Parallel Application Memory Scheduling

Eiman Ebrahimi† Rustam Miftakhutdinov† Chris Fallin§
Chang Joo Lee‡ José A. Joao† Onur Mutlu§ Yale N. Patt†

†Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, rustam, joao, patt}@ece.utexas.edu

§Carnegie Mellon University
{cfallin,onur}@cmu.edu

‡Intel Corporation
chang.joo.lee@intel.com

ABSTRACT

A primary use of chip-multiprocessor (CMP) systems is to speed up a single application by exploiting thread-level parallelism. In such systems, threads may slow each other down by issuing memory requests that interfere in the shared memory subsystem. This *inter-thread memory system interference* can significantly degrade parallel application performance. Better memory request scheduling may mitigate such performance degradation. However, previously proposed memory scheduling algorithms for CMPs are designed for multi-programmed workloads where each core runs an independent application, and thus do not take into account the inter-dependent nature of threads in a parallel application.

In this paper, we propose a memory scheduling algorithm designed specifically for parallel applications. Our approach has two main components, targeting two common synchronization primitives that cause inter-dependence of threads: locks and barriers. First, the runtime system estimates threads holding the locks that cause the most serialization as the set of limiter threads, which are prioritized by the memory scheduler. Second, the memory scheduler shuffles thread priorities to reduce the time threads take to reach the barrier. We show that our memory scheduler speeds up a set of memory-intensive parallel applications by 12.6% compared to the best previous memory scheduling technique.

Categories and Subject Descriptors: C.1.0 [Processor Architectures]: General; C.1.2 [Multiple Data Stream Architectures (Multiprocessors)]
General Terms: Design, Performance.
Keywords: Parallel Applications, Shared Resources, CMP, Memory Controller, Multi-core, Memory Interference.

1. INTRODUCTION

Chip multiprocessors (CMPs) are commonly used to speed up a single application using multiple threads that concurrently execute on different cores. These parallel threads share memory system resources beyond some level in the memory hierarchy (e.g., bandwidth to main memory and a shared last level cache). Memory requests from concurrently executing threads can interfere with one another in the shared memory subsystem, slowing the threads down significantly. Most importantly, the *critical path* of execution can also be significantly slowed down, resulting in increased application execution time.

To illustrate the importance of DRAM-related inter-thread interference to parallel application performance, Figure 1 shows the potential performance improvement that can be obtained for six different parallel applications run on a 16-core system.[1] In this experiment, we ideally eliminate all *inter-thread* DRAM-related interference. Thread i's DRAM-related interference cycles are those extra cycles that thread i has to wait on memory due to bank or row-buffer conflicts caused by concurrently executing threads (compared to if thread i were accessing the same memory system alone). In the ideal, unrealizable system we model for this experiment: 1) thread i's memory requests wait for DRAM banks only if the banks are busy servicing requests from that same thread i, and 2) no DRAM row-conflicts occur as a result of some other thread j ($i \neq j$) closing a row that is accessed by thread i (i.e., we model each thread as having its own row buffer in each bank). This figure shows that there is significant potential performance to be obtained by better management of memory-related inter-thread interference in a parallel application: ideally eliminating inter-thread interference reduces the average execution time of these six applications by 45%.

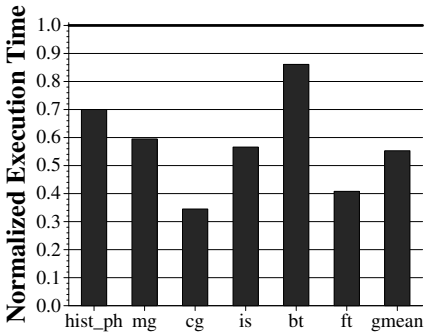


Figure 1: Potential execution time reduction if all inter-thread interference is ideally eliminated

[1]Our system configuration and benchmark selection are discussed in Section 4.

MICRO'11 December 3-7, 2011, Porto Alegre, Brazil

Previous work on managing memory system related *inter-application* interference [13, 15, 27, 14, 25, 23, 26, 6, 7, 16, 17, 9] addresses the problem of improving system performance (system throughput or average job turnaround time) and/or system fairness in the context of multi-programmed workloads where different cores of the CMP execute independent single-threaded applications.[2] None of these works directly address parallel *multi-threaded* applications as we do in this paper where our goal of managing memory system inter-thread interference is very different: reducing the execution time of a single parallel application. Managing the interference between threads of a parallel application poses a different challenge than previous works: threads in a parallel application are likely to be inter-dependent on each other, whereas such inter-dependencies are assumed to be non-existent between applications in these previous works. Techniques for reducing inter-application memory interference for improving system performance and fairness of multi-programmed workloads may result in improved parallel application performance by reducing overall interference. However, as we show in this paper, designing a technique that specifically aims to maximize parallel application performance by taking into account the inter-dependence of threads within an application can lead to significantly higher performance improvements.

Basic Idea: We design a memory scheduler that reduces parallel application execution time by managing inter-thread DRAM interference. Our solution consists of two key parts:

First, we propose estimating the set of threads likely to be on the critical path using *limiter thread* estimation (for lock-based synchronization) and *loop progress* measurement (for barrier-based synchronization). For lock-based synchronization, we extend the runtime system (e.g., runtime library that implements locks) with a mechanism to estimate a set of *limiter threads* which are likely critical (i.e., make up the critical path of the application). This estimate is based on lock contention, which we quantify as the time threads spend waiting to acquire a lock. For barrier-based synchronization used with parallel `for` loops, we add hardware iteration counters to estimate the progress of each thread towards the barrier at the end of the loop. We identify threads that fall behind as more likely to be critical.

Second, we design our memory controller based on two key principles: a) we prioritize threads that are likely to be on the critical path (which are either limiter threads or threads falling behind in parallel loops), and b) among a group of limiter threads, non-limiter threads, or parallel-for-loop threads that have made the same progress towards a synchronizing barrier (i.e. threads that are equally critical), we shuffle thread priorities in a way that reduces the time all threads collectively make progress.

Summary of Evaluations: We evaluate our parallel application memory scheduling (PAMS) technique on a 16-core CMP system in comparison to the commonly used FR-FCFS [31, 33] memory scheduler and a state-of-the-art memory scheduler, TCM [17], which is designed for high performance and fairness on multi-programmed workloads. Experimental results across six memory intensive parallel workloads show that PAMS outperforms FR-FCFS and TCM by 16.7% and 12.6% respectively.

Contributions: To our knowledge, our proposal is the first design that manages inter-thread memory system interference at the memory controller to improve parallel application performance. We make the following contributions:

1. We propose a runtime-system mechanism to periodically estimate a set of *limiter threads*, which is likely to include the thread on the *critical path* in lock-based synchronization, for the purpose of memory request prioritization.
2. We propose a memory request prioritization mechanism that reduces inter-thread interference among threads that are equally critical, by periodically shuffling their priorities, in order to reduce the time it takes the threads to collectively reach their synchronization point.
3. We propose a memory scheduling algorithm that estimates likely-critical threads based on lock contention information and progress of threads in parallel `for` loops. Our design makes memory scheduling decisions based on its prediction of likely-critical threads and dynamically gathered information about the memory intensity of concurrently-executing threads. We show that by intelligently prioritizing requests in a thread-aware manner, our memory controller significantly improves the performance of parallel applications compared to three state-of-the-art memory controller designs.

2. BACKGROUND

We briefly describe a state-of-the-art memory scheduling technique, Thread Cluster Memory Scheduling (TCM) [17], which we will extensively discuss and compare to in this paper. Here we will explain ideas from this previous work that are important to understand our mechanisms.

