Parallelism-Aware Batch Scheduling: Paving the Way to High-Performance and Fair Memory Controllers*

Onur Mutlu†§ Thomas Moscibroda†
†Microsoft Research §Carnegie Mellon University

ABSTRACT

In modern processors, the DRAM system is shared among concurrently-executing threads. Memory requests from a thread can delay requests from other threads by causing bank/bus/row-buffer conflicts. Conventional DRAM controllers are unaware of inter-thread interference, which causes two problems. First, some threads are unfairly penalized and denied DRAM service for long time periods. Second, as we show in our ISCA-35 paper, each thread's memory-level parallelism can be destroyed. A thread's outstanding requests that would have been serviced in parallel can effectively become serialized, exposing the latency of each request. As a result, both single-thread performance and system performance/fairness degrade.

Our ISCA-35 paper proposes *parallelism-aware batch scheduling* (PAR-BS), a new approach to designing a shared DRAM controller. PAR-BS is based on two new basic building blocks which collectively reduce inter-thread interference in DRAM, ensure fairness, and preserve the memory-level parallelism of each thread. As a result, PAR-BS reduces the memory-related stall-time experienced by the threads. In addition, PAR-BS provides fairness, avoids starvation of any thread, and seamlessly incorporates support for system-level thread priorities. Our evaluations show that PAR-BS significantly improves both fairness and system performance compared to four previous DRAM controllers across a wide variety of workloads and systems.

1 Summary

1.1 The Problem: Uncontrolled Inter-Thread Interference in the DRAM System

The DRAM memory system is one of the major limiters of computer system performance. In modern processors, which are overwhelmingly multi-core (or multithreaded), the DRAM system is shared among the concurrently-executing threads. Different threads running on different cores can delay each other by causing resource contention. One thread's memory requests can cause DRAM bank conflicts, row-buffer conflicts, and data/address bus conflicts to another's. As the number of on-chip cores increases, the pressure on the DRAM system and hence the interference among threads sharing it increases.

Unfortunately, conventional DRAM controllers are unaware of this interference. They schedule requests to simply maximize DRAM data throughput. For example, the commonly-employed FR-FCFS scheduling policy [20, 19] is thread-unaware: it prioritizes 1) row-hit requests and 2) all else being equal, older requests over others. Uncontrolled inter-thread interference in DRAM scheduling results in two major problems, which are impediments to building viable and scalable multi-core systems.

1. Unfairness and Denial of DRAM Service: First, as previous work [16, 11, 14] showed, some threads can be unfairly prioritized, while more important threads can be starved for long time periods waiting to access memory.[1] In fact, programs can be written to deny DRAM service to more important programs running on the same chip [11]. Such unfairness 1) results in low system performance/utilization [16, 11, 14], 2) makes the system vulnerable to denial of service [11], and 3) makes the system uncontrollable, i.e., unable to enforce thread priorities [14].

2. Destruction of Memory-Level Parallelism (MLP): Second, our ISCA-35 paper shows that inter-thread interference results in another, new problem: it can destroy the *bank-level access parallelism* of individual threads, effectively serializing the memory requests whose latencies would otherwise have been largely overlapped (had there been no interference). Many sophisticated single-thread performance improvement techniques, such as out-of-order execution [21], non-blocking caches [10], and runahead execution [5, 13] are used/designed to amortize the cost of long DRAM memory latencies by generating multiple outstanding DRAM requests (by exploiting memory-level parallelism [8]). The effectiveness of these techniques critically depends on whether the thread's outstanding DRAM requests are actually serviced in parallel by different DRAM banks (i.e., whether or not intra-thread bank-level parallelism is maintained). In a single-threaded system, this is not a problem: since the thread has exclusive access to DRAM banks, its requests are serviced in parallel.[2] However, in a multi-threaded/multi-core system, multiple threads share the DRAM controller. As existing controllers make no attempt to preserve the bank-level parallelism of each thread, each thread's outstanding requests can be serviced serially (due to interference from other threads' requests), instead of in parallel. *This new problem makes conventional single-thread memory latency tolerance techniques less effective in systems where multiple threads share the DRAM memory.* As a result, each thread's performance can degrade significantly, which in turn degrades overall system performance.

