Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Shared Memory Controllers

Uncontrolled interthread interference in main memory can destroy individual threads' memory-level parallelism, effectively serializing the memory requests of a thread whose latencies would otherwise have largely overlapped, thereby reducing single-thread performance. The parallelism-aware batch scheduler preserves each thread's memory-level parallelism, ensures fairness and starvation freedom, and supports system-level thread priorities.

Onur Mutlu
Carnegie Mellon University

Thomas Moscibroda
Microsoft Research

••••••The main memory (dynamic RAM) system is a major limiter of computer system performance. In modern processors, which are overwhelmingly multicore (or multithreaded), the concurrently executing threads share the DRAM system, and different threads running on different cores can delay each other through resource contention. One thread's memory requests can cause DRAM bank conflicts, row-buffer conflicts, and data/address bus conflicts with another's. As the number of on-chip cores increases, the pressure on the DRAM system increases, as does the interference among threads sharing the system. Unfortunately, many conventional DRAM controllers are unaware of this interthread interference. They schedule requests simply to maximize DRAM data throughput. For example, the commonly used row-hit-first (FR-FCFS, or first ready, first come, first served) scheduling policy is thread unaware.[1,2]

Uncontrolled interthread interference in DRAM scheduling results in two major problems. First, as previous work showed, a state-of-the-art DRAM controller can unfairly prioritize some threads while starving more important threads for long time periods, as they wait to access memory (see the "Related Work on Memory Controllers" sidebar). For example, FR-FCFS unfairly prioritizes threads with high row-buffer hit rates over those witho low row-buffer hit rates. Similarly, an oldest-first scheduling policy implicitly prioritizes memory-intensive threads over memory-nonintensive ones. In fact, it is possible to write programs to deny DRAM service to more important programs running on the same chip, as we showed in our previous work.[3] Such unfairness in the DRAM controller results in low system performance and utilization, makes the system vulnerable to denial-of-service attacks, and makes the system uncontrollable—that is, unable to enforce operating-system-level thread priorities.

Interthread interference in the DRAM system can also destroy individual threads' bank-level access parallelism, effectively serializing the memory requests whose latencies would have largely overlapped had there been no interference. Many sophisticated single-thread performance improvement techniques aim to amortize the cost of long DRAM latencies by generating multiple outstanding DRAM requests (by exploiting memory-level parallelism[4]). These techniques' effectiveness depends on whether different DRAM banks actually service the thread's outstanding DRAM requests in parallel (that is, whether they maintain intrathread bank-level parallelism). In a single-threaded system, this isn't a problem: because the thread has exclusive access to DRAM banks, its requests are serviced in parallel. However, in a multithreaded or multicore system, multiple threads share the DRAM controller. Because existing controllers make no attempt to preserve each thread's bank-level parallelism, each thread's outstanding requests can be serviced serially (due to interference from other threads' requests), instead of in parallel. Thus, conventional single-thread memory latency tolerance techniques are less effective in systems in which multiple threads share the DRAM memory. As a result, each thread's performance can degrade significantly, which in turn degrades overall system performance.

To solve these problems, we designed the parallelism-aware batch scheduler (PAR-BS), a memory controller that controls and limits interthread interference. Our International Symposium on Computer Architecture (ISCA) paper provides more detail on the work described here.[5]

Destruction of memory-level parallelism in shared memory controllers

DRAM requests are 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 can't commit that (and any subsequent) instruction because instructions are committed in program order to support precise exceptions.[6] The processor's instruction window becomes full a small number of cycles after a last-level cache miss,[7] and the processor stalls until DRAM services the miss. Current processors try to reduce performance loss due to a DRAM access by servicing other DRAM accesses in parallel. Techniques such as out-of-order execution,[8] nonblocking caches,[9] and runahead execution[10,11] strive to overlap the latency of future DRAM accesses with the current access so the processor doesn't need to stall long for future DRAM accesses. Instead of stalling once for each access in a serialized fashion, the processor stalls, at an abstract level, once for all overlapped accesses.[7] The concept of generating and servicing multiple DRAM accesses in parallel is called *memory-level parallelism* (MLP).[4]

In a single-threaded, single-core system, a thread has exclusive access to the DRAM banks, so its concurrent DRAM accesses are serviced in parallel as long as they aren't to the same bank. The simple, conceptual example in Figure 1 illustrates this. Request 1's (Req1) latency is hidden by Request 0's (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 decoding and execution of future instructions. When Load 1 becomes the oldest instruction in the window, its miss has already been serviced, so the processor can continue computation without stalling.

