Thread Cluster Memory Scheduling

Memory schedulers in multicore systems should carefully schedule memory requests from different threads to ensure high system performance and fair, fast progress of each thread. No existing memory scheduler provides both the highest system performance and highest fairness. Thread Cluster Memory scheduling is a new algorithm that achieves the best of both worlds by differentiating latency-sensitive threads from bandwidth-sensitive ones and employing different scheduling policies for each.

Yoongu Kim
Michael Papamichael
Onur Mutlu
Mor Harchol-Balter
Carnegie Mellon University

High latency of off-chip memory accesses has long been a critical bottleneck in thread performance—particularly in chip-multiprocessors where memory is shared among concurrently executing threads. When a thread accesses memory, it contends with other threads and, as a result, can slow down. Inter-thread memory contention, if not properly managed, can have devastating effects on individual thread performance as well as overall system throughput, leading to system underuse and even thread starvation.[1]

A memory scheduling algorithm's effectiveness is commonly evaluated on the basis of two objectives: fairness[2-4] and system throughput.[3-5] No single thread should be disproportionately slowed down, but on the other hand, the overall system's throughput should remain high. Intuitively, fairness and high system throughput ensure that all threads progress at a relatively even and fast pace. Designing a memory scheduler that achieves both high fairness and high system throughput is a difficult task.

Applying a single memory scheduling policy across all threads, an approach commonly employed by existing memory scheduling algorithms, can't address the disparate needs of different threads. Therefore, existing algorithms can't decouple the system throughput and fairness goals and achieve them simultaneously.

By using a multifaceted approach, our TCM (Thread Cluster Memory) scheduling algorithm can achieve both the highest system throughput and the highest system fairness of any known memory scheduling approach.

Problems with existing approaches

TCM can solve several problems that existing approaches can't handle. (See the "Related work in memory scheduling algorithms" sidebar for more information.)

Bias toward one metric

Previously proposed memory scheduling algorithms are biased toward either fairness or system throughput. To illustrate this, Figure 1 plots the unfairness (the slowdown of the thread that incurs the highest slowdown compared to when it is run alone) and system throughput (weighted speedup) of four previous, state-of-the-art algorithms. An ideal memory scheduling algorithm would fall near the plot's lower (better fairness) right (better system throughput). Unfortunately, no previous scheduling algorithm achieves the best fairness and the best system throughput simultaneously.

Single-faceted approaches are inadequate

Previous memory scheduling algorithms can't balance fairness and system throughput, because they employ the same policy for all threads. Such a single-faceted approach can't address different threads' disparate needs.

For example, in one extreme, by trying to equalize the amount of bandwidth each thread receives, a scheduling algorithm could achieve some notion of fairness, but at a large expense to system throughput.[2] At the opposite extreme, strictly prioritizing certain favorable threads over all others would increase system throughput, but at a large expense to fairness.[5] As a result, such relatively single-faceted approaches can't provide the highest fairness and system throughput simultaneously.

Balancing fairness and system throughput

A workload can consist of a diverse mix of threads that exhibit different memory access behavior. A well-designed memory scheduling algorithm should strive to maximize overall system throughput, but at the same time, it should bound the worst-case slowdown experienced by any one thread. These two goals often conflict and form a trade-off between fairness and system throughput.

To simultaneously achieve high fairness and system throughput, the memory scheduling algorithm must consider threads' memory access behavior because memory access behavior of threads affects how those threads interfere in the memory system (as we show later).

Key insights

Our new scheduling algorithm exploits differences in thread memory behavior to optimize for both system throughput and fairness. It is based on several key insights that we developed and integrated in our paper presented at the 43rd Annual IEEE/ACM International Symposium on Microarchitecture.[6]

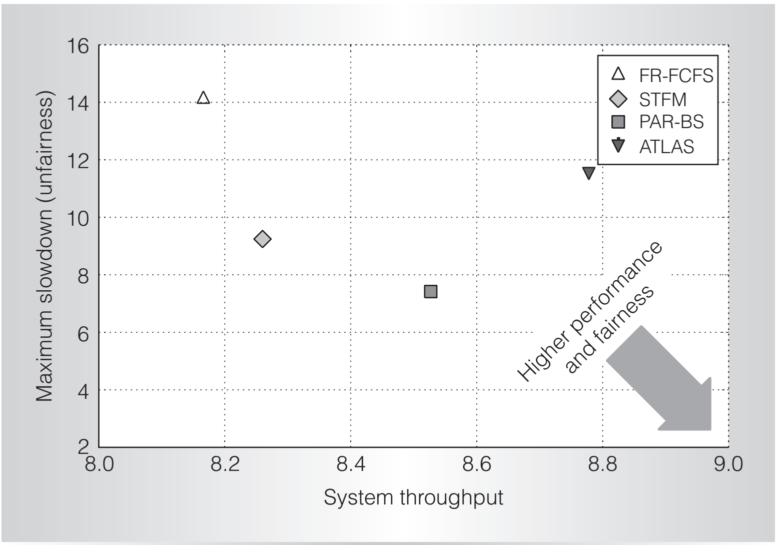

Figure 1. Performance and fairness of state-of-the-art memory scheduling algorithms. The lower right corner is toward the ideal operation point.

