ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers

Yoongu Kim Dongsu Han Onur Mutlu Mor Harchol-Balter

Carnegie Mellon University

Abstract

Modern chip multiprocessor (CMP) systems employ multiple memory controllers to control access to main memory. The scheduling algorithm *employed by these memory controllers has a significant effect on system throughput, so choosing an efficient scheduling algorithm is important. The scheduling algorithm also needs to be scalable – as the number of cores increases, the number of memory controllers shared by the cores should also increase to provide sufficient bandwidth to feed the cores. Unfortunately, previous memory scheduling algorithms are inefficient with respect to system throughput and/or are designed for a single memory controller and do not scale well to multiple memory controllers, requiring significant fine-grained coordination among controllers.*

This paper proposes ATLAS (Adaptive per-Thread Least-Attained-Service memory scheduling), a fundamentally new memory scheduling technique that improves system throughput without requiring significant coordination among memory controllers. The key idea is to periodically order threads based on the service they have attained from the memory controllers so far, and prioritize those threads that have attained the least *service over others in each period. The idea of favoring threads with least-attained-service is borrowed from the queueing theory literature, where, in the context of a single-server queue it is known that least-attained-service optimally schedules jobs, assuming a Pareto (or any decreasing hazard rate) workload distribution. After verifying that our workloads have this characteristic, we show that our implementation of least-attained-service thread prioritization reduces the time the cores spend stalling and significantly improves system throughput. Furthermore, since the periods over which we accumulate the attained service are long, the controllers coordinate very infrequently to form the ordering of threads, thereby making ATLAS scalable to many controllers.*

We evaluate ATLAS on a wide variety of multiprogrammed SPEC 2006 workloads and systems with 4-32 cores and 1-16 memory controllers, and compare its performance to five previously proposed scheduling algorithms. Averaged over 32 workloads on a 24-core system with 4 controllers, ATLAS improves instruction throughput by 10.8%, and system throughput by 8.4%, compared to PAR-BS, the best previous CMP memory scheduling algorithm. ATLAS's performance benefit increases as the number of cores increases.

1. Introduction

In modern chip-multiprocessor (CMP) systems, main memory is a shared resource among the cores. Memory requests made by different cores interfere with each other in the main memory system, causing bank/row-buffer/bus conflicts [35] and serializing each core's otherwise parallel accesses in banks [36]. As the growth in the number of cores integrated on-chip far exceeds the growth in off-chip pin bandwidth [24], contention for main memory continues to increase, making main memory bandwidth one of the major bottlenecks in increasing overall system performance. If requests from different cores are not properly prioritized in the main memory system, overall system throughput can degrade and some cores can be denied service for long time periods [32].

The memory controller (MC) is the intermediary between the cores and main memory that prioritizes and schedules memory requests. Cutting-edge processors [21, 2, 20, 54] employ multiple memory controllers each of which controls a different portion (channel) of main memory. To provide large physical memory space to each core, each core can access the memory controlled by any of the memory controllers. As a result, different cores contend with each other in multiple controllers. Ideally, the scheduling algorithm employed by a memory controller in a multiple-memory-controller system should have three properties: *(i)* the MC should maximize system performance (throughput) without starving any cores, *(ii)* the MC should be able to communicate with the system software, enforcing the thread priority structure dictated by the system software, allowing the implementation of the QoS/fairness policies, *(iii)* the MC should be scalable to a large number of memory controllers: its implementation should not require significant coordination and information exchange between different controllers to provide high system performance.

Unfortunately, no previous scheduling algorithm satisfies all these requirements. Some existing scheduling algorithms [46, 30, 45] do not require coordination among multiple controllers, but result in low system performance (throughput) in multi-core systems. Other scheduling algorithms [35, 36] provide higher system throughput and are configurable, but they require significant coordination to achieve these benefits when implemented in a multiple-memory-controller system. As we show in this paper, the best previous algorithm in terms of system throughput, parallelism-aware batch scheduling (PAR-BS) [36], is not scalable because it requires significant coordination (information exchange) between different memory controllers that may be located far apart on the chip. Coordination is important because controllers need to agree on a consistent ranking of threads to ensure threads are serviced in the same order in each controller. This is needed to preserve bank-level parallelism of each thread and thereby ensure high system throughput. Since thread ranking is computed at the beginning of every batch of requests (approximately every 2000 cycles), either per-thread information within each controller needs to be *frequently* broadcast to all controllers, or a global meta-controller needs to be designed that frequently gathers thread information from each controller, computes a thread ranking, and broadcasts the ranking to all controllers. Neither option is scalable to a large number of controllers. Ideally, we would like memory controllers to exchange as little information as possible when scheduling memory requests because coordination incurs additional hardware complexity and power consumption costs, especially in a large scale many-core system with potentially tens/hundreds of memory controllers.

Our goal in this paper is to design a configurable memory scheduling algorithm that provides the highest system throughput without requiring significant coordination between controllers. To this end, we develop a fundamentally new approach to memory scheduling, called ATLAS (Adaptive per-Thread Least-Attained-Service memory scheduling).

Key Ideas and Basic Operation ATLAS is based on two key principles: *Least-Attained-Service (LAS) based thread ranking* to maximize system throughput, and a *long time quantum* to provide scalability. The basic mechanism is as follows. Execution time is divided into long time intervals or periods, called *quanta*. During each quantum, controllers keep track of how much service each thread has attained from the memory system. At the beginning of a quantum, controllers coordinate to determine a consistent ranking of threads, where threads that have attained the least service from the memory

controllers so far are ranked highest. During the course of a quantum, every controller uses this ranking to prioritize higher-ranked threads' requests (i.e. threads that have attained the least service) over other requests when making scheduling decisions. The idea of favoring threads with least-attained-service is borrowed from the queueing theory literature, where, in the context of a single-server queue it is known that least-attained-service optimally schedules jobs whose size (service requirement) is unknown, assuming a Pareto (or any decreasing hazard rate) workload distribution. After verifying that our workloads have these characteristics, we show that our implementation of least-attained-service thread prioritization reduces the time the cores spend stalling and significantly improves system throughput. Thread ranking also enables ATLAS to ensure each thread's bank-level parallelism is preserved, thereby preserving single-thread performance. In addition to high system throughput, ATLAS achieves 1) high scalability to a large number of memory controllers because long quanta ensure that information exchange between controllers is very infrequent, 2) starvation-freedom by using *thresholding*, which forces the servicing of a request that has been outstanding for too long.