TCM is designed to address system throughput and fairness with the goal of achieving the best of both for multi-programmed workloads. Note that TCM is designed for multi-programmed workloads (where different cores of a CMP are executing independent single-threaded applications) and has only been evaluated on these workloads. The algorithm detects and exploits differences in memory access behavior across applications. TCM periodically (every 10M cycles in [17]) groups applications into two clusters: *latency-sensitive* and *bandwidth-sensitive*. This is based on the applications' memory intensity measured in last level cache misses per thousand instructions (MPKI). The least memory-intensive threads are placed in the latency-sensitive cluster, and others in the bandwidth-sensitive cluster. To improve system throughput, TCM always prioritizes latency-sensitive applications over bandwidth-sensitive ones. To improve fairness, the priorities of applications in the bandwidth-sensitive cluster are periodically shuffled (every 800 cycles in [17]).

As we show in Section 5, TCM can improve the performance of multi-threaded workloads compared to a standard FR-FCFS (First Ready, First Come First Serve) memory scheduling algorithm [31, 33] by mitigating some memory related inter-thread interference. However, as we will demonstrate later, a memory scheduling algorithm targeted at managing DRAM interference specifically for multi-threaded applications can significantly reduce application runtime compared to such state-of-the-art techniques.

[2]We refer to interference between independent applications running on different cores as inter-application interference, and to interference between threads of a parallel application running on different cores as inter-thread interference.

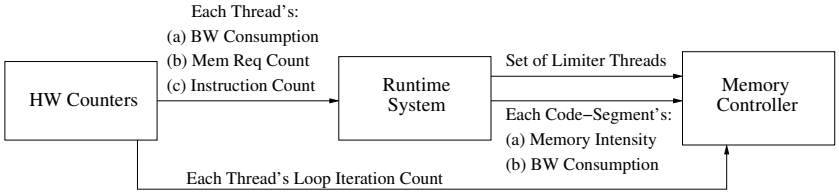


Figure 2: Overview of parallel application memory scheduling

3. MECHANISM

Our parallel application memory scheduler (PAMS):

1. Estimates likely-critical threads using limiter estimation (Section 3.1.1) and loop progress measurement (Section 3.1.2).
2. Prioritizes likely-critical threads (Section 3.2.2) and shuffles priorities of non-likely-critical threads (Section 3.2.3) to reduce inter-thread memory interference.

Figure 2 provides an overview of the interactions between the major components of our design. The runtime system (e.g., runtime library that implements locks) uses hardware monitors to characterize memory behavior of *code-segments* (parts of the parallel program, see Section 3.2.1 for details) and passes this information to the memory controller. In addition, the runtime system provides the memory controller with a set of *limiter threads* (those likely to be on the critical path). Finally, the memory controller has access to iteration counts of parallel `for` loops. The following sections describe each component in detail.

3.1 Runtime System Extensions

In parallel applications, the *critical path* of execution determines the running time of the program. In each execution cycle, the critical path lies on one of the concurrently executing threads. Hence, to improve performance, the memory scheduler should minimize memory-related interference suffered by memory requests issued by the thread on the critical path. Unfortunately, identifying exactly which thread is on the critical path at runtime with low/acceptable overhead is difficult. However, we find that even a coarse estimation of the critical path can be very useful.

We propose to estimate the critical path via limiter thread estimation and loop progress measurement. Limiter thread estimation is a runtime system mechanism which identifies a set of threads likely to contain the thread on the critical path by analyzing lock contention. We call these threads *limiter threads*, since one of them likely limits the application running time. Loop progress measurement is a cooperative compiler/hardware mechanism which estimates the progress of each thread within a parallel `for` loop, for programs structured with such barrier-synchronized loops across threads.

The memory controller uses limiter thread and loop progress information to manage inter-thread interference in the DRAM system and improve application performance.

3.1.1 Estimating Limiter Threads

When multiple threads concurrently execute and access shared data, correctness is guaranteed by the *mutual exclusion* principle: multiple threads are not allowed to access shared data concurrently. This mutual exclusion is achieved by encapsulating accesses to shared data in code regions guarded by synchronization primitives such as locks. Such guarded code is referred to as *critical section* code.

Prior work [32] shows that accelerating *critical sections* by executing them on high performance cores in a heterogeneous CMP can significantly reduce application running time. This is because contended critical sections are often on the *critical path*. We find that performance can be greatly improved by exposing information about contended critical sections to the memory controller, which uses this information to make better memory scheduling decisions. The rest of this subsection describes how this information is gathered by the runtime system and passed to the memory controller. We describe how the runtime system informs the memory controller of the single most contended critical section for ease of explanation; in general, however, the runtime system can detect any number of most contended critical sections.

As more and more threads contend over the lock protecting some shared data, it is more likely that threads executing the critical section guarded by the contended lock will be on the critical path of execution. As such, at a high level, the runtime system periodically identifies the most contended lock. The thread holding that lock is estimated to be a limiter thread. Limiter thread information is passed to the memory controller hardware using the *LimiterThreadBitVector* which has a bit per thread.[3] The runtime system identifies thread i as a *limiter* thread by setting the corresponding bit i in this bit-vector. This information is used by the memory controller in the following interval. The runtime system provides two main pieces of information which our algorithm uses to estimate limiter threads: the ID of the thread currently holding each lock, and the time a thread starts waiting for a lock.

Algorithm 1 explains limiter thread estimation in detail. The goal of the algorithm is to a) find the lock that causes the most contention in a given interval, and b) record the thread that owns this lock in *LimiterThreadBitVector* so that the memory controller can prioritize that thread. To implement the algorithm, the runtime system maintains one counter per lock which accumulates the total cycles threads wait in that lock's queue, and keeps two variables to record the currently most-contended lock and the thread that owns it.

Every interval (i.e., every *LimiterEstimationInterval* lock acquires), the runtime system finds the most-contended lock. To do so, it compares the lock queue waiting times accumulated for all of the locks. The system identifies the lock for which threads spent the most time waiting in the queue during the previous interval and saves it as $Lock_{longest}$. It then determines which thread is holding that lock, and sets the corresponding bit in the *LimiterThreadBitVector*.

[3]In this paper, we consider one thread of execution per core, but in systems with simultaneous multithreading (SMT) support, each thread context would have its own bit in *LimiterThreadBitVector*.

To keep track of each lock's waiting time, every time a lock is successfully acquired by some thread i, the runtime system adds the time thread i spent waiting on the lock to the lock's waiting time counter (See Section 3.3 for implementation details). Finally, when a thread acquires the lock that had the longest waiting time in the previous interval ($Lock_{longest}$), *LimiterThreadBitVector* is updated: the bit corresponding to the previous owner of the lock is reset in the vector, the bit for the thread acquiring the lock is set, and the new owner is recorded as $LastOwner_{longest}$. This updated bit-vector is communicated to the memory controller in order to prioritize the limiter thread.

Algorithm 1 Runtime Limiter Thread Estimation

Every *LimiterEstimationInterval* **lock acquires**
  Find lock with longest total waiting time in previous interval
  Set $Lock_{longest}$ to the lock with the longest waiting time
  Set $LastOwner_{longest}$ to the thread that holds $Lock_{longest}$
  Set bit for $LastOwner_{longest}$ in *LimiterThreadBitVector*
Every successful lock acquire
  Increment *waitingTime* counter of acquired lock by the number of cycles spent in the lock's queue by the acquiring thread
  if acquired lock is $Lock_{longest}$ **then**
    Reset bit for $LastOwner_{longest}$ in *LimiterThreadBitVector*
    Record new $Lock_{longest}$ owner in $LastOwner_{longest}$
    Set bit for $LastOwner_{longest}$ in *LimiterThreadBitVector*
  end if

3.1.2 Measuring Loop Progress

Parallel `for` loops are a common parallel programming construct which allows for critical path estimation in a different way. Each iteration of a parallel `for` loop identifies an independent unit of work. These loops are usually statically scheduled by dividing iterations equally among threads. After the threads complete their assigned iterations, they typically synchronize on a barrier.