First, prior studies have demonstrated the system throughput benefits of prioritizing non-memory-intensive threads over memory-intensive threads.[4,5,7] Non-memory-intensive threads seldom generate memory requests and have greater potential for making fast progress in the processor. Therefore, to maximize system throughput, a memory scheduling algorithm should prioritize non-memory-intensive threads. Doing so doesn't degrade fairness, because non-memory-intensive threads are "light" and rarely interfere with memory-intensive threads.

Second, we observe that unfairness problems usually stem from interference among memory-intensive threads. When we prioritize threads on the basis of memory intensity,[5] medium-intensity threads deny memory access to higher-intensity threads. As a result, higher-intensity threads experience large slowdowns and suffer from unfairness or even starvation.

Third, we observe that periodically shuffling the priority order among memory-intensive threads reduces unfairness because threads take turns gaining prioritized access to memory. However, we find that shuffling symmetrically (where each thread has an equal chance of being at all priority levels) can be ineffective because threads aren't equal in their propensity to interfere with others. Therefore, a memory scheduling

Related work in memory scheduling algorithms

We describe related work on memory scheduling and qualitatively compare Thread Cluster Memory to several previous designs. Our main article compares TCM quantitatively with four state-of-the-art schedulers.

Thread-unaware schedulers

Researchers have examined memory controller designs that don't distinguish between different threads[1-7] within the context of single-threaded, vector, or streaming architectures. Existing systems commonly employ variants of the FR-FCFS (first-ready, first-come first-serve) scheduling policy,[1,2] which prioritizes row-hit requests over other requests. Recent work explored reducing the cost of the FR-FCFS design for accelerators.[8] These policies aim to maximize DRAM throughput. Thread-unaware scheduling policies have been shown to be low-performance and prone to starvation when multiple competing threads share the memory controller in general-purpose multicore and multithreaded systems.[9-15]

Thread-aware schedulers

In contrast, researchers have recently designed thread-aware memory schedulers to improve fairness and provide quality of service (QoS). Fair-queueing memory schedulers adapt variants of the fair-queueing algorithm from computer networks to build a memory scheduler that provides QoS to each thread.[10,11] The Stall-Time Fair Memory (STFM) scheduler uses heuristics to estimate each thread's slowdown compared to when it runs alone, and it prioritizes the thread that has slowed down the most.[13] These algorithms aim to maximize fairness, although they can also lead to throughput improvements by increasing system utilization.

The ATLAS (Adaptive Per-Thread Least Attained Service) scheduler strives to maximize system throughput by prioritizing threads that have attained the least service from the memory controllers.[15] This increase in system throughput comes at the cost of fairness because the most memory-intensive threads receive the lowest priority and experience very high slowdowns.

Parallelism-aware batch scheduling (PAR-BS) strives to balance fairness and system throughput.[14] For fairness, PAR-BS groups memory requests into batches and prioritizes the oldest batch. For throughput, PAR-BS prioritizes threads that are less memory intensive, within the oldest batch. However, the batching policy limits system throughput; it implicitly penalizes non-memory-intensive threads because memory-intensive threads usually insert many more requests into a batch.

Ipek et al. leverage machine-learning techniques to implement memory scheduling policies;[12] Zhu et al. describe memory scheduling optimizations for simultaneous multithreading (SMT) processors.[16] Neither considers fairness or system throughput when threads compete. Lee et al. describe a mechanism to adaptively prioritize between prefetch and demand requests in a memory scheduler;[17] their mechanism can be combined with ours.

References

1. W.K. Zuravlev 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, to Microunity Systems Eng., Inc., 1997.

algorithm should shuffle thread priorities such that a thread that's more likely to cause interference has a smaller chance of being assigned a high priority.

Fourth, as previous work has shown,[4,5,8] scheduling decisions should be made in a synchronized manner across all banks such that each thread's concurrent requests are serviced in parallel, without being serialized because of interference from other threads.

Memory access behavior

TCM defines a thread's memory access behavior using three components from previous work: memory intensity,[5] bank-level parallelism,[4] and row-buffer locality.[9]

Memory intensity

Memory intensity is the frequency at which a thread misses in the last-level cache and generates memory requests. It is measured in the unit of (cache) misses per thousand instructions (MPKI).

Bank-level parallelism

A thread's bank-level parallelism is the average number of banks to which it has outstanding memory requests when the thread has at least one outstanding request. In the extreme case where a thread concurrently accesses all banks at all times, its bank-level parallelism would equal the total number of banks in the memory subsystem.

Row-buffer locality

A memory bank is internally organized as a 2D structure of rows and columns. The column is the smallest addressable unit of memory, and multiple columns make up a single row. When a thread accesses a particular column within a particular row, the memory bank places that row into a small internal