Figure 1 demonstrates the problem pictorially: if two threads (T0 and T1) each have two outstanding requests to two different banks, an existing DRAM controller may first service T0's request to Bank 0 in parallel with T1's request to Bank 1, and subsequently T1's request to Bank 0 in parallel with T0's request to Bank 1. This service order exposes two bank access latencies to each thread. In contrast, a parallelism-aware controller would service each thread's requests in parallel (e.g., T0's requests first, then T1's), thereby exposing only one bank access latency to one of the threads, without slowing down the other. Ultimately, this improves both single-thread and system performance.

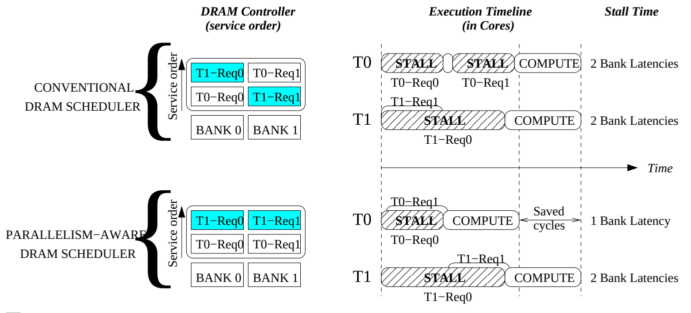

Figure 1: Destruction of memory-level parallelism in DRAM control (top) and how a parallelism-aware controller can do better (bottom)

The goal of our ISCA-35 paper is to design a DRAM controller that controls and limits inter-thread interference to solve the above two problems. To achieve this goal, our paper describes a new DRAM scheduler that 1) provides fairness and starvation-freedom to threads sharing the DRAM system, 2) preserves each thread's DRAM bank parallelism, thereby improving each thread's memory latency tolerance, and 3) is configurable enough to enable different service levels (i.e., Quality of Service) to threads with different priorities.

*The full paper is: "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," O. Mutlu and T. Moscibroda, ISCA-35, June 2008. The full paper can be downloaded from http://research.microsoft.com/∼onur/pub/parbs_isca08.pdf. An animated presentation summarizing the paper can be downloaded from http://research.microsoft.com/∼onur/pub/mutlu_isca08_talk.ppt

[1] For example, the FR-FCFS policy unfairly prioritizes threads with high row-buffer hit rates over those with low row-buffer hit rates.

[2] As long as they are not to the same bank.

1.2 Solution: Parallelism-Awareness and Batching

Our PAR-BS controller is based on two key principles.

1. Parallelism-Awareness: To preserve a thread's bank-level parallelism, a DRAM controller needs to service a thread's requests (to different banks) back-to-back (i.e., one right after another, without any interfering requests from other threads), because this way, each thread's request service latencies overlap.

2. Request Batching: Parallelism-aware scheduling by itself could cause unfairness and even starvation of requests. 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 one are fully serviced. Since there is no out-of-order servicing of requests across batches, no thread can indefinitely deny service to another. Thus, batching ensures fairness and forward progress. It also provides a convenient granularity (i.e., a batch) within which the PAR-BS scheduler can service requests according to the first principle, in a possibly unfair but parallelism-aware manner.

1.2.1 Operation of PAR-BS

The operation of PAR-BS is explained in detail with examples and animations in our ISCA-35 paper [15] and presentation [12], respectively. Here, we briefly describe its key components.

