2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture

Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior

Yoongu Kim yoonguk@ece.cmu.edu
Michael Papamichael papamix@cs.cmu.edu
Onur Mutlu onur@cmu.edu
Mor Harchol-Balter harchol@cs.cmu.edu

Carnegie Mellon University

Abstract

In a modern chip-multiprocessor system, memory is a shared resource among multiple concurrently executing threads. The memory scheduling algorithm should resolve memory contention by arbitrating memory access in such a way that competing threads progress at a relatively fast and even pace, resulting in high system throughput and fairness. Previously proposed memory scheduling algorithms are predominantly optimized for only one of these objectives: no scheduling algorithm provides the best system throughput and best fairness at the same time.

This paper presents a new memory scheduling algorithm that addresses system throughput and fairness separately with the goal of achieving the best of both. The main idea is to divide threads into two separate clusters and employ different memory request scheduling policies in each cluster. Our proposal, Thread Cluster Memory scheduling (TCM), dynamically groups threads with similar memory access behavior into either the latency-sensitive (memory-non-intensive) or the bandwidth-sensitive (memory-intensive) cluster. TCM introduces three major ideas for prioritization: 1) we prioritize the latency-sensitive cluster over the bandwidth-sensitive cluster to improve system throughput; 2) we introduce a "niceness" metric that captures a thread's propensity to interfere with other threads; 3) we use niceness to periodically shuffle the priority order of the threads in the bandwidth-sensitive cluster to provide fair access to each thread in a way that reduces inter-thread interference. On the one hand, prioritizing memory-non-intensive threads significantly improves system throughput without degrading fairness, because such "light" threads only use a small fraction of the total available memory bandwidth. On the other hand, shuffling the priority order of memory-intensive threads improves fairness because it ensures no thread is disproportionately slowed down or starved.

We evaluate TCM on a wide variety of multiprogrammed workloads and compare its performance to four previously proposed scheduling algorithms, finding that TCM achieves both the best system throughput and fairness. Averaged over 96 workloads on a 24-core system with 4 memory channels, TCM improves system throughput and reduces maximum slowdown by 4.6%/38.6% compared to ATLAS (previous work providing the best system throughput) and 7.6%/4.6% compared to PAR-BS (previous work providing the best fairness).

1. Introduction

High latency of off-chip memory accesses has long been a critical bottleneck in thread performance. This has been further exacerbated 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 be slowed down compared to when it has the memory entirely to itself. 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 underutilization and potentially thread starvation [11].

The effectiveness of a memory scheduling algorithm is commonly evaluated based on two objectives: *fairness* [16, 13, 14] and *system throughput* [14, 13, 5]. On the one hand, no single thread should be disproportionately slowed down, while on the other hand, the throughput of the overall system should remain high. Intuitively, fairness and high system throughput ensure that all threads progress at a relatively even and fast pace.

Previously proposed memory scheduling algorithms are biased towards either fairness or system throughput. In one extreme, by trying to equalize the amount of bandwidth each thread receives, some notion of fairness can be achieved, but at a large expense to system throughput [16]. In the opposite extreme, by strictly prioritizing certain favorable (memory-non-intensive) threads over all other threads, system throughput can be increased, but at a large expense to fairness [5]. As a result, such relatively single-faceted approaches cannot provide the highest fairness and system throughput at the same time.

Our new scheduling algorithm exploits differences in threads' memory access behavior to optimize for both system throughput and fairness, based on several key observations. First, prior studies have demonstrated the system throughput benefits of prioritizing "light" (i.e., memory-non-intensive) threads over "heavy" (i.e., memory-intensive) threads [5, 14, 26]. Memory-non-intensive threads only seldom generate memory requests and have greater potential for making fast progress in the processor. Therefore, to maximize system throughput, it is clear that a memory scheduling algorithm should prioritize memory-non-intensive threads. Doing so also does not degrade fairness because light threads rarely interfere with heavy threads.

Second, we observe that unfairness problems usually stem from interference among memory-intensive threads. The most memory-intensive threads become vulnerable to starvation when less memory-intensive threads are statically prioritized over them (e.g., by forming a priority order based on a metric that corresponds to memory intensity, as done in [5]). As a result, the most memory-intensive threads can experience disproportionately large slowdowns which lead to unfairness.

Third, we observe that periodically shuffling the priority order among memory-intensive threads allows each thread a chance to gain prioritized access to the memory banks, thereby reducing unfairness. However, how to best perform the shuffling is not obvious. We find that shuffling in a symmetric manner, which gives each thread equal possibility to be at all priority levels, causes unfairness because not all threads are equal in terms of their propensity to interfere with others; some threads are more likely to slow down other threads. Hence, thread priority order should be shuffled such that threads with higher propensity to interfere with others have a smaller chance of being at higher priority.

Finally, as previous work has shown, it is desirable that scheduling decisions are made in a synchronized manner across all banks [5, 14, 12], so that concurrent requests of each thread are serviced in parallel, without being serialized due to interference from other threads.

1072-4451/10 $26.00 © 2010 IEEE
DOI 10.1109/MICRO.2010.51

Overview of Mechanism. Based on the above observations, we propose Thread Cluster Memory scheduling (TCM), an algorithm that detects and exploits differences in memory access behavior across threads. TCM dynamically groups threads into two clusters based on their memory intensity: a *latency-sensitive* cluster comprising memory-non-intensive threads and a *bandwidth-sensitive* cluster comprising memory-intensive threads.

Threads in the latency-sensitive cluster are always prioritized over threads in the bandwidth-sensitive cluster to maximize system throughput. To ensure that no thread is disproportionately slowed down, TCM periodically shuffles the priority order among threads in the bandwidth-sensitive cluster.

TCM's intelligent shuffling algorithm ensures that threads that are likely to slow down others spend less time at higher priority levels, thereby reducing the probability of large slowdowns. By having a sufficiently long shuffling period and performing shuffling in a synchronized manner across all banks, threads are able to exploit both row-buffer locality and bank-level parallelism. Combined, these mechanisms allow TCM to outperform any previously proposed memory scheduler in terms of both fairness and system throughput.

Contributions. In this paper, we make the following contributions:

- We introduce the notion of thread clusters for memory scheduling, which are groups of threads with similar memory intensity. We show that by dynamically dividing threads into two separate clusters (latency-sensitive and bandwidth-sensitive), a memory scheduling algorithm can satisfy the disparate memory needs of both clusters simultaneously. We propose a simple, dynamic clustering algorithm that serves this purpose.
- We show that threads in different clusters should be treated differently to maximize both system throughput and fairness. We observe that prioritizing latency-sensitive threads leads to high system throughput, while periodically perturbing the prioritization order among bandwidth-sensitive threads is critical for fairness.
- We propose a new metric for characterizing a thread's memory access behavior, called *niceness*, which reflects a thread's susceptibility to interference from other threads. We observe that threads with high row-buffer locality are less nice to others, whereas threads with high bank-level parallelism are nicer, and monitor these metrics to compute thread niceness.
- Based on the proposed notion of niceness, we introduce a shuffling algorithm, called *insertion shuffle*, which periodically perturbs the priority ordering of threads in the bandwidth-sensitive cluster in a way that minimizes inter-thread interference by ensuring nicer threads are prioritized more often over others. This reduces unfairness within the bandwidth-sensitive cluster.
- We compare TCM against four previously proposed memory scheduling algorithms and show that it outperforms all existing memory schedulers in terms of both fairness (maximum slowdown) and system throughput (weighted speedup) for a 24-core system where the results are averaged across 96 workloads of varying levels of memory intensity. Compared to ATLAS [5], the best previous algorithm in terms of system throughput, TCM improves system throughput and reduces maximum slowdown by 4.6%/38.6%. Compared to PAR-BS [14], the best previous algorithm in terms of fairness, TCM improves system throughput and reduces maximum slowdown by 7.6%/4.6%.
- We show that TCM is configurable and can be tuned to smoothly and robustly transition between fairness and system throughput goals, something which previous schedulers, optimized for a single goal, are unable to do.