2. S. Rixner et al., ''Memory Access Scheduling,'' *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 00), ACM Press, 2000, pp. 128-138.
3. L. Zhang et al., ''The Impulse Memory Controller,'' *IEEE Trans. Computers,* vol. 50, no. 11, 2001, pp. 1117-1132.
4. S.A. McKee et al., ''Dynamic Access Ordering for Streamed Computations,'' *IEEE Trans. Computers,* vol. 49, no. 11, 2000, pp. 1255-1271.
5. I. Hur and C. Lin, ''Adaptive History-Based Memory Schedulers,'' *Proc. 37th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS Press, 2004, pp. 343-354.
6. J. Shao and B.T. Davis, ''A Burst Scheduling Access Reordering Mechanism,'' *Proc. IEEE 13th Int'l Symp. High Performance Computer Architecture* (HPCA 07), IEEE CS Press, 2007, pp. 285-294.
7. C. Natarajan et al., A Study of Performance Impact of Memory Controller Features in Multi-processor Server Environment,'' *Proc. 3rd Workshop Memory Performance Issues* (WMPI 04), ACM Press, 2004, pp. 80-87.
8. G.L. Yuan, A. Bakhoda, and T.M. Aamodt, ''Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures,'' *Proc. 42nd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* ACM Press, 2009, 34-44.
9. T. Moscibroda and O. Mutlu, ''Memory Performance Attacks: Denial of Memory Service in Multi-core Systems,'' *Proc. 16th USENIX Security Symp.* (SS 07), Usenix Assoc., 2007, pp. 257-274.
10. K.J. Nesbit et al., ''Fair Queuing Memory Systems,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS Press, 2006, pp. 208-222.
11. 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.
12. E. Ipek et al., ''Self-Optimizing Memory Controllers: A Reinforcement Learning Approach,'' *Proc. 35th Ann. Int'l Symp. Computer Architecture* (ISCA 08), IEEE CS Press, 2008, pp. 39-50.
13. O. Mutlu and T. Moscibroda, ''Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors,'' *Proc. 40th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS Press, 2007, pp. 146-160.
14. O. Mutlu and T. Moscibroda, ''Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems,'' *Proc. 35th Ann. Int'l Symp. Computer Architecture* (ISCA 08), IEEE CS Press, 2008, pp. 63-74.
15. Y. Kim et al., ''ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers,'' *Proc. IEEE 16th Int'l Symp. High Performance Computer Architecture* (HPCA 10), IEEE Press, 2010, doi:10.1109/HPCA.2010.5416658.
16. 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.
17. C.J. Lee et al., ''Prefetch-Aware DRAM Controllers,'' *Proc. 41st Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS Press, 2008, pp. 200-209.

buffer called the row buffer. If a subsequent memory request accesses the same row that's in the row buffer (called a row-buffer hit), it can be serviced much more quickly. A thread's row-buffer locality is the row buffer's average hit rate across all banks.

Latency- versus bandwidth-sensitive threads

From a memory intensity perspective, we classify threads as either latency sensitive or bandwidth sensitive. Latency-sensitive threads spend most of their time at the processor and issue memory requests sparsely. Although the number of generated memory requests is low, latency-sensitive threads are sensitive to the memory subsystem's latency; every additional cycle spent waiting on memory is a wasted cycle that could have been spent on computation. In contrast, bandwidth-sensitive threads experience frequent cache misses and thus spend a lot of time waiting on pending memory requests. Therefore, the memory subsystem's throughput greatly affects their progress. Even if a memory request is quickly serviced, subsequent memory requests will once again stall execution.

Exploiting differences in memory access behavior

Threads are not created equal in terms of memory access behavior. By being aware of their differences, the scheduling algorithm can decouple the two objectives of system throughput and fairness to achieve them simultaneously.

System throughput: Prioritize light threads

Exploiting differences in threads' memory intensity lets us improve system throughput. Prior studies have demonstrated the system throughput benefits of prioritizing

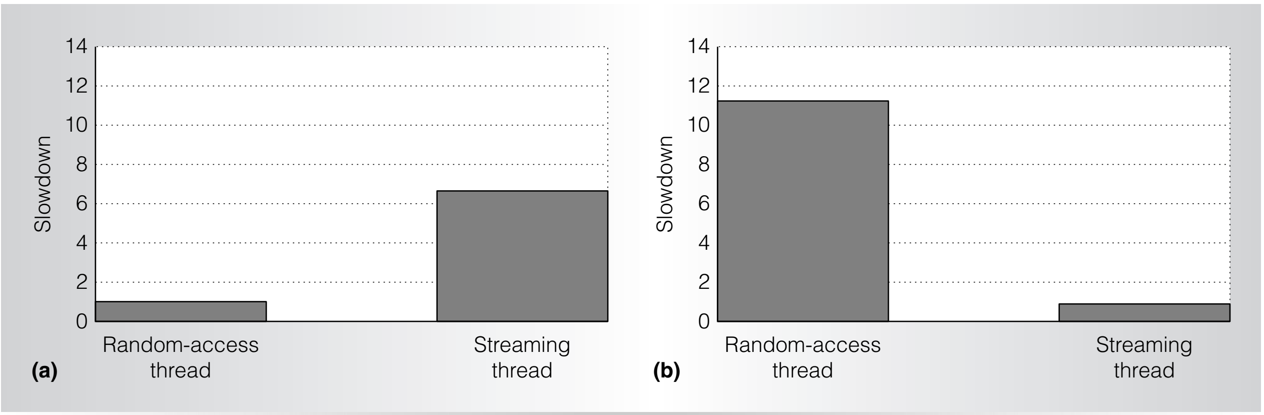

Figure 2. Effect of prioritization choices between the random-access thread and the streaming thread. Strictly prioritizing the random-access thread (a) vs. the streaming thread (b).

Table 1. Two examples of bandwidth-sensitive threads: random-access vs. streaming.

	Memory access behavior		
Thread type	**Memory intensity**	**Bank-level parallelism**	**Row-buffer locality**
Random-access	High (100 MPKI)	High (72.7% of maximum)	Low (0.1%)
Streaming	High (100 MPKI)	Low (0.3% of maximum)	High (99%)

*MPKI: misses per thousand instructions.