Unfortunately, if multiple threads are generating memory requests concurrently (as in a multicore system), modern DRAM controllers schedule the outstanding requests in a way that completely ignores the threads' inherent memory-level parallelism. Instead, they exclusively seek to maximize the DRAM data throughput—that is, the number of DRAM requests serviced per second.[1,2] As we show here, blindly maximizing the DRAM data throughput doesn't minimize a thread's stall time (which

Related Work on Memory Controllers

A modern synchronous dynamic RAM (SDRAM) chip consists of multiple DRAM banks to let multiple outstanding memory accesses proceed in parallel if they require data from different banks. Each DRAM bank is a 2D array, consisting of columns and rows. Rows typically store data in consecutive memory locations and are 1 to 2 Kbytes in size. The memory controller can access the data in a bank 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 row buffer's status:

- *Row-hit:* The request is to the open row in the row buffer. The DRAM controller issues only a read or write command to the DRAM bank, resulting in a bank access latency of t_{CL}.
- *Row-closed:* The row buffer does not have an open row. The DRAM controller issues an activate command to open the required row and then a read or 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 closes the row by issuing a precharge command, then opens the required row (activate), and issues a read or 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 wait to be serviced and a (possibly two-level) scheduler that selects the next request to be serviced.[1–3] 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 request's state. It schedules a DRAM command for a request only if the request's scheduling doesn't cause any resource (bank and address, data, or command bus) conflicts and doesn't violate any DRAM timing constraints. Such a DRAM command is said to be *ready*.

When multiple threads share the DRAM system, the row-hit-first scheduling policy (FR-FCFS, or first ready, first come, first served) tends to unfairly prioritize threads with high row-buffer locality (that is, high row-buffer hit rate) over those with relatively low row-buffer locality. It also tends to unfairly prioritize memory-intensive threads over nonintensive ones due to the oldest-first prioritization rule. As a result, even though FR-FCFS achieves high DRAM data throughput, it might starve requests or threads for long time periods, causing unfairness and relatively low overall system throughput.[2,3]

Previous research proposed new, fairer scheduling policies that provide quality of service to different threads.[2,3] Nesbit and colleagues applied network fair-queuing (NFQ) techniques to DRAM controllers to divide the DRAM bandwidth among multiple threads sharing the DRAM system.[2] Rafique and colleagues proposed an improvement to the NFQ scheme.[4] However, although fair queuing is a good fairness abstraction for stateless network wires without any parallelism, it isn't directly applicable to DRAM systems because it doesn't account for row-buffer state and bank parallelism, two critical determinants of DRAM system performance.[3] Our stall-time fair memory scheduler (STFM) aims to equalize the slowdowns experienced by threads as

directly correlates with system throughput). Even though we can maximize DRAM throughput, some threads can stall 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 multicore system throughput and increased stall times. We assume two cores, each running a single thread: Thread 0 (T0) and Thread 1 (T1). Each 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 (Figure 2a). The top part of Figure 2b shows how a current DRAM scheduler can destroy intrathread

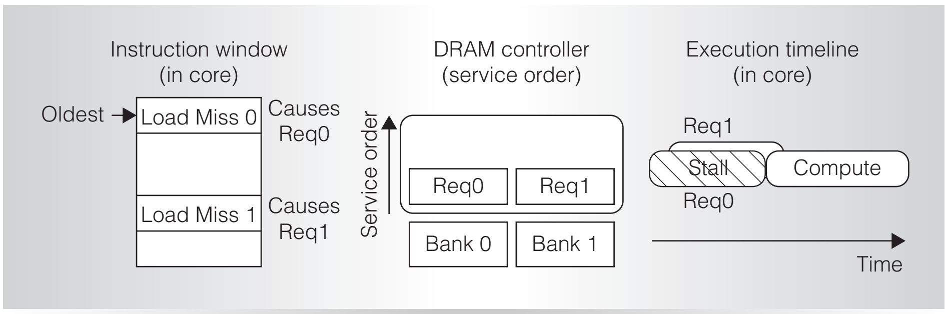

Figure 1. Example showing how latencies of two DRAM requests are overlapped in a single-core system.

compared to when each is run alone.[3] Several DRAM controllers achieve hard real-time guarantees at the cost of reduced throughput and flexibility that is unacceptable in high-performance general-purpose systems.[5] None of these previous controllers accounts for intrathread bank parallelism, and so can significantly degrade system performance when requests of different threads interfere in the DRAM system.

Several publications have examined the effect of new memory controller policies to optimize DRAM throughput in multiprocessor and multithreaded systems[6,7] and in single-threaded systems.[1,8,9] These techniques don't consider fairness or intrathread bank parallelism. Recently proposed prefetch-aware DRAM schedulers[10] adaptively prioritize prefetch versus demand requests. We could combine the basic principles of prefetch-awareness used in that scheduler with our proposed DRAM controller.