2. Background and Motivation

2.1. Defining Memory Access Behavior

TCM defines a thread's *memory access behavior* using three components as identified by previous work: memory intensity [5], bank-level parallelism [14], and row-buffer locality [19].

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 or MPKI.

Memory is not a monolithic resource but consists of multiple memory banks that can be accessed in parallel. It is the existence of multiple memory banks and their particular internal organization that give rise to bank-level parallelism and row-buffer locality, respectively.

Bank-level parallelism (BLP) of a thread is the average number of banks to which there are 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 be equal to the total number of banks in the memory subsystem.

A memory bank is internally organized as a two-dimensional structure consisting 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 in a small internal memory called the row-buffer. If a subsequent memory request accesses the same row that is in the row-buffer, it can be serviced much more quickly; this is called a row-buffer hit. The *row-buffer locality* (RBL) of a thread is the average hit-rate of the row-buffer across all banks.

2.2. Latency- vs. Bandwidth-Sensitive Threads

From a memory intensity perspective, we classify threads into one of two distinct groups: latency-sensitive or bandwidth-sensitive. *Latency-sensitive* threads spend most of their time at the processor and issue memory requests sparsely. Even though the number of generated memory requests is low, the performance of latency-sensitive threads is very sensitive to the latency of the memory subsystem; every additional cycle spent waiting on memory is a wasted cycle that could have been spent on computation. *Bandwidth-sensitive* threads experience frequent cache misses and thus spend a large portion of their time waiting on pending memory requests. Therefore, their rate of progress is greatly affected by the throughput of the memory subsystem. Even if a memory request is quickly serviced, subsequent memory requests will once again stall execution.

2.3. Our Goal: Best of both System Throughput and Fairness

A multiprogrammed workload can consist of a diverse mix of threads including those which are latency-sensitive or bandwidth-sensitive. A well-designed memory scheduling algorithm should strive to maximize overall system throughput, but at the same time bound the worst case slowdown experienced by any one of the threads. These two goals are often conflicting and form a trade-off between system throughput and fairness.

Intuitively, latency-sensitive threads (which cannot tolerate high memory latencies) should be prioritized over others to improve system throughput, while bandwidth-sensitive threads (which can tolerate high memory latencies) should be scheduled in a fairness-aware manner to limit the amount of slowdown they experience. Applying a single memory scheduling policy across all threads, an approach commonly taken by existing memory scheduling algorithms, cannot address the disparate needs of different threads. Therefore, existing algorithms are unable to decouple the system throughput and fairness goals and achieve them simultaneously.

To illustrate this problem, Figure 1 compares the unfairness (maximum thread slowdown compared to when run alone on the system) and system throughput (weighted speedup) of four state-of-the-art memory scheduling algorithms (FR-FCFS [19], STFM [13], PAR-BS [14], and ATLAS [5]) averaged over 96 workloads.[1]

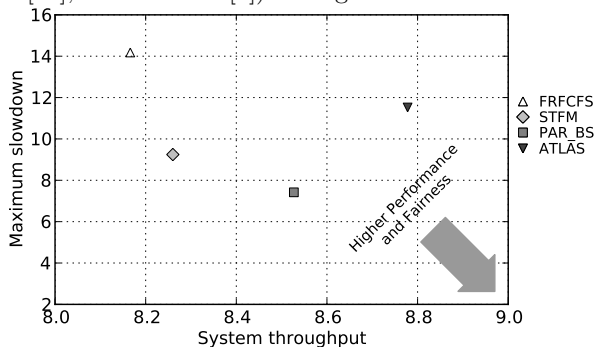


Figure 1. Performance and fairness of state-of-the-art scheduling algorithms. Lower right corner is the ideal operation point.

An ideal memory scheduling algorithm would be placed towards the lower (better fairness) right (better system throughput) part of the plot in Figure 1. Unfortunately, no previous scheduling algorithm achieves the best fairness and the best system throughput at the same time. While PAR-BS provides the best fairness, it has 2.9% lower system throughput than the highest-performance algorithm, ATLAS. On the other hand, ATLAS provides the highest system throughput but its maximum slowdown is 55.3% higher than the most fair algorithm, PAR-BS. Hence, existing scheduling algorithms are good at either system throughput or fairness, but not both. **Our goal** in this paper is to design a memory scheduling algorithm that achieves the best of both worlds: highest system throughput and highest fairness at the same time.

[1] Our evaluation methodology and baseline system configuration are described in Section 6.

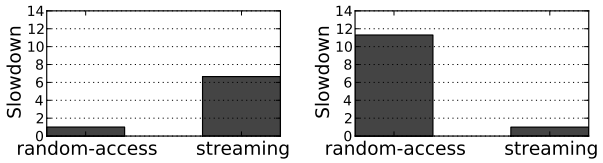
2.4. Varying Susceptibility of Bandwidth-Sensitive Threads to Interference

We motivate the importance of differentiating between threads' memory access behavior by showing that not all bandwidth-sensitive threads are equal in their vulnerability to interference. To illustrate this point, we ran experiments with two bandwidth-sensitive threads that were specifically constructed to have the same memory intensity, but very different bank-level parallelism and row-buffer locality. As shown in Table 1, the random-access thread has low row-buffer locality and high bank-level parallelism, while the streaming thread has low bank-level parallelism and high row-buffer locality.

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

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

Which of the two threads is more prone to large slowdowns when run together? Figure 2 shows the slowdown experienced by these two threads for two different scheduling policies: one where the random-access thread is strictly prioritized over the streaming thread and one where the streaming thread is strictly prioritized over the random-access thread. Clearly, as shown in Figure 2(b), the random-access thread is more susceptible to being slowed down since it experiences a slowdown of more than 11x when it is deprioritized, which is greater than the slowdown of the streaming thread when it is deprioritized.



(a) Strictly prioritizing random-access thread (b) Strictly prioritizing streaming thread

Figure 2. Effect of prioritization choices between the random-access thread and the streaming thread

This is due to two reasons. First, the streaming thread generates a steady stream of requests to a bank at a given time, leading to 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 since a bank conflict leads to the loss of bank-level parallelism, resulting in the 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 will use this insight to develop a new memory scheduling algorithm that intelligently prioritizes between bandwidth-sensitive threads.

3. Mechanism

3.1. Overview of TCM

Clustering Threads. To accommodate the disparate memory needs of concurrently executing threads sharing the memory, TCM dynamically groups threads into two clusters based on 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 is able to decouple the system throughput and fairness goals and optimize for each one separately.

Prioritizing the Latency-Sensitive Cluster. Memory requests from threads in the latency-sensitive cluster are always strictly prioritized over requests from threads in the bandwidth-sensitive cluster. As shown previously [5, 14, 26], prioritizing latency-sensitive threads (which access memory infrequently) increases overall system throughput, because they have greater potential for making progress. Servicing memory requests from such "light" threads allows them to continue with their computation. To avoid starvation issues and ensure sufficient bandwidth is left over 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.