Given this common computation pattern, we can easily measure the progress of each thread towards the barrier by the number of loop iterations they have completed, as has also been proposed by Cai et al. [2]. We employ the compiler to identify this computation pattern and pass the address of the loop branch to the PAMS hardware. For each thread, we add a hardware loop iteration counter which tracks the number of times the loop branch is executed (i.e., the number of loop iterations completed by the thread) The runtime system resets these counters at every barrier.

The memory controller uses this loop progress information to prioritize threads that have lower executed iteration counts, as described in Section 3.2.3.

3.2 Memory Controller Design

At a high level, our memory controller enforces three priorities in the following order (see Algorithm 2): First, we prioritize row-buffer hit requests over all other requests because of the significant latency benefit of DRAM row-buffer hits compared to row-buffer misses. Second, we prioritize limiter threads over non-limiter threads, because our runtime system mechanism deems limiter threads likely to be on the critical path. We describe prioritization among limiter threads in detail in Section 3.2.2. We prioritize remaining non-limiter threads according to *loop progress* information described in Section 3.1.2. Prioritization among non-limiter threads is described in detail in Section 3.2.3. Algorithm 2 serves as a high level description and outline for the subsections that follow.

Algorithm 2 Request Prioritization for Parallel Application Memory Scheduler (PAMS)

1. Row-hit first
2. Limiter threads (Details of the following are explained in Section 3.2.2)
- Among limiter threads, latency-sensitive threads are prioritized over bandwidth-sensitive threads
- Among latency-sensitive group: lower-MPKI threads are ranked higher
- Among bandwidth-sensitive group: periodically shuffle thread ranks

3. Non-Limiter threads (Details of the following are explained in Section 3.2.3)
if loop progress towards a synchronizing barrier is known **then**
  - Prioritize threads with lower loop-iteration counts first
  - Among threads with same loop-iteration count: shuffle thread ranks
else
  - Periodically shuffle thread ranks of non-limiter threads
end if

3.2.1 Terminology

Throughout the subsections that follow, we will be using the term **code-segment** which we define as: a program region between two consecutive synchronization operations such as lock acquire, lock release, or barrier. Code-segments starting at a lock acquire are also distinguished based on the address of the acquired lock. Hence, a code-segment can be identified with a 2-tuple:

<beginning IP, lock address (zero if code is not encapsulated within a critical section)>

Code-segments are an important construct in classifying threads as latency- vs. bandwidth-sensitive (as we describe in the next subsection), and also in defining the intervals at which classification and shuffling are performed.

3.2.2 Prioritization Among Limiter Threads

The goal for the limiter thread group is to achieve high performance in servicing the requests of the group as the runtime system has estimated the critical path to lie on one of the threads of this group, while also ensuring some level of fairness in progress between them as we do not know exactly which one is on the critical path. To this end, we propose classifying limiter threads into two groups: *latency-sensitive* and *bandwidth-sensitive*. Latency-sensitive threads (which are generally the less memory intensive threads) are prioritized over bandwidth-sensitive ones. As Algorithm 2 shows, among latency-sensitive threads, threads with lower MPKI are prioritized as they are less-memory intensive and servicing them quickly will allow for better utilization of the cores. Prioritization among bandwidth-sensitive threads is done using a technique called *rank shuffling* [17]. This technique is also used to prioritize non-limiter threads and, in fact, is more important in that context; hence, we defer discussion of rank shuffling to Section 3.2.3. The rest of this subsection describes how the latency- vs. bandwidth-sensitive classification is done.

Latency-sensitive vs. bandwidth-sensitive classification of threads: As described in [17], a less memory intensive thread has greater potential to make progress and keep its core utilized than a more memory intensive one. Hence, classifying it as latency-sensitive and prioritizing it in the memory controller improves overall system throughput because it allows the thread to quickly return to its compute

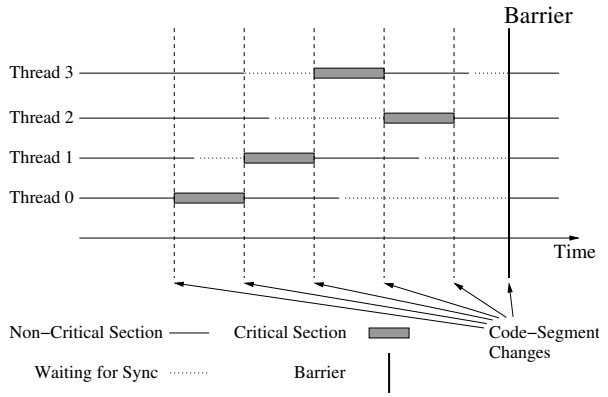


Figure 3: Code-segment based classification

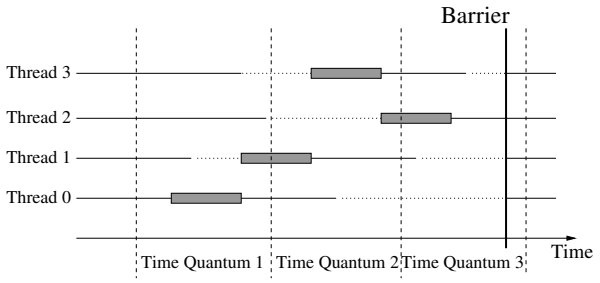


Figure 4: Time based classification

phase and utilize its core. To do this classification, the main question is how to predict the future memory intensity of the code a thread is about to execute.

We propose classifying threads as latency- or bandwidth-sensitive based on the memory intensity of the *code-segment* that thread is executing. The key idea is that we can estimate the memory intensity of the code-segment that the thread is entering based on the memory intensity of that code-segment last time it was executed. Figure 3 illustrates this strategy. Classification of threads is performed at each code-segment change (indicated by a vertical dotted line in the figure). Algorithm 3 presents the details of the classification algorithm used by the memory controller. This algorithm is a modified version of the original thread clustering algorithm by Kim et al. [17] adapted to be invoked at every code-segment change.[4] The algorithm requires information about the memory intensity (number of misses per thousand instructions) and bandwidth consumption of the code-segment to be executed (number of cycles that at least one memory bank is busy servicing the code-segment's requests). The runtime system predicts this future behavior based on monitored behavior from when that code-segment was executed last (as we will explain further shortly).

Algorithm 3 sets aside a threshold fraction (*ClusterThreshold*) of the total bandwidth per cycle for latency-sensitive threads. It uses previous bandwidth consumption of currently executing code-segments to predict their current behavior. To do so, it sums up the previous bandwidth consumption of the least memory intensive currently-executing code-segments up to a *ClusterThreshold* fraction of total bandwidth consumption. The threads that are included in this sum are classified as latency-sensitive.

Note that in the original algorithm, Kim et al. [17] measure each cores' memory intensity every 10M cycles in a multi-core system where each core executes an independent application. In other words, they classify threads on a time interval basis rather than on the basis of a change in the code-segment. We find that with parallel workloads there is little information to be gained by looking back at a thread's memory behavior over a fixed time interval. Figure 4 shows why. In the figure, thread 2 spends a long time waiting on a lock in time quantum 2. However, its memory behavior measured during that time interval has nothing to do with its memory behavior in the following time interval (time quantum 3), during which it happens to be not waiting. For this reason, we perform classification not on a time interval basis but on the basis of a code-segment change.

[4]We refer the reader to Algorithm 1 in the original TCM [17] paper for details on the original algorithm.