Results Our extensive experimental evaluation shows that ATLAS provides the highest system throughput compared to five previous memory schedulers in both single-controller and multiple-controller systems. Compared to the best previous memory scheduling algorithm, PAR-BS, ATLAS improves system throughput by 8.4% on a 24-core system with 4-MCs and by 10.8% on a 32-core system with 4 MCs for a diverse set of multiprogrammed workloads. We show that ATLAS requires significantly less coordination between memory controllers than PAR-BS and is therefore more scalable. We describe the reasons for performance and scalability benefits and compare our work extensively (both qualitatively and quantitatively) to five previous memory scheduling algorithms on a wide variety of systems with 4-32 cores and 1-16 memory controllers.

Contributions We make the following new contributions:

- We show that coordination across multiple memory controllers is important to make good scheduling decisions that maximize system performance, and that frequent coordination hinders scalability of memory request scheduling algorithms. We propose a novel, scalable, high-performance memory request scheduling algorithm that significantly reduces the amount of coordination needed between controllers. By monitoring thread behavior over long periods of time (quanta), our proposed scheme performs well even in the absence of coordination.
- We introduce the concept of *Least-Attained-Service (LAS) based memory request scheduling* to maximize system throughput. We analyze the characteristics of a large number of workloads in terms of memory access behavior, and, based on this analysis, provide a theoretical basis for why LAS scheduling improves system throughput within the context of memory request scheduling.
- We qualitatively and quantitatively compare the ATLAS scheduler to five previously proposed schedulers and show that it provides the best system throughput. ATLAS is also more scalable than the best previously-proposed memory access scheduler, PAR-BS, in that it does not require frequent and large information exchange between multiple memory controllers.

2. Background and Motivation

2.1. Background on CMP Memory Scheduling

Long-latency memory accesses are a significant performance limiter in modern systems. When an instruction misses in the last-level cache and needs to access memory, the processor soon stalls once its instruction window becomes full [25, 34]. This problem becomes more severe when multiple cores/threads[1] share the memory system. Since cores' memory requests interfere with each other in the memory controllers and DRAM banks/buses/row-buffers, each core's request experiences additional delay, thereby increasing the time spent by the core stalling. To maximize system performance, memory scheduling algorithms need to minimize the total time cores spend stalling by controlling inter-core interference.

Parallelism-aware batch scheduling (PAR-BS) [36] is the best previous CMP memory scheduling algorithm that attempts to minimize the average stall time by scheduling the thread with the shortest stall-time first at a given instant during execution. Since we compare to it extensively, we briefly describe its operation. PAR-BS operates using two principles. First, it forms a batch of requests among the outstanding ones in the DRAM request buffers and prioritizes that batch over all other requests to prevent starvation. Second, when a batch is formed, it forms a ranking of threads based on their estimated stall time. The thread with the shortest queue of memory requests (number of memory requests to any bank) is heuristically considered to be the thread with the shortest stall-time (i.e. the shorter job) and is ranked higher than others. By servicing higher-ranked threads first within a batch, PAR-BS aims to 1) improve system throughput and 2) preserve the bank-level-parallelism of each thread, thereby preserving the benefits of latency tolerance techniques. PAR-BS was shown to provide the highest system performance compared to a wide variety of memory scheduling algorithms using a single memory controller. Unfortunately, PAR-BS's, as well as several other scheduling algorithms', scalability is limited with multiple memory controllers, as we show below.

2.2. Need for Coordination between Multiple Memory Controllers in Previous Schedulers

Modern multi-core systems employ multiple memory controllers. While previous proposals for memory scheduling algorithms were focused on reducing inter-thread interference in a single MC, it is now necessary to do the same for multiple MCs. The key aspect that distinguishes a multiple-MC system from a single-MC system is the need for coordination among MCs. Without coordination, each MC is oblivious of others and prone to make locally greedy scheduling decisions that conflict with other MCs' decisions [33]. Specifically, coordination is defined to be the exchange of information between MCs so that they agree upon and make globally beneficial (instead of purely local) scheduling decisions.

Figure 1 illustrates the need for coordination by comparing the performance of two uncoordinated MCs against two coordinated MCs. We assume that the MCs each implement PAR-BS, but later present analysis for other scheduling algorithms in Section 4. For illustration purposes, we will assume that a thread can continue computation only when all of its memory requests are serviced.[2] First, consider the case where the two MCs are uncoordinated, shown in Figure 1 (left). Each controller forms a ranking of threads based purely on information about its own DRAM request buffers. From the perspective of MC0, Thread 0 and Thread 1 have queue lengths of one and two memory requests, respectively. Therefore, according to the heuristic used by PAR-BS, Thread 0 is deemed to be the shorter job and is scheduled first. While this may have been a good decision if MC0 was the only controller in the system, MC0 neglects the fact that Thread 0's queue length in MC1 is larger and hence Thread 0 is actually the longer job from the viewpoint of the entire memory system. As a result, both Thread 0 and Thread 1 experience three bank access latencies until all their memory requests are serviced.

In contrast, if the two MCs were coordinated (i.e. aware of the queue lengths of each thread in each other's request buffers), MC0 would realize that Thread 0 has a queue length of 3 requests in MC1, and therefore no matter what it does, Thread 0 would experience three bank access latencies. Therefore, MC0 would rank Thread 1 higher and service it first even though it has a larger queue length than Thread 0 in MC0's buffers. Doing so reduces Thread 1's stall time to two bank access latencies without affecting Thread 0's stall

[1] Without loss of generality, we will assume one core can execute one thread, and use the terms core and thread interchangeably.

[2] For simplicity and to ease understanding, this diagram abstracts many details of the DRAM system, such as data bus conflicts and the row buffer. Our evaluations model the DRAM system faithfully with all bus/bank/row-buffer conflicts, queueing delays, and timing constraints.