Different Clusters, Different Policies. 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 towards high performance and low latency, since threads in that cluster 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 towards maximizing fairness, since threads in that cluster have heavy memory bandwidth demand and are susceptible to detrimental slowdowns if not given a sufficient share of the memory bandwidth.

Within the latency-sensitive cluster, TCM enforces a strict priority, with the least memory-intensive thread receiving the highest priority. Such a policy ensures that requests from threads spending most of their time at the processor (i.e., accessing memory infrequently), are always promptly serviced; this allows them to quickly resume their computation and ultimately make large contributions to overall system throughput.

Within the bandwidth-sensitive cluster, threads share the remaining memory bandwidth, so that no thread is disproportionately slowed down or, even worse, starved. TCM accomplishes this by periodically shuffling the priority ordering among the threads in the bandwidth-sensitive cluster. To minimize thread slowdown, TCM introduces a new shuffling algorithm, called *insertion shuffle*, that tries to reduce the amount of inter-thread interference and at the same time maximize row-buffer locality and bank-level parallelism. To monitor inter-thread interference, we introduce a new composite metric, called *niceness*, which captures both a thread's propensity to cause interference and its susceptibility to interference. TCM monitors the niceness values of threads and adapts its shuffling decisions to ensure that nice threads are more likely to receive higher priority. Niceness and the effects of shuffling algorithms for the bandwidth-sensitive cluster are discussed in Section 3.3.

3.2. Grouping Threads into Two Clusters

TCM periodically ranks all threads based on their memory intensity at fixed-length time intervals called quanta. The least memory-intensive threads are placed in the latency-sensitive cluster while the remaining threads are placed in the bandwidth-sensitive cluster. Throughout each quantum TCM monitors the *memory bandwidth usage* of each thread in terms of the memory service time it has received: summed across all banks in the memory subsystem, a thread's memory service time is defined to be the number of cycles that the banks were kept busy servicing its requests. The *total memory bandwidth usage* is defined to be the sum of each thread's memory bandwidth usage across all threads.

TCM groups threads into two clusters at the beginning of every quantum by using a parameter called *ClusterThresh* to specify the amount of bandwidth to be consumed by the latency-sensitive cluster (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, i.e., 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 good performance-fairness trade-off points. A thorough analysis of the effect of different *ClusterThresh* values is presented in Section 7.1.

Grouping of threads into clusters happens in a synchronized manner across all memory controllers to better exploit bank-level parallelism [5, 14]. In order for all memory controllers to agree upon the same thread clustering, they periodically exchange information, every quantum. The length of our time quantum is set to one million cycles, which, based on experimental results, is short enough to detect phase changes in the memory behavior of threads and long enough to minimize the communication overhead of synchronizing multiple memory controllers. Algorithm 1 shows the pseudocode for the thread clustering algorithm used by TCM.

3.3. 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, the thread priority order for the bandwidth-sensitive cluster needs to be periodically shuffled. As mentioned earlier, to preserve bank-level parallelism, this shuffling needs to happen 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 Round-Robin. Shuffling the priority order in a round-robin fashion among bandwidth-sensitive threads would appear to be a simple solution to this problem, but our experiments revealed two problems. The first problem is that a round-robin shuffling algorithm is oblivious to inter-thread interference: it is not aware of which threads are more likely

Algorithm 1 Clustering Algorithm

Initialization:
$LatencyCluster \leftarrow \varnothing$; $BandwidthCluster \leftarrow \varnothing$
$Unclassified \leftarrow \{thread_i : 1 \leq i \leq N_{threads}\}$
$SumBW \leftarrow 0$

Per-thread parameters:
· $MPKI_i$: Misses per kiloinstruction of $thread_i$
· $BWusage_i$: BW used by $thread_i$ during previous quantum

Clustering: (beginning of quantum)
$TotalBWusage \leftarrow \sum_i BWusage_i$
while $Unclassified \neq \varnothing$ **do**
  $j = \arg\min_i MPKI_i$ `//find thread with lowest MPKI`
  $SumBW \leftarrow SumBW + BWusage_j$
  if $SumBW \leq ClusterThresh \cdot TotalBWusage$ **then**
    $Unclassified \leftarrow Unclassified - \{thread_j\}$
    $LatencyCluster \leftarrow LatencyCluster \cup \{thread_j\}$
  else
    break
  end if
end while
$BandwidthCluster \leftarrow Unclassified$

Algorithm 2 Insertion Shuffling Algorithm

Definition:
· N: number of threads in bandwidth-sensitive cluster
· $threads[N]$: array of bandwidth-sensitive threads;
  we define a thread's rank as its position in the array
  (N^{th} position occupied by highest ranked thread)
· $incSort(i, j)$: sort subarray $threads[i..j]$ in inc. niceness
· $decSort(i, j)$: sort subarray $threads[i..j]$ in dec. niceness

Initialization: (beginning of quantum)
$incSort(1, N)$ `//nicest thread is highest ranked`

Shuffling: (throughout quantum)
while *true* **do**
  `//each iteration occurs every` *ShuffleInterval*
  for $i = N$ to 1 **do**
    $decSort(i, N)$
  end for
  `//each iteration occurs every` *ShuffleInterval*
  for $i = 1$ to N **do**
    $incSort(1, i)$
  end for
end while

to slow down others. The second problem 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, then the next highest priority thread is considered and so forth. As a result, a thread does not have to be necessarily 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. In fact, in our experiments we often encountered cases where memory service was "leaked" all the way to the fifth or sixth highest priority thread in a 24-thread system.

This memory service "leakage" effect is the second reason the simple round-robin algorithm performs poorly. In particular, the problem with round-robin is that a thread always maintains its relative position with respect to other threads. This means "lucky" threads scheduled behind "leaky" threads will consistently receive more service than other threads that are scheduled behind "non-leaky" threads, resulting in unfairness. This problem becomes more evident if one considers the different memory access behavior of threads. 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 is currently accessing.

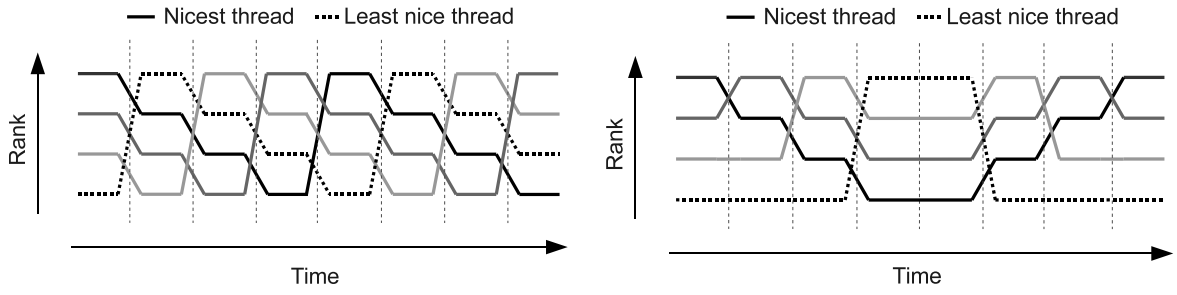
Thread Niceness and Insertion Shuffle. To alleviate the problems stemming from memory service "leakage" and to minimize inter-thread interference, TCM employs a new shuffling algorithm, called *insertion shuffle*[2], that reduces memory interference and increases fairness by exploiting heterogeneity in the bank-level parallelism and row-buffer locality among different threads.