Algorithm 3 Latency-sensitive vs. Bandwidth-sensitive classification for limiter threads

Per-thread parameters:
$CodeSegMPKI_i$: MPKI of code-segment currently running on thread i the last time it occurred
$CodeSegBWConsumedPerCycle_i$: BW consumed per cycle by code-segment currently running on thread i the last time it occurred
$BWConsumed_i$: Bandwidth consumed by thread i during previous interval
Classification: (every code-segment change)
TotalBWConsumedPerCycle = $(\Sigma_i\ BWConsumed_i)$ / $Length\ Of\ Previous\ Interval\ In\ Cycles$
while Threads left to be classified **do**
  Find thread with lowest MPKI (thread i)
  SumBW += $CodeSegBWConsumedPerCycle_i$
  if SumBW $\leq$ $ClusterThreshold$ $\times$ TotalBWConsumedPerCycle **then**
    thread i classified as *LatencySensitive*
  else
    thread i classified as *BandwidthSensitive*
  end if
end while

Keeping track of code-segment memory behavior: When a thread executes a code-segment, the memory controller maintains a counter for the number of memory requests generated by that code-segment. Another counter maintains the number of instructions executed in the code-segment. When the code-segment ends, the runtime system takes control because a synchronization event has occurred. The runtime system reads both counters and calculates the memory intensity of that code-segment which it stores for later use. It also keeps track of a *bandwidth consumed per cycle* count for the completed code-segment. When that code-segment is started on the same or another thread in the future, the runtime system loads two registers in the memory controller with the memory intensity and bandwidth consumed per cycle that were last observed for that code-segment. The memory controller uses this information to (i) classify threads into latency- vs. bandwidth-sensitive clusters and (ii) prioritize latency-sensitive limiter threads with lower memory intensity.

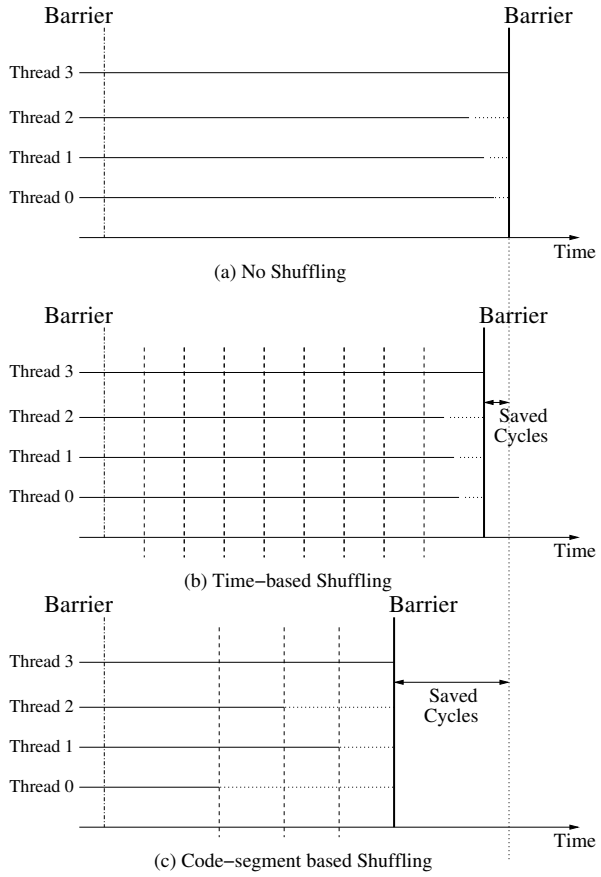


Figure 5: Threads have similar memory behavior

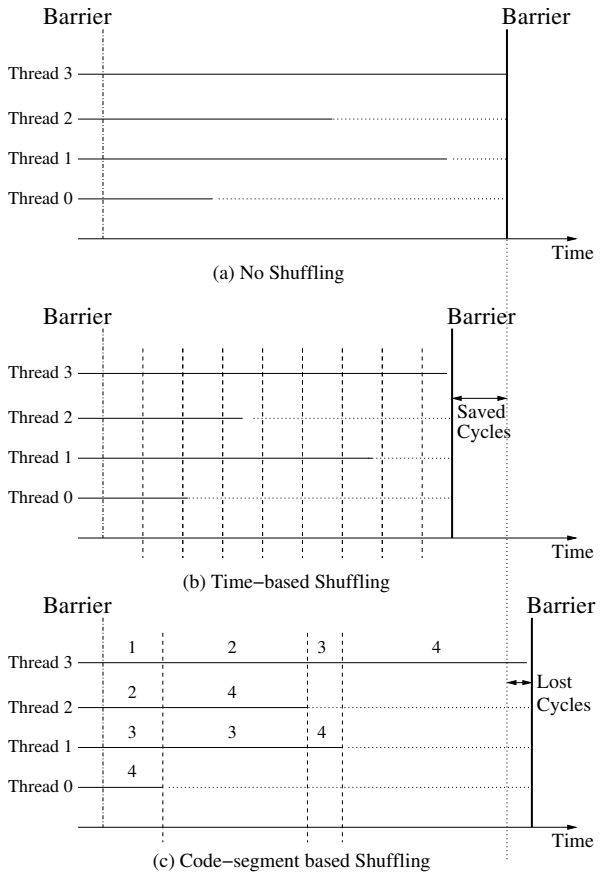


Figure 6: Threads have different memory behavior

3.2.3 Prioritization Among Non-limiter Threads

When the application is executing a parallel `for` loop, the memory controller uses loop progress information (Section 3.1.2) to ensure balanced thread execution. The measured loop progress information is used by the memory controller to create priorities for different threads in order of their loop progress: threads with lower iteration counts—those falling behind—are prioritized over those with higher iteration counts. This prioritization happens on an interval by interval basis, where the priorities assigned based on loop progress are maintained for a while to give threads that have fallen behind a chance to fully exploit their higher priority in the memory system (e.g., exploit row buffer locality). Subsequently, priorities are re-evaluated and assigned at the end of the interval for the next interval.

Among a set of threads that have the same loop progress or in the absence of such information, the memory controller aims to service all bandwidth-sensitive threads in a manner such that none become a new bottleneck as a result of being deprioritized too much in the memory system. To achieve this, we perform interval-based *rank shuffling* of the threads.

Shuffling of bandwidth-sensitive threads: At the beginning of each interval, we assign a random rank to each of the bandwidth-sensitive threads and prioritize their memory requests based on that ranking in that interval. The main question in the design of shuffling for parallel threads is: when should an interval end and new rankings be assigned?

We find that a group of threads that have similar memory behavior should be treated differently than a group of threads that do not.[5] When threads have similar memory behavior, we find that maintaining a given random ranking until one of the threads finishes executing the code-segment it is currently executing can significantly improve performance. This is because when a code-segment ends (e.g., when the thread reaches a barrier), the inter-thread interference it was causing for the other threads is removed, and the other threads can make faster progress in its absence. We call this *code-segment based shuffling*: new thread ranks are assigned when a code-segment change happens. On the other hand, when a group of threads have very different memory behavior, we find that changing the thread ranking only on a code-segment change can sometimes lead to performance loss. For example, if the thread that is going to reach the barrier first is assigned the highest rank, keeping it prioritized until it reaches the barrier delays the thread that would be last to reach the barrier, lengthening the critical path of the program. As such, for threads with very different memory behavior, fixed-interval time-based shuffling of thread ranking performs better. This allows each thread to get quick service for its memory requests for a while and make proportional progress toward the barrier. We call this *time-based shuffling*.

[5]When the ratio between the largest memory intensity and the smallest memory intensity of all threads within a group of threads is small (less than 1.2 in our experiments), we refer to the group as a group of threads with similar memory behavior.

Figures 5 and 6 illustrate how each of these two shuffling policies performs when applied to two very different scenarios for threads concurrently executing between two barriers.

When the set of threads have similar memory behavior as shown in Figure 5 (a), code-segment based shuffling can be significantly better than time-based shuffling. Behavior similar to this exists in the applications *ft* and *is*. Time-based shuffling (Figure 5 (b)) improves performance over no shuffling by allowing different threads to be prioritized during different time intervals and thus make proportional progress toward the barrier. However, all threads continue to interfere with one another in the memory system until they all reach the barrier at a similar time. Code-segment based shuffling reduces this interference between threads by ensuring some threads reach the barrier earlier and once they reach the barrier, they stop exerting pressure on the memory system. As shown in Figure 5 (c) and described above, maintaining a given random ranking until a code-segment change happens (i.e., a thread reaches a barrier) allows the prioritized thread to reach its barrier before the deprioritized one. After that, the deprioritized thread can make much faster progress because previously-prioritized threads stop exerting memory interference as they are waiting at the barrier. For this very reason, code-segment based shuffling can significantly improve performance over time-based shuffling, as shown in the longer "Saved Cycles" of Figure 5 (c) compared to that of Figure 5 (b).