References

1. S. Rixner et al., ''Memory Access Scheduling,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 00), IEEE CS Press, 2000, pp. 128-138.
2. K.J. Nesbit et al., ''Fair Queuing Memory Systems,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 06), IEEE CS Press, 2006, pp. 208-222.
3. O. Mutlu and T. Moscibroda, ''Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 07), IEEE CS Press, 2007, pp. 146-160.
4. N. Rafique, W.-T. Lim, and M. Thottethodi, ''Effective Management of DRAM Bandwidth in Multicore Processors,'' *Proc. 16th Int'l Conf. Parallel Architecture and Compilation Techniques* (PACT 07), IEEE CS Press, 2007, pp. 245-258.
5. C. Macian, S. Dharmapurikar, and J. Lockwood, ''Beyond Performance: Secure and Fair Memory Management for Multiple Systems on a Chip,'' *Proc. 2003 IEEE Int'l Conf. Field-Programmable Technology* (FPT 03), IEEE CS Press, 2003, pp. 348-351.
6. Z. Zhu and Z. Zhang, ''A Performance Comparison of DRAM Memory System Optimizations for SMT Processors,'' *Proc. 11th Int'l Symp. High-Performance Computer Architecture* (HPCA 05), IEEE CS Press, 2005, pp. 213-224.
7. E. Ipek et al., ''Self-optimizing Memory Controllers: A Reinforcement Learning Approach,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 08), IEEE CS Press, 2008, pp. 39-50.
8. S.A. McKee et al., ''Dynamic Access Ordering for Streamed Computations,'' *IEEE Trans. Computers,* vol. 49, no. 11, Nov. 2000, pp.1255-1271.
9. I. Hur and C. Lin, ''Adaptive History-Based Memory Schedulers,'' *Proc. 37th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 04), IEEE CS Press, 2004, pp. 343-354.
10. C.J. Lee et al., ''Prefetch-Aware DRAM Controllers,'' *Proc. 41st Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 08), IEEE CS Press, 2008.

bank parallelism, thereby increasing a thread's stall time. The bottom part shows how a parallelism-aware scheduler can schedule the requests more efficiently.

A conventional parallelism-unaware DRAM scheduler[1,2] can service requests in their arrival order, as Figure 2b (top) shows. First, the controller services T0's request to Bank 0 in parallel with T1's request to Bank 1. Later, it services T1's request to Bank 0 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 the execution timeline in Figure 2c (top) shows, instead of stalling once (that is, for one bank-access latency) for the two requests, both cores stall twice. Core 0 first stalls for Load 0, and shortly thereafter 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 in Figure 2b (bottom) and execution timeline in Figure 2c (bottom). The scheduler preserves bank parallelism by first scheduling both of T0's requests in parallel, and then T1's requests. This lets Core 0 execute faster (shown as "saved cycles" in the figure) because it stalls for only one bank-access latency. Core 1's stall time remains unchanged. Although the controller services the second request (T1-Req1) later than a conventional scheduler would, 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 2 bank-access latencies with the conventional scheduler (Figure 2c, top), but only 1.5 bank-access latencies with the parallelism-aware scheduler (Figure 2c, bottom). Although this example shows only two cores for simplicity, the destruction of intrathread bank parallelism