We introduce a new metric, called *niceness*, that 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 small number of banks and cause them to be congested. Under such circumstances, another thread with high bank-level parallelism becomes vulnerable to memory interference since it is subject to transient high loads at any of the many banks it is concurrently accessing. Hence, a thread with high bank-level parallelism is fragile (more likely to be interfered by others), whereas one with high row-buffer locality is hostile (more likely to cause interference to others), as we have empirically demonstrated in Section 2.4. We define a thread's niceness to increase with the relative fragility of a thread 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, threads are sorted based on 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 in 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 visualizes successive permutations of the priority order for both the round-robin and the insertion shuffle algorithms for four threads. It is interesting to note that in the case of insertion shuffle, the least nice thread spends most of its time at the lowest priority position, while the remaining nicer threads are at higher priorities and are thus able to synergistically leak their memory service time among themselves. Algorithm 2 shows the pseudocode for the insertion shuffle algorithm. Note that the pseudocode does not reflect the actual hardware implementation—the implementation is simple because the permutation is regular.

Handling Threads with Similar Behavior. If the bandwidth-sensitive cluster consists of homogeneous

[2]The name is derived from the similarity to the insertion sort algorithm. Each intermediate state during an insertion sort corresponds to one of the permutations in insertion shuffle.



(a) Round-robin shuffle (b) Insertion shuffle

Figure 3. Visualizing two shuffling algorithms

threads with very similar memory behavior, TCM disables insertion shuffle and falls back to random shuffle to prevent unfair treatment of threads based on marginal differences in niceness values. To do this, TCM inspects whether threads exhibit a sufficient amount of diversity in memory access behavior before applying insertion shuffling. First, TCM calculates the largest difference between any two threads in terms of bank-level parallelism ($\max \Delta BLP$) and row-buffer locality ($\max \Delta RBL$). Second, if both values exceed a certain fraction (*ShuffleAlgoThresh*) of their maximum attainable values, then insertion shuffling is applied. Specifically, $\max \Delta BLP$ must exceed $ShuffleAlgoThresh \times NumBanks$ and $\max \Delta RBL$ must exceed *ShuffleAlgoThresh*. In our experiments we set *ShuffleAlgoThresh* to be 0.1, which intuitively means that TCM falls back to random shuffling if BLP and RBL differ by less than 10% across all threads in the system.

Random Shuffling. When random shuffling is employed, a random permutation of threads is generated every shuffling interval which serves as the thread ranking for the next shuffling interval. In contrast to insertion shuffling, random shuffling is oblivious to thread niceness and does not follow a predetermined shuffling pattern. Random shuffling is also different from round-robin in that it does not preserve the relative position of threads across shuffles, thereby preventing cases where a nice thread remains stuck behind a highly interfering or malicious thread.

The major advantage of random shuffling over insertion shuffling is the significantly lower implementation complexity; it does not require the monitoring of BLP and RBL or the calculation of niceness values for each thread. However, random shuffling pays the penalty of increased unfairness, since it is unable to successfully minimize the interference among heterogeneous threads with large differences in niceness, as we empirically show in Section 7.3. TCM can be forced to always employ random shuffling by setting *ShuffleAlgoThresh* to 1. Section 7.5 provides sensitivity results for *ShuffleAlgoThresh*; Section 7.3 evaluates the effect of different shuffling algorithms.

3.4. Monitoring Memory Access Behavior of Threads

To implement TCM, the L2 cache and memory controller collect statistics for each thread by continuously monitoring its memory intensity, row-buffer locality (RBL), and bank-level parallelism (BLP) over time. If there are multiple memory controllers, this information is sent to a centralized meta-controller at the end of a quantum, similarly to what is done in ATLAS [5]. The meta-controller aggregates the information, computes thread clusters and ranks as described previously, and communicates them to each of the memory controllers to ensure that the thread prioritization order is the same in all controllers.

Memory intensity. A thread's L2 MPKI (L2 cache misses per kiloinstruction) is computed at the L2 cache controller and serves as the measure of memory intensity.

Row-buffer locality. Each memory controller estimates the inherent row-buffer locality of a thread. Doing so requires the memory controller to keep track of a *shadow row-buffer index* [11] for each thread for each bank, which keeps track of the row that would have been open in that bank if the thread were running alone on the system. RBL is simply calculated as the number of shadow row-buffer hits divided by the number of accesses during a quantum.

Bank-level parallelism. Each memory controller counts the number of banks that have at least one memory request from a thread as an estimate of the thread's instantaneous BLP had it been running alone. Throughout a quantum, each controller takes samples of a thread's instantaneous BLP and computes the average BLP for that thread, which is sent to the meta-controller at the end of the quantum. The meta-controller then computes the average BLP for each thread across all memory controllers.

3.5. Summary: Thread Cluster Memory Scheduling (TCM) Prioritization Rules

Algorithm 3 summarizes how TCM prioritizes memory requests from threads. When requests from multiple threads compete to access a bank, the higher ranked thread (where ranking depends on the thread cluster) is prioritized as we have described previously. If two requests share the same priority, row-buffer hit requests are favored. All else being equal, older requests are favored.

3.6. System Software Support

Thread Weights. TCM supports thread weights (or priorities) as assigned by the operating system, such that threads with larger weights are prioritized in the memory. Unlike previous scheduling algorithms, TCM prioritizes a thread based on its weight while also striving to preserve the performance of other threads. Given a thread with a very large thread weight, blindly prioritizing it over all other threads without regard to both its and others'

Algorithm 3 TCM: Request prioritization

1. **Highest-rank first:** *Requests from higher ranked threads are prioritized.*
   - Latency-sensitive threads are ranked higher than bandwidth-sensitive threads (Section 3.1).
   - Within latency-sensitive cluster: lower-MPKI threads are ranked higher than others (Section 3.1).
   - Within bandwidth-sensitive cluster: rank order is determined by insertion shuffling (Section 3.3).
2. **Row-hit first:** *Row-buffer hit requests are prioritized over others.*
3. **Oldest first:** *Older requests are prioritized over others.*

memory access behavior would lead to destruction of the performance of all other threads and, as a result, severely degrade system throughput and fairness.

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 does not prioritize it over the latency-sensitive threads because doing so would significantly degrade the performance of all latency-sensitive threads without significantly improving the performance of the higher-weight thread (as latency-sensitive threads rarely interfere with it).

To enforce thread weights within the latency-sensitive cluster, TCM scales down each thread's MPKI by its weight. Thus, a thread with a larger weight is more likely to be ranked higher than other latency-sensitive threads because its scaled MPKI appears to be low. Within the bandwidth-sensitive cluster, TCM implements weighted shuffling where the time a thread spends at the highest priority level is proportional to its weight.

Fairness/Performance Trade-off Knob. TCM's *ClusterThresh* is exposed to the system software such that the system software can select a value that favors its desired metric. We discuss the effect of *ClusterThresh* on fairness and performance in Section 7.1.

3.7. Multithreaded Workloads

Multithreaded applications can be broadly categorized into two types: those whose threads execute mostly independent of each other and those whose threads require frequent synchronization. Since the first type of multithreaded applications resemble, to a certain extent, multiprogrammed workloads, they are expected to perform well under TCM. In contrast, the execution time of the second type of multithreaded applications is determined by slow-running critical threads [22, 1, 2]. For such applications, TCM can be extended to incorporate the notion of thread criticality to properly identify and prioritize critical threads. Furthermore, we envision TCM to be applicable to composite workloads that consist of an assortment of different applications (e.g., multiple multithreaded applications), by reducing inter-application memory interference.