"light" (that is, latency-sensitive) threads over "heavy" (that is, bandwidth-intensive) threads.[4,5,7] Latency-sensitive threads only seldom generate memory requests and have greater potential for making fast progress in the processor. Therefore, to maximize system throughput, a memory scheduling algorithm should prioritize such threads and shield them from the heavy threads' interference. Doing so doesn't degrade fairness, because light threads rarely interfere with heavy threads.

Fairness: Minimize interference among heavy threads

We've observed that unfairness problems usually stem from interference among bandwidth-sensitive threads. Their numerous memory requests conflict with one another and destroy each thread's parallelism and locality. However, depending on their bank-level parallelism and row-buffer locality, memory-intensive threads have varying susceptibility to interference.

To illustrate this point, we ran experiments with two bandwidth-sensitive threads that we specifically constructed to have the same memory intensity but different bank-level parallelism and row-buffer locality. As Table 1 shows, the random-access thread has low row-buffer locality and high bank-level parallelism, whereas the streaming thread has low bank-level parallelism and high row-buffer locality.

Which of the two threads is more prone to large slowdowns when run together? Figure 2 shows the slowdown these two threads experience for two scheduling policies: one which strictly prioritizes the random-access thread over the streaming thread, and one which strictly prioritizes the streaming thread over the random-access thread. Clearly, as Figure 2 shows, the random-access thread is more susceptible to being slowed down because it experiences a slowdown of more than 11× when it's deprioritized, which is significantly greater than the streaming thread's slowdown when it's deprioritized.

The random-access thread is more susceptible to slowdown for two reasons. First, the streaming thread generates a steady stream of requests to a bank at a given time, leading to a temporary denial of service to any thread that accesses the same bank. Second, a thread with high bank-level parallelism is more susceptible to memory interference from another thread, because a bank conflict leads to the loss of bank-level parallelism, thus leading to serialization of otherwise parallel requests. Therefore, all else being the same, a scheduling algorithm should favor the thread with higher bank-level parallelism when distributing the memory bandwidth among bandwidth-sensitive threads. We used this insight to develop a new memory scheduling algorithm that intelligently prioritizes between bandwidth-sensitive threads.

Thread Cluster Memory scheduler

Our algorithm comprises three main components: clustering threads, prioritizing the latency-sensitive cluster, and employing different scheduling policies for different clusters.

First, to accommodate the disparate memory needs of concurrently executing threads sharing memory, TCM dynamically groups threads into two clusters on the basis of their memory intensity: a latency-sensitive cluster containing lower-memory-intensity threads and a bandwidth-sensitive cluster containing higher-memory-intensity threads. By employing different scheduling policies within each cluster, TCM can decouple the system throughput and fairness goals and optimize for each one separately.

Second, TCM always strictly prioritizes memory requests from the latency-sensitive cluster's threads over requests from the bandwidth-sensitive cluster's threads. As we explained earlier, prioritizing latency-sensitive threads increases overall system throughput because they have greater potential for making progress. Servicing memory requests from such light threads lets them continue with their computation.

To ensure sufficient bandwidth is left for the bandwidth-sensitive cluster, TCM limits the number of threads placed in the latency-sensitive cluster, such that they consume only a small fraction of the total memory bandwidth.

Third, to achieve high system throughput and to minimize unfairness, TCM employs a different scheduling policy for each cluster. The policy for the latency-sensitive cluster is geared toward high performance and low latency, because that cluster's threads have the greatest potential for making fast progress if their memory requests are serviced promptly. By contrast, the policy for the bandwidth-sensitive cluster is geared toward maximizing fairness, because that cluster's threads have heavy memory bandwidth demand and are susceptible to detrimental slowdowns if not given a sufficient share of the memory bandwidth.

Grouping threads into two clusters

TCM periodically ranks all threads according to their memory intensity at fixed-length time intervals called quanta. It places the least memory-intensive threads in the latency-sensitive cluster and the remaining threads in the bandwidth-sensitive cluster. Throughout each quantum, TCM monitors each thread's *memory bandwidth usage* in terms of the memory service time it has received; calculated across all banks in the memory subsystem, TCM defines a thread's memory service time as the number of cycles that the banks were kept busy servicing the thread's requests. The total memory bandwidth usage is the sum of each thread's memory bandwidth usage.

TCM groups the threads into two clusters at the beginning of every quantum using a parameter called ClusterThresh to specify the amount of bandwidth the latency-sensitive cluster consumes (as a fraction of the previous quantum's total memory bandwidth usage). Our experimental results show that for a system with N threads, a ClusterThresh value ranging from $2/N$ to $6/N$—that is, forming the latency-sensitive cluster such that it consumes $2/N$ to $6/N$ of the total memory bandwidth usage can provide a smooth transition between different performance-fairness trade-off points.

TCM groups threads into clusters in a synchronized manner across all memory controllers to better exploit bank-level parallelism.[4,5,8] To agree on the same thread clustering, the memory controllers exchange information at the end of every quantum, as has been done in previous work.[5] We set our time quantum's length to 1 million cycles, which is short enough to detect phase changes in the threads' memory behavior and long enough to minimize the communication overhead of synchronizing multiple memory controllers. Our original paper shows the pseudocode for the thread-clustering algorithm.[6]

Latency-sensitive cluster: Maximize system throughput

Within the latency-sensitive cluster, TCM enforces a strict priority, with the least memory-intensive thread receiving the highest priority. Such a thread is more likely to make large contributions to overall system throughput.