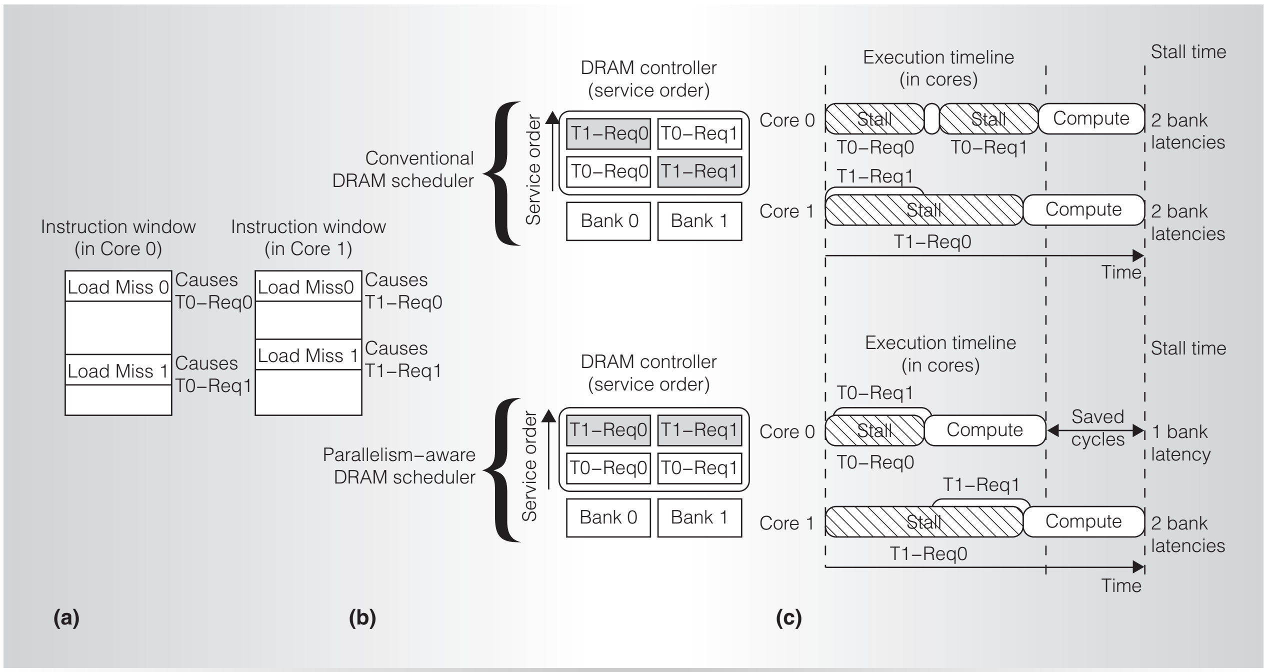

Figure 2. Conceptual example showing the importance of including parallelism awareness in DRAM scheduling decisions: instruction windows in cores 1 and 2 (a), DRAM controllers for conventional and parallelism-aware schedulers (b), and execution timelines (c).

becomes worse as more cores share the DRAM system.

Parallelism-aware batch scheduling

Our PAR-BS controller is based on two key principles. The first principle is *parallelism-awareness*. To preserve a thread's bank-level parallelism, a DRAM controller must service a thread's requests (to different banks) back to back (that is, one right after another, without any interfering requests from other threads). This way, each thread's request service latencies overlap.

The second principle is *request batching*. If performed greedily, servicing requests from a thread back to back could cause unfairness and even request starvation. To prevent this, PAR-BS groups a fixed number of oldest requests from each thread into a batch, and services the requests from the current batch before all other requests. The controller forms a new batch when all requests belonging to the previous batch are fully serviced. Because out-of-order request servicing doesn't occur across batches, no thread can indefinitely deny service to another. Thus, batching ensures fairness and forward progress. It also provides a convenient granularity (that is, a batch) within which the PAR-BS scheduler can service requests according to the first principle, in a possibly unfair but parallelism-aware manner.

The following sections detail these two basic principles' operation.

PAR-BS operation

Batching involves consecutively grouping outstanding requests in the memory request buffer into larger units, or batches. The DRAM scheduler avoids request reordering across batches by prioritizing requests belonging to the current batch over other requests. Once all requests of a batch are serviced (that is, when the batch is finished), the scheduler forms a new batch consisting of outstanding requests in the memory request buffer that weren't included in the last batch. By 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 fine granularity and enforces steady and fair progress across all threads. At the same time, batch formation gives the

Rule 1. Batch formation

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

Figure 3. Rule 1 describes how the PAR-BS scheduler forms batches of requests.

Rule 2. PAR-BS scheduler: Request prioritization

1. **BS—Marked-requests-first:** Marked requests are prioritized over unmarked requests (batch requests to ensure fairness and avoid starvation).
2. **RH—Row-hit-first:** Row-hit requests are prioritized over row-conflict and closed requests (exploit row-buffer locality).
3. **Rank—Higher-rank-first:** Requests from threads with higher rank are prioritized over requests from lower-ranked threads (preserve memory level parallelism).
4. **FCFS—Oldest-first:** Older requests are prioritized over younger ones.

Figure 4. Rule 2 describes how the PAR-BS scheduler prioritizes requests.

scheduler the flexibility to reorder requests within a batch to maximally exploit row-buffer locality and bank parallelism without significantly disturbing thread fairness.

PAR-BS's batching component 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*. PAR-BS forms batches using the steps listed in Rule 1 (Figure 3).

`Marking-Cap` is a system parameter (determined dynamically by the system or empirically by the designer) 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 seven outstanding requests for a bank, PAR-BS marks only the five 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 over nonmarked requests in a given bank. However, PAR-BS neither wastes bandwidth nor unnecessarily delays requests. If there are no marked requests to a given bank, PAR-BS schedules outstanding nonmarked requests to that bank.

Batching naturally provides a convenient granularity (the batch) within which a scheduler can optimize scheduling decisions to obtain high performance. Within a batch of requests (that is, the set of marked requests), we could use any DRAM command scheduling policy (for example, FR-FCFS or FCFS) to prioritize requests. However, no existing policy preserves a thread's bank parallelism in the presence of interthread interference. PAR-BS prioritizes requests as shown in Rule 2 (Figure 4) to achieve two objectives: first, to exploit row-buffer locality; and second, to preserve each thread's bank parallelism.