When the set of threads have different memory behavior as shown in Figure 6 (a), time-based shuffling can outperform code-segment based shuffling. Behavior similar to this can be observed in the *mg* application. With time-based shuffling (Figure 6 (b)), threads are assigned different random rankings for each fixed-length interval, which allows each thread to get quick service for its memory requests for a while. This reduces the time it takes for all threads to get to the barrier at the end of the interval. Figure 6(c) shows how code-segment based shuffling can easily perform poorly. The numbers shown above the threads in the different intervals are an example of random ranks assigned to the threads every time one of the threads' code-segment finishes (i.e., every time a thread reaches the barrier, in this example). Because the threads which would have anyway reached the barrier earlier end up receiving a high rank over the thread that would reach the barrier last (thread 3) after every code-segment change, code-segment based shuffling delays the "critical thread" by causing more interference to it and therefore results in performance loss compared to time-based shuffling and even compared to no shuffling, as shown in "Lost Cycles" in Figure 6(c).

Dynamic shuffling policy: Since neither of the two policies always performs best, we propose a dynamic shuffling policy that chooses either time-based shuffling or code-segment based shuffling based on the similarity in the memory behavior of threads. Our dynamic shuffling policy operates on an interval-basis. An interval ends when each thread executes a threshold number of instructions (we empirically determined this interval as 5000 instructions). Our proposed policy continuously monitors the memory intensity of the threads to be shuffled. At the end of each interval, depending on the similarity in memory intensity of the threads involved, the memory controller chooses a time-based or code-segment-based shuffling policy for the following interval. As we will show in Section 5, this policy performs better than either time-based shuffling or code-segment based shuffling employed for the length of the application.

3.3 Implementation Details

Table 1 breaks down the modest storage required for our mechanisms, 1552 bits in a 16-core configuration. Additionally, the structures we add or modify require little energy to access and are not accessed very often. As such, significant overhead is not introduced in terms of power consumption.

PAMS	Closed form for N cores (bits)	N=16 (bits)
Loop iteration counters	32 x N	512
Bandwidth consumption counters	16 x N	256
Number of generated memory requests counters	16 x N	256
Past code-segment information registers	2 x 16 x N	512
Limiter thread bit-vector	N	16
Total storage required for PAMS	97 x N	1552

Table 1: Hardware storage cost of PAMS

Limiter Estimation: In Algorithm 1, to keep track of the total time all threads spend waiting on lock l in an interval, we modify the runtime system (i.e., the threading library) to perform the following: When any thread attempts to acquire lock l, a timestamp of this event is recorded locally. Once lock l is successfully acquired by some thread i, the runtime system adds the waiting time for that thread (obtained by subtracting the recorded timestamp for thread i from the current time) to the waiting time counter of lock l. Note that the waiting time counter for lock l is protected by the lock itself as it is only modified by a thread once that thread has successfully acquired the lock.

The overhead of the runtime limiter estimation described in Algorithm 1 is insignificant as it does not occur very often. In our evaluations we empirically determine *LimiterEstimationInterval* to be equal to five. Among our benchmarks, *hist* has the highest frequency of lock acquires, averaging one lock acquire every 37k cycles. Assuming sixteen locks are being tracked, the limiter estimation algorithm incurs the latency of sorting sixteen waiting times (each a 32-bit value) once every 185k cycles. A back-of-the-envelope calculation shows that this latency adds an overhead of less than 1% (even for the benchmark that has the most frequent lock acquires).

Alternative Hardware-Based Limiter Estimation: Even though the overhead of tracking total waiting time for each lock in the runtime system is very small in our implementation and evaluation, it could become more significant in the context of a locking library that is highly-optimized for fine-grain synchronization and when there is high lock contention. An alternative implementation of our proposal could track waiting time in hardware to further reduce the overhead. Although we did not evaluate this alternative, we outline its general idea here. In this implementation, two new instructions delimit the beginning and the end of each thread's wait for a lock: *LOCK_WAIT_START* <*lock_address*> and *LOCK_WAIT_END* <*lock_address*>. Each instruction takes a lock address, and updates a centralized lock table after commit, i.e. off the critical path.

This table contains one entry for each lock which contains the current number of threads waiting on that lock (*num_wait*) and the associated cumulative waiting time (*wait_time*). *LOCK_WAIT_START* increments *num_wait*

Benchmark	Description	Input Set	Length (Instr.)	MPKI	Critical Sections	Barriers	Barrier Interval
hist	Histogram (Phoenix)	minis	50M	2.66	405	1	N/A
mg	Multigrid solver (NPB)	W	225M	4.07	0	300	201–501
cg	Conjugate gradient solver (NPB)	A	113M	22.26	256	60	31–91
is	Integer sort (NPB)	W	140M	17.32	112	25	1–26
bt	Block tridiagonal solver (NPB)	W	397M	6.45	0	310	171–481
ft	Fast fourier transform (NPB)	W	161M	5.41	16	5	21–26

Table 2: Benchmark summary

and *LOCK_WAIT_END* decrements *num_wait* for the specified lock. Periodically, the hardware increments *wait_time* by *num_wait*, and estimates the limiter by finding the lock with the the highest *wait_time* and storing its address in a $Lock_{longest}$ register associated with the lock table. Since *LOCK_WAIT_END* executes right before a thread starts the critical section, the instruction also compares the lock address with $Lock_{longest}$ and in case of a match, it reports the thread ID to the memory controller as the current owner of $Lock_{longest}$, and the memory controller prioritizes requests from this thread.

4. METHODOLOGY

Benchmarks: We use a selection of benchmarks from NAS Parallel Benchmarks (NPB 2.3) [5] and the *hist* benchmark from Phoenix [30]. For each NPB benchmark, we manually choose a representative execution interval delimited by global barriers (Table 2 lists the barriers used). We do this in order to simulate a tractable number of instructions with a large enough input set that will produce a meaningful number of memory requests. However, this was not possible for three of the NAS benchmarks *ep*, *lu*, and *sp*. This is because, with a large enough input set to exert pressure on memory, we were unable to pick a tractable execution interval. We run the *hist* benchmark to completion.

All benchmarks are compiled using the Intel C Compiler with the `-O3` option. Table 2 summarizes the benchmarks. The memory intensity values reported in this table are obtained from simulations on the system described by Table 3. The benchmarks we evaluate use Pthreads and OpenMP threading libraries. We modify the threading library to intercept library calls and detect locks. Also, we assume gang scheduling [28, 10] of threads where all the threads of a parallel application are concurrently scheduled to execute. As a result, thread preemption does not skew the threads' measured waiting times.

Processor Model: We use an in-house cycle-accurate x86 CMP simulator for our evaluation. We faithfully model cache coherence, port contention, queuing effects, bank conflicts, and other major memory system constraints. Table 3 shows the baseline configuration of each core and the shared resource configuration for the 16-core CMP system we use. Table 4 shows the parameter values used in our evaluations.