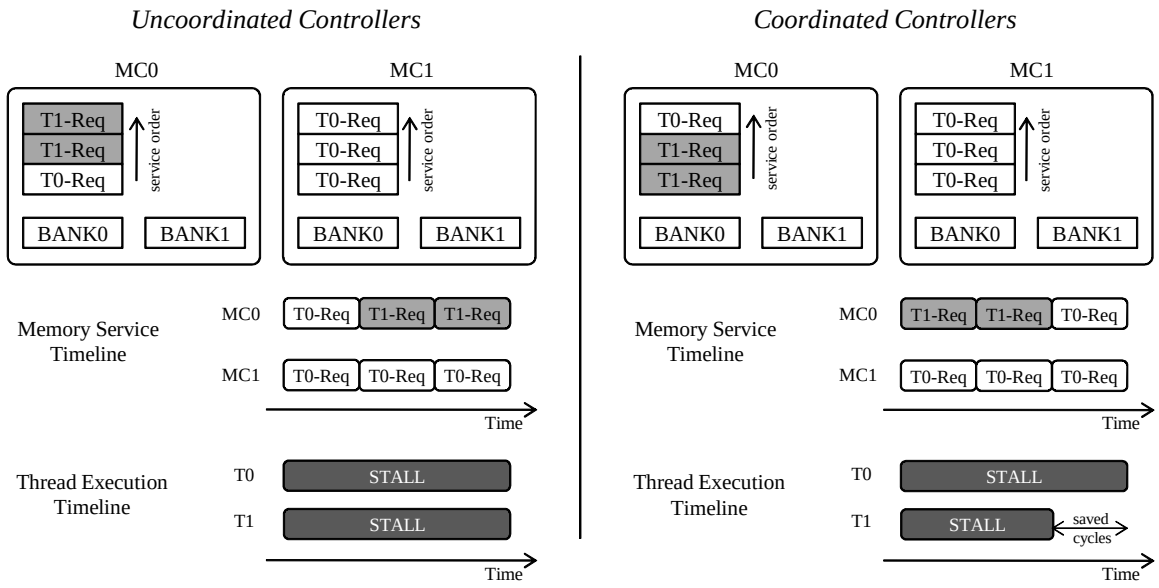


Figure 1. Conceptual example showing the importance of coordinating the actions of multiple memory controllers

time, leading to an overall improvement in system performance, as shown in Figure 1 (right).

The critical observation is that, when the controllers are aware of each others' state, they can take coordinated actions that can improve system performance. However, coordination does not come for free. There are two ways to achieve coordination: 1) a centralized meta-controller can collect information from all controllers, determine a thread ranking, and broadcast this ranking to all controllers, 2) each controller sends its information to every other controller, and each controller independently computes the same ranking based on this information. In either case, *for the scheduling algorithm to be scalable to tens/hundreds of controllers, it is preferable to communicate a small amount of information, as infrequently as possible between the controllers*. If communication is frequent, the pressure on the links connecting the controllers increases and other traffic needing these links can be delayed. Furthermore, the on-chip network may be unable to provide latency guarantees to support frequent exchange of information. This particularly pertains to cases where MCs are placed far apart on the die and the on-chip network distances between them are large [1].

For example, in PAR-BS, at the end of every batch, each controller needs to send two pieces of its local information to either a centralized meta-controller or to every other controller: each thread's 1) maximum number of requests to any bank, 2) total number of requests. With N threads and M controllers, the amount of information to be sent to a centralized meta-controller (and the amount of information the meta-controller sends back) is on the order of $O(N \cdot M)$. All this information needs to be transmitted via the on-chip network, which requires time. Since batches are very short (the average batch length is 2000 cycles) and their length does not change significantly as N and M increases, PAR-BS quickly becomes unscalable as N and M increases: the time it takes to communicate information between controllers to determine the ranking quickly starts exceeding the length of the batch for which the ranking needs to be computed, leading to an ineffective algorithm.

Our goal in this work is to fundamentally re-design the memory scheduling algorithm such that it provides high system throughput yet requires little or no coordination among controllers and therefore scales well to multiple-memory-controller systems.

2.3. Background on the Pareto Distribution and LAS

Many empirical measurements of computer workloads have found that the service requirements of jobs follow a Pareto distribution. Examples include Unix process lifetimes [16], sizes of files transferred through the Web [9, 10], sizes of files stored in Unix file systems [23], durations of FTP transfers in the Internet [40], and CPU requirements for supercomputing jobs [48].

Mathematically, a Pareto distribution with parameter α is defined by:

$$Probability\{\text{Job size} > x\} = k \cdot x^{-\alpha}$$

where $k, \alpha > 0$ are constants. Practically, a Pareto distribution has 3 important characteristics [15]: *(i)* very high (or infinite) variability, *(ii)* the heavy-tailed property (also known as "mice and elephants"), whereby just 1% of the largest jobs (the elephants) comprise half the total load, and *(iii) decreasing hazard rate* (DHR), which roughly states that the longer a job has run so far, the longer it is expected to run. Thus, if a job has run for a short time so far, then it is expected to end soon, but the longer the job runs without completing, the longer it is expected to continue running. It is this DHR property that is exploited by LAS scheduling.

It is well-known that scheduling to favor jobs which will complete soonest – Shortest-Remaining-Processing-Time (SRPT) – is optimal for minimizing latency [47] since it minimizes the number of jobs in the system. However in many situations, including that in this paper, the job's size (service requirement) is not known a priori. Fortunately, if the job sizes follow a distribution with DHR, then favoring those jobs which have received the least service so far is equivalent to favoring jobs which are expected to complete soonest. The Least-Attained-Service (LAS) scheduling policy prioritizes those jobs which have attained the least service so far. LAS is provably optimal under job size distributions with DHR, and unknown job sizes [44]. The LAS policy has been applied in various settings, most notably flow scheduling, where the flow duration is not known a priori, but prioritizing towards flows which have transmitted the fewest packets so far ends up favoring short flows (via the DHR property of flow durations) [42, 5, 49].

3. Mechanism

This section builds step-by-step the basic ideas and notions that are used to arrive at the ATLAS scheduling algorithm, which satisfies our goals of high system throughput with little coordination needed between the controllers. Section 3.1 provides the final resulting algorithm and describes its qualitative properties.