To achieve the first objective, the controller prioritizes row-hit requests within a batch. This increases row-buffer locality and ensures that the controller makes the best possible use of any rows left open by the previous batch's requests in the next batch. To achieve the second objective, when a new batch is formed, the DRAM scheduler computes a ranking among all threads with requests in the batch. While the controller processes the batch, it prioritizes requests from higher-ranked threads over those from lower-ranked threads (and the computed ranking remains the same). This ensures that each thread's requests are serviced back to back within the batch. The effect of thread-rank-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.

The thread-ranking scheme affects system throughput and fairness. A good ranking scheme should satisfy two objectives:

- maximize system throughput and
- minimize stall-time unfairness (that is, equalize thread slowdown compared to when each is run alone to allow proportional progress of each thread, a property assumed by existing operating system schedulers).

As our previous publication explains,[5] these two objectives call for the same ranking scheme. We can maximize system throughput within a batch by minimizing the average stall

Rule 3. Thread ranking: Shortest stall-time first within batch

For each thread, the scheduler finds 1) the maximum number of marked requests to any given bank (max-bank-load) and 2) the total number of marked requests (total-load).

1. **Max rule:** A thread with lower max-bank-load is ranked higher than a thread with higher max-bank-load (shortest-job-first).
2. **Total rule:** In case of a tie, a thread with lower total-load is ranked higher than a thread with higher total-load. Any remaining ties are broken randomly.

Figure 5. Rule 3 describes how the PAR-BS controller ranks threads using their estimated stall times.

time of threads within the batch (that is, minimizing the average completion time of threads as consistent with generalized machine scheduling theory[12]). By minimizing the average stall time, we maximize the amount of time threads spend on useful computation instead of stalling for DRAM access. We achieve this by servicing threads with inherently low memory stall time (memory-nonintensive threads) early within the batch by assigning them high ranks. Doing so also minimizes stall-time unfairness. The insight is that if a thread has low stall time to begin with (that is, it is nonintensive), delaying it results in a higher slowdown and increased unfairness as opposed to delaying a memory-intensive thread. To achieve both objectives, PAR-BS uses the shortest-job-first principle to rank threads[12] when it forms a new batch. The controller estimates each thread's stall time within the batch. It then gives a higher rank to threads with shorter estimated stall-time. Rule 3 (Figure 5) shows how this is done.

The maximum number of outstanding requests to any bank correlates with the "shortness of the job"—that is, with the minimal memory latency required to serve all requests from a thread if they're 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 nonintensive threads or threads with high bank parallelism make fast progress and aren't delayed unnecessarily long.

Within-batch scheduling example

The example in Figure 6 indicates why our proposed within-batch prioritization scheme preserves intrathread bank parallelism and improves system throughput. The figure abstracts away many details of DRAM scheduling, such as DRAM data, address, and command bus contention, and complex interactions between timing constraints, but provides a framework for understanding the parallelism and locality

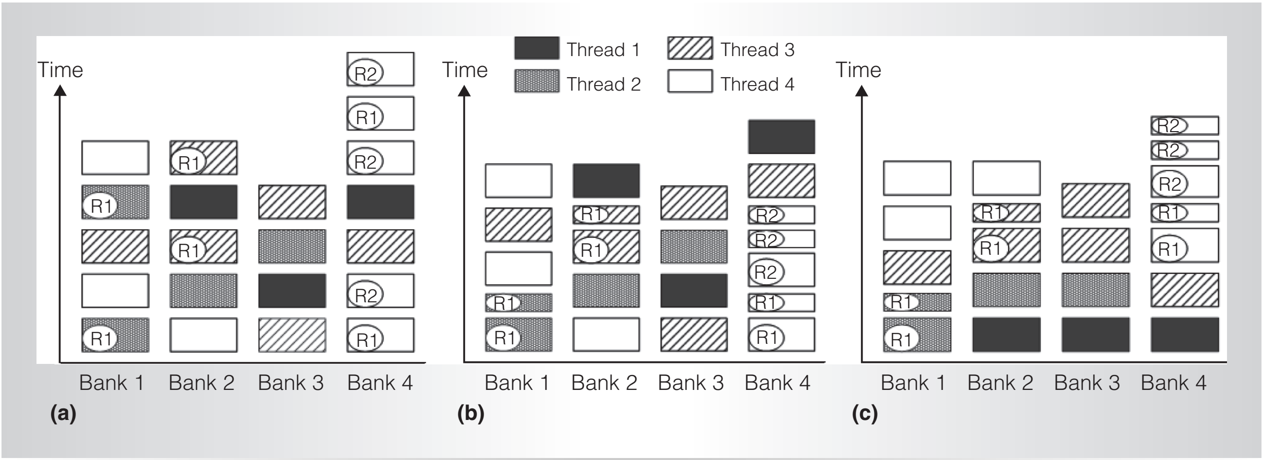

Figure 6. A simplified abstraction of scheduling within a batch containing requests from four threads: arrival order (and FCFS schedule) (a), FR-FCFS schedule (b), and PAR-BS schedule (c). Rectangles represent marked requests from different threads; bottom-most requests are the oldest requests for the 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. We assume the first request to each bank to be a row conflict.