5. RESULTS AND ANALYSIS

We first present performance results for each of the 6 benchmarks on a 16-core system normalized to their performance on a system using an FR-FCFS memory scheduler. Figure 7 shows results for the following six configurations from left to right for each benchmark, with each succeeding configuration introducing only one new component to the previous configuration: 1) thread cluster memory scheduling (TCM) [17], which uses time-based classification of latency-sensitive vs. bandwidth-sensitive threads with time-based shuffling, 2) code-segment based classification of latency-sensitive vs. bandwidth-sensitive threads (Section 3.2.2) with time-based shuffling, 3) code-segment based classification of threads with code-segment based shuffling (Section 3.2.3), 4) limiter information based thread prioritization (Section 3.1.1) with code-segment based classification and code-segment based shuffling, 5) limiter information based thread prioritization with code-segment based classification and dynamic shuffling policy, and 6) the combination of all our proposed mechanisms (PAMS): limiter information based thread prioritization, code-segment based thread classification with dynamic shuffling policy, and loop progress measurement based thread prioritization (note that no configuration except for this last one takes into account loop progress information in barrier based synchronization, described in Section 3.1.2). We find that among all evaluated mechanisms, PAMS provides the best performance, reducing execution time by 16.7% compared to a system with FR-FCFS memory scheduling, and by 12.6% compared to TCM, a state-of-the-art memory scheduling technique. Several observations are in order:

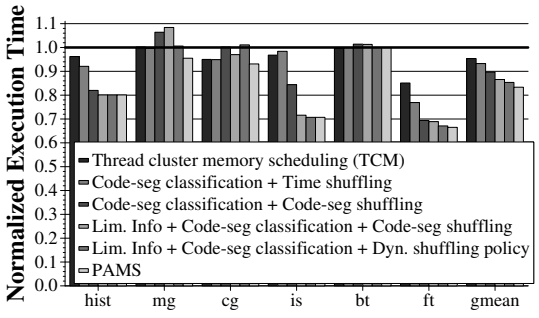


Figure 7: Overall results

1. Applying TCM, which is a memory scheduling technique primarily designed for improving system performance and fairness in multi-programmed workloads, to parallel applications improves average performance by 4.6%. This is because even though this technique does not consider interdependencies between threads, it still reduces inter-thread memory system interference, providing quicker service to threads (average memory latency reduces by 4.8%), thus enabling faster application progress.

2. Using code-segment based classification of latency-sensitive vs. bandwidth-sensitive threads (second bar from the left for each benchmark) as explained in Section 3.2.2 improves performance significantly compared to the time-based

Execution core	15-stage out of order processor, decode/retire up to 2 instructions Issue/execute up to 4 micro-instructions; 64-entry reorder buffer
Front end	Fetch up to 2 branches; 4K-entry BTB; 64K-entry hybrid branch predictor
On-chip caches	L1 I-cache: 32KB, 4-way, 2-cycle, 64B line ; L1 D-cache: 32KB, 4-way, 2-cycle, 64B line Shared unified L2: 4MB , 16-way, 16-bank, 20-cycle, 1 port, 64B line size
DRAM controller	On-chip, FR-FCFS [31, 33] scheduling 128-entry MSHR and memory request queue
DRAM and bus	667MHz bus cycle, DDR3 1333MHz [22] 8B-wide data bus, 8 DRAM banks, 16KB row buffer per bank Latency: 15-15-15ns (^tRP-^tRCD-CL), corresponds to 100-100-100 processor cycles Round-trip L2 miss latency: Row-buffer hit: 36ns, conflict: 51ns

Table 3: Baseline system configuration

LimiterEstimation Interval	TCM Time Quanta	TCM Shuffling Period	Time-based Period (also used within Dynamic Shuffling)	Instruction Sampling Period in Dynamic Shuffling
5	2M cycles	100k cycles	100k cycles	5k insts

Table 4: Parameters used in evaluation

classification done by TCM on two of the shown benchmarks (*hist* and *ft*). This is mainly because by using code-segments as interval delimiters to classify threads as latency- vs. bandwidth-sensitive (See Figure 3), we can make a more accurate classification of the thread's future memory behavior using information from the last time the starting code-segment executed.

3. When code-segment based shuffling is used instead of time-based shuffling (third bar from left, compared to second), performance improves significantly on three benchmarks (*hist*, *is*, and *ft*). This is primarily due to behavior shown in Figure 5. As explained in Section 3.2.3, when the group of concurrently executing threads have similar memory behavior, using code-segment based intervals for shuffling thread rankings outperforms time-based shuffling. On the other hand, in benchmarks *mg* and *cg*, execution time increases by as much as 6.8% (for *mg*) when code-segment based shuffling is used. This is because the threads have significantly different memory behavior, which can lead to performance degradation with code-segment based shuffling, as shown in Figure 6 (c). However, because of large improvements on *hist* (11%), *is* (14%), and *ft* (10%), average performance with code-segment based shuffling improves by 3.9% compared to time-based shuffling.

4. When limiter information is used to prioritize threads likely to be on the critical path (fourth bar from left), as described in Section 3.1.1, further benefits can be gained on applications that have contended locks. This can be seen in benchmarks such as *hist* and *is*. In these applications (one of which we will analyze in detail in a case study in Section 5.1), memory requests from limiter threads estimated by the runtime system are prioritized over non-limiter threads' requests, resulting in further execution time reduction. Note that when limiter information is used (in the three rightmost bars of Figure 7), latency- vs. bandwidth-sensitive classification of threads is performed only for limiter threads (as described by Algorithm 3 in Section 3.2.2).

5. Using the dynamic shuffling policy described in Section 3.2.3 (fifth bar for each benchmark) mitigates the performance loss seen due to code-segment based shuffling on benchmarks that have threads with different memory behavior, such as *mg* and *cg*. The dynamic shuffling policy monitors the memory intensity of concurrently executing threads and dynamically chooses code-segment based shuffling (when threads have similar intensities) or time-based shuffling (when threads have different intensities). With our dynamic shuffling policy, time-based shuffling is used for 74% and 52% of the time on *mg* and *cg* respectively.

6. *mg* and *cg* are also the benchmarks that benefit the most from prioritization of lagging threads enabled by loop progress measurement. This is expected since parallel `for` loops dominate the execution time of both benchmarks. In fact, *mg* and *cg* have very few critical sections, leaving loop progress measurement as the only way to estimate the critical path. Hence, performance of both benchmarks improves the most when loop progress measurement is enabled (4.5% and 6.9% over FR-FCFS, respectively).

5.1 Case Study

To provide insight into the dynamics of our mechanisms, we use the benchmark *is*, which has a combination of barriers and critical sections, as a case study. This benchmark performs a bucket sort, each iteration of which consists of two phases: counting the integers belonging to each bucket and partially computing the starting index of each integer in the sorted integer array. The first phase is done in parallel; the second, however, modifies a shared array of partial results and hence requires a critical section. Figures 8(a)–(d) show *thread activity* plots generated by running *is* on the following configurations: a baseline system with an FR-FCFS memory controller, a system with TCM [17], a system that uses code-segment based shuffling and code-segment based classification of latency-sensitive vs. bandwidth-sensitive threads, and finally a system using our proposed PAMS.[6]

In each *thread activity* plot shown in Figure 8, each thread's execution is split into three different states (as indicated by the legend on top of the figure): non-critical section execution (normal line), critical section execution (bold line), and waiting for a lock or barrier (dotted line). Vertical lines represent barriers where all threads synchronize.

Several observations are in order: First, by using TCM [17], overall inter-thread interference is reduced compared to a baseline system with FR-FCFS, resulting in 3% reduction in execution time. This is mainly due to the reduction in execution time when threads are executing the

[6]The total amount of work performed is the same across the different configurations. Due to space constraints, we only show the first 7 of the total 14 barrier intervals that were executed. Observing the total execution of 14 barrier intervals yields the exact same insights described in this section. For graphs with all 14 barriers, see our technical report [8].

non-critical section code that comes right after each barrier. This happens due to TCM's shuffling of priorities between the threads on time-based intervals, which leads to relatively similar improvement in the execution of all threads.

