PAR-BS Scheduler: Request Prioritization

1: **BS—Marked-requests-first:** Marked ready requests are prioritized over requests that are not marked.
2: **RH—Row-hit-first:** Row-hit requests are prioritized over row-conflict/closed requests.
3: **RANK—Higher-rank-first:** Requests from threads with higher-rank are prioritized over requests from lower-ranked threads.
4: **FCFS—Oldest-first:** Older requests are prioritized over younger requests.

Thread Ranking: PAR-BS uses a *rank-based thread prioritization* scheme within a batch to maximize the intra-thread bank-parallelism while maintaining row-buffer locality. When a new batch is formed, the DRAM scheduler computes a ranking among all threads that have requests in the batch. While the batch is processed, the computed ranking remains the same and requests from higher-ranked threads are prioritized over those from lower-ranked threads. The effect of ranking-based scheduling is that different threads are prioritized in the same order *across all banks* and thus, each thread's requests are more likely to be serviced in parallel by all banks.

How to Rank Threads Within a Batch: Although conceptually any ranking-based scheme enhances within-batch intra-thread bank-parallelism, the specific ranking procedure has a significant impact on CMP throughput and fairness.

[6] In fact, several combinatorial formalizations of this optimization problem can be shown to be NP-complete and hence no efficient algorithmic solutions are expected to exist.

A good ranking scheme must effectively differentiate between memory-intensive and non-intensive threads (and threads with high bank-parallelism). If a non-intensive thread with few requests is ranked lower than an intensive thread, its requests may be overly delayed within a batch. As explained in [25], a fair DRAM scheduler should equalize the *DRAM-related slowdown* of each thread compared to when the thread is running alone on the same memory system. As a non-intensive thread or a thread with high bank-parallelism inherently has a low DRAM-related stall-time when running alone, delaying its requests within a batch results in a much higher slowdown than it would for an intensive thread, whose DRAM-related stall-time is already high even when running alone. To avoid this unfairness (and loss of system throughput as explained below), our ranking scheme is based on the *shortest job first* principle [35]: it ranks the non-intensive threads higher than the intensive ones.

Besides fairness, the key rationale behind the *shortest job first* principle is that it tends to reduce the *average batch-completion time* of threads within a batch.[7] A thread's batch-completion time is the time between the beginning of a batch and the time the thread's last marked request from the batch is serviced. It directly corresponds to the thread's memory-related stall-time within a batch. By reducing the average batch-completion time, *shortest job first* scheduling improves overall system throughput as the threads stall less for DRAM requests, on average, thereby making faster progress in the execution of their instruction streams.

Concretely, PAR-BS uses the following *Max-Total ranking scheme*, to compute each thread's rank within a batch:

Rule 3 PAR-BS Scheduler: Thread Ranking

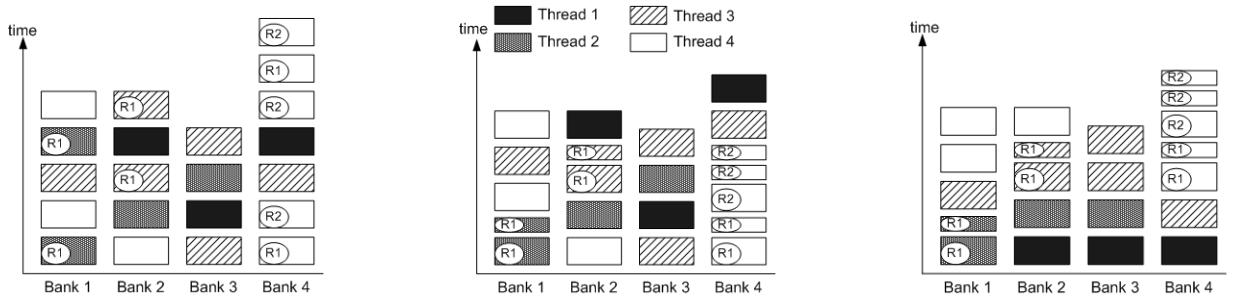
1: **Max rule:** For each thread, the scheduler finds the maximum number of marked requests to any given bank, called max-bank-load. A thread with a lower max-bank-load is ranked higher than a thread with a higher max-bank-load.
2: **Tie-breaker Total rule:** For each thread, the scheduler keeps track of the total number of marked requests, called total-load. If threads are ranked the same based on the Max rule, a thread with a lower total-load is ranked higher than a thread with a higher total-load.

The maximum number of outstanding requests to any bank correlates with the "shortness of the job," i.e., with the minimal memory latency that is required to serve all requests from a thread if they were processed completely in parallel. A highly-ranked thread has few marked requests going to the same bank and hence can be finished fast. By prioritizing requests from such high-ranked threads within a batch, PAR-BS ensures that non-intensive threads or threads with high bank-parallelism make fast progress and are not delayed unnecessarily long.

Example: Figure 3 shows an example that provides insight into why our proposed within-batch prioritization scheme preserves intra-thread bank-parallelism and improves system throughput. The figure abstracts away many details of DRAM scheduling[8] but provides a framework for understanding the parallelism and locality trade-offs. We assume a latency unit of 1 for row-conflict requests and 0.5 for row-hit requests. Figure 3(a) depicts the arrival order of requests in each bank, which is equivalent to their service order with an FCFS scheduler. FCFS neither exploits locality nor preserves intra-thread bank-parallelism and therefore results in the largest average completion time of the four threads (5 latency units). FR-FCFS maximizes row-buffer hit rates by reordering row-hit requests over others, but as shown in Figure 3(b), it does not preserve intra-thread bank-parallelism. For example, although Thread 1 has only three requests that are all intended for different banks, FR-FCFS services all three requests sequentially. Depending on the history of

[7] In the classic single-machine job-scheduling problem and many of its generalizations, shortest-job-first scheduling is optimal in that it minimizes the average job completion time [35].

[8] Such as DRAM data/address/command bus contention and complex interactions between timing constraints.



(a) Arrival order (and FCFS schedule) (b) FR-FCFS schedule (c) PAR-BS schedule

FCFS schedule batch-completion (stall) times					FR-FCFS schedule batch-completion (stall) times					PAR-BS schedule batch-completion (stall) times				
Thread 1	Thread 2	Thread 3	Thread 4	AVG	Thread 1	Thread 2	Thread 3	Thread 4	AVG	Thread 1	Thread 2	Thread 3	Thread 4	AVG
4	4	5	7	5	5.5	3	4.5	4.5	4.375	1	2	4	5.5	3.125

Figure 3. A simplified abstraction of scheduling within a batch containing requests from 4 threads. Rectangles represent marked requests from different threads; bottom-most requests are the oldest requests for the bank. Those requests that affect or result in row-hits are marked with the row number they access; if two requests to the same row are serviced consecutively, the second request is a row-hit with smaller access latency.

memory requests, the schedule shown in Figure 3(b) for FR-FCFS is also a possible execution scenario when using the QoS-aware NFQ [27] or STFM [25] schedulers since those schedulers are unaware of intra-thread bank-parallelism.

Figure 3(c) shows how PAR operates within a batch. Thread 1 has at most one request per bank (resulting in the lowest max-bank-load of 1) and is therefore ranked highest in this batch. Both Threads 2 and 3 have a max-bank-load of two, but since Thread 2 has fewer total number of requests, it is ranked above Thread 3. Thread 4 is ranked the lowest because it has 5 requests to Bank 4. As Thread 1 is ranked highest, its three requests are scheduled perfectly in parallel, before other requests. Similarly, Thread 2's requests are scheduled as much in parallel as possible. As a result, PAR maximizes the bank-parallelism of non-intensive threads and finishes their requests as quickly as possible, allowing the corresponding cores to make fast progress. Compared to FR-FCFS or FCFS, PAR significantly speeds up Threads 1, 2, and 3 while not substantially slowing down Thread 4. The average completion time is reduced to 3.125 latency units.