4. Implementation and Hardware Cost

TCM requires hardware support to 1) monitor threads' memory access behavior and 2) schedule memory requests as described. Table 2 shows the major hardware storage cost incurred in each memory controller to monitor threads' memory access behavior. The required additional storage cost within a controller on our baseline 24-core system is less than 4 Kbits. (If pure random shuffling is employed, it is less than 0.5 Kbits.) TCM requires additional logic to rank threads by aggregating monitored thread metrics. Both ranking and aggregation logic are utilized only at the end of each quantum and are not on the critical path of the processor. Ranking can be implemented using priority encoders, as was done in ATLAS [5].

At the end of every quantum, a central meta-controller (similar to ATLAS [5]) gathers data from every memory controller to cluster threads and to calculate niceness. Subsequently, the central meta-controller broadcasts the results to all the memory controllers so that they can make consistent scheduling decisions throughout the next quantum. At any given point in time, each memory controller prioritizes threads according to their ranking (Algorithm 3). Even though the ranking of the bandwidth-sensitive cluster is shuffled, it is consistent for all memory controllers since shuffling is deterministic and occurs at regular time intervals.

The meta-controller exists only to reduce hardware complexity by consolidating parts of the processing logic at a single location rather than replicating it across separate memory controllers. Although the meta-controller is centralized, it is unlikely to impact scalability since only small amounts of data (4 bytes per hardware context per controller) are exchanged infrequently (once every million cycles). Furthermore, the communication is not latency critical because the previous ranking can be used in the controllers while the next ranking is being computed or transferred.

5. Related Work: Comparison with Other Memory Schedulers

We describe related work on memory scheduling and qualitatively compare TCM to several previous designs. Section 7 compares TCM quantitatively with four state-of-the-art schedulers [19, 13, 14, 5].

Thread-Unaware Memory Schedulers. Memory controller designs that do not distinguish between different threads [28, 19, 25, 9, 3, 20, 15] have been examined within the context of single-threaded, vector, or streaming architectures. The FR-FCFS scheduling policy [19] that prioritizes row-hit requests over other requests is commonly employed in existing processors. Recent work [24] explored reducing the cost of the FR-FCFS design for accelerators. The goal of these policies is 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/multithreaded systems [11, 16, 18, 4, 13, 14, 5].

Thread-Aware Memory Schedulers. Recent work designed thread-aware memory schedulers with the goal of improving fairness and providing QoS. Fair queueing memory schedulers [16, 18] adapted variants of the fair queueing algorithm from computer networks

Memory intensity		
Storage	**Function**	**Size (bits)**
MPKI-counter (monitored by processor)	A thread's cache misses per kilo-instruction	$N_{thread} \cdot \log_2 MPKI_{max} = 240$
Bank-level parallelism		
Storage	**Function**	**Size (bits)**
Load-counter	Number of outstanding thread requests to a bank	$N_{thread} \cdot N_{bank} \cdot \log_2 Queue_{max} = 576$
BLP-counter	Number of banks for which *load-counter* > 0	$N_{thread} \cdot \log_2 N_{bank} = 48$
$BLP_{average}$	Average value of *load-counter*	$N_{thread} \cdot \log_2 N_{bank} = 48$
Row-buffer locality		
Storage	**Function**	**Size (bits)**
Shadow row-buffer index	Index of a thread's last accessed row	$N_{thread} \cdot N_{bank} \cdot \log_2 N_{rows} = 1344$
Shadow row-buffer hits	Number of row-buffer hits if a thread were running alone	$N_{thread} \cdot N_{bank} \cdot \log_2 Count_{max} = 1536$

Table 2. Storage required for monitoring threads' memory access behavior

Processor pipeline	128-entry instruction window
Fetch/Exec/Commit width	3 instructions per cycle in each core; only 1 can be a memory operation
L1 Caches	32 K-byte per core, 4-way set associative, 32-byte block size
L2 Caches	512 K-byte per core, 8-way set associative, 32-byte block size
DRAM controller (on-chip)	128-entry request buffer, 64-entry write data buffer, reads prioritized over writes
DRAM chip parameters	Micron DDR2-800 timing parameters (see [10]) t_{CL}=15ns, t_{RCD}=15ns, t_{RP}=15ns, $BL/2$=10ns; 4 banks, 2K-byte row-buffer per bank
DIMM configuration	Single-rank, 8 DRAM chips put together on a DIMM
Round-trip L2 miss latency	For a 32-byte cache block uncontended: row-buffer hit: 40ns (200 cycles), closed: 60ns (300 cycles), conflict: 80ns (400 cycles)
Cores and DRAM controllers	24 cores, 4 independent DRAM controllers (1 controller has 6.4 GB/s peak DRAM bandwidth)

Table 3. Baseline CMP and memory system configuration

to build a memory scheduler that provides QoS to each thread. Stall-time fair memory scheduler (STFM) [13] uses heuristics to estimate the slowdown of each thread, compared to when it is run alone, and prioritizes the thread that has been slowed down the most. These algorithms aim to maximize fairness, although they can also lead throughput improvements by improving system utilization.

Parallelism-aware batch scheduling (PAR-BS) [14] aims to achieve a balance between fairness and throughput. To avoid unfairness, PAR-BS groups memory requests into batches and prioritizes older batches over younger ones. To improve system throughput, PAR-BS prioritizes less-intensive threads over others to exploit bank-level parallelism. As we will show in Section 7, PAR-BS's batching policy implicitly penalizes memory-non-intensive threads because memory-intensive threads usually insert many more requests into a batch, leading to long delays for memory-non-intensive threads and hence relatively low system throughput.

ATLAS [5] aims to maximize system throughput by prioritizing threads that have attained the least service from the memory controllers. However, as shown in [5], this increase in system throughput comes at the cost of fairness because the most memory-intensive threads receive the lowest priority and incur very high slowdowns.

Ipek et al. [4] leverage machine learning techniques to implement memory scheduling policies that maximize DRAM throughput. Zhu and Zhang [27] describe memory scheduling optimizations for SMT processors to improve DRAM throughput. Neither of these consider fairness or system throughput in the presence of competing threads. Lee et al. [6] describe a mechanism to adaptively prioritize between prefetch and demand requests in a memory scheduler; their mechanism can be combined with ours.

Comparison with TCM. Overall, previous thread-aware memory scheduling algorithms have three major shortcomings, which we address in TCM. First, they are mainly biased towards either fairness or system throughput—no previous algorithm achieves the best system throughput and fairness at the same time. We will show that TCM achieves this by employing multiple different prioritization algorithms, each tailored for system throughput or fairness. Second, previous algorithms do not provide a knob that allows a smooth and gradual trade-off between system throughput and fairness. TCM's ability to group threads into two clusters with different policies optimized for fairness or system throughput allows it to trade off between fairness and system throughput by varying the clustering threshold. Third, previous algorithms do not distinguish different threads' propensity for causing interference to others. As a result, they cannot customize their prioritization policies to the specific needs/behavior of different threads. TCM, by tracking memory access characteristics of threads, determines a prioritization order that favors threads that are likely to cause less interference to others, leading to improvements in fairness and system throughput.