Motivating Ideas During its life cycle, a thread alternates between two episodes as shown in Figure 2: 1) *memory episode*, where

the thread is waiting for at least one memory request,[3] 2) *compute episode*, where there are no memory requests by the thread. Instruction throughput (Instructions Per Cycle) is high during the compute episode, but low during the memory episode. When a thread is in its memory episode, it is waiting for at least one memory request to be serviced and, as mentioned previously, is likely to be stalled, degrading core utilization. Therefore, to maximize system throughput, *our goal in designing a scalable memory scheduling algorithm is to minimize the time threads spend in their memory episodes.*

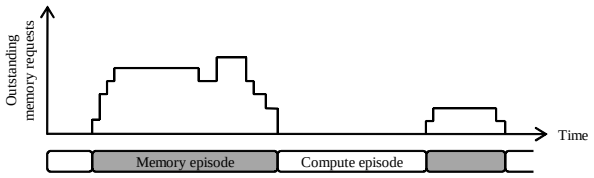


Figure 2. Memory vs. compute episodes in a thread's execution time

Shortest-Remaining-Memory-Episode Scheduling To minimize the time threads spend in memory episodes, we want to prioritize the thread whose memory episode will end soonest. By prioritizing any other thread in the memory controller (i.e., one whose memory episode will end later), or by prioritizing none, all threads experience prolonged memory episodes and contribute very little to overall system throughput. Prioritizing the thread whose memory episode will complete soonest is reminiscent of SRPT (Shortest-Remaining-Processing-Time) scheduling, which is provably optimal for job scheduling in a single-server queue, see Section 2.3.

How to Predict Which Memory Episode will End Soonest Unfortunately, the MC does not know which thread's memory episode will end soonest. The lengths of memory episodes are not known a priori and are hard to predict because a thread may initially have a few outstanding memory requests, but may continuously generate more requests as soon as some are serviced and, eventually, turn out to have a very long memory episode. But, what the MC does know is the attained service of an episode. *Attained service* is defined as the total amount of memory service (in cycles) that a memory episode has received since it started, see Figure 3.

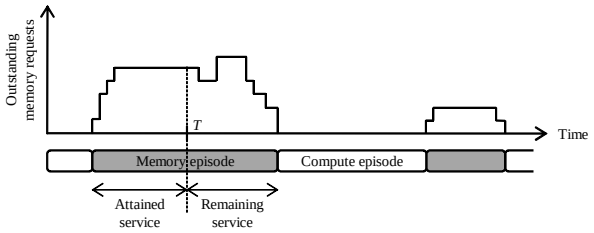


Figure 3. Attained service versus remaining service within a memory episode, viewed at time T

The *key point* that we exploit is that the *attained* service of a memory episode is an excellent predictor of the *remaining length (service)* of the memory episode, *if the memory episode lengths follow a distribution with decreasing hazard rate (DHR)*. Specifically, as explained in Section 2.3, under DHR, the longer the attained service of a given memory episode, the longer its expected remaining service will be. Thus, under DHR, favoring episodes with least attained service (LAS scheduling) will result in favoring memory episodes which will end soonest.

Fortunately, our measurements show that memory episode lengths indeed follow a distribution with DHR, namely a Pareto distribution. We collected statistics on memory episode lengths across tens of SPEC 2006 workloads and found that memory episode lengths consistently follow a Pareto (DHR) distribution. Figure 4 shows the distribution of memory episode lengths for three representative SPEC 2006 applications, soplex, bzip2, and calculix. The Pareto distribution is obvious from the linear fit on the log-log scale. Note that the R^2 value is very high, indicating that the Pareto distribution is indeed a good fit. We found that 26 of the 29 SPEC 2006 benchmarks have Pareto-distributed memory episode lengths.

Because of the DHR property of episode lengths, maximal throughput is achieved by a LAS policy which favors those threads with smallest attained memory episode time.

Taking Long-Term Thread Behavior into Account Favoring threads whose memory episodes will end soonest will certainly maximize system throughput in the short term. However it does not take *longer-term* thread behavior into account. The issue is that a thread does not consist of a single memory-plus-compute cycle, but rather many cycles, and different threads can have different long-term memory *intensities*, where the intensity denotes the long-run fraction of time spent in memory episodes. Consider an example where two threads, A and B, of differing memory-intensity are sharing the memory. Thread A is a highly *memory-intensive* thread with many short memory episodes and even shorter compute episodes in between the memory episodes. On the other hand, Thread B is a very *memory non-intensive* thread that has a few longer memory episodes, with very long compute episodes in between. If we performed simple LAS scheduling, Thread A would be prioritized and would reach its compute episode faster. However, since its compute episode is so short, soon afterwards it would go back to stalling in another memory episode, thereby hindering system throughput. On the other hand, if the memory scheduler prioritizes Thread B's episode, Thread B would reach its very long compute episode faster and afterwards it would not compete with Thread A in memory for a long time. Hence, it is clear that one would like to service Thread B first because doing so would result in very long stretches of compute episodes. This is in contrast to the short-term optimizations made by per-episode LAS scheduling.

To take into account both short-term and long-term optimizations, we generalize the notion of LAS to include a larger time interval than just a single episode. The ATLAS (Adaptive per-Thread LAS) memory controller divides time into large but fixed-length intervals called *quanta*. During each quantum, the memory controller tracks each thread's total attained service for that quantum. At the beginning of the next quantum, the memory controller *ranks* the threads based on their attained service in the past, weighting the attained service in the recent past quanta more heavily than the attained service in older quanta. Specifically, for any given thread, we define:

$$TotalAS_i = \alpha TotalAS_{i-1} + (1-\alpha)AS_i \quad (1)$$

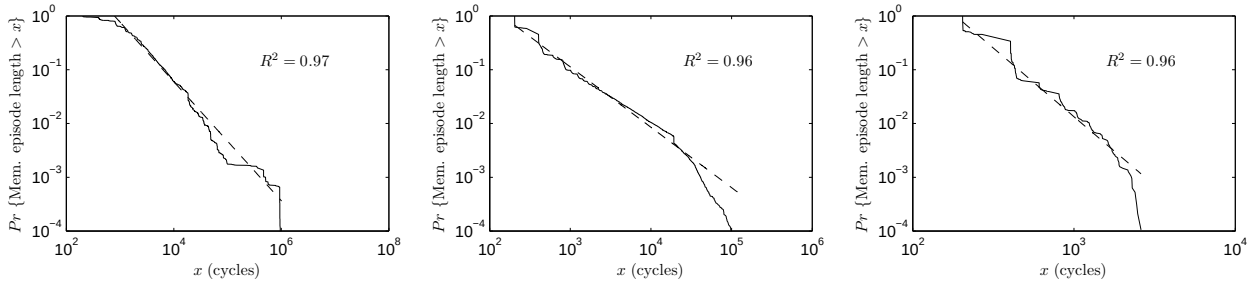
AS_i: Attained service during quantum i alone (reset at the beginning of a quantum)
$TotalAS_i$: Total attained service summed over all quanta up to the end of quantum i (reset at a context switch)

Here α is a parameter $0 \leq \alpha < 1$, where lower α indicates a stronger bias towards the most recent quantum. We generally use $\alpha = 0.875$, but evaluate the effect of different α in Section 7.3.

During the course of quantum $i+1$, the controller uses the above thread ranking based on $TotalAS_i$, favoring threads with lower $TotalAS_i$, that is lower total attained service. Observe that when using thread-based attained service, the memory-intensive Thread A gets served a lot at the beginning, since Thread B is idle. However, once Thread B starts a memory episode, Thread B has the lowest attained service, since Thread A has at this point already accumulated a lot of service. In fact, Thread B is able to complete its memory episode, entering a long compute episode, before Thread A is scheduled again.

Multiple Memory Controllers When there are multiple memory controllers, the attained service of a thread is the sum of the individual service it has received from each MC. As a result, at the beginning of a new quantum, each MC needs to coordinate with other MCs to determine a consistent thread ranking across all MCs. Controllers achieve this coordination by sending the local attained service of each thread to a centralized agent in the on-chip network that computes the global attained service of each thread, forms a ranking based on least-attained-service, and broadcasts the ranking back to each controller. Since the quantum is relatively long (as demon-

[3]There may be multiple outstanding requests by a thread due to the use of techniques that exploit memory-level parallelism, such as out-of-order execution, runahead execution, and non-blocking caches.



(a) 450.soplex (b) 401.bzip2 (c) 454.calculix

Figure 4. Pareto distribution of memory episode lengths in three applications

strated in our experimental results, Section 7.3), controllers need to coordinate very infrequently, making the approach scalable.

More Advantages of LAS-Based Thread Ranking LAS-based thread ranking provides two major benefits. First, as explained above, it maximizes system throughput within a quantum by minimizing the time spent in memory episodes. Second, it ensures that a thread's concurrent requests are serviced in parallel in the memory banks instead of being serialized due to inter-thread interference, thereby preserving the bank-level parallelism of each thread [36]. It is important to note that thread ranking does not imply that the memory requests of a lower-ranked thread are serviced only after those of higher-ranked threads have all been serviced. If a lower-ranked thread has a request to a bank where there are no higher-ranked thread's requests, then the lower-ranked thread's request is scheduled.

Guaranteeing Starvation Freedom Since the thread that was serviced least in the past is assigned the highest ranking for the next quantum, ATLAS ensures that no thread is left behind for too long. However, this non-starvation guarantee can take a while to take effect because the rankings are not updated very frequently. To provide a stricter non-starvation guarantee, ATLAS uses *thresholding*: when a memory request has not been serviced for T cycles after entering the memory controller, it is prioritized over all requests. We empirically find that a threshold value of $T = 100K$ cycles provides a reasonable non-starvation guarantee while retaining high system throughput (Section 7.3).

3.1. Putting It All Together: The ATLAS Algorithm

Rule 1 summarizes the ATLAS scheduling algorithm by showing the request prioritization rules employed by each memory controller when scheduling requests. Each controller maximizes system performance (via least-attained-service scheduling and exploiting row-buffer locality) and prevents starvation (via thresholding and updating attained service values at the end of each quantum).

Rule 2 describes the actions taken by controllers at the end of each quantum to determine a consistent LAS-based ranking among all threads. Note that each controller keeps only the AS value in the last quantum for each hardware thread. The meta-controller keeps the TotalAS value for each thread and performs the computation shown in Equation 1.

Rule 1 ATLAS: Request prioritization in each memory controller

1: **TH—Over-threshold-requests-first:** Requests that have been outstanding for more than T cycles in the memory controller are prioritized over all others (to prevent starvation for a long time).
2: **LAS—Higher-LAS-rank-thread-first:** Requests from threads with higher LAS-based-rank are prioritized over requests from threads with lower LAS-based rank (to maximize system performance and preserve per-thread bank-level parallelism).
3: **RH—Row-hit-first:** Row-hit requests are prioritized over row-conflict/closed requests (to exploit row buffer locality).
4: **FCFS—Oldest-first:** Older requests are prioritized over younger requests.

Rule 2 ATLAS: Coordination at the end of a quantum

1: Each MC sends each thread's local attained service (AS) in the last quantum to a central meta-controller. Afterwards, AS value is reset.
2: Meta-controller accumulates all local AS values for each thread and updates the TotalAS value of each thread according to Equation 1.
3: Meta-controller ranks threads based on TotalAS values; threads with lower TotalAS values are ranked higher.
4: Meta-controller broadcasts the ranking to all controllers.
5: Each MC updates thread ranks when it receives the broadcast ranking. New ranking is used for scheduling in the next quantum.

3.2. Support for System Software

So far, we described ATLAS assuming all threads are of equal importance. If this is the case, ATLAS aims to maximize system throughput by prioritizing memory non-intensive threads over others. However, this is not the right prioritization choice if some threads are more important (i.e. have higher *weight*, or *thread priority*) than others. To enforce thread weights, system software communicates thread weights to the ATLAS memory controller, as in [35, 36]. When updating the attained service of a thread, ATLAS scales the attained service with the weight of the thread as follows:

$$TotalAS_i = \alpha TotalAS_{i-1} + \frac{(1-\alpha)}{thread_weight} AS_i \qquad (2)$$

Observe that a thread with a higher weight appears as if it attained less service than it really did, causing such threads to be favored. Section 7.6 quantitatively evaluates ATLAS's support for thread weights.