Batching: Each memory request has a bit (*marked* bit) associated with it. Rule 1 describes batching using this bit:

Rule 1 Batch Formation

1: **Forming a new batch:** A new batch is formed when there are no marked requests left in the memory request buffer.
2: **Marking:** When forming a new batch, PAR-BS marks up to `Marking-Cap` outstanding requests per bank for each thread; all marked requests constitute the batch. (Marking-Cap is determined by the system or empirically by the designer)

Within-batch prioritization: Within a batch of requests (i.e., the set of marked requests), any DRAM command scheduling policy (e.g., FR-FCFS, FCFS, or [16, 14]) could be employed to prioritize requests. However, no existing policy preserves a thread's bank parallelism in the presence of inter-thread interference. PAR-BS prioritizes requests as shown in Rule 2 in order to achieve two objectives: 1) exploit row-buffer locality, 2) preserve each thread's bank parallelism. To achieve the latter objective, 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, requests from higher-ranked threads are prioritized 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.

Rule 2 PAR-BS Scheduler: Request Prioritization

1: **BS—Marked-requests-first:** Marked requests are prioritized over requests that are not marked (batching: ensure fairness and avoid starvation).
2: **RH—Row-hit-first:** Row-hit requests are prioritized over row-conflict/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 over younger ones.

Thread ranking: The thread ranking scheme affects both system throughput and fairness. A good ranking scheme should 1) maximize system throughput and 2) minimize stall-time unfairness [14] (i.e., equalize the slowdown of threads compared to when each is run alone in order to allow proportional progress of each thread, a property assumed by existing operating system schedulers). As we explain in our paper (Section 4.2), these two objectives call for the same ranking scheme. Maximizing system throughput within a batch is achieved by minimizing the average stall-time of threads within the batch. This, in turn, is achieved by servicing threads with inherently low memory stall-time (i.e., memory non-intensive threads) early within the batch. Doing so also improves stall-time fairness. The insight is that if a thread has low stall-time to begin with (i.e., is non-intensive), 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. The controller estimates each thread's stall-time within the batch. Then, it ranks threads with shorter estimated stall-time higher. Rule 3 shows how this is done:

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 (called 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.
2: **Total rule:** In case of a tie, a thread with lower total-load is ranked higher than a thread with higher total-load.

1.2.2 Configurability: Support for Thread Priorities

PAR-BS seamlessly incorporates support for system-level thread priorities. First, it marks requests from lower-priority threads less frequently. Lowest-priority threads' requests are never marked. Second, given the choice between two requests, PAR-BS prioritizes the higher priority thread's request. For details and a quantitative evaluation, see Sections 5 and 8.4 of our paper.

1.2.3 Implementation and Hardware Cost

PAR-BS's storage cost on an 8-core CMP is only 1412 bits (see Sec. 6 of [15]). PAR-BS is solely based on simple request prioritization rules, similarly to existing DRAM scheduling policies. It does not require any complex operations (e.g., division) unlike other QoS-aware schedulers (e.g., [16, 14]).

1.2.4 Comparisons with Other DRAM Controllers

Our ISCA-35 paper comprehensively compares PAR-BS with four previously-proposed throughput- or fairness-oriented DRAM controllers (FR-FCFS [20], FCFS [20], NFQ [16], and STFM [14]) qualitatively (Sec. 2, 4, 8) and quantitatively (Sec. 8). None of the previous controllers try to preserve the memory-level parallelism of individual threads. In addition, each of them unfairly penalizes threads with certain properties (Sec. 8) whereas PAR-BS's request batching provides a high degree of fairness and starvation-freedom for *all* threads.

1.2.5 Experimental Results

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.[3] Figure 2 summarizes our main results. PAR-BS provides the best fairness, the highest system throughput (weighted-speedup), and the best thread turnaround time (in terms of hmean-speedup) [7] averaged over all workloads.[4] On the 4-core system, PAR-BS improves fairness by 1.11X/2.56X, hmean-speedup by 8.3%/32.6%, and weighted-speedup by 4.4%/12.4% compared to respectively the best previous technique, stall-time fair memory (STFM) scheduler [14], and the commonly-employed FR-FCFS scheduler. Hence, PAR-BS significantly outperforms the best-performing DRAM controller, while requiring substantially simpler hardware. PAR-BS is also robust: it does not degrade fairness, performance, or maximum latency on any workload.