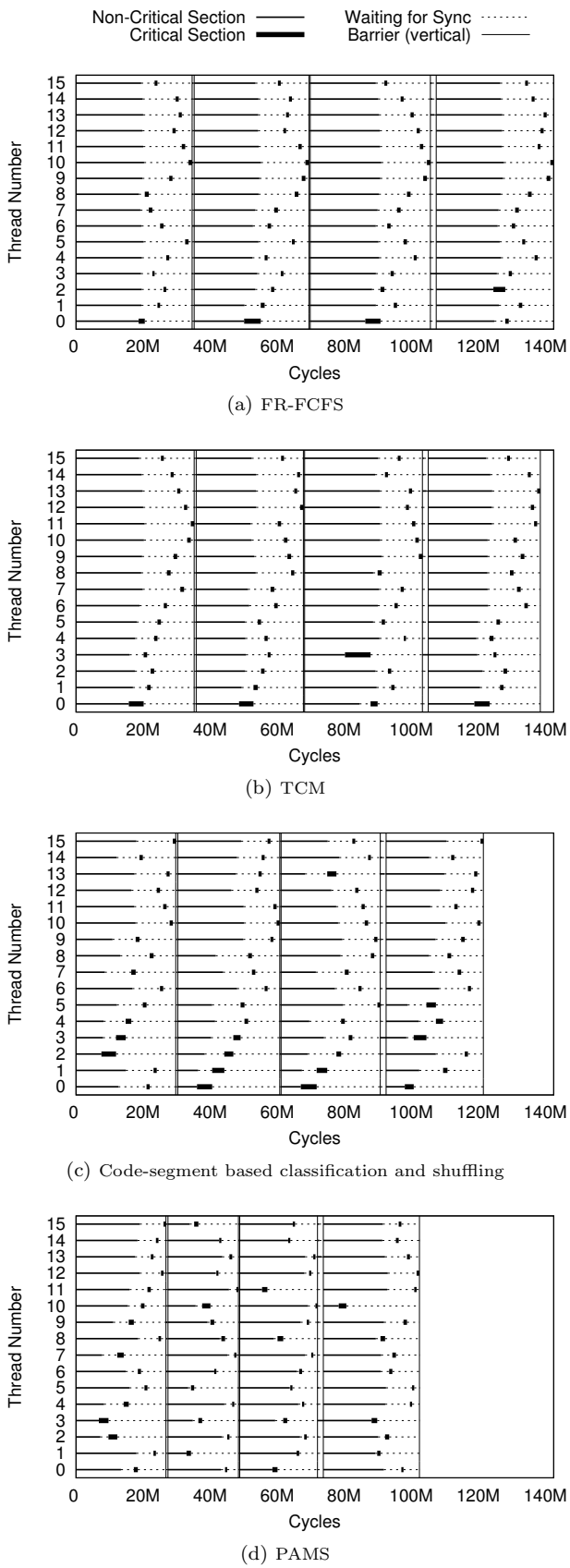
Second, performance can be significantly improved by using the code-segment based thread classification and shuffling that we propose in Sections 3.2.2 and 3.2.3 respectively. Figure 8(c) is a good real benchmark example of the behavior shown in Figure 5. Comparing the intervals between each pair of barriers across Figures 8(c) and (b) clearly shows the benefits of code-segment based shuffling vs. time-based shuffling in a benchmark where parallel threads executing non-critical section code have similar memory behavior.

By keeping an assigned ranking constant until a code-segment change happens (which triggers the end of an interval and the assignment of a new ranking across threads) three benefits occur: 1) when a prioritized thread reaches the barrier, it starts waiting and stops interfering with other threads enabling their faster progress (as explained in Section 3.2.3), 2) with time-based shuffling all threads reach the point where they attempt to acquire the lock at a similar time resulting in high contention and waiting for the lock. Code-segment based shuffling reduces this lock contention: with it, accesses to the critical section are spread over time and the first thread to reach the lock acquire in each barrier interval gets to that point earlier than with time-based shuffling (as seen in Figure 8(c)), and 3) code-segment based shuffling enables some threads to reach the critical section earlier than others as opposed to all threads reaching it at the same time (the latter happens in Figures 8(a) and (b)). This leads to the overlapping of the critical section latency with the execution of non-critical section code, and ultimately a reduction in critical path of execution. As a result of these three major benefits, using code-segment based shuffling reduces execution time by 15.6% and 12.8% compared to the FR-FCFS baseline and TCM respectively.

Finally, adding limiter information detected by the runtime system can significantly improve performance when combined with code-segment based classification and shuffling. Consider those critical sections that are part of the critical path in Figure 8(c). As this figure shows, some threads enter their critical section early while other threads are still executing non-critical section code. Hence, memory requests from threads executing *non-critical* code can interfere with memory requests of the *critical* thread. However, by prioritizing memory requests from the thread identified as critical by the runtime system (Section 3.1), PAMS reduces the total time spent in the critical section by 29% compared to code-segment based classification and shuffling without limiter thread information (as shown by the improvement in Figure 8(d) compared to (c)). Overall, PAMS improves execution time by 28.4% and 26% compared to the FR-FCFS baseline and TCM respectively.

5.2 Comparison to Memory Scheduling Using Thread Criticality Predictors [1]

Bhattacharjee and Martonosi [1] propose thread criticality predictors (TCP) to predict thread criticality based on memory hierarchy statistics. Although they do not demonstrate how their thread criticality predictor can be used for reducing inter-thread interference in the memory system, they do mention that it can be used in the design of memory controllers. We implement a memory scheduling technique



(a) FR-FCFS

(b) TCM

(c) Code-segment based classification and shuffling

(d) PAMS

Figure 8: Execution of *is* benchmark with different memory scheduling techniques

based on the information TCP provides as a comparison point to PAMS. TCP uses L1 and L2 cache miss counts and the penalty incurred by such misses to determine a criticality count for a thread, defined in [1] as:

$$N(Crit.Count.) = N(L1miss) + \frac{LLCpenalty\ .\ N(LLCmiss)}{L1penalty}$$

In the TCP-based memory scheduling technique we developed, the criticality of each thread is obtained once every 100k cycles, and a set of rankings is assigned to threads based on their criticality. Threads with higher estimated criticality are given higher priority for that interval. At the end of each interval, thread criticalities are re-evaluated and a new set of priorities are assigned for the next interval. As Table 5 shows, we find that our technique, PAMS, outperforms this TCP-based memory scheduler by 6.6% on average. PAMS outperforms TCP significantly on three of the benchmarks. This improvement is mainly due to the following which TCP does not address: 1) PAMS uses information about the multi-threaded application such as lock contention and loop progress to estimate thread criticality, and 2) PAMS also addresses how to schedule requests of non-critical threads (e.g., shuffling of non-limiter bandwidth-sensitive threads). As such, the TCP idea is orthogonal to some of our proposals and could be used within our PAMS proposal as part of the basis for predicting critical/limiter threads, which we leave to future work.

Benchmark name	hist	mg	cg	is	bt	ft	gmean
Δ Execution time	-9.9%	-15%	-9.8%	-1.3%	0.2%	-2.5%	-6.6%

Table 5: Reduction in execution time of PAMS compared to TCP-based [1] memory scheduling

5.3 Sensitivity to System Parameters

Table 6 shows how PAMS performs compared to FR-FCFS and TCM on systems with 8MB/16MB shared last level caches or two/four independent memory channels. Even though using a larger cache or multiple memory channels reduces interference in main memory, PAMS still provides significantly higher performance than both previous schedulers. We conclude that our mechanism provides performance benefits even on more costly systems with higher memory bandwidth or larger caches.

Channels	LLC	Δ wrt FR-FCFS	Δ wrt TCM
Single	4MB	-16.7%	-12.6%
Single	8MB	-15.9%	-13.4%
Single	16MB	-10.5%	-5.0%
Dual	4MB	-11.6%	-10.0%
Quad	4MB	-10.4%	-8.9%

Table 6: Sensitivity of PAMS performance benefits to memory system parameters

6. RELATED WORK

To our knowledge, this paper is the first to design a memory controller that manages inter-thread memory system interference to improve parallel application performance. Our major contribution is a new memory scheduling technique that takes into account inter-dependencies introduced by common parallel program constructs (locks and barriers) to make memory request scheduling decisions that reduce the length of the critical path of the application.