In addition, we make the quantum length and starvation threshold (in terms of cycles) configurable by system software such that it can enforce a desired balance between system throughput and fairness.

4. Qualitative Comparison with Previous Scheduling Algorithms

We compare the basic properties of the ATLAS scheduler with major previous memory scheduling algorithms. The fundamental difference of ATLAS from all previous algorithms is that *no previous memory scheduling algorithm tries to prioritize threads by taking into account their long-term memory intensity*, aiming to prioritize memory non-intensive threads over memory-intensive threads in the long run. Previous memory scheduling algorithms have one or more of the following shortcomings: 1) their system throughput is low mainly because they perform locally greedy scheduling, 2) they require very high coordination between multiple MCs, 3) they depend on heuristics that become increasingly inaccurate as the number of MCs or the number of threads in the system increases.

First-come-first-serve (FCFS) services memory requests in arrival order. Therefore, unlike ATLAS, it does not distinguish between different threads or different memory episodes. FCFS inher-

Register	Description and Purpose	Size (additional bits)
Per-request registers in each controller		
Over-threshold	Whether or not the request is over threshold	1
Thread-rank	The thread rank associated with the request	$\log_2 NumThreads$ (=5)
Thread-ID	ID of the thread that generated the request	$\log_2 NumThreads$ (=5)
Register in each controller		
QuantumDuration	How many cycles left in the quantum	$\log_2 QuantumLength$ (=24)
Per-thread registers in each controller		
Local-AS	Local Attained Service of the thread in the controller	$\log_2 (QuantumLength \cdot NumBanks)$ (=26)
Per-thread registers in meta-controller		
TotalAS	Total Attained Service of the thread	$\log_2 (NumControllers \cdot QuantumLength \cdot NumBanks)$ (=28)

Table 1. Additional state (over FR-FCFS) required for a possible ATLAS implementation

ently favors memory-intensive threads because their requests naturally appear older to the memory controller as they arrive more frequently than requests of non-intensive threads. In addition, FCFS does not exploit row-buffer locality, nor does it preserve bank-level parallelism of threads across banks or controllers. As a result, it leads to low system throughput [38, 35, 36] even though it requires no coordination across controllers.

First-ready FCFS (FR-FCFS) [59, 46, 45] is commonly implemented in existing controllers. It prioritizes: 1) row-hit requests over all others, 2) older requests over younger ones. By prioritizing row-hit requests, it aims to maximize DRAM throughput, but as shown in previous work [35, 36, 32] this thread-unaware policy leads to low system throughput and starvation of threads with low row-buffer locality. FR-FCFS also shares the shortcomings of FCFS as described above.

Fair queueing memory scheduler (FQM) [38, 41] is based on the fair queueing algorithm [11] from computer networks. It attempts to partition memory bandwidth equally among threads. For each thread, in each bank, FQM keeps a counter called *virtual time* and increases this counter when a memory request of the thread is serviced. FQM prioritizes the thread with the earliest virtual time, trying to balance each thread's progress in each bank. As a result, FQM does not take into account the long-term memory intensity of threads and therefore cannot prioritize memory non-intensive threads over others. In addition, FQM 1) degrades bank-level parallelism of each thread because each bank makes scheduling decisions independently, 2) does not exploit row-buffer locality to a large extent, both of which lead to low system throughput compared to other scheduling algorithms [35, 36]. Since each bank acts independently in FQM, FQM does not require coordination across controllers, but this comes at the expense of relatively low system performance.

Stall-time fair memory scheduler (STFM) [35] estimates the slowdown of each thread compared to when it is run alone by quantifying the interference between threads. If unfairness in the memory controller is greater than a threshold, it prioritizes the thread that has been slowed down the most. STFM has three shortcomings compared to ATLAS. First, like FQM, it does not consider the long-term memory-intensity and does not preserve bank-level parallelism of threads. Second, it is designed for a single memory controller and requires extensive and very fine-grained coordination among controllers to be effective. To estimate a thread's slowdown, STFM keeps a per-thread interference counter, which is incremented by a certain amount each time another thread's request is scheduled instead of a request from this thread. The amount of increment depends on the estimate of bank-level parallelism of the thread in the memory system at that particular instant. In a multiple-MC system, MCs need to coordinate to determine this amount of increment since a thread's bank-level parallelism in the entire system cannot be locally known by a single-MC. This coordination is very fine-grained because it happens when a request is scheduled. Third, STFM quantifies memory interference using heuristics that are only approximations of the true value [35, 36]. Compared to a single-MC system, a multiple-MC system supports many more threads that interfere with each other in increasingly complex ways. As a result, the accuracy of estimated memory interference values (especially bank-level parallelism) degrades in a multiple-MC system (as shown for a multi-bank system in [36]).

Parallelism-aware batch scheduling (PAR-BS) [36] was described previously in Section 2.1. Compared to ATLAS, PAR-BS has three shortcomings. First, in contrast to the long quantum size in ATLAS, the batch duration in PAR-BS (on average ∼2000 cycles) is not large enough to capture the entirety of a memory episode. Therefore, a relatively short memory episode may be broken into pieces and serviced separately across batches, causing longer episodes (or memory-intensive threads) to be prioritized over shorter episodes (or non-intensive threads). Second, since the batch size is small, PAR-BS's coordination overhead to form a thread ranking at the end of each batch is very significant as described in Section 2.2 making it unscalable to a large number of controllers.[4] Third, PAR-BS's mechanisms become increasingly inaccurate as the number of threads and MCs in the system increases. When a thread has requests to multiple banks, the bank to which it has the most requests is considered by PAR-BS to dominate the memory latency of that thread. Therefore, to prioritize shorter threads within a batch, PAR-BS assigns the highest rank to the thread that has the lowest number of requests to any bank. In reality, the bank to which the thread has the most requests may service those requests more quickly than the other banks because other banks might be very heavily loaded with *many other less-intensive threads'* requests. As more threads compete for more MCs, such load imbalance across banks potentially grows and results in suboptimal thread ranking in PAR-BS.