Bandwidth-sensitive cluster: Fairly sharing the memory

Bandwidth-sensitive threads should fairly share memory bandwidth to ensure no single thread is disproportionately slowed down. To achieve this, we must periodically shuffle the bandwidth-sensitive cluster's thread priority order. As we mentioned earlier, to preserve bank-level parallelism, this shuffling must occur in a synchronized manner across all memory banks, such that at any point in time all banks agree on a global thread priority order.

The problem with a round-robin approach. Shuffling the priority order in a round-robin fashion among bandwidth-sensitive threads seems like a simple solution, but our experiments revealed two problems. First, a round-robin shuffling algorithm is oblivious to inter-thread interference: it doesn't know which threads are more likely to slow down others. The second issue is more subtle and is tied to the way memory banks handle thread priorities. When choosing which memory request to service next, each bank first considers the requests from the highest-priority thread according to the current priority order. If that thread has no requests, the bank then considers the next-highest-priority thread, and so on. Therefore, a thread doesn't have to be at the top priority position to get some of its requests serviced. In other words, memory service leaks from highest-priority levels to lower ones.

This memory service leakage is the second reason the simple round-robin algorithm performs poorly. In particular, the problem with a round-robin approach is that a thread always maintains its relative position with respect to other threads. This means that fortunate threads scheduled behind leaky threads will consistently receive more service than other threads scheduled behind nonleaky threads, resulting in unfairness. This problem becomes more evident when we consider the threads' different memory access behavior. For instance, a streaming thread that exhibits high row-buffer locality and low bank-level parallelism will severely leak memory service time at all memory banks except for the single bank it's currently accessing.

Thread niceness and insertion shuffle. To alleviate the problems stemming from memory service leakage and minimize inter-thread interference, TCM employs a new shuffling algorithm, called *insertion shuffle*. (We derived the name from its similarity to the insertion sort algorithm. Each intermediate state during an insertion sort corresponds to a permutation in insertion shuffle.) Insertion shuffle reduces memory interference and increases fairness by exploiting heterogeneity in the bank-level parallelism and row-buffer locality among different threads.

We introduce a new metric, called *niceness*, which captures a thread's propensity to cause interference and its susceptibility to interference. A thread with high row-buffer locality is likely to make consecutive accesses to a few banks and thus make them congested. Under such circumstances, another thread with high bank-level parallelism becomes vulnerable to memory interference because it's subject to transient high loads at any of the many banks it's concurrently accessing. So, a thread with high bank-level parallelism is fragile (more likely to be interfered with), whereas one with high row-buffer locality is hostile (more likely to cause interference to others). We define a thread's niceness to increase with

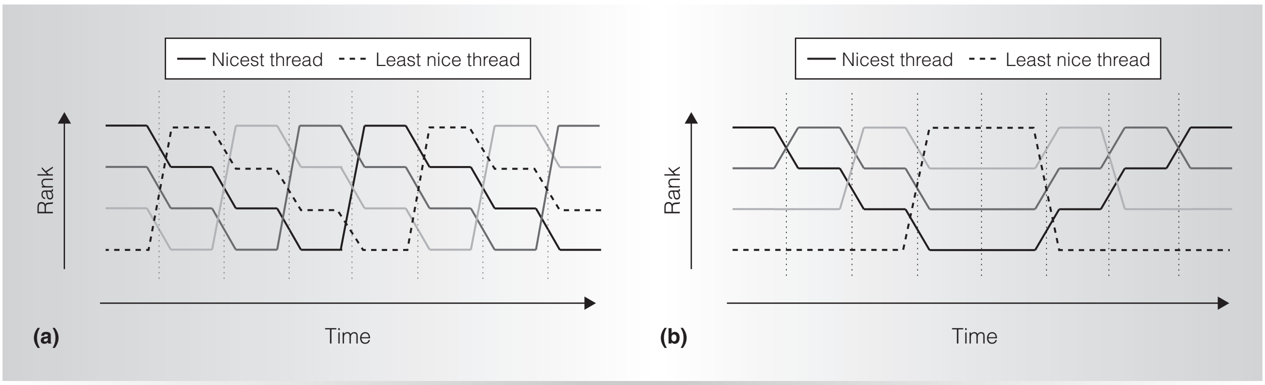

Figure 3. Visualizing two shuffling algorithms: round-robin shuffle (a) and insertion shuffle (b).

a thread's relative fragility and to decrease with its relative hostility. Within the bandwidth-sensitive cluster, if *thread*$_i$ has the (b_i)th highest bank-level parallelism and the (r_i)th highest row-buffer locality, we formally define its niceness as follows:

$$Niceness_i \equiv b_i - r_i.$$

Every quantum, TCM sorts threads by their niceness value to yield a ranking, where the nicest thread receives the highest rank. Subsequently, every ShuffleInterval cycles, the insertion shuffle algorithm perturbs this ranking. This perturbation is done in such a way that reduces the time during which the least nice threads are prioritized over the nicest threads, ultimately resulting in less interference. Figure 3 shows successive permutations of the priority order for both the round-robin and insertion shuffle algorithms for four threads. In the case of insertion shuffle, the least nice thread spends most of its time at the lowest priority position, whereas the remaining nicer threads are at higher priorities and thus can synergistically leak their memory service time among themselves. (Algorithm 2 in our original paper shows the pseudocode for the insertion shuffle algorithm.[6])