Notice that in addition to good bank-parallelism, our proposal achieves as good a row-buffer locality as FR-FCFS within a batch, because within a batch PAR-BS always prioritizes marked row-hit requests over row-conflict requests.[9]

4.3. Advantages, Disadvantages, Trade-offs

Request Batching component of our proposal has the following major advantages:

- **Fairness and Starvation Avoidance:** Batching guarantees the absence of short-term or long-term starvation: every thread can make progress in every batch, regardless of the memory access patterns of other threads.[10] The number of requests from a thread scheduled before requests of another thread is strictly bounded with the size of a batch. Apart from FCFS, no existing scheduler provides a similar notion of starvation avoidance. In FR-FCFS, a memory-intensive thread with excellent row-buffer locality can capture a bank for a very long time, if it can issue a large number of row-hit requests to the same bank in succession. Depending on the history of access patterns, short-term starvation is also possible in NFQ and STFM, especially due to the *idleness and bank access balance problems* [25] associated with NFQ and inaccurate slowdown estimates in STFM [25]. In PAR-BS, memory-intensive threads are unable to delay requests from non-intensive threads for a long time.

[9] However, this might not be the case across batches. PAR-BS can reduce locality at batch boundaries because marked requests are prioritized over row-hit requests. This locality reduction depends on how large `Marking-Cap` is. Section 8.3 evaluates the trade-offs of `Marking-Cap`.

[10] Starvation freedom of "batched (or grouped) scheduling" was proven formally within the context of disk scheduling [7].

- **Substrate for Exploiting Bank Parallelism:** Batching enables the use of highly efficient within-batch scheduling policies (such as PAR). Without batches (or any similar notion of groups of requests in time), devising a parallelism-aware scheduler is difficult as it is unclear within what context bank-parallelism should be optimized.
- **Flexibility and Simple Implementation:** While most beneficial in combination with PAR, the idea of batching can be used in combination with any existing or future DRAM command scheduling policy. Batching thus constitutes a simple and flexible framework that can be used to enhance the fairness of existing scheduling algorithms. We explore the performance and fairness of using FCFS and FR-FCFS policies within a batch in Section 8.3.3.

A possible disadvantage of our scheme is that it requires careful determination of `Marking-Cap`. If `Marking-Cap` is large, PAR-BS could suffer from similar unfairness problems as FR-FCFS, although not to the same extent. If a non-memory-intensive thread issues a request that just misses the formation of a new batch, the request has to wait until all requests from the current batch to the same bank are serviced, which slows down the non-intensive thread. On the other hand, a small `Marking-Cap` can slow down memory-intensive threads, since at most `Marking-Cap` requests per thread and per bank are included in a batch, the remaining ones being postponed to the next batch. There is a second important downside to having small batches: *The lower the `Marking-Cap`, the lower the row-buffer hit rate of threads with high inherent row-buffer locality.* Across a batch boundary, a marked row-conflict request is prioritized over an unmarked row-hit request. The smaller the batches (the smaller the `Marking-Cap`), the more frequently a stream of row-hit accesses can be broken in this way, which increases the access time of requests. Section 8.3.1 analyzes in detail the fairness and performance trade-offs of various `Marking-Cap` settings.

Parallelism-Aware Within Batch Scheduling simultaneously achieves a high degree of bank-parallelism and row-buffer locality, as described in the previous section. No other DRAM scheduling scheme we know of optimizes for intra-thread bank-parallelism. Consistent with the general machine scheduling theory [35], using the *Max-Total* ranking scheme to prioritize threads with fewer requests reduces the average stall time of threads within a batch. While this "shortest-job-first" principle may appear to unfairly penalize memory-intensive threads, our experimental evaluations in Section 8 show that this effect is not significant. There are two reasons: 1) the overlying batching scheme ensures a high degree of fairness, 2) delaying a memory intensive thread results in a relatively smaller slowdown since the inherent DRAM-related stall-time of an intensive thread is higher. Within a batch, a scheduler should therefore freely optimize for reduced stall-times by finishing threads with few and bank-parallel requests as quickly as possible.

4.4. Design Alternatives

We have experimented with a variety of novel, alternative batching and within-batch scheduling schemes. We briefly describe these schemes for completeness. Our evaluations in Section 8 show that averaged over a large and varied set of workload mixes, these alternative designs perform worse than our PAR-BS scheme.

The batching method in PAR-BS can be referred to as *full batching* because it requires that a batch of requests be completed in full before the next batch is started. There are alternative ways to perform batching.

Time-Based Static Batching: In this approach, outstanding requests are marked periodically using a static time interval, regardless of whether or not the previous batch is completed. The scheme is characterized by a system parameter `Batch-Duration` that describes at what time interval a new batch is formed. At the outset of a new batch, unmarked requests are marked subject to the `Marking-Cap`, while requests that are already marked from the previous batch

remain so. In comparison to PAR-BS, this batching approach does not provide strict starvation-avoidance guarantees and can lead to significant unfairness as we show in Section 8.3.2.

Empty-Slot (Eslot) Batching: If in PAR-BS, a request arrives in the DRAM system slightly after a new batch was formed, it may be delayed until the beginning of a new batch, causing a large stall time especially for a non-intensive thread. Empty-slot batching attempts to alleviate this problem by allowing requests to be *added* to the *current batch* if less than `Marking-Cap` requests from that thread for the specific bank were marked so far in the batch. In other words, if at the time a new batch is formed, a thread does not utilize its entire allotted share of marked requests (i.e. has "empty slots") within the batch, it is allowed to add late-coming requests to the batch until the `Marking-Cap` threshold is met.

Alternative Within-Batch Scheduling Policies: Within a batch, many different alternative request/command prioritization techniques can be employed. Aside from *Max-Total* ranking, we have also evaluated *Total-Max* (where the order of the *Max rule* and *Total rule* is reversed), *random*, and *round-robin* ranking schemes. Furthermore, we have evaluated using FCFS and FR-FCFS within a batch –without any ranking– to isolate the effect of parallelism-awareness in our proposal. Section 8.3.3 describes the trade-offs involved with alternative within-batch scheduling techniques.

5. Incorporating Thread Priorities and Software Support

We have so far described PAR-BS assuming that all threads have equal priority and, in terms of fairness, should experience equal DRAM-related slowdowns when run together. The system software (the operating system or virtual machine monitor), however, would likely want to assign priorities to threads to convey that some threads are more/less important than others. PAR-BS seamlessly incorporates the notion of *thread priorities* to provide support for the system software. The priority of each thread is conveyed to PAR-BS in terms of *priority-levels* $1, 2, 3, \ldots$, where level 1 indicates the most important thread (highest priority) and a larger number indicates a lower priority. Equal-priority threads should be slowed down equally [25], but the lower a thread's priority, the more tolerable its slowdown. We adjust PAR-BS in two ways to incorporate thread priorities.

- **Priority-Based Marking:** Requests from a thread with priority X are marked only every Xth batch. For example, requests from highest priority threads with level 1 are marked every batch, requests from threads with level 2 are marked every other batch, and so forth. The batching mechanism otherwise remains the same, i.e., a new batch is formed whenever there are no marked requests in the buffer.
- **Priority-Based Within-Batch Scheduling:** An additional rule is added to the within-batch request prioritization rules shown in Rule 2. Between rules `1.BS---Marked-requests-first` and `2.RH---Row-hit-first`, we add the new rule `PRIORITY---Higher-priority-threads-first`. That is, given the choice between two marked or two unmarked requests, PAR-BS prioritizes the request from the thread with higher priority. Between requests of equal-priority threads, other request prioritization rules remain the same.

The effect of these two changes to PAR-BS is that higher-priority threads are naturally scheduled faster: they are marked more frequently and thus take part in more batches, and they are prioritized over other requests within a batch.

Purely Opportunistic Service: In addition to the integer-based priority-levels, PAR-BS provides one particular priority-level, L, that indicates the lowest-priority threads. Requests from such threads are never marked and they are assigned the lowest priority among unmarked requests. Consequently, requests from threads at level L are scheduled

purely opportunistically—only scheduled if the memory system is free—to minimize their disturbance on other threads.

Finally, we provide the system software with the ability to set `Marking-Cap`, which serves as a lever to determine how much unfairness exists in the system (see Section 8.3.1).