We have already provided extensive quantitative and qualitative comparisons to a state-of-the-art memory scheduling technique, TCM [17], which is designed for high system performance and fairness on multi-programmed workloads. We have also discussed how a state-of-the-art critical thread prediction mechanism [1] can be used in designing a memory controller and compared our proposal to it quantitatively and qualitatively in Section 5.2. Here, we briefly discuss other related work in memory controllers, shared resource management for parallel applications, and critical path prediction of parallel applications.

6.1 Memory Controllers

Many prior works have focused on management and reduction of inter-application memory system interference by better scheduling of memory requests at the memory controller [27, 25, 26, 23, 24, 29, 16, 17, 9]. However, like TCM, all of these techniques focus on multi-programmed workloads and do not consider the inter-dependencies between threads in their prioritization decisions. Ipek et al. [12] propose a memory controller that uses reinforcement learning to dynamically optimize its scheduling policy, which improves performance of parallel applications. Their technique observes the system state and estimates long-term performance impact of different actions, but does not explicitly take into account thread criticality or loop progress. In contrast to our PAMS design, this technique requires relatively more complex black box implementation of reinforcement learning in hardware. Lin et al. propose hierarchical memory scheduling for multimedia MPSoCs [19]. This design reduces interference between requests coming from different IP blocks of the SoC working on the same application by applying the parallelism-aware batch scheduling mechanism of Mutlu and Moscibroda [26]. As such, it does not take into account the inter-dependencies of parallel applications that PAMS takes into account to shorten the critical path and only attempts to fairly service different request streams from different IP blocks.

Thread-unaware memory scheduling policies (e.g., [33, 31, 21, 11]) are designed to improve DRAM throughput, but do not take into account inter-thread interference or thread criticality, leading to relatively low performance as we showed in comparison to FR-FCFS.

6.2 Shared Resource Management for Parallel Applications

Cheng et al. [4] propose throttling memory requests generated by threads in streaming parallel applications to reduce memory system interference. Their mechanism is a software-based approach that allows only an analytically determined number of threads to send out requests to memory at any given time to limit memory interference. Contrary to PAMS, which is not restricted to a particular programming model, their solution requires applications to be written in a gather-compute-scatter style of stream programming. Chen et al. [3] address inter-thread interference in shared caches as opposed to managing interference at the memory controller and propose a thread scheduling mechanism that aims to increase constructive cache sharing among threads of a parallel application. These two mechanisms are complementary to PAMS and can be combined with it to further improve parallel application performance.

6.3 Critical Thread Prediction

Cai et al. [2] propose a mechanism for dynamically detecting which threads in a parallel region are the slowest to reach a barrier (i.e., critical) by keeping track of the executed iteration counts of each thread. They use this loop progress measurement to delay non-critical threads to save energy, and to give higher priority to critical threads in the issue queue of an SMT core. We use a variant of their technique to identify and prioritize critical threads in barrier based programs as a component of our overall PAMS design as described in Section 3.1.2. Our evaluation in Section 5 shows that of the 16.7% performance gain of PAMS over the baseline, 2.3% is due to this optimization enabled by loop progress measurement (comparing the last two bars of Figure 7). As such, most of our proposals are orthogonal to this prior work. Other prior techniques exploit the idleness of threads that arrive early at a barrier to save power [18, 20], which Cai et al. [2] improve over.

7. CONCLUSION

We introduced the Parallel Application Memory Scheduler (PAMS), a new memory controller design that manages inter-thread memory interference in parallel applications to reduce the overall execution time. To achieve this, PAMS employs a hardware/software cooperative approach that consists of two new components. First, the runtime system estimates likely-critical threads due to lock-based and barrier-based synchronization using different mechanisms and conveys this information to the memory scheduler. Second, the memory scheduler 1) prioritizes the likely-critical threads' requests since they are the performance bottleneck, 2) periodically shuffles the priorities of non-likely-critical threads to reduce memory interference between them and enable their fast progress. To our knowledge, PAMS is the first memory controller design that explicitly aims to reduce inter-thread interference between inter-dependent threads of a parallel application.

Our experimental evaluations show that PAMS significantly improves parallel application performance, outperforming the best previous memory scheduler designed for multi-programmed workloads and a memory scheduler we devised that uses a previously-proposed thread criticality prediction mechanism to estimate and prioritize critical threads. We conclude that the principles used in the design of PAMS can be beneficial in designing memory controllers that enhance parallel application performance and hope our design inspires new approaches in managing inter-thread memory system interference in parallel applications.

ACKNOWLEDGMENTS

Many thanks to Veynu Narasiman, other HPS members, Carlos Villavieja, and the anonymous reviewers for their comments and suggestions. We also thank Aater Suleman for his help and insights in the early stages of this work. We gratefully acknowledge the support of the Cockrell Foundation, Intel, and Samsung. This research was partially supported by grants from the Gigascale Systems Research Center, the National Science Foundation (CAREER Award CCF-0953246), and the Intel Corporation ARO Memory Hierarchy Program. José A. Joao was supported by an Intel PhD Fellowship. Chris Fallin was supported by an NSF Graduate Research Fellowship.

REFERENCES

[1] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *ISCA*, 2009.
[2] Q. Cai et al. Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. In *PACT*, 2008.
[3] S. Chen et al. Scheduling threads for constructive cache sharing on CMPs. In *SPAA*, 2007.
[4] H.-Y. Cheng et al. Memory latency reduction via thread throttling. In *MICRO*, 2010.
[5] D. H. Bailey et al. NAS parallel benchmarks. Technical report, NASA Ames Research Center, 1994.
[6] E. Ebrahimi et al. Coordinated control of multiple prefetchers in multi-core systems. In *MICRO*, 2009.
[7] E. Ebrahimi et al. Fairness via source throttling: A configrable and high-performance fairness substrate for multi-core memory systems. In *ASPLOS*, 2010.
[8] E. Ebrahimi et al. Parallel application memory scheduling. Technical Report TR-HPS-2011-001, UT-Austin, Oct. 2011.
[9] E. Ebrahimi et al. Prefetch-aware shared resource management for multi-core systems. In *ISCA*, 2011.
[10] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *JPDC*, 16(4):306–318, 1992.
[11] I. Hur and C. Lin. Adaptive history-based memory scheduler. In *MICRO*, 2004.
[12] E. Ipek et al. Self-optimizing memory controllers: A reinforcement learning approach. In *MICRO*, 2008.
[13] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *ICS*, 2004.
[14] R. Iyer et al. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, 2007.
[15] S. Kim et al. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.
[16] Y. Kim et al. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*, 2010.
[17] Y. Kim et al. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *MICRO*, 2010.
[18] J. Li et al. The thrifty barrier: energy-aware synchronization in shared memory multiprocessors. 2004.
[19] Y.-J. Lin et al. Hierarchical memory scheduling for multimedia MPSoCs. In *ICCAD*, 2010.
[20] C. Liu et al. Exploiting barriers to optimize power consumption of CMPs. In *IPDPS*, 2005.
[21] S. A. McKee et al. Dynamic access ordering for streamed computations. *IEEE TC*, 49:1255–1271, Nov. 2000.
[22] Micron. *Datasheet: 2Gb DDR3 SDRAM, MT41J512M4 - 64 Meg x 4 x 8 banks, http://download.micron.com/pdf/datasheets/dram/ddr3.*
[23] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security*, 2007.
[24] T. Moscibroda and O. Mutlu. Distributed order scheduling and its application to multi-core DRAM controllers. In *PODC*, 2008.
[25] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO*, 2007.
[26] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA*, 2008.
[27] K. J. Nesbit et al. Fair queuing memory systems. In *MICRO*, 2006.
[28] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *IEEE Distributed Computer Systems*, 1982.
[29] N. Rafique et al. Effective management of DRAM bandwidth in multicore processors,.
[30] C. Ranger et al. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*, 2007.
[31] S. Rixner et al. Memory access scheduling. In *ISCA*, 2000.
[32] M. A. Suleman et al. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, 2009.
[33] W. Zuravleff and T. Robinbson. 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, 1997.