trade-offs. We assume a latency unit of 1 for row-conflict requests and 0.5 for row-hit requests. Figure 6a depicts the arrival order of requests in each bank, which is equivalent to their service order with an FCFS scheduler. FCFS neither exploits row-buffer locality nor preserves intrathread bank parallelism, so it results in the largest average completion time of the four threads (5 latency units, as Table 1 shows). FR-FCFS maximizes row-buffer hit rates by reordering row-hit requests over others, but as Figure 6b shows, it doesn't preserve intrathread 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 memory requests, the schedule shown in Figure 6b for FR-FCFS is also a possible execution scenario when using the quality-of-service-aware network fair-queuing (NFQ) or stall-time fair memory (STFM) schedulers, because those schedulers are unaware of intrathread bank parallelism.

Figure 6c shows how PAR-BS operates within a batch. Thread 1 has at most one request per bank (resulting in the lowest max-bank-load, 1) so is ranked highest in this batch. Threads 2 and 3 have a max-bank-load of 2, but because thread 2 has fewer total requests, it's ranked above thread 3. Thread 4 is ranked the lowest because it has a max-bank-load of 5. 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-BS maximizes the bank parallelism of nonintensive threads and finishes their requests as quickly as possible, letting the corresponding cores make fast progress. Compared to FR-FCFS or FCFS, PAR-BS 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.

In addition to good bank parallelism, our proposal achieves row-buffer locality as well as FR-FCFS within a batch, because within a batch, PAR-BS always prioritizes marked row-hit requests over row-conflict requests.

Table 1. Stall times for three DRAM schedulers.

Stall times	Thread 1	Thread 2	Thread 3	Thread 4	Average
FCFS schedule	4	4	5	7	5
FR-FCFS schedule	5.5	3	4.5	4.5	4.375
PAR-BS schedule	1	2	4	5.5	3.125

Support for system software

So far, we've 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 thread priorities to convey that some threads are more or less important than others. PAR-BS seamlessly incorporates the notion of thread priorities to support the system software. The system software conveys each thread's priority to PAR-BS in terms of priority-levels 1, 2, 3, and so on, where level 1 indicates the most important thread (highest priority) and larger numbers indicate lower priority. Equal-priority threads should be slowed down equally, but the lower a thread's priority, the more tolerable its slowdown. We adjust PAR-BS in two ways to incorporate thread priorities.

First, we apply *priority-based marking*. PAR-BS marks requests from a thread with priority *X* only every *X*th batch. For example, it marks requests from highest-priority threads with level 1 every batch, requests from threads with level 2 every other batch, and so forth.

We also use *priority-based within-batch scheduling*. We add another rule to the within-batch request prioritization rules shown in Rule 2 (Figure 4). Between BS—Marked-requests-first and RH—Row-hit-first, we add 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're marked more frequently and thus take part in more batches, and they're prioritized over other requests within a batch.

In addition to the integer-based priority levels, PAR-BS provides a priority level L, which indicates the lowest-priority threads. Requests from such threads are never marked and they're assigned the lowest priority among unmarked requests. Consequently, PAR-BS schedules requests from threads at level L purely opportunistically—only if the memory system is free—to minimize their disturbance on other threads.

Finally, we let the system software set `Marking-Cap`, which serves as a lever to determine how much unfairness exists in the system.

Evaluation

PAR-BS's storage cost on an 8-core system is only 1,412 bits.[5] PAR-BS is based solely on simple request prioritization rules, similarly to existing DRAM scheduling policies. It doesn't require complex operations (such as division) unlike other QoS-aware schedulers.

Our ISCA paper comprehensively compares PAR-BS with four previously proposed throughput or fairness-oriented DRAM controllers (FR-FCFS, FCFS, NFQ, and STFM) both qualitatively and quantitatively in terms of fairness, system throughput, and configurability.[5] None of the previous controllers try to preserve individual threads' memory-level parallelism. In addition, each of them unfairly penalizes threads with certain properties,[5] whereas PAR-BS's request batching provides a high degree of fairness and starvation freedom for all threads.

We evaluated PAR-BS on a wide variety of workloads consisting of SPEC CPU2006 and Windows applications on 4-, 8-, and 16-core systems using an x86 CMP simulator. Our simulator models the DDR2-800 memory system in detail, faithfully capturing bandwidth limitations, contention, and enforcing bank, port, channel, and bus conflicts. Our previous work details our experimental methodology.[5] Figure 7 summarizes our main results. PAR-BS provides the best fairness,[5] the highest system throughput (weighted speedup),[13] and the best thread turnaround time (in terms of harmonic mean speedup)[13] averaged over all 100, 16, and 12 randomly selected multiprogrammed workloads on the 4-, 8-, and 16-core systems.