6. Implementation and Hardware Cost

PAR-BS requires the implementation of batching (Rule 1) and the request prioritization policy described in Section 4.2 (Rules 2 and 3). Modern FR-FCFS based controllers already implement prioritization policies. Each DRAM request is assigned a priority and the DRAM command belonging to the highest priority request is scheduled among all *ready* commands. PAR-BS extends the priority of each DRAM request using two additional pieces of information: 1) whether or not the request is marked, and 2) the rank of the thread the request belongs to (using *Max-Total* ranking). To keep track of this additional information, the scheduler requires the additional state shown in Table 1. Assuming an 8-core CMP, 128-entry request buffer and 8 DRAM banks, the extra hardware state, including request priorities, required to implement PAR-BS (beyond FR-FCFS) is 1412 bits.

Register	Description and Purpose	Size (additional bits)
Per-request registers		
$Marked$	Whether or not the request is marked	1
$Priority$	The priority of the request including marked status, row-hit status, thread rank, and request ID	$\log_2 NumThreads$ (3) See Figure **??**
$Thread-ID$	ID of the thread that generated the request	$\log_2 NumThreads$ (3)
Per-thread per-bank registers to compute Max rule in Max-Total ranking		
$ReqsInBankPerThread$	Number of requests from this thread to this bank	$\log_2 RequestBufferSize$ (7)
Per-thread registers to compute Total rule in Max-Total ranking		
$ReqsPerThread$	Number of total requests from this thread in the request buffer	$\log_2 RequestBufferSize$ (7)
Individual registers		
$TotalMarkedRequests$	Number of marked requests in the request buffer (used to determine when to mark requests)	$\log_2 RequestBufferSize$ (7)
$Marking-Cap$	Stores the system-configurable `Marking-Cap` value	5

Table 1. Additional state required for a possible PAR-BS implementation

The counters $ReqsInBankPerThread$ and $ReqsPerThread$ are incremented/decremented when a new request enters/leaves the memory request buffer. When a marked request is fully serviced, the DRAM controller decrements $TotalMarkedRequests$. When $TotalMarkedRequests$ reaches zero, the controller starts a new batch by 1) marking the oldest `Marking-Cap` requests per bank from each thread, 2) computing the new *Max-Total* ranking of threads using the $ReqsInBankPerThread$ and $ReqsPerThread$ registers. Thus, the additional logic required by PAR-BS consists of logic that 1) marks requests (marking logic), 2) determines thread ranking (ranking logic), and 3) computes request priorities based on marked-status and thread rank (prioritization logic). Both marking and ranking logic are utilized only when a new batch is formed and implemented using priority encoders that take as input the relevant information in each case. Prioritization logic takes as input the marked status, row-hit status, thread rank, and request ID of a request to form a single priority value (see Figure **??**) for each request every DRAM cycle.

Notice that none of this logic is on the critical path of the processor because an on-chip DRAM controller runs at a higher frequency than DRAM and needs to make a scheduling decision only every DRAM cycle. Similar prioritization policies have been implemented in instruction schedulers, which are on the critical path. If needed, the marking/ranking logic can take multiple cycles since marking/ranking is done only when a new batch is formed.

PAR-BS is simpler to implement than the previous-best scheduler STFM, which requires significant logic, including dividers, to estimate thread slowdowns [25]. In contrast to STFM, PAR-BS is based only on simple prioritization rules that depend on request counts and therefore does not require complex arithmetic operations.

7. Experimental Methodology

We evaluate our proposal using a cycle-accurate x86 CMP simulator. The functional front-end of the simulator is based on Pin [17] and iDNA [1]. We model the memory system in detail, faithfully capturing bandwidth limitations, contention, and enforcing bank/port/channel/bus conflicts. Table 2 shows the major DRAM and processor parameters. We scale DRAM bandwidth with the number of cores.

Processor pipeline	4 GHz processor, 128-entry instruction window (64-entry issue queue, 64-entry store queue), 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, 32 MSHRs
DRAM controller (on-chip)	FR-FCFS; 128-entry request buffer, 64-entry write data buffer, reads prioritized over writes, XOR-based address-to-bank mapping [6, 41]
DRAM chip parameters	Micron DDR2-800 timing parameters (see [21]), 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 channel to DRAM
Round-trip L2 miss latency	For a 64-byte cache line, uncontended: row-buffer hit: 40ns (160 cycles), closed: 60ns (240 cycles), conflict: 80ns (320 cycles)
Cores and DRAM bandwidth	DRAM channels scaled with cores: 1, 2, 4 parallel lock-step channels for 4, 8, 16 cores (1 channel has 6.4 GB/s peak bandwidth)

Table 2. Baseline CMP and memory system configuration

We use the SPEC CPU2006 benchmarks and two Windows desktop applications (Matlab and an xml parsing application) for evaluation.[11] Each benchmark was compiled using gcc 4.1.2 with -O3 optimizations and run for 150 million instructions chosen from a representative execution phase [28].

We classify the benchmarks into eight categories based on their memory intensiveness (low or high), row-buffer locality (low or high), and bank-level parallelism (low or high). We define bank-level parallelism (BLP) as the average number of requests being serviced in the DRAM banks when there is at least one request being serviced in the DRAM banks. This definition follows the memory-level parallelism (MLP) definition of Chou et al. [2]. We characterize a thread based on the *average stall time per DRAM request (AST/req)* metric, which is computed by dividing the number of cycles in which the thread cannot commit instructions because the oldest instruction is a miss to DRAM by the total number of DRAM load requests generated by the thread.[12] Table 3 shows the category and memory system characteristics of the benchmarks when they run alone in one core of the baseline 4-core CMP. Note that benchmarks with high levels of BLP also have relatively low AST/req. In all figures, benchmarks are ordered based on their category in Table 3.

#	Benchmark	Type	MCPI	L2 MPKI	RB hit rate	BLP	AST/req	Category	#	Benchmark	Type	MCPI	L2 MPKI	RB hit rate	BLP	AST/req	Category
1	437.leslie3d	FP	7.30	51.52	62.8%	1.90	139	7 (111)	15	453.povray	FP	0.00	0.03	79.9%	1.75	123	3
2	450.soplex	FP	6.18	47.58	78.8%	1.81	125	7	16	464.h264ref	INT	0.48	2.65	76.5%	1.29	161	2 (010)
3	470.lbm	FP	3.57	43.59	61.1%	3.37	77	7	17	445.gobmk	INT	0.11	0.60	61.1%	1.46	162	2
4	482.sphinx3	FP	3.05	24.89	75.0%	1.89	117	7	18	447.dealII	FP	0.07	0.41	90.3%	1.21	133	2
5	matlab	DSK	15.4	78.36	93.7%	1.08	192	6 (110)	19	444.namd	FP	0.06	0.33	86.6%	1.27	160	2
6	462.libquantum	INT	9.10	50.00	98.4%	1.10	181	6	20	481.wrf	FP	0.05	0.28	83.6%	1.20	164	2
7	433.milc	FP	4.65	32.48	86.4%	1.51	139	6	21	454.calculix	FP	0.04	0.19	75.9%	1.30	157	2
8	xml-parser	DSK	2.92	18.23	95.3%	1.32	158	6	22	400.perlbench	INT	0.02	0.13	75.4%	1.69	128	2
9	429.mcf	INT	6.45	98.68	41.5%	4.75	63	5 (101)	23	471.omnetpp	INT	1.96	22.15	26.7%	3.78	86	1 (001)
10	459.GemsFDTD	FP	4.08	29.95	20.4%	2.40	126	5	24	401.bzip2	INT	0.49	3.56	52.0%	2.05	127	1
11	483.xalancbmk	INT	2.80	23.52	59.8%	2.27	113	5	25	473.astar	INT	1.82	9.25	50.2%	1.45	177	0 (000)
12	436.cactusADM	FP	2.78	11.68	6.75%	1.60	219	4 (100)	26	456.hmmer	INT	1.50	5.67	33.8%	1.26	231	0
13	403.gcc	INT	0.05	0.37	63.9%	1.87	127	3 (011)	27	435.gromacs	FP	0.18	0.68	58.2%	1.04	220	0
14	465.tonto	FP	0.02	0.13	70.7%	1.92	108	3	28	458.sjeng	INT	0.10	0.41	16.8%	1.53	192	0

Table 3. Benchmark characteristics. MCPI: Memory Cycles Per Instruction (cycles spent waiting for memory divided by number of instructions), L2 MPKI: L2 Misses per 1000 Instructions, RB Hit Rate: Row-buffer hit rate, BLP: bank-level parallelism, AST/req: Average stall-time per DRAM request, **Categories** are determined based on MCPI (1:High, 0:Low), RB hit rate (1:High, 0:Low), and BLP (1:High, 0:Low)