We provide insights into *why* PAR-BS performs better than other techniques via detailed case studies and analyses (Sec. 2, 4, 8.1), analyze tradeoffs in the PAR-BS design, evaluate batching and parallelism-awareness in isolation (Sec. 8.3), evaluate alternative designs we examined (Sec. 8.3), and quantitatively show PAR-BS's support for thread priorities is effective (Sec. 8.4). Our presentation [12] shows that PAR-BS is very effective with multiple memory controllers, even without any coordination.

2 Novelty and Long-Term Impact

The DRAM system is a critical shared resource that determines the performance and scalability of multi-core systems: if the cores cannot be supplied data in an efficient manner, they cannot perform useful computation. A DRAM controller's scheduling policy determines both the individual thread performance and overall system performance. As the number of cores on

[3]Our experimental methodology is described in Section 6 of [15].
[4]100, 16, and 12 workloads on the 4-,8-, and 16-core systems.

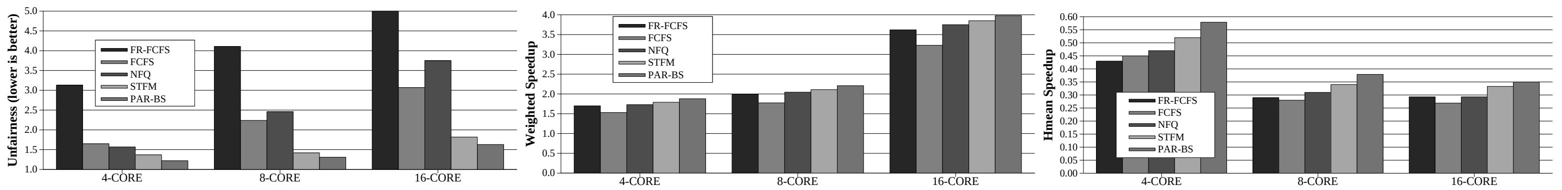

Figure 2: Performance of PAR-BS vs. other DRAM scheduling techniques in terms of (a) unfairness [14], (b) system throughput (weighted-speedup) [7], and (c) inverse of thread turnaround time (hmean-speedup) [7]. The best achievable unfairness value is 1 in these workloads.

a die continues to increase faster than off-chip DRAM bandwidth, techniques that distribute the limited DRAM performance fairly across threads while maximizing single-thread performance become increasingly necessary. No existing DRAM controller recognizes the destruction of a thread's memory-level parallelism as a problem. Our ISCA-35 paper presents a simple, low-complexity DRAM controller that both preserves single-thread MLP (and hence performance) and provides fairness/QoS/starvation-freedom to threads sharing the DRAM.

2.1 Contributions and Impact on Future Research

Our paper makes three major contributions which will likely have long-term impact on both industry and academia:

1. The Problem of Memory-Level Parallelism Destruction: We identify a new problem in shared memory systems: inter-thread interference can destroy the MLP and serialize requests of individual threads, leading to significant degradation in both single-thread and system performance in multi-core/multi-threaded systems. This problem did not exist in single-core/single-threaded systems. Computer architects (and compiler designers) strive very hard to parallelize a thread's memory requests to tolerate memory latency and have developed/used many techniques (e.g., [21, 10, 5, 8, 17, 13, 23, 1, 4, 2, 3, 18, 22, 6]), to exploit MLP. Our paper shows that these techniques, including out-of-order execution, can become ineffective 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 hard-extracted single-thread performance) in other shared system resources as well as memory controllers.

2. A Building Block for Preserving Memory Parallelism: Our paper introduces the idea of *thread ranking* and *rank-based scheduling* to preserve the MLP of individual threads. 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 is not a problem*, preserving MLP significantly improves single-thread and system performance (see Sec. 8.1.3). In the long term, the idea of thread ranking can be used as a building block for new techniques to preserve MLP in other shared resources (e.g., caches, interconnects, the I/O system).