5. Implementation and Hardware Cost

ATLAS requires the implementation of request prioritization rules in each controller (Rule 1) as well as coordination of controllers at the end of each quantum (Rule 2). Modern FR-FCFS-based controllers already implement request prioritization policies that take into account row-hit status and age of each request. ATLAS adds to them the consideration of the thread-rank and over-threshold status of a request. Whether a request is over-threshold is determined by comparing its age to threshold T, as done in some existing controllers [52] to prevent starvation.

To implement quanta, each memory controller keeps a quantum duration counter. To enable LAS-based thread ranking, each controller needs to maintain the local AS (attained service) value for each thread. AS for a thread is incremented every cycle by the number of banks that are servicing that thread's requests. At the end of a quantum, each thread's local AS value is sent to the meta-controller and then reset to zero. The meta-controller keeps the $TotalAS$ value for each thread and updates it as specified in Equation 1.

Table 1 shows the additional state information required by ATLAS. Assuming a 24-core CMP with 4 MCs, each of which with 128-entry request buffers and 4 banks, one meta-controller, and a quantum length of 10M cycles, the extra hardware state, including storage needed for Thread-IDs, required to implement ATLAS beyond FR-FCFS is 8,896 bits. None of the logic required by ATLAS is on the critical path of execution because the MC makes a decision only every DRAM cycle.

6. Evaluation Methodology

We evaluate ATLAS using an in-house cycle-precise x86 CMP simulator. The functional front-end of the simulator is based on Pin [28] and iDNA [4]. We model the memory system in detail,

[4] Note that there is no easy way to increase batch size in PAR-BS because thread ranking within a batch is formed based on the requests currently in the DRAM request buffers.

Processor pipeline	5 GHz processor, 128-entry instruction window (64-entry issue queue, 64-entry store queue), 12-stage pipeline
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, 2-cycle latency
L2 Caches	512 K-byte per core, 8-way set associative, 32-byte block size, 12-cycle latency, 32 MSHRs
Each DRAM controller (on-chip)	FR-FCFS; 128-entry request buffer, 64-entry write data buffer, reads prioritized over writes, XOR-based address-to-bank mapping [14, 57]
DRAM chip parameters	Micron DDR2-800 timing parameters (see [31]), 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 (dual in-line memory module) to provide a 64-bit wide channel to DRAM
Round-trip L2 miss latency	For a 64-byte cache line, uncontended: row-buffer hit: 40ns (200 cycles), closed: 60ns (300 cycles), conflict: 80ns (400 cycles)
Cores and DRAM controllers	4-32 cores, 1-16 independent DRAM controllers (1 controller has 6.4 GB/s peak DRAM bandwidth)

Table 2. Baseline CMP and memory system configuration

#	Benchmark	L2 MPKI	RB hit rate	Mem-fraction	Epi. length	#	Benchmark	L2 MPKI	RB hit rate	Mem-fraction	Epi. length
1	429.mcf	107.87	9.0%	99.2%	33512	14	401.bzip2	5.29	63.1%	52.3%	810
2	450.soplex	52.89	68.3%	99.4%	4553	15	464.h264ref	2.38	77.8%	35.9%	751
3	462.libquantum	52.88	98.4%	100.0%	$200M$	16	435.gromacs	2.14	55.6%	40.6%	498
4	459.GemsFDTD	49.72	11.7%	99.2%	1970	17	445.gobmk	0.73	45.8%	18.6%	483
5	470.lbm	45.68	48.5%	94.0%	1049	18	458.sjeng	0.38	1.8%	17.3%	437
6	437.leslie3d	32.38	42.6%	94.4%	2255	19	403.gcc	0.33	45.5%	6.2%	588
7	433.milc	27.90	62.7%	99.1%	1118	20	481.wrf	0.25	62.5%	10.5%	339
8	482.sphinx3	25.00	72.7%	96.6%	805	21	447.dealII	0.21	65.0%	6.3%	345
9	483.xalancbmk	23.25	42.8%	90.0%	809	22	444.namd	0.19	82.3%	4.6%	424
10	436.cactusADM	22.23	1.5%	86.3%	1276	23	400.perlbench	0.12	54.9%	2.1%	867
11	471.omnetpp	9.66	29.2%	82.0%	4607	24	454.calculix	0.10	67.0%	2.9%	334
12	473.astar	8.57	35.4%	86.5%	526	25	465.tonto	0.05	73.3%	0.9%	587
13	456.hmmer	5.63	32.4%	77.3%	815	26	453.povray	0.03	74.3%	0.7%	320

Table 3. Benchmark characteristics when run on the baseline (L2 MPKI: L2 Misses per 1000 Instructions; RB Hit Rate: Row-buffer hit rate; Mem-fraction: Fraction of execution time spent in memory episodes; Epi. length: Average length of memory episodes in cycles)

faithfully capturing bandwidth limitations, contention, and enforcing bank/port/channel/bus conflicts. Table 2 shows the major DRAM and processor parameters in the baseline configuration. Our main evaluations are done on a 24-core system with 4 memory controllers.

Workloads We use the SPEC CPU2006 benchmarks for evaluation.[5] We compiled each benchmark using gcc 4.1.2 with -O3 optimizations and chose a representative simulation phase using PinPoints [39]. Table 3 shows benchmark characteristics.

We classified the benchmarks into two categories: memory-intensive (those with greater than 10 L2 misses per 1000 instructions using the baseline L2 cache) and memory non-intensive (with less than 10 MPKI). We form multi-programmed workloads of varying heterogeneity. For our main evaluations on the 24-core system, we use 32 workloads, each comprising 24 benchmarks, of which 12 are memory-intensive and 12 are non-intensive. This provides a good mix of heavy and light applications in terms of memory behavior, that is likely to be a common mix in future many-core based systems, e.g. in cloud computing. Table 4 lists eight of the representative workloads that we show results for in Section 7. As we scale the number of cores (i.e. benchmarks in a workload), we keep the fraction of memory-intensive benchmarks constant (at 50%) in the workload. We use 32 workloads to evaluate each core-controller configuration. We also varied the fraction of memory-intensive benchmarks in each workload from 0%, 25%, 50%, 75%, 100% and constructed 32 workloads for each category to evaluate the effect of ATLAS with varying memory load, for the experiments described in Section 7.2.