We evaluate combinations of multiprogrammed workloads running on 4, 8, and 16-core CMPs. For 4-core simulations, we evaluated 100 different combinations, each of which was formed by pseudo-randomly selecting a benchmark from each category such that different category combinations are evaluated. For 8-core simulations, we evaluated 16 dif-

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

[12] AST/req is similar to the average cost of an L2 cache miss, described by Qureshi et al. [29], except AST/req is based on processor stall time rather than L2 miss latency.

ferent combinations; and for 16-core, 12 different combinations. Space limitations prevent us from listing all evaluated combinations, but we try to show as many results with representative individual combinations as possible in Section 8.[13]

7.1. Evaluation Metrics

We measure fairness using the *unfairness index* proposed in [25, 8].[14] This is the ratio between the maximum memory-related slowdown and the minimum memory-related slowdown among all threads sharing the DRAM system. The memory related slowdown of a thread *i* is the memory stall time per instruction it experiences when running together with other threads divided by the memory stall time per instruction it experiences when running alone in the same memory system: $$MemorySlowdown_i = \frac{MCPI_i^{shared}}{MCPI_i^{alone}} \quad , \quad UnfairnessIndex = \frac{\max_i MemorySlowdown_i}{\min_j MemorySlowdown_j}$$

We measure system throughput using ***Weighted-Speedup*** [36] and ***Hmean-Speedup*** [18], which balances fairness and throughput [18]: $$Weighted\ Speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}} \quad , \quad Hmean\ Speedup = NumThreads / \sum_i \frac{1}{IPC_i^{shared}/IPC_i^{alone}}$$

7.2. Evaluated Schemes: Parameters and Configuration

Our baseline controller uses the FR-FCFS scheduling policy. All evaluated schedulers prioritize DRAM read requests over DRAM write requests because read requests can directly block forward progress in processing cores and are therefore more performance critical. Unless otherwise stated, we use PAR-BS with a Marking-Cap of 5 in our experiments. When comparing PAR-BS to other schedulers, we use the following parameters. **STFM:** We set $\alpha = 1.10$ and $IntervalLength = 2^{24}$ as proposed by Mutlu and Moscibroda [25]. **NFQ:** We use Nesbit et al.'s best scheme (FQ-VFTF) [27], including its priority inversion prevention optimization with a threshold of t_{RAS} [27].

8. Experimental Results

8.1. Results on 4-core Systems

We first analyze the fairness and throughput of PAR-BS in comparison to previously proposed DRAM scheduling techniques using three case studies on 4-core systems that highlight the typical behavior of different scheduling algorithms. Aggregate results over 100 workloads are provided in Section 8.1.4.

8.1.1. Case Study I: Memory-intensive workload This workload includes four memory-intensive benchmarks, one with very high bank-level parallelism (mcf). Figure 5(left) shows the memory slowdown of each benchmark with different memory schedulers. Figure 5(right) compares the five different schedulers in terms of system throughput.

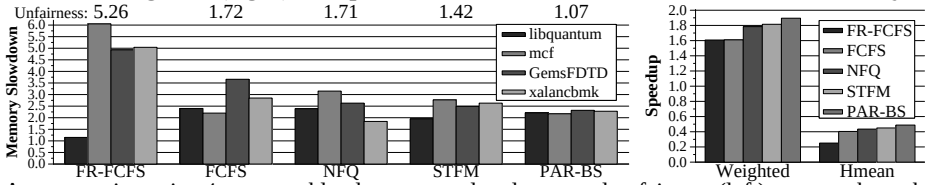


Figure 5. A memory intensive 4-core workload: memory slowdowns and unfairness (left), system throughput (right)

- FR-FCFS and FCFS: The commonly-used FR-FCFS scheduling policy is very unfair, slowing down the three less-intensive benchmarks significantly more than libquantum because of libquantum's very high row-buffer hit rate (98.4%) and memory intensiveness. Such unfairness results in the lowest system throughput as cores running the three less-intensive programs make very slow progress. FCFS improves fairness over FR-FCFS because it prevents libquantum's row-buffer hit requests from being continuously prioritized over other threads' requests. Nonetheless,

[13] Should the paper get accepted, we will post the evaluated combinations on a website that will be referenced in the paper.

[14] Gabor et al.'s fairness metric [8] is essentially the inverse of Mutlu and Moscibroda's unfairness index [25].

FCFS still unfairly prioritizes memory-intensive libquantum and mcf as their requests are more likely to be older than other threads' requests. Since the fairness and throughput characteristics of both FR-FCFS and FCFS were analyzed in detail in previous research [25], we concentrate our analysis primarily on the other scheduling algorithms.

- NFQ slightly improves fairness over FCFS, although it overly slows down mcf (by 3.15X). Mcf has very high bank-parallelism when run alone. NFQ's scheduling policy is to balance the requests from different threads in each bank, without any coordination among banks. As the other threads have bursty access patterns in some banks, NFQ prioritizes their requests over mcf's requests in those banks during bursts (this is due to the *idleness problem* inherent in NFQ's design, as described in [25, 30]). Therefore, NFQ destroys mcf's bank-parallelism: in some banks mcf's requests are unhindered by requests from other threads, while in other banks, requests from the bursty threads are prioritized. Mcf's requests in these banks are delayed, although they could have been serviced in parallel with its other requests. We found that mcf's BLP of 4.75 when run alone reduces to only 2.05 with NFQ and its average stall-time per DRAM access (AST/req) increases from 64 to 193 processor cycles.
- STFM results in better fairness and throughput than all previous policies. However, it also penalizes (slows down) mcf significantly, by 2.77X. This is due to two reasons. First, STFM tries to provide fairness by estimating the memory-slowdown of each thread and prioritizing requests from the threads that are slowed down the most. STFM penalizes mcf because its heuristics to estimate mcf's inherent bank-parallelism are not always accurate [25] and hence, it underestimates mcf's slowdown. Second, like NFQ, STFM is not parallelism-aware: it does not try to service requests from a thread in parallel. Instead, it prioritizes requests from threads that it estimates to have incurred the highest memory-slowdowns—in this case, libquantum and GemsFDTD. These threads' requests often take precedence over mcf's requests in the banks they access, increasing mcf's AST/req from 64 to 174 cycles.
- PAR-BS provides both the best fairness and system throughput. It reduces unfairness from 1.42 (STFM) to 1.07, and improves weighted-speedup by 4.4% and hmean-speedup by 8.4% over STFM. The *Request batching* component of PAR-BS fairly distributes memory-slowdowns by effectively containing libquantum's impact on other threads. We found that request batching is more effective and robust in providing fairness than both NFQ's and STFM's techniques because it is not vulnerable to 1) the *idleness* and *bank access balance* problems of the NFQ approach [25], 2) incorrect estimation of thread slowdowns in the STFM approach. *Parallelism-aware scheduling within a batch allows* PAR-BS to better exploit mcf's bank-parallelism, keeping its AST/req at 146 cycles, lower than NFQ and STFM. Consequently, PAR-BS slows down mcf (by 2.17X) less than NFQ (3.15X) and STFM (2.77X).

8.1.2. Case Study II: Non-intensive workload Figure 6 shows unfairness and throughput on a workload including three non-intensive benchmarks and a single intensive one. Only one application (omnetpp) has high bank-parallelism (3.78), which results in an average stall-time per DRAM access of 86 cycles when omnetpp is run alone.