6. Methodology and Metrics

We evaluate TCM using an in-house cycle-level x86 CMP simulator the front-end of which is based on Pin [7]. The memory subsystem is modeled using DDR2 timing parameters [10], which were verified using DRAMSim [23] and measurements from real hardware. Table 3 shows the major DRAM and processor parameters in the baseline configuration. Unless stated otherwise, we assume a 24-core CMP with 4 memory controllers.

Workloads. We use the SPEC CPU2006 benchmarks for evaluation. We compiled each benchmark using gcc 4.1.2 with -O3 optimizations and chose a representative simulation phase using PinPoints [17]. From these benchmarks, we formed multiprogrammed workloads of varying memory intensity, which were run for 100 million cycles.

#	Benchmark	MPKI	RBL	BLP	#	Benchmark	MPKI	RBL	BLP
1	429.mcf	97.38%	42.41	6.20	14	464.h264ref	2.30	90.34%	1.19
2	462.libquantum	50.00%	99.22	1.05	15	435.gromacs	0.98	89.25%	1.54
3	437.leslie3d	49.35%	91.18	1.51	16	445.gobmk	0.77	65.76%	1.52
4	450.soplex	46.70%	88.84	1.79	17	458.sjeng	0.39	12.47%	1.57
5	470.lbm	43.52%	95.17	2.82	18	403.gcc	0.34	70.92%	1.96
6	459.GemsFDTD	31.79%	56.22	3.15	19	447.dealII	0.21	86.83%	1.22
7	482.sphinx3	24.94%	84.78	2.24	20	481.wrf	0.21	92.34%	1.23
8	483.xalancbmk	22.95%	72.01	2.35	21	444.namd	0.19	93.05%	1.16
9	471.omnetpp	21.63%	45.71	4.37	22	400.perlbench	0.12	81.59%	1.66
10	436.cactusADM	12.01%	19.05	1.43	23	454.calculix	0.10	88.71%	1.20
11	473.astar	9.26%	75.24	1.61	24	465.tonto	0.03	88.60%	1.81
12	456.hmmer	5.66%	34.42	1.25	25	453.povray	0.01	87.22%	1.43
13	401.bzip2	3.98%	71.44	1.87					

Table 4. Individual benchmark characteristics (MPKI: Misses per kiloinstruction, RBL: Row-buffer locality, BLP: Bank-level parallelism)**.**

Workload	Memory-intensive benchmarks	Memory-non-intensive benchmarks
A	calculix(3), dealII, gcc, gromacs(2), namd, perl, povray, sjeng, tonto	mcf, soplex(2), lbm(2), leslie, sphinx3, xalancbmk, omnetpp, astar, hmmer(2)
B	gcc(2), gobmk(3), namd(2), perl(3), sjeng, wrf	bzip(2), cactusADM(3), GemsFDTD, h264ref(2), hmmer, libquantum(2), sphinx3
C	calculix(2), dealII(2), gromacs(2), namd, perl(2), povray, tonto, wrf	GemsFDTD(2), libquantum(3), cactusADM, astar, omnetpp, bzip, soplex(3)
D	calculix, dealII, gcc, gromacs, perl, povray(2), sjeng(2), tonto(3)	omnetpp, bzip2(2), h264ref, cactusADM, astar, soplex, lbm(2), leslie, xalancbmk(2)

Table 5. Four representative workloads (figure in parentheses is the number of instances spawned)

We classify benchmarks based on their memory intensity; benchmarks with an average MPKI greater than one are labeled as memory-intensive, while all other benchmarks are labeled as memory-non-intensive. The memory intensity of a workload is defined as the fraction of memory-intensive benchmarks in that workload. Unless stated otherwise, results are for workloads that are 50% memory-intensive (i.e., consisting of 50% memory-intensive benchmarks). For each memory intensity category (50%, 75% and 100%), we simulate 32 multiprogrammed workloads, for a total of 96 workloads.

Evaluation Metrics. We measure system throughput using *weighted speedup* [21] and fairness using *maximum slowdown*. We also report *harmonic speedup* [8], which measures a balance of fairness and throughput.

$$Weighted\ Speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}}$$

$$Harmonic\ Speedup = \frac{N}{\sum_i \frac{IPC_i^{alone}}{IPC_i^{share}}}$$

$$Maximum\ Slowdown = \max_i \frac{IPC_i^{alone}}{IPC_i^{shared}}$$

Parameters of Evaluated Schemes. Unless stated otherwise, we use a *BatchCap* of 5 for PAR-BS [14], a *QuantumLength* of 10M cycles and *HistoryWeight* of 0.875 for ATLAS [5] and a *FairnessThreshold* of 1.1 and *IntervalLength* of 2^{24} for STFM [13]. FR-FCFS [19] has no parameters. For TCM we set *ClusterThresh* to 4/24, *ShuffleInterval* to 800, and *ShuffleAlgoThresh* to 0.1.

7. Results

We compare TCM's performance against four previously proposed memory scheduling algorithms, FR-FCFS [19], STFM [13], PAR-BS [14] (best previous algorithm for fairness) and ATLAS [5] (best previous algorithm for system throughput). Figure 4 shows where each scheduling algorithms 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 achieves the best system throughput and the best fairness, outperforming every algorithm with regard to weighted speedup, maximum slowdown, and harmonic speedup (the last shown in Fig. 6(b)).[3]

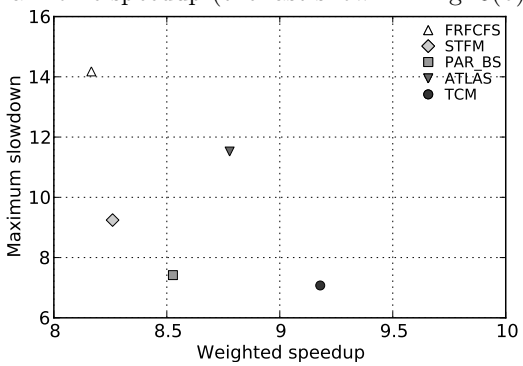


Figure 4. Performance and fairness of TCM vs. other algorithms across all 96 workloads

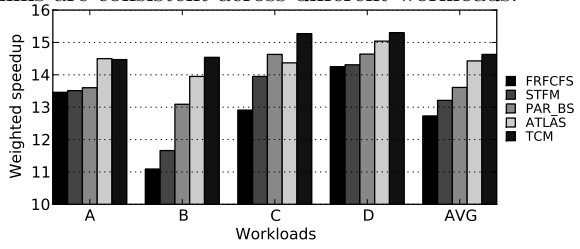
Compared to ATLAS, the highest-performance previous algorithm, TCM provides significantly better fairness (38.6% lower maximum slowdown) and better system throughput (4.6% higher weighted speedup). ATLAS suffers from unfairness because it is a strict priority-based scheduling algorithm where the thread with the lowest priority can access memory only when no other threads have outstanding memory requests to the same bank. As a result, the most deprioritized threads (those which are the most memory-intensive) become vulnerable to starvation and large slowdowns. TCM avoids this problem by using shuffling to ensure that no memory-intensive thread is disproportionately deprioritized.

[3]The performance of TCM as shown here is for just a single operating point. As we will show in Section 7.1, TCM provides the flexibility of smoothly transitioning along a wide range of different performance-fairness trade-off points.