Handling threads with similar behavior. If the bandwidth-sensitive cluster consists of homogeneous threads with similar memory behavior, TCM disables insertion shuffle and falls back to random shuffle to prevent unfair treatment of threads because of marginal differences in niceness values. To do this, TCM inspects whether threads exhibit sufficient diversity in memory access behavior before applying insertion shuffling. Our original paper explains exactly how TCM does this, describes random shuffling, and evaluates the performance and fairness benefits of random shuffling.[6]

1. **Highest-rank first:** *Requests from higher-ranked threads have priority over others.*
   - Latency-sensitive threads have higher rank than bandwidth-sensitive threads.
   - Within the latency-sensitive cluster, lower-MPKI threads have higher rank than others.
   - Within the bandwidth-sensitive cluster, insertion shuffling determines rank order.
2. **Row-hit first:** *Row buffer hit requests have priority over others.*
3. **Oldest first:** *Older requests have priority over others.*

Figure 4. TCM request prioritization rules.

Summary: TCM prioritization rules

Figure 4 summarizes how TCM prioritizes threads' memory requests. When requests from multiple threads compete to access a bank, the higher-ranked thread (where ranking depends on the thread cluster) has priority. If two requests have the same thread rank, TCM favors row-buffer hit requests. All else being equal, TCM favors older requests.

System software support

TCM supports thread weights (or priorities) that the operating system assigns, such

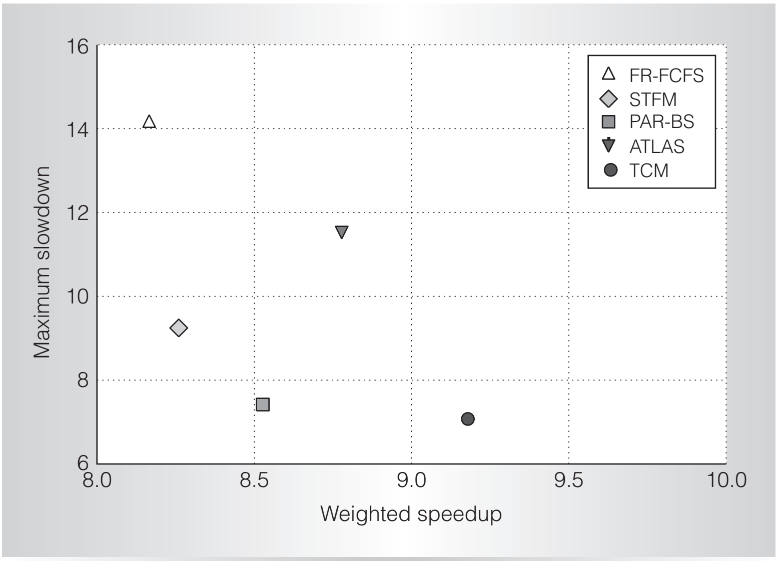

Figure 5. Performance and fairness of TCM vs. other algorithms across all 96 workloads.

that threads with larger weights have higher priority in the memory controller. However, given a thread with a large thread weight, blindly prioritizing it can potentially lead to the destruction of all other threads' performance.

TCM solves this problem by honoring thread weights within the context of thread clusters. For example, even if the operating system assigns a large weight to a bandwidth-sensitive thread, TCM doesn't prioritize it over the latency-sensitive threads, because doing so would significantly degrade all latency-sensitive threads' performance without significantly improving the higher-weight thread's performance. Thus, TCM prioritizes threads only within their respective cluster: within the bandwidth-sensitive cluster, TCM employs a weighted shuffling that takes into account thread weight; within the latency-sensitive cluster, TCM interprets a thread's memory intensity as its MPKI scaled down by its weight.

Fairness and performance trade-off knob

ClusterThresh is exposed to the system software such that the software can adjust its value to achieve the desired balance between throughput and fairness, as we will soon show.

Evaluation

We evaluated TCM using an in-house cycle-level x86 CMP simulator. We modeled the memory subsystem using DDR2 (double data rate) timing parameters,[10] which we verified using DRAMSim[11] and measurements from real hardware. Our main results are from a 24-core CMP with four memory controllers, where each core has a 512-Kbyte private level-two (L2) cache. Our original paper explains our model and methodology in detail and evaluates the sensitivity of our results to changes in system and TCM parameters.[6]

We used the SPEC CPU2006 benchmarks for evaluation. We formed 96 multiprogrammed workloads of varying memory intensity, which we simulated for 100 million cycles.

We measured system throughput using weighted speedup,[12] and we measured fairness using maximum slowdown;[13] these are common metrics for multiprogrammed workload evaluation. We also report harmonic speedup,[14] which measures a balance of fairness and throughput.