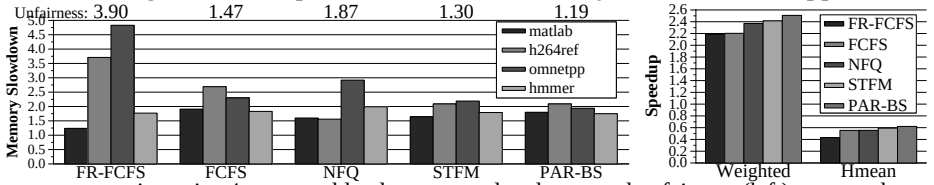


Figure 6. A non-memory-intensive 4-core workload: memory slowdowns and unfairness (left), system throughput (right)

PAR-BS is the only scheduler that does not significantly penalize the thread with high bank-parallelism (omnetpp). NFQ and STFM reduce unfairness compared to FR-FCFS because they successfully mitigate the problems caused by

FR-FCFS' rigid row-hit-first policy. However, neither NFQ nor STFM can recover omnetpp's loss in bank-parallelism and both slow down this thread the most. In fact, NFQ is even more unfair than FCFS because its earliest-virtual-deadline-first scheme prioritizes h264ref's (and to a lesser degree also hmmer's) bursty requests over omnetpp's requests in the banks they concurrently access [25]. This causes omnetpp's accesses that would otherwise proceed in parallel to get out-of-sync and become serialized, which degrades omnetpp's performance. The processor stalls for the bank access latency of each access rather than amortizing this latency by overlapping the latencies of multiple outstanding accesses. The result is an AST/req of 256 cycles for omnetpp. While STFM reduces this measure to 182 cycles, it still overly slows down omnetpp as it fails to optimize omnetpp's bank-parallelism and underestimates this thread's slowdown. In contrast, parallelism-aware PAR-BS reduces omnetpp's AST/access down to 150 cycles.

PAR-BS outperforms all existing schemes, achieving the best fairness while also improving weighted-speedup and hmean-speedup by 3.1% and 5.2% over STFM, respectively. In contrast to the other schemes, it is the least memory-intensive thread (h264ref) that is slowed down the most by PAR-BS, but this thread's slowdown is nonetheless smaller than with the other schedulers. Some of h264ref's less-frequent requests are likely to miss the formation of a batch, in which case they are not serviced until the batch completes. However, this does not result in a large slowdown because 1) batches are quick to process due to the small `Marking-Cap` of 5; we found that the average batch is completed in 1269 cycles, 2) even if h264ref's requests are not marked, they are still serviced if there is no marked request for the required bank, 3) because h264ref's requests are infrequent, they are prioritized within a batch due to our *Max-Total* thread ranking scheme; thus even if a request misses a batch it will be serviced first in the next batch.

8.1.3. Case Study III: Memory-intensive benchmark with high bank-parallelism running with copies of itself Our last case study is intended to explicitly demonstrate the parallelism behavior of the PAR-BS scheduler. For this, we minimize the variance among threads and run four identical copies of lbm together on a CMP. As expected, all schedulers are perfectly fair in this case (Figure 7(left)), but they differ significantly in their memory-slowdown and hence system throughput. FCFS drastically slows down each copy of lbm compared to FR-FCFS because it does not exploit row-buffer locality. NFQ's performance is even worse because it not only limits the row-buffer locality that can be exploited by the memory controller (using the priority-inversion optimization in [27]) but also frequently interleaves requests from different copies of lbm to a bank to keep the virtual deadline of each lbm copy in balance. This destroys the row-buffer hit rate of each lbm copy, reducing it from 61% to only 31%, and therefore reduces system throughput by 29.7%. STFM provides the same throughput as FR-FCFS because it never switches to a fairness-oriented scheduling policy as it correctly estimates the unfairness in the system to be 1.

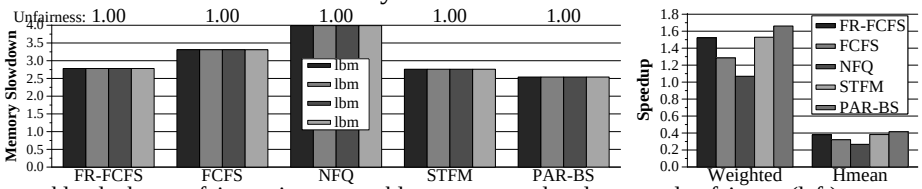


Figure 7. A 4-core workload where unfairness is not a problem: memory slowdowns and unfairness (left), system throughput (right)

PAR-BS achieves the best system throughput by servicing each lbm's concurrent requests in parallel, reducing the average stall-time a DRAM access inflicts upon a thread (from 222 (FR-FCFS and STFM) and 322 (NFQ) to 199 cycles). Therefore, PAR-BS improves both weighted- and hmean-speedup by 8.6%. Hence, making the DRAM scheduler parallelism-aware improves system throughput even in uniform application mixes where unfairness is not a problem.

8.1.4. 4-Core Experiments: Average Results Figure 8(left) compares the unfairness of the five schedulers across 10 other diverse workloads as well as averaged over all the 100 examined workloads. Figure 8(right) shows the average system throughput across 100 workloads. PAR-BS provides both the best fairness and the best throughput. Unfairness is reduced from 1.36 (STFM) to 1.22. At the same time, system throughput is improved by 4.4% (weighted-speedup) and by 8.3% (hmean-speedup) compared to the best previously-proposed scheduling scheme (STFM).

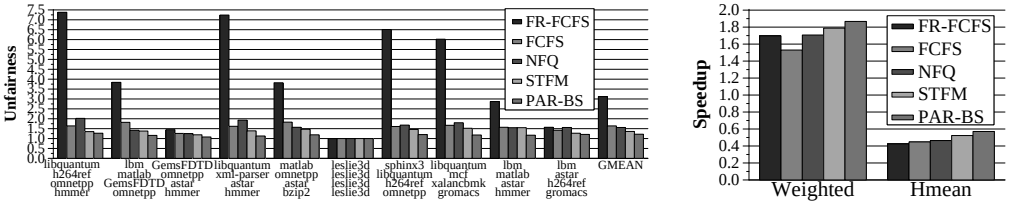


Figure 8. Unfairness (left) and system throughput (right) averaged (using geometric mean) over all 100 workloads run in the 4-core system

8.2. PAR-BS on 8-Core and 16-Core Systems

The DRAM system will become a bigger QoS and performance bottleneck as the number of cores sharing it increases. We briefly examine the scalability of PAR-BS on 8-core and 16-core systems. Figure 9 shows an 8-core workload consisting of 3 memory-intensive and 5 non-intensive applications. Mcf is the only program with very high inherent bank-parallelism. All previous schedulers consistently slow down mcf (by at least 3.5X) because they fail to control the serialization of mcf's concurrent DRAM accesses due to interference from the other seven applications.[15] On the other hand, PAR-BS increases mcf's bank-level parallelism, reducing its slowdown to 2.8X (and its AST/access from 330 (NFQ) and 221 (STFM) to only 173 cycles). As a result, PAR-BS provides both the best fairness and system throughput.

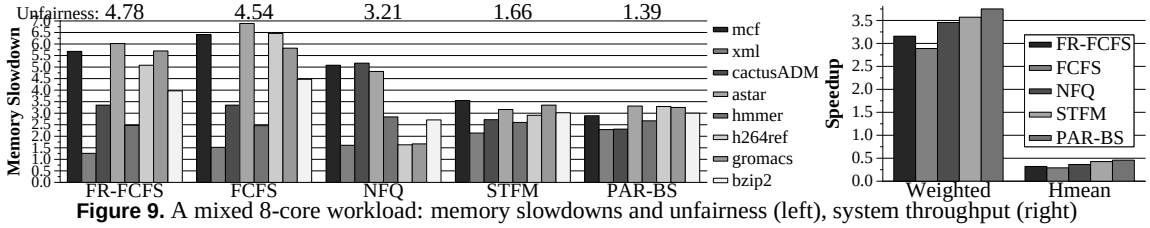


Figure 9. A mixed 8-core workload: memory slowdowns and unfairness (left), system throughput (right)

Figure 10 provides unfairness and throughput results on the 16-core system for five sample workloads as well as averaged over all 12 workloads. PAR-BS reduces unfairness from 1.81 (STFM) to 1.63, while improving weighted-speedup by 3.2% and hmean-speedup by 5.1% compared to STFM.

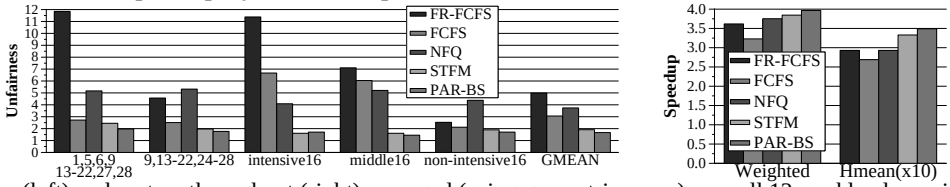


Figure 10. Unfairness (left) and system throughput (right) averaged (using geometric mean) over all 12 workloads run in the 16-core system

Summary: Table 4 summarizes our evaluation by comparing the geometric-mean of unfairness and system-throughput of PAR-BS to the previous schemes. PAR-BS provides the lowest average stall time per request, which indicates that it effectively reduces the average cost of a DRAM request on performance. Also, PAR-BS provides significantly lower worst-case request latency than other QoS-aware techniques. We found that both NFQ and STFM can delay requests from particular threads for a very long time in order to enforce fairness.[16] In contrast, the batching component of PAR-BS

[15] The likelihood that mcf's concurrent requests are serialized increases when 7 other threads are running together with it instead of 3.