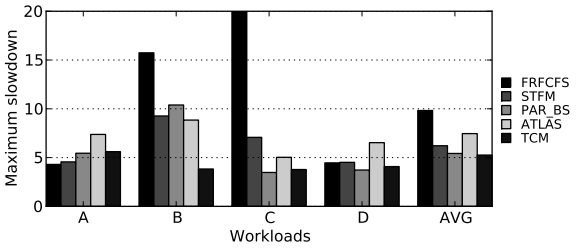
Compared to PAR-BS, the most fair previous algorithm, TCM provides significantly better system throughput (7.6% higher weighted speedup) and better fairness (4.6% lower maximum slowdown). PAR-BS suffers from relatively low system throughput since memory requests from memory-intensive threads can block those from memory-non-intensive threads. PAR-BS periodically forms batches of memory requests and strictly prioritizes older batches. Batch formation implicitly favors memory-intensive threads because such threads have more requests that can be included in the batch. As a result, memory-non-intensive threads are slowed down because their requests (which arrive infrequently) have to wait for the previous batch of requests—mostly full of memory-intensive threads' requests—to be serviced. TCM avoids this problem by ensuring that memory-non-intensive threads are *always* strictly prioritized over memory-intensive ones.

TCM outperforms STFM in weighted speedup by 11.1% and in maximum slowdown by 23.5%. TCM also outperforms the thread-unaware FR-FCFS in both system throughput (12.4%) and maximum slowdown (50.1%). We conclude that TCM provides the best fairness and system performance across all examined previous scheduling algorithms.

Individual Workloads. Figure 5 shows individual results for four, randomly selected, representative workloads described in Table 5. We find that the performance and fairness improvements of TCM over all other algorithms are consistent across different workloads.



(a) Weighted speedup for individual workloads



(b) Maximum slowdown for individual workloads

Figure 5. TCM vs. other algorithms for 4 sample workloads and averaged across 32 workloads

7.1. Trading off between Performance and Fairness

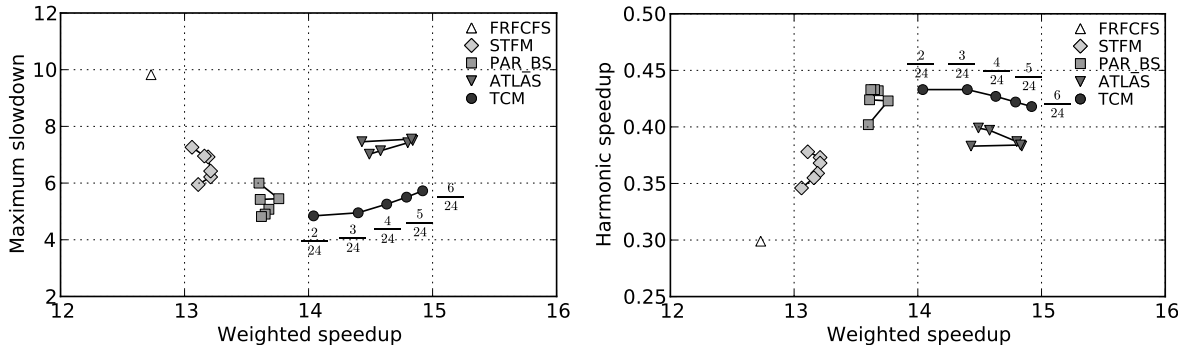
To study the robustness of each memory scheduler, as well as the ability to adapt to different performance and fairness goals, we varied the most salient configuration parameters of each scheduler. We evaluated ATLAS for a *QuantumLength* ranging from 1K (conservative) to 20M cycles (aggressive), PAR-BS for a *BatchCap* ranging from 1 (conservative) to 10 (aggressive), STFM for a *FairnessThreshold* ranging from 1 (conservative) to 5 (aggressive), and FR-FCFS (has no parameters). Finally, for TCM, we vary the *ClusterThresh* from 2/24 to 6/24 in 1/24 increments. The performance and fairness results are shown in Figure 6. The lower right and upper right parts of Figures 6(a) and 6(b) respectively, correspond to better operating points in terms of 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, system throughput and fairness can be gently traded off for one another. As a result, TCM has a wide range of balanced operating points that provide both high system throughput and fairness. None of the previously proposed algorithms provide nearly the same degree of flexibility as TCM. For example, ATLAS always remains biased towards system throughput (i.e., its maximum slowdown changes by little), regardless of its *QuantumLength* setting. Similarly, PAR-BS remains biased towards fairness (i.e., its weighted speedup changes by little).

For TCM, an aggressive (large) *ClusterThresh* value provides more bandwidth for the latency-sensitive cluster and allows relatively "lighter" threads among the bandwidth-sensitive cluster to move into the latency-sensitive cluster. As a result, system throughput is improved since the "lighter" threads are prioritized over the "heavier" threads. But the remaining threads in the bandwidth-sensitive cluster now compete for a smaller fraction of the memory bandwidth and experience larger slowdowns, leading to higher unfairness. In contrast, a conservative (small) *ClusterThresh* value provides only a small fraction of the memory bandwidth for the latency-sensitive cluster so that most threads are included in the bandwidth-sensitive cluster and, as a result, take turns sharing the memory. We conclude that TCM provides an effective knob for trading off between fairness and performance, enabling operation at different desirable operating points depending on system requirements.

7.2. Effect of Workload Memory Intensity

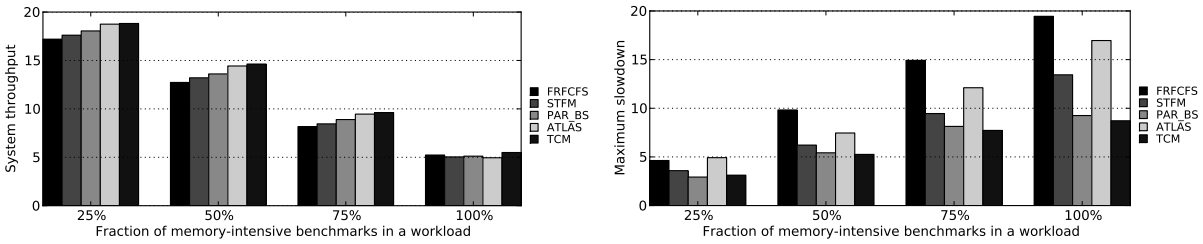
Figure 7 compares the performance of TCM to previously proposed scheduling algorithms for four sets of 32 workloads that are 25%, 50%, 75% and 100% memory-intensive. (We include 25%-intensity workloads for completeness, even though memory is not a large bottleneck for them.) TCM's relative advantage over PAR-BS and ATLAS becomes greater as the workload becomes more memory-intensive and memory becomes more heavily contended. When all the threads in the workload are memory-intensive, TCM provides 7.4% and 10.1% increase in weighted speedup and 5.8% and 48.6% decrease in maximum slowdown compared to PAR-BS and ATLAS. TCM provides higher gains for very memory-intensive workloads because previous algorithms are either unable to prioritize less memory-intensive threads (due to batching policy in PAR-BS) or cause severe deprioritization of the most memory-intensive threads (due to strict ranking in ATLAS) in such heavily contended systems.