Evaluation: Best of both throughput and fairness

We compared TCM's performance against four previous memory scheduling algorithms: FR-FCFS (first-ready, first-come, first-serve),[9] STFM (Stall-Time Fair Memory) scheduler,[3] PAR-BS[4] (parallelism-aware batch scheduling—the best existing algorithm for fairness), and ATLAS[5] (Adaptive Per-Thread Least Attained Service—the best existing algorithm for system throughput). Figure 5 shows where each scheduling algorithm lies with regard to fairness and system throughput, averaged across all 96 workloads of varying memory intensity. The lower right part of the figure corresponds to better fairness (lower maximum slowdown) and better system throughput (higher weighted speedup). TCM improves system throughput and reduces maximum slowdown by 4.6 percent and 38.6 percent compared to ATLAS (previous work providing the best system throughput) and 7.6 percent and 4.6 percent compared to PAR-BS (previous work providing the best fairness). We conclude that TCM achieves the best system throughput and the best fairness, outperforming every algorithm with regard to

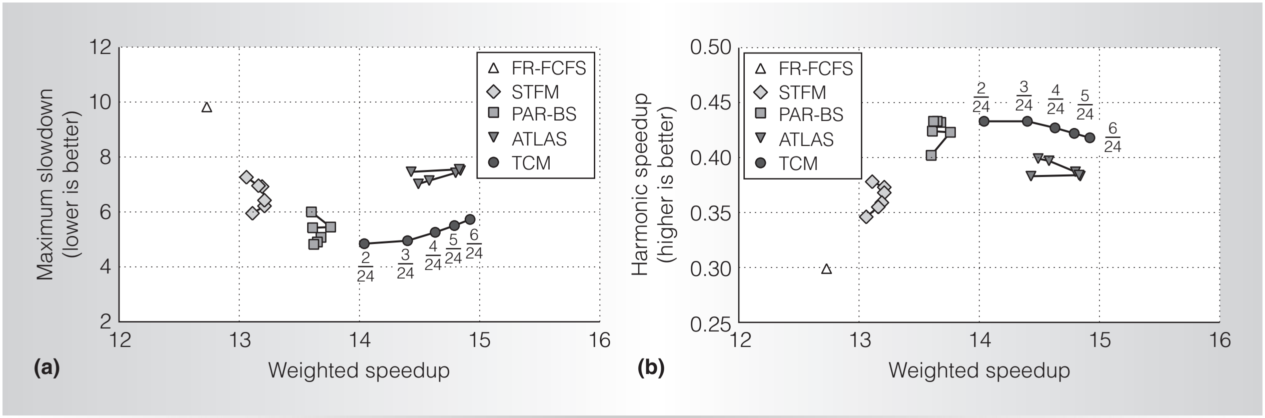

Figure 6. Performance and fairness trade off as algorithmic parameters are varied (for 32 workloads): maximum slowdown (a) and harmonic speedup (b) vs. weighted speedup.

weighted speedup, maximum slowdown, and harmonic speedup (shown in Figure 6b).

Note that TCM's performance as shown here is for just a single operating point. As the next section shows, TCM provides the flexibility to smoothly transition along a range of performance-fairness trade-off points.

Evaluation: Trading off between performance and fairness

To study each memory scheduler's robustness, as well as the ability to adapt to different performance and fairness goals, we varied each scheduler's most salient configuration parameters, as described in our original paper.[6] For TCM, we varied the ClusterThresh from 2/24 to 6/24 in 1/24-increments. Figure 6 shows the performance and fairness results. The lower right part of Figure 6a and the upper right part of Figure 6b correspond to better operating points for both performance and fairness.

In contrast to previous memory scheduling algorithms, TCM exposes a smooth continuum between system throughput and fairness. By adjusting the clustering threshold between latency- and bandwidth-sensitive clusters, TCM can trade system throughput and fairness for one another. As a result, TCM has a wide range of balanced operating points that provide both high system throughput and fairness. No previous algorithm provides the same degree of flexibility as TCM. For example, ATLAS always remains biased toward system throughput (that is, its maximum slowdown changes little). Similarly, PAR-BS remains biased toward fairness (that is, its weighted speedup changes little). TCM provides an effective knob for trading off between fairness and performance, enabling operation at various desirable operating points, depending on system requirements.

Further analyses

Our original paper shows how TCM supports thread weights assigned by the system software, evaluates TCM using different workload compositions where the fraction of memory-intensive threads varies, and analyzes different shuffling algorithms' effect on fairness.[6] We also provide sensitivity results where the system configurations are varied (including 512-Kbyte to 2-Mbyte caches, 4 to 32 cores, and 1 to 16 memory controllers), showing that TCM provides the best system throughput and fairness across all configurations.

We've developed a new way of thinking about memory controller design: treating thread groups differently instead of having a single unified policy for all threads. We've also developed basic principles for fair and high-performance management of memory bandwidth. As we describe below, these principles would likely apply to other shared hardware resources, hopefully inspiring innovative research beyond memory controllers.

First, thread clusters can be a building block for shared hardware resource management. Our paper introduces the idea of clustering threads into multiple groups according to their resource usage patterns and treating each group differently to maximize system performance and fairness. Architects can use the notion of thread clustering as a building block for managing not only memory bandwidth, but also other shared resources, such as cache bandwidth and capacity, interconnect bandwidth, I/O bandwidth, and system energy and power.

Second, priority shuffling provides a substrate for fair service and starvation avoidance. Our paper introduces the idea of thread priority shuffling in hardware to provide memory-intensive threads with fair service. We show that even simple shuffling algorithms can alleviate the unfairness problem, and we also introduce more elaborate shuffling algorithms that perturb thread priorities in a coordinated manner, such that inter-thread interference is minimized. We envision these same mechanisms being used to provide fair access to other on-chip and off-chip resources.