[16] For example, STFM delays requests from threads that are estimated to be slowed down much less than others. Similarly, NFQ delays requests of a thread to a bank, if the thread had used that bank very intensively for a long time and accumulated a large virtual deadline.

achieves fairness while bounding the amount of time a thread's requests can be delayed. PAR-BS consistently provides better fairness and throughput than the best previous technique (STFM) for all examined systems. *We conclude that PAR-BS is very effective in providing the best fairness and highest system performance in 4-, 8-, and 16-core systems.*

	4-core system					8-core system					16-core system				
	Unf.	Weighted	Hmean-sp	AST/req	WC lat.	Unf.	Weighted-sp	Hmean-sp	AST/req	WC lat.	Unf.	Weighted-sp	Hmean-sp	AST/req	WC lat.
FR-FCFS	3.12	1.70	0.43	374	18481	4.10	1.99	0.29	605	34655	4.99	3.62	2.93	968	35117
FCFS	1.64	1.53	0.45	364	13728	2.23	1.77	0.28	633	20114	3.06	3.23	2.69	964	36549
NFQ	1.56	1.73	0.47	346	19801	2.45	2.04	0.31	525	59117	3.74	3.75	2.93	774	88732
STFM	1.36	1.79	0.52	301	20305	1.41	2.11	0.34	484	57764	1.81	3.85	3.33	712	86577
PAR-BS	1.22	1.87	0.57	281	13866	1.31	2.20	0.37	457	25614	1.63	3.97	3.50	676	41115
Δ vs. STFM	**1.11X**	4.4%	**8.3%**	7.1%	1.46X	**1.08X**	4.3%	**6.1%**	5.9%	2.26X	**1.11X**	3.2%	**5.1%**	5.3%	2.11X

Table 4. STFM vs. others: unfairness (Unf.), throughput (weighted/hmean-speedup), AST/req, and worst-case request latency (WC lat.) over all workloads

8.3. Analysis

8.3.1. Effect of Marking-Cap `Marking-Cap` determines the duration of a batch by changing the number of requests that are marked when a new batch is formed. Varying this parameter affects PAR-BS's fairness and throughput properties because it changes 1) the amount of row-buffer locality exploited, 2) the amount of delay unmarked requests experience, and 3) the degree of bank-level parallelism that can be exploited.

Figure 11(left) shows the effect of varying `Marking-Cap` from 1 to 20 and not using `Marking-Cap` at all (no-c) on unfairness and throughput averaged over the 100 workloads on the 4-core system. When `Marking-Cap` is smallest, system throughput is at its lowest because the resulting batches are too small. For example, with a cap of 1, a thread can have at most 1 request per bank in a batch. Such a small batch size significantly reduces our scheduler's ability to 1) exploit row-buffer locality and 2) find concurrent accesses from threads with high bank-parallelism. If, in a bank, Thread A has 5 outstanding requests to one row, and Thread B has 5 requests to another row, a cap of 1 results in the interleaving of Thread A and B's requests because only 1 request to the bank can finish from each thread in a batch. This interleaving results in a row-conflict for each access and therefore significantly increases the latency experienced by each thread. In contrast, with a `Marking-Cap` of 5, PAR-BS would service A's 5 requests first and B's 5 requests next with all accesses except for the first from each thread being row-hits. A small cap also results in poor fairness because it penalizes threads with high row-buffer locality (e.g. libquantum and matlab in Figure 11(middle) and (left)).

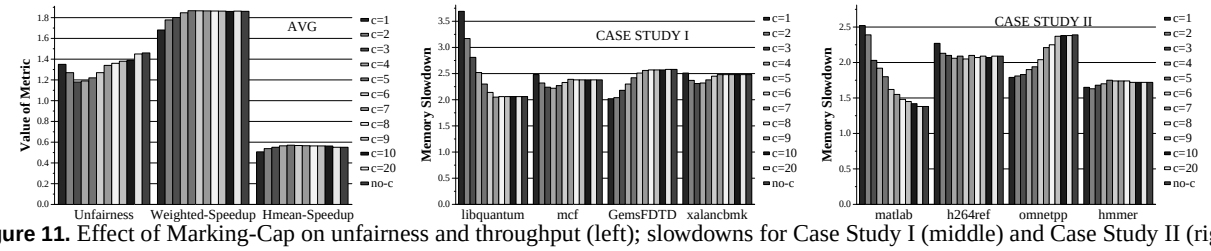


Figure 11. Effect of Marking-Cap on unfairness and throughput (left); slowdowns for Case Study I (middle) and Case Study II (right)

As `Marking-Cap` increases, unfairness decreases and system throughput increases, until a certain point beyond which unfairness increases due to two reasons. First, a large cap allows memory-intensive threads to insert more requests into a batch and thus delays non-intensive threads that "miss" the formation of a batch. As such, a large cap penalizes less memory-intensive threads as shown in memory slowdowns for GemsFDTD and xalancbmk in Figure 11(middle) and for omnetpp and hmmer in Figure 11(right). Second, because PAR-BS prioritizes threads with high row buffer locality within a batch, a large cap exacerbates the delay of requests from threads with low row-buffer locality within a batch.

According to Figure 11(left) a `Marking-Cap` of 5 provides the best average system throughput (both weighted-speedup and hmean-speedup) while providing very good fairness. Therefore, we use a `Marking-Cap` of 5 in our experiments. Note that it is possible to improve our mechanism by making the `Marking-Cap` adaptive.

8.3.2. Effect of Batching Choice Figure 12(left) compares the unfairness and throughput of *static batching* with various choices for `BatchDuration` (varied from 400 to 25600 cycles), *eslot batching*, and *full-batching* as used in PAR-BS, which were described in Section 4.4. Figure 12(middle) and (right) show the effect of the batching choice on the threads' memory-slowdowns in two case studies. On average, full batching provides the best fairness and throughput.

Static batching is unfair if `BatchDuration` is too small (e.g. 400 or 800 cycles). Because most requests in the request buffer become marked with a small `BatchDuration`, the scheme prioritizes memory-intensive threads with high row-buffer hit rates. Therefore, a small `BatchDuration` effectively eliminates request batching and degenerates to a row-hit-first, rank-first, oldest-first prioritization policy, which (similar to FR-FCFS) penalizes less-intensive threads with low row-buffer locality, as shown in Figure 12(middle) and (right)). Conversely, if `BatchDuration` is too large, most requests in the buffer are unmarked. This also effectively eliminates request batching and behaves similarly to FR-FCFS. The sweet-spot in static batching is with a `BatchDuration` of 3200 cycles but this does not provide as good performance or fairness as full batching since it is rigid/unadaptive and prone to starvation.

Eslot batching reduces the probability of penalizing non-intensive threads. Unfortunately, as shown in Figure 12(middle) and (right), it penalizes memory-intensive threads too much by allowing requests from less intensive ones into a current batch, which reduces the row-buffer hit rate of intensive threads. While this can result in system throughput improvement in some cases (e.g. for Case Study II in Figure 12(right) – not shown in the figure), full batching provides better average fairness and system throughput. We conclude that full batching is the most effective batching policy for PAR-BS.