On the 4-core system, compared to the FR-FCFS scheduler, PAR-BS improves fairness by 2.56×, harmonic mean speedup by 32.6 percent, and weighted speedup by 12.4 percent. Compared to the STFM scheduler, PAR-BS improves fairness by 1.11×, harmonic mean speedup by 8.3 percent, and weighted speedup by 4.4 percent. Hence, PAR-BS significantly outperforms the best-performing DRAM controller while requiring substantially simpler hardware. PAR-BS is also robust. It doesn't significantly degrade fairness, performance, or maximum request latency on any evaluated multiprogrammed workload.

Our previous work uses detailed case studies and analyses to provide insights into why PAR-BS performs better than other techniques.[5] The main reasons for PAR-BS's performance improvement are its ability to preserve single-thread memory-level parallelism and its shortest-job-first within-batch scheduling policy. Batching and thread ranking together result in the large fairness improvements. PAR-BS is effective with multiple memory controllers, even without any coordination between the different controllers, as our previous ISCA presentation shows.

Our article makes three major contributions, which we hope will serve as building blocks for future research in both industry and academia.

First, we identify a new problem in shared-memory systems: interthread interference can destroy the MLP and serialize requests of individual threads, leading to significant degradation in single-thread and system performance in multicore and multithreaded systems. Computer architects and compiler designers strive to parallelize a thread's memory requests to tolerate memory latency and have developed and used many techniques to exploit MLP (and thus improve single-thread performance). These techniques, including out-of-order execution, can become less effective when multiple threads interfere in the memory system. Over time, this new research problem can lead to novel techniques to preserve MLP (and thus the difficult-to-extract single-thread performance) in other shared system resources as well as memory controllers.

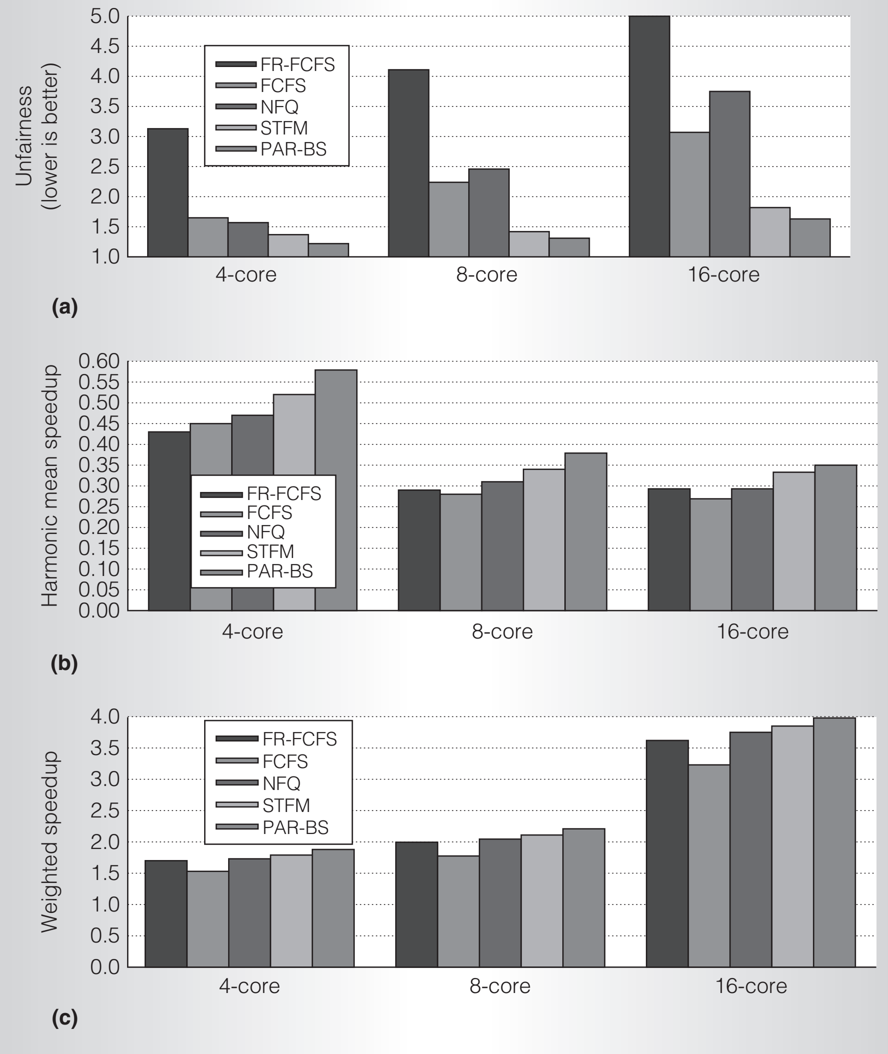

Figure 7. Performance of PAR-BS and other DRAM scheduling techniques in terms of unfairness (a), inverse of thread turnaround time (harmonic mean speedup) (b), and system throughput (weighted speedup) (c). The best achievable unfairness value is 1 in these workloads.

We also introduce the idea of thread ranking and rank-based scheduling to preserve individual threads' MLP. We propose several specific mechanisms to rank threads within a batch, and show that shortest-stall-time-first ranking performs the best. Even when unfairness isn't a problem, preserving MLP significantly improves single-thread and system performance.[5] In the long term, thread ranking can serve as a building block for new techniques to preserve MLP in other shared system resources.

Finally, we adapt the idea of request batching from I/O systems[14] to provide fairness and starvation-freedom to threads sharing the DRAM system. As we show elsewhere,[5] this idea can be used with any existing or future DRAM scheduling technique to improve fairness.

Request batching provides not only fairness at a low hardware cost but also a framework for new scheduling optimizations within the batch. In the long term, we hope this framework will enable sophisticated within-batch DRAM scheduling policies that aggressively exploit the increasingly valuable DRAM bandwidth. For example, a DRAM controller can use this framework to provide fairness while maximizing DRAM throughput within a batch using machine learning or by aggressively prioritizing accurate prefetches. Batching can also serve as a building block for fairness in other shared system resources. In fact, batching and awareness of thread ranking and parallelism can help provide effective sharing of cache bandwidth. We believe and hope the problem discovered in this article will inspire new solutions and that our building blocks will enable both industry and academia to design fair and high-performance shared resource-management techniques. MICRO

References

1. W.K. Zuravleff and T. Robinson, *Controller for a Synchronous DRAM that Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order,* US patent 5,630,096, Patent and Trademark Office, May 1997.
2. S. Rixner, ''Memory Controller Optimizations for Web Servers,'' *Proc. 37th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 04), IEEE CS Press, 2004, pp. 355-366.
3. T. Moscibroda and O. Mutlu, ''Memory Performance Attacks: Denial of Memory Service in Multicore Systems,'' *Proc. Usenix Security,* Usenix Assoc., 2007, pp. 257-274.
4. A. Glew, ''MLP Yes! ILP No!'' *8th Int'l Conf. Architectural Support for Programming Languages and Operating Systems, Wild and Crazy Ideas Session* (ASPLOS WACI), 1998.
5. O. Mutlu and T. Moscibroda, ''Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 08), IEEE CS Press, 2008, pp. 63-74.
6. J.E. Smith and A.R. Pleszkun, ''Implementation of Precise Interrupts in Pipelined Processors,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 85), IEEE CS Press, 1985, pp. 36-44.
7. O. Mutlu, H. Kim, and Y.N. Patt, ''Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance,'' *IEEE Micro,* vol. 26, no. 1, Jan./Feb. 2006, pp. 10-20.
8. R.M. Tomasulo, ''An Efficient Algorithm for Exploiting Multiple Arithmetic Units,'' *IBM J. Research and Development,* vol. 11, no. 1, Jan. 1967, pp. 25-33.
9. D. Kroft, ''Lockup-Free Instruction Fetch/Prefetch Cache Organization,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 8), IEEE CS Press, 1981, pp. 81-88.
10. J. Dundas and T. Mudge, ''Improving Data Cache Performance by Pre-executing Instructions under a Cache Miss,'' *Proc. 11th Int'l Conf. Supercomputing,* ACM Press, 1997, pp. 68-75.
11. O. Mutlu et al., ''Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors,'' *Proc. 9th Int'l Symp. High-Performance Computer Architecture* (HPCA 03), IEEE CS Press, 2003, pp. 129-140.
12. W.E. Smith, ''Various Optimizers for Single Stage Production,'' *Naval Research Logistics Quarterly,* vol. 3, 1956, pp. 59-66.
13. S. Eyerman and L. Eeckhout, ''System-Level Performance Metrics for Multiprogram Workloads,'' *IEEE Micro,* vol. 28, no. 3, May/June 2008, pp. 42-53.
14. H. Frank, ''Analysis and Optimization of Disk Storage Devices for Time-Sharing Systems,'' *J. ACM,* vol. 16, no. 4, Oct. 1969, pp. 602-620.

Onur Mutlu is an assistant professor of electrical and computer engineering at Carnegie Mellon University. His research interests include computer architecture and systems research. Mutlu has a PhD in electrical and computer engineering from the University of Texas, Austin.

Thomas Moscibroda is a researcher at Microsoft Research in Redmond. His research focuses on distributed computing, networking, and algorithmic aspects of computer architecture. Moscibroda has a PhD in computer science from ETH Zurich.

Direct questions and comments about this article to Onur Mutlu, Carnegie Mellon University, ECE Dept., 5000 Forbes Ave., Pittsburgh, PA 15213; onur@cmu.edu.