Third, different treatment of latency-sensitive versus bandwidth-sensitive threads can benefit the management of other system resources. Our scheduler is the first to distinguish between latency-sensitive and bandwidth-sensitive threads and employ different scheduling policies based on this distinction. To maximize resource efficiency and maintain fairness, we prioritize latency-sensitive threads (which use little bandwidth) over bandwidth-sensitive ones and divide the remaining unused bandwidth fairly between the bandwidth-sensitive threads. Architects can apply this key idea to efficiently manage other shared system resources.

Fourth, our paper demonstrates one way of quantifying a thread's propensity to cause interference and its susceptibility to interference, which we call niceness, in the memory system. The notion of niceness can inspire other research that develops similar or better metrics useful for the management of memory controllers and other resources.

Fifth, TCM is the first memory scheduler that enables a robust, smooth, and flexible trade-off between fairness and performance, which allows it to be tailored to the specific fairness and performance requirements of different systems and workloads. As quality-of-service requirements and performance demands continue to vary widely across applications, especially in consolidated systems that run a large number of workloads (such as data centers and cloud computing), we expect different shared resources will be designed to provide similar configurability and fairness-performance trade-off to allow the system software to select the best operation points. MICRO

Acknowledgments

We thank the anonymous reviewers and members of the Computer Architecture Lab at Carnegie Mellon University for their valuable feedback. Yoongu Kim is supported by a PhD fellowship from the Korea Foundation for Advanced Studies. We gratefully acknowledge the support of the Gigascale Systems Research Center, Intel, and CyLab. This research was partially supported by a National Science Foundation Career Award (CCF-0953246).

References

1. T. Moscibroda and O. Mutlu, ''Memory Performance Attacks: Denial of Memory Service in Multi-core Systems,'' *Proc. 16th USENIX Security Symp.* (SS 07), Usenix Assoc., 2007, pp. 257-274.
2. K.J. Nesbit et al., ''Fair Queuing Memory Systems,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS Press, 2006, pp. 208-222.
3. O. Mutlu and T. Moscibroda, ''Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors,'' *Proc. 40th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS Press, 2007, pp. 146-160.
4. O. Mutlu and T. Moscibroda, ''Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems,'' *Proc. 35th Ann. Int'l Symp. Computer Architecture* (ISCA 08), IEEE CS Press, 2008, pp. 63-74.
5. Y. Kim et al., ''ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers,'' *Proc. IEEE*

16th Int'l Symp. High Performance Computer Architecture (HPCA 10), IEEE Press, 2010, doi:10.1109/HPCA.2010.5416658.
6. Y. Kim et al., ''Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior,'' *Proc. 43rd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS Press, 2010.
7. H. Zheng et al., ''Memory Access Scheduling Schemes for Systems with Multi-core Processors,'' *Proc. 37th Int'l Conf. Parallel Processing* (ICPP 08), IEEE CS Press, 2008, pp. 406-413.
8. T. Moscibroda and O. Mutlu, ''Distributed Order Scheduling and Its Application to Multi-core DRAM Controllers,'' *Proc. 27th ACM Symp. Principles of Distributed Computing* (PODC 08), ACM Press, 2008, pp. 365-374.
9. S. Rixner et al., ''Memory Access Scheduling,'' *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 00), ACM Press, 2000, pp. 128-138.
10. ''1Gb DDR2 SDRAM Component: MT47H128M8HQ-25,'' data sheet, Micron, 2010.
11. D. Wang et al., ''DRAMsim: A Memory System Simulator,'' *SIGARCH Computer Architecture News,* vol. 33, no. 4, 2005, pp. 100-107.
12. A. Snavely and D.M. Tullsen, ''Symbiotic Job Scheduling for a Simultaneous Multithreading Processor,'' *Proc. 9th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS IX), ACM Press, 2000, pp. 234-244.
13. R. Das et al., ''Application-Aware Prioritization Mechanisms for On-Chip Networks,'' *Proc. 42nd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* ACM Press, 2009, pp. 280-291.
14. K. Luo, J. Gummaraju, and M. Franklin, ''Balancing Throughput and Fairness in SMT Processors,'' *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software,* IEEE CS Press, 2001, pp. 164-171.

Yoongu Kim is a PhD student in the Department of Electrical and Computer Engineering at Carnegie Mellon University. His research interests include scheduling problems that arise in computer architecture. He has a BS in electrical engineering from Seoul National University.

Michael Papamichael is a PhD candidate in the Department of Computer Science at Carnegie Mellon University. His research interests include computer architecture, particularly uncore modeling and FPGA (field-programmable gate array)-accelerated simulation and instrumentation for multiprocessor architectures. He has an MS in computer science from the University of Crete.

Onur Mutlu is an assistant professor in the Department of Electrical and Computer Engineering at Carnegie Mellon University. His research interests include computer architecture and systems. He has a PhD in electrical and computer engineering from the University of Texas at Austin. He's a member of IEEE and the ACM.

Mor Harchol-Balter is an associate professor in the Department of Computer Science at Carnegie Mellon University. Her research interests include designing new resource allocation policies, power management policies, and scheduling policies for server farms and distributed systems. She has a PhD in computer science from the University of California, Berkeley. She's a member of IEEE, the ACM, and Informs (the Institute for Operations Research and the Management Sciences).

Direct questions and comments to Yoongu Kim at Carnegie Mellon Univ., 5000 Forbes Ave., Hamerschlag Hall, A-313, Pittsburgh, PA 15213; yoonguk@ece.cmu.edu.