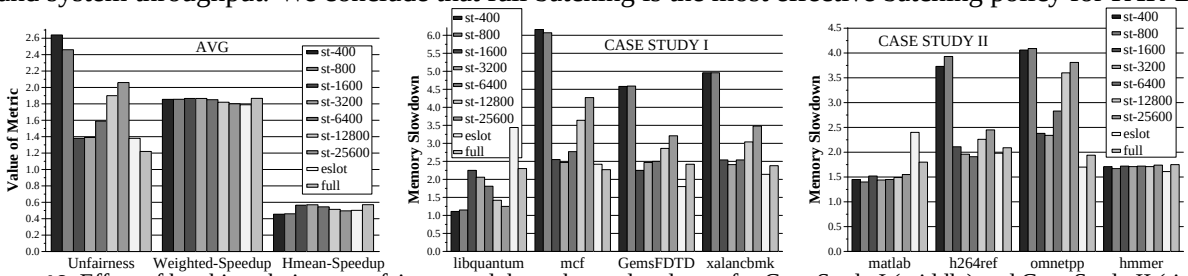


Figure 12. Effect of batching choice on unfairness and throughput; slowdowns for Case Study I (middle) and Case Study II (right)

8.3.3. Effect of Parallelism Awareness and Different Within-Batch Scheduling Schemes Figure 13(left) explores the effect of changing the within-batch ranking scheme or removing it altogether and simply using FR-FCFS or FCFS to prioritize among commands within a batch. We study three alternative within-batch ranking schemes, two of which do not adhere to the shortest-job-first principle: the *random* ranking scheme assigns random ranks to threads when a batch is formed; the *round-robin* scheme alternates the rank of each thread in a round-robin fashion in consecutive batches.

Figure 13(left) shows these alternative non-shortest-job-first within-batch scheduling techniques significantly degrade both fairness and system throughput because they increase the average completion time of threads. Specifically, changing the ranking scheme from *Max-Total* or *Total-Max* (which perform similarly) to a random or round-robin ranking scheme reduces weighted-speedup/hmean-speedup by respectively 5.7% and 9.8%. Using no ranking (i.e., FR-FCFS or FCFS) within a batch completely eliminates *parallelism-awareness* from our proposal while keeping the *request batching* component intact. The result is a decrease in both fairness and throughput. Using the FR-FCFS policy within a batch results in a weighted-speedup/hmean-speedup loss of 4.7% and 10.7% compared to PAR-BS. As expected, FCFS provides better fairness than FR-FCFS but significantly worse throughput.

We conclude that parallelism-awareness is a key component of our proposal. However, even without parallelism-

awareness, the concept of request-batching itself results in designs that are almost competitive with the best previously-proposed scheduler, STFM. As Figure 13(left) shows, round-robin ranking within a batch achieves slightly worse fairness and only 2.1%/1.5% smaller weighted-speedup/hmean-speedup than STFM.

Figure 13(middle and right) shows that the throughput improvement due to parallelism-aware prioritization is significant when threads have high inherent bank-level parallelism (4 copies of lbm), but negligible when threads have low parallelism (4 copies of matlab). We conclude that the parallelism-awareness component of our proposal is independent of the fairness component and it can be used to improve solely system throughput even when fairness is not a problem.

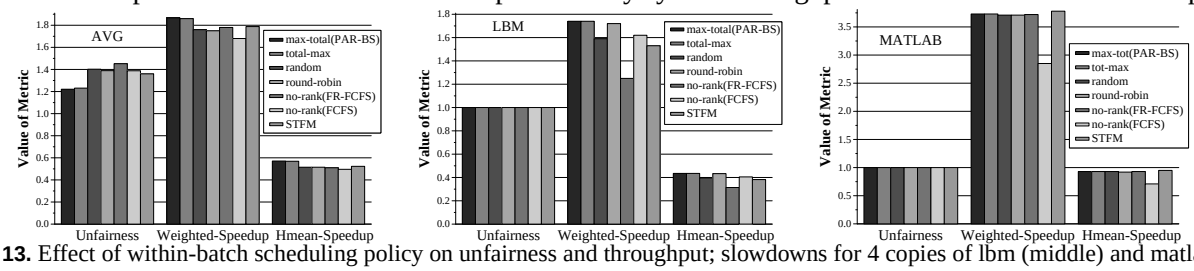


Figure 13. Effect of within-batch scheduling policy on unfairness and throughput; slowdowns for 4 copies of lbm (middle) and matlab (right)

8.4. Evaluation of Support for Thread Priorities

We evaluated PAR-BS's support for thread priorities in a variety of scenarios and present two representative case studies to highlight its effectiveness. Figure 14(left) shows the memory slowdowns of 4 lbm programs with different weights (for NFQ and STFM) and corresponding priorities (for PAR-BS). Two programs have a priority of 1 (corresponding to a weight of 8 in NFQ/STFM) and two have priorities of 2 and 8. While all three schedulers respect the relative priorities of threads, PAR-BS is much more efficient: it results in the lowest slowdown for the highest-priority programs because it preserves their bank-parallelism. Lbm with priority 1 experiences a slowdown of 2.09 and 2.15 with NFQ and STFM, but only 1.88 with PAR-BS. In addition, we found that PAR-BS provides higher system throughput even for low-priority programs (e.g. the lowest-priority lbm has a much smaller slowdown with PAR-BS than with other schemes).

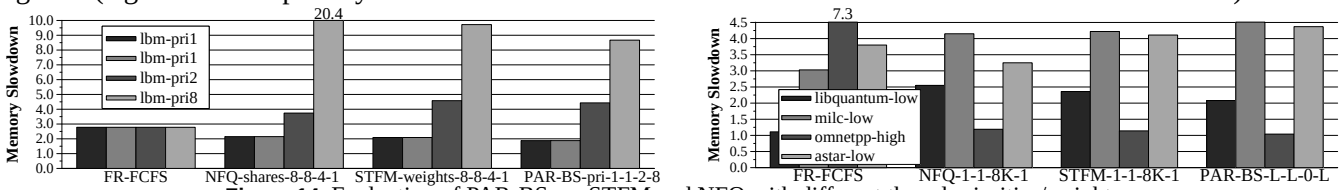


Figure 14. Evaluation of PAR-BS vs. STFM and NFQ with different thread priorities/weights

Figure 14(right) presents a scenario in which omnetpp is the most important thread to the user whereas the other three co-scheduled threads are not important. Therefore, the system software designates the other threads as "opportunistic," i.e. they should be serviced only when there is available bandwidth. As explained in Section 5, PAR-BS easily accommodates this notion of "opportunistic service" by never including these threads' requests in a batch. For NFQ and STFM, there is no notion of "opportunistic service," so we approximated it by assigning a very large weight (8192) to the high-priority omnetpp and very small weights (1) to low-priority threads.[17] PAR-BS provides much higher throughput to the high-priority thread. Omnetpp's slowdown is only 1.04 with PAR-BS whereas it is 1.14 with STFM and 1.19 with NFQ. Hence, from both examples, we conclude that PAR-BS treats higher-priority applications better than alternate approaches for enforcing thread priorities/weights in the DRAM controller.

[17]Note that such a large range of weights might be difficult to implement in NFQ or STFM hardware, whereas PAR-BS's ability to handle opportunistic threads is very easy to implement: it simply consists of not marking the requests of opportunistic threads.

8.5. Sensitivity to System Parameters

Finally, Table 5 shows the effect of key system parameters on the fairness and system throughput provided by PAR-BS as compared to FR-FCFS and STFM (averaged over 100 workloads on the 4-core system). The results show that PAR-BS consistently outperforms the best previous scheduler for different numbers of DRAM banks, memory latencies, row-buffer/L2/instruction-window sizes. We conclude that PAR-BS is effective in a wide variety of system configurations.