Experiments and Metrics We run each simulation for 200 million cycles. During the simulation, if a benchmark finishes all of its representative instructions before others, its statistics are collected but it continues to execute so that it exerts pressure on the memory system. We use two main metrics to evaluate performance. *Instruction throughput* is the average number of instructions retired by all cores per cycle during the entire run. We measure *system throughput* using the commonly-employed *weighted speedup* metric [51], which sums the Instructions Per Cycle (IPC) slowdown experienced by each benchmark compared to when it is run alone for the same number of instructions as it executed in the multi-programmed workload:

$$Sys.\,Throughput = Weighted\,Speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}} \quad (3)$$

[5] 410.bwaves, 416.gamess, and 434.zeusmp are not included because we were not able to collect representative traces for them.

Parameters of Evaluated Schemes For ATLAS, we use a quantum length of 10M cycles, $\alpha = 0.875$, and $T = 100K$ cycles. For STFM and PAR-BS, we use the best parameters described in [35, 36].

7. Results

We first present the performance of ATLAS in comparison with four previously proposed scheduling algorithms (FCFS, FR-FCFS, STFM, and PAR-BS) implemented in each controller and an ideally-coordinated version of the PAR-BS algorithm (referred to as PAR-BSc) on a baseline with 24 cores and 4 memory controllers. In PAR-BSc, each controller is assumed to have instant global knowledge about the information available to every other controller, and based on this each controller determines a consistent ranking of threads at the beginning of a batch. As such, this algorithm is idealistic and un-implementable (because it assumes instantaneous global knowledge). However, since this algorithm provides an upper bound on what is achievable with the addition of coordination to previous scheduling algorithms, we provide comparisons to it in our evaluation for thoroughness.

Figure 5 shows the instruction and system throughput provided by the six algorithms on eight representative workloads and averaged over all 32 workloads run on the 24-core system with 4 memory controllers. On average, ATLAS provides 10.8% higher instruction throughput and 8.4% higher system throughput compared to the best previous implementable algorithm, PAR-BS, which significantly outperforms FCFS, FR-FCFS, and STFM. Even compared to the un-implementable, idealized coordinated PAR-BSc, ATLAS provides 7.3%/5.3% higher average instruction/system throughput. The performance improvement of ATLAS is consistent across all 32 workloads we studied. ATLAS's maximum and minimum system performance improvement over PAR-BS is 14.5% and 3.4%, respectively. Note that ATLAS improves average instruction/system throughput by 17.1%/13.2% over the commonly-implemented FR-FCFS algorithm. We conclude that ATLAS performs better than previous scheduling algorithms.

Figure 6 compares the instruction/system throughput of the six algorithms while varying the number of memory controllers from 1 to 16 in the 24-core system. Percentages on top of bars indicate ATLAS's improvement over the best previous implementable algorithm (STFM in single-MC and PAR-BS in multiple-MC systems). As expected, as the number of MCs increases, overall system performance also increases because the memory system expe-

Workload	Memory-intensive benchmarks	Memory non-intensive benchmarks
A	cactusADM, GemsFDTDs(2), lbm, leslie(2), mcf, milc, soplex(2), xalancbmk(2)	astar, bzip2, calculix(3), deall, gobmk(2), omnetpp(2), sjeng, tonto
B	GemsFDTDs, lbm, leslie, libquantum, mcf(2), milc, soplex(3), sphinx3(2)	astar(3), bzip2(2), calculix(2), dealII, hmmer, namd, povray, wrf
C	cactusADM, GemsFDTD(2), lbm(3), mcf, milc(2), soplex(2), sphinx3,	bzip2(2), dealII(2), gcc(2), h264ref, hmmer, perl, sjeng, tonto(2)
D	GemsFDTD(3), lbm, leslie, libquantum, milc, soplex, sphinx3(3), xalancbmk	bzip2, dealII, calculix, gobmk, gromacs, h264ref(2), hmmer, perl, povray, sjeng, wrf
E	GemsFDTD(2), leslie(2), mcf(2), soplex(2), sphinx3, xalancbmk(3)	astar(2), dealII, gcc(2), hmmer, namd(2), perl(2), wrf(2)
F	GemsFDTD, lbm, leslie(2), libquantum(3), mcf(2), milc, sphinx3, xalancbmk	astar(2), bzip2, dealII, gcc(3), gobmk, sjeng, tonto(2), wrf
G	cactusADM, lbm, leslie(4), libquantum , mcf, sphinx3(3), xalancbmk	astar, bzip2(2), gobmk, hmmer(3), omnetpp, perl(2), povray, tonto
H	cactusADM, GemsFDTD, libquantum, mcf(4), milc(2), sphinx3, xalancbmk(2)	astar(3), bzip2, gcc, hmmer, namd, sjeng, tonto(3), wrf

Table 4. Eight representative workloads evaluated (figure in parentheses denotes the number of instances spawned)

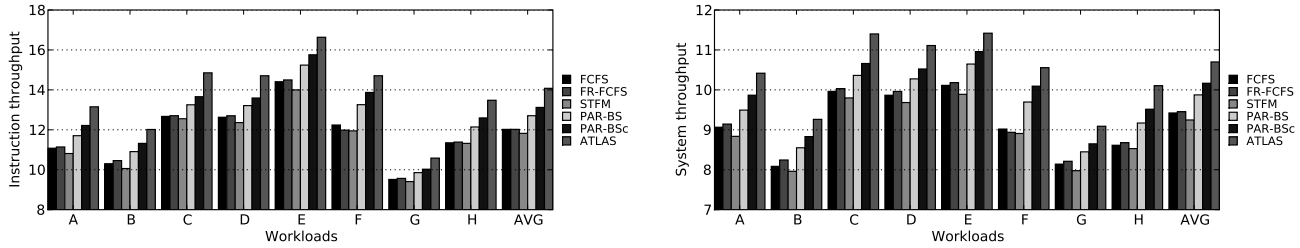


Figure 5. ATLAS vs