(a) Maximum slowdown vs. Weighted speedup

(b) Harmonic speedup vs. Weighted speedup

Figure 6. Performance-Fairness trade-off as algorithmic parameters are varied (for 32 50%-intensity workloads)



(a) System throughput

(b) Unfairness

Figure 7. Performance and fairness of TCM and previous algorithms for different workload mixes

7.3. Effect of Shuffling Algorithm

In Table 6, we compare the fairness of four shuffling algorithms (round-robin, random, insertion, and TCM) when evaluated across 32 workloads. We focus on fairness, because the goal of shuffling is to limit unfairness among bandwidth-sensitive threads. Of the four shuffling algorithms, round-robin shuffle is the most unfair (highest average maximum slowdown) since it is vulnerable to disadvantageous thread rankings where a nice thread becomes stuck behind less nice threads (Section 3.3).

	Shuffling algorithm			
	Round-robin	Random	Insertion	TCM
MS Average	5.58	5.13	4.96	4.84
MS Variance	1.61	1.53	1.45	0.85

Table 6. Average and variance of maximum slowdown (MS) across 32 workloads for four shuffling algorithms

Although random shuffle and insertion shuffle provide higher fairness (low average maximum slowdown), their effectiveness is not consistent but rather fluctuates across different workloads (high variance in maximum slowdown). As described in Section 3.3, this is because random shuffle and insertion shuffle are each geared towards particular types of workloads, homogeneous and heterogeneous workloads, respectively.

The TCM shuffling algorithm combines the strengths of random shuffle and insertion shuffle by dynamically switching back and forth between the two shuffling algorithms, depending on the homogeneity or heterogeneity in workload composition. As a result, it provides the best fairness as well as the smallest variability in fairness across workloads.

7.4. Operating System Thread Weights

TCM enforces thread weights assigned by the operating system in a way that preserves high system throughput and fairness. Figure 8 shows an example where six different thread weights are applied to a heterogeneous workload comprising benchmarks of varying memory intensity. Thread weights are assigned in the worst possible manner for system throughput, i.e., higher weights to more intensive threads. ATLAS blindly adheres to thread weights and thus severely slows down memory-non-intensive benchmarks due to interference. In contrast, TCM provides comparable speedups for the highest-weighted memory-intensive threads and, at the same time, vastly accelerates the memory-non-intensive threads. As a result, TCM increases system throughput and reduces maximum slowdown by 82.8% and 44.2%, respectively.

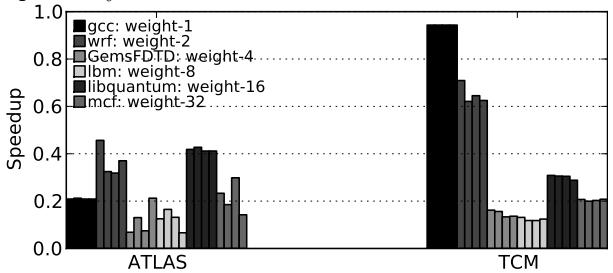


Figure 8. Performance of threads with different weights

7.5. Sensitivity Analysis

As shown in Table 7, the performance of TCM is robust against variations in algorithmic parameters. However, we do observe a small decrease in performance as

		ShuffleAlgoThresh			*ShuffleInterval*			
		0.05	0.07	0.10	500	600	700	800
TCM	System throughput	14.3	14.2	14.2	14.2	14.3	14.2	14.7
	Maximum slowdown	5.5	5.7	5.9	6.0	5.4	5.9	5.5

Table 7. Sensitivity of TCM to algorithmic parameters (32 workloads)

		Number of memory controllers					Number of cores					Cache size		
		1	2	4	8	16	4	8	16	24	32	512KB	1MB	2MB
Compared to ATLAS	Sys. throughput	5%	2%	1%	2%	3%	0%	3%	2%	1%	1%	1%	3%	4%
	Max. slowdown	-50%	-44%	-30%	-40%	-53%	-4%	-30%	-29%	-30%	-41%	-30%	-28%	-29%

Table 8. Sensitivity of TCM to system configuration (32 workloads)

the *ShuffleInterval* value is decreased, due to the reduced row-buffer locality. Table 8 compares the performance of TCM against ATLAS (best previous scheduler in terms of performance), as the configuration of the system is varied (cache size, number of controllers and cores). TCM consistently provides significantly better fairness and better system throughput across all system configurations.

8. Conclusion

We presented Thread Cluster Memory scheduling (TCM), a new approach to memory scheduling that provides the best system throughput and fairness. TCM achieves this by dynamically grouping threads into two clusters with different needs (latency- vs. bandwidth-sensitive) and employing different scheduling policies within each cluster: one tailored to maximize system throughput, the other tailored to maximize fairness. To our knowledge, TCM is the first memory scheduling algorithm that employs multiple different scheduling policies for different threads based on threads' memory access and interference characteristics.

Our experimental evaluations show that TCM provides better system throughput than the highest performing previous algorithm and better fairness than the most fair previous algorithm. TCM provides a control knob that allows it to smoothly trade off between system throughput and fairness over a wide range, which was not possible in previous scheduling algorithms.

We conclude that TCM can be an effective memory scheduling substrate for multi-core systems: one that provides very high levels of performance and fairness, as well as flexible mechanisms for smoothly transitioning between these two goals.

Acknowledgments

We thank the anonymous reviewers and members of CALCM (Computer Architecture Lab at Carnegie Mellon) for valuable feedback. Yoongu Kim is supported by a Ph.D. fellowship from KFAS (Korea Foundation for Advanced Studies). We gracefully acknowledge the support of Gigascale Systems Research Center, Intel, and CyLab. This research was partially supported by an NSF CAREER Award CCF-0953246.

References

[1] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *ISCA-36*, 2009.

[2] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *PACT-17*, 2008.

[3] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *MICRO-37*, 2004.

[4] E. İpek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA-35*, 2008.

[5] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA-16*, 2010.

[6] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware DRAM controllers. In *MICRO-41*, 2008.

[7] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[8] K. Luo, J. Gummaraju, and M. Franklin. Balancing thoughput and fairness in smt processors. In *ISPASS*, 2001.

[9] S. A. McKee et al. Dynamic access ordering for streamed computations. *IEEE TC*, 49(11):1255–1271, Nov. 2000.

[10] Micron. *1Gb DDR2 SDRAM: MT47H128M8HQ-25*.

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

[12] T. Moscibroda and O. Mutlu. Distributed order scheduling and its application to multi-core DRAM controllers. In *PODC*, 2008.

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

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

[15] C. Natarajan et al. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI-3*, 2004.

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

[17] H. Patil et al. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.

[18] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective management of DRAM bandwidth in multicore processors. In *PACT-16*, 2007.

[19] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000.

[20] J. Shao and B. T. Davis. A burst scheduling access reordering mechanism. In *HPCA-13*, 2007.

[21] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *ASPLOS-IX*, 2000.

[22] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS-14*, 2009.

[23] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. Dramsim: a memory system simulator. *SIGARCH Comput. Archit. News*, 33(4):100–107, 2005.

[24] G. L. Yuan, A. Bakhoda, and T. M. Aamodt. Complexity effective memory access scheduling for many-core accelerator architectures. In *MICRO-42*, 2009.

[25] L. Zhang et al. The impulse memory controller. *IEEE TC*, 50(11):1117–1132, Nov. 2001.

[26] H. Zheng, J. Lin, Z. Zhang, and Z. Zhu. Memory access scheduling schemes for systems with multi-core processors. In *ICPP*, 2008.

[27] Z. Zhu and Z. Zhang. A performance comparison of DRAM memory system optimizations for SMT processors. In *HPCA-11*, 2005.

[28] 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. U.S. Patent Number 5,630,096, May 1997.