	DRAM banks						Row-buffer Size						Instruction Window Size					
	4		8		16		2 KB		4 KB		8 KB		32		128		512	
	Unf.	Hmean-sp	Unf.	Hmean-sp	Unf.	Hmean-sp	Unf.	Hmean-sp	Unf.	Hmean-sp	Unf.	Hmean-sp	Unf.	Hmean-sp	Unf.	Hmean-sp	Unf.	Hmean-sp
FR-FCFS	3.23	0.35	3.12	0.43	2.72	0.49	3.12	0.43	3.93	0.34	5.01	0.28	2.69	0.48	3.12	0.43	3.36	0.41
STFM	1.33	0.44	1.36	0.52	1.32	0.61	1.36	0.52	1.35	0.51	1.34	0.50	1.33	0.59	1.36	0.52	1.29	0.47
PAR-BS	1.22	0.47	1.22	0.57	1.24	0.65	1.22	0.57	1.23	0.55	1.23	0.54	1.23	0.63	1.22	0.57	1.24	0.50
Improvement	1.09X	7.1%	1.11X	8.3%	1.07X	6.2%	1.11X	8.3%	1.10X	7.6%	1.09X	7.8%	1.08X	6.5%	1.11X	8.3%	1.04X	6.4%

	Minimum DRAM Latency						L2 Size									
	80 cycles		160 cycles		240 cycles		Private 512 KB		Private 1 MB		Private 2 MB		Shared 4MB		Shared 8MB	
	Unf.	Hmean-sp	Unf.	Hmean-sp	Unf.	Hmean-sp	Unf.	Hmean-sp	Unf.	Hmean-sp	Unf.	Hmean-sp	Unf.	Hmean-sp	Unf.	Hmean-sp
FR-FCFS	2.81	0.45	3.12	0.43	3.20	0.41	3.12	0.43	3.23	0.43	2.91	0.49	5.92	0.33	3.63	0.41
STFM	1.29	0.55	1.36	0.52	1.30	0.51	1.36	0.52	1.31	0.53	1.32	0.60	2.02	0.49	1.57	0.55
PAR-BS	1.21	0.60	1.22	0.57	1.20	0.55	1.22	0.57	1.23	0.57	1.23	0.64	1.90	0.53	1.47	0.59
Improvement	1.06X	8.4%	1.11X	8.3%	1.08X	7.8%	1.11X	8.3%	1.07X	7.6%	1.07X	7.0%	1.06X	7.7%	1.07X	7.3%

Table 5. Sensitivity of gmean unfairness (Unf.) and system throughput (hmean-speedup) of PAR-BS to various system parameters

9. Related Work

Fair DRAM Controllers: Fair and QoS-aware DRAM controller design in shared memory systems has received increasing attention in the last two years. We already provided extensive qualitative and quantitative comparisons to two very recently proposed DRAM controllers that aim to provide QoS, Nesbit et al.'s network-fair-queueing (NFQ) based scheduler [27] and Mutlu and Moscibroda's stall-time fair memory (STFM) scheduler [25]. Rafique et al. [30] proposed an improvement to the NFQ scheme by employing start-time fair queueing, which provides better fairness than virtual finish-time fair queueing. As explained in [25], while fair queueing is a good fairness abstraction for *stateless* network wires without any *parallelism (i.e., banks)*, it is not directly applicable to DRAM systems because it does not take into account row-buffer state and bank-parallelism, two critical determinants of DRAM performance. In comparison, our design provides not only fairness, QoS, and starvation freedom but also significantly improves system throughput via better intra-thread overlapping of DRAM accesses.

Iyer et al. [11] sketch a design that allows requests from only higher priority threads to bypass other requests in the memory controller. However, their solution does not provide fairness to equal-priority threads. Several DRAM controllers [19, 16] achieve hard real-time guarantees at the cost of a reduction in throughput and flexibility that is unacceptable in high-performance general-purpose systems.

Batching: The general concept of "batching" has been used in disk scheduling [7, 38, 12] to prevent starvation of I/O requests. We apply a similar concept, *request batching*, in our PAR-BS design and evaluate the trade-offs associated with batching in DRAM controllers. However, the locality, bandwidth, parallelism trade-offs in DRAM memory are very different from those in sequential-access disk drives since disk drives do not have 1) a banked structure or 2) row-buffers.

Parallelism Awareness: The concept of memory-level parallelism awareness was exploited in processor caches to improve the cache replacement policy [29]. The authors observed that cache misses that are likely to be serviced in parallel with other misses are less costly on processor performance than misses that occur in isolation. They proposed a replacement policy that tries to keep costly blocks in the cache. Our proposal is orthogonal: it proactively tries to improve the probability that cache misses from a given thread will be serviced in parallel and can 1) be used together with and 2) improve the effectiveness of MLP-aware cache replacement.

DRAM Throughput Optimizations: A number of papers examined the effect of different memory controller policies and DRAM throughput optimizations in multiprocessor/multithreaded [26, 42] and single-threaded systems [32, 20, 31, 10, 33]. These techniques do not consider fairness or intra-thread bank-parallelism.

Fairness in On-Chip Resources: Proposed techniques for fair sharing of CMP caches (e.g., [37, 14]) and multithreaded processor resources (e.g., [36, 18, 8]) are complementary to our work and can be used in conjunction with PAR-BS.

10. Conclusion

We introduced a novel, comprehensive solution to both high-performance and QoS-aware DRAM scheduler design. Compared to existing DRAM schedulers, our parallelism-aware batch scheduler (PAR-BS) significantly improves both fairness and system throughput in systems where DRAM is a shared resource among multiple threads. Our technique combines two orthogonal ideas: 1) it provides thread-fairness and better prevents short-term and long-term starvation through the use of *request batching* and 2) within a batch, it explicitly reduces average thread stall times via a *parallelism-aware DRAM scheduling policy* that improves intra-thread bank-level parallelism, using the *shortest job first* scheduling principle. While effective at improving both fairness and system performance, PAR-BS is also configurable and simple to implement. Our future work will focus on formally analyzing the parallelism, locality, and fairness properties of PAR-BS to further refine the employed request prioritization heuristics.

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] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS-11*, 1997.
[6] J. M. Frailong, W. Jalby, and J. Lenfant. XOR-Schemes: A flexible data organization in parallel memories. In *ICPP*, 1985.
[7] H. Frank. Analysis and optimization of disk storage devices for time-sharing systems. *Journal of the ACM*, 16(4):602–620, Oct. 1969.
[8] R. Gabor, S. Weiss, and A. Mendelson. Fairness and throughput in switch on event multithreading. In *MICRO-39*, 2006.
[9] A. Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Idea Session '98*, Oct. 1998.
[10] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *MICRO-37*, 2004.
[11] R. Iyer et al. QoS policies and architecture for cache/memory in cmp platforms. In *SIGMETRICS*, 2007.
[12] D. M. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPLCSP917rev1, HP Labs, 1991.
[13] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Second Workshop on Memory Performance Issues*, 2002.
[14] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. *PACT-13*, 2004.
[15] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA-8*, 1981.
[16] T.-C. Lin et al. Quality-aware memory controller for multimedia platform SoC. In *IEEE Workshop on Signal Processing Systems*, 2003.
[17] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.
[18] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*, 2001.
[19] C. Macian et al. Beyond performance: secure and fair memory management for multiple systems on a chip. In *FPT*, 2003.
[20] S. A. McKee et al. Dynamic access ordering for streamed computations. *IEEE Transactions on Computers*, 49(11):1255–1271, Nov. 2000.
[21] Micron. *1Gb DDR2 SDRAM Component: MT47H128M8HQ-25*, May 2007. http://download.micron.com/pdf/datasheets/dram/ddr2/1GbDDR2.pdf.
[22] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security*, 2007.
[23] O. Mutlu et al. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA-9*, 2003.
[24] O. Mutlu, H. Kim, and Y. N. Patt. Efficient runahead execution: Power-efficient memory latency tolerance. *IEEE Micro*, 26(1):10–20, 2006.
[25] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO-40*, 2007.
[26] C. Natarajan et al. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI*, 2004.
[27] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO-39*, 2006.
[28] H. Patil et al. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.
[29] M. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA-33*, 2006.
[30] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective management of DRAM bandwidth in multicore processors. In *PACT*, 2007.
[31] S. Rixner. Memory controller optimizations for web servers. In *MICRO-37*, 2004.
[32] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000.
[33] J. Shao and B. T. Davis. A burst scheduling access reordering mechanism. In *HPCA-13*, 2007.
[34] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *ISCA-12*, 1985.
[35] W. E. Smith. Various optimizers for single stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.
[36] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *ASPLOS-IX*, 2000.
[37] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. *HPCA-8*, 2002.
[38] T. J. Teorey and T. B. Pinkerton. A comparative analysis of disk scheduling policies. *Communications of the ACM*, 15(3):177–184, 1972.
[39] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, 1967.
[40] D. H. Woo et al. Analyzing performance vulnerability due to resource denialofservice attack on chip multiprocessors. In *CMP-MSI*, 2007.
[41] Z. Zhang et al. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *MICRO-33*, 2000.
[42] Z. Zhu and Z. Zhang. A performance comparison of DRAM memory system optimizations for SMT processors. In *HPCA-11*, 2005.