3. A Building Block for Memory System Fairness/QoS: Our paper introduces the idea of *request batching* to provide fairness and starvation-freedom to threads sharing the DRAM system. As we show (in Sec. 8.3.3), this idea can be employed with any existing/future DRAM scheduling technique to improve fairness. The concept of "request batching" provides not only fairness at very low hardware cost but also a framework for new scheduling optimizations within the batch. In the long term, we believe 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 [9]. Batching can also be used as a building block for fairness in other shared system resources.

2.2 Long-Term Impact on Industry

The importance of both the problems (MLP destruction and unfairness) and the solution presented in our paper will increase in future systems. The importance of "preserving MLP" is likely to increase as memory latency tolerance techniques continue to be necessary to keep/improve single-thread performance in multi-core systems. The importance of effectively/fairly managing the DRAM system will also increase as the number of cores sharing the DRAM system is increasing faster than memory performance (bandwidth, speed, latency, and parallelism). In addition, future systems will likely execute increasingly diverse workloads (e.g., using virtualization on data centers for server consolidation) that have very different memory performance requirements and access characteristics. As more and more diverse threads share the DRAM, both the serialization of otherwise-parallel requests and unfairness will increase.

As a result, DRAM controllers will likely be one of the primary performance/QoS limiters in future computer systems. To keep area, power, testing and verification overheads minimal, the modifications to the DRAM controller should be low-cost and simple. Our paper offers a low-cost, implementable solution that preserves MLP and provides fairness in the DRAM system.

2.3 Conclusion

The two major building blocks proposed in this paper can be used/extended to improve the management of other shared resources in multi-core/multithreaded systems. For example, the ideas of batching and thread-ranking/parallelism-awareness are directly applicable to provide effective sharing of cache bandwidth. We believe that these ideas are also applicable to preserve both MLP and fairness in shared resources, such as on-chip interconnects, caches, prefetchers.

Our ISCA paper presents a new problem and practical ideas for designing fair and MLP-preserving memory systems. We believe and hope the problem discovered in this paper will inspire new solutions and our building blocks will enable both industry and academia to design fair and high-performance shared resource management techniques.

REFERENCES

[1] H. Akkary et al. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO-36*, 2003.

[2] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-performance throughput computing. *IEEE Micro*, 25(3):32–45, May 2005.

[3] Y. Chou et al. Store memory-level parallelism optimizations for commercial applications. In *MICRO-38*, 2005.

[4] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA-31*, 2004.

[5] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS-11*, 1997.

[6] S. Eyerman and L. Eeckhout. A memory-level parallelism aware fetch policy for SMT processors. In *HPCA-13*, 2007.

[7] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.

[8] A. Glew. MLP yes! ILP no! In *ASPLOS WACI*, 1998.

[9] E. Ipek et al. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA-35*, 2008.

[10] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA-8*, 1981.

[11] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security*, 2007.

[12] O. Mutlu. Parallelism-Aware Batch Scheduling. `http://research.microsoft.com/~onur/pub/mutlu_isca08_talk.ppt`.

[13] O. Mutlu et al. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA-9*, 2003.

[14] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO-40*, 2007.

[15] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA-35*, 2008.

[16] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO-39*, 2006.

[17] V. S. Pai and S. Adve. Code transformations to improve memory parallelism. In *MICRO-32*, 1999.

[18] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA-33*, 2006.

[19] S. Rixner. Memory controller optimizations for web servers. In *MICRO-37*, 2004.

[20] S. Rixner et al. Memory access scheduling. In *ISCA-27*, 2000.

[21] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of RD*, 11:25–33, Jan. 1967.

[22] J. Tuck, L. Ceze, and J. Torrellas. Scalable cache miss handling for high memory-level parallelism. In *MICRO-39*, 2006.

[23] H. Zhou and T. M. Conte. Enhancing memory level parallelism via recovery-free value prediction. In *ICS-17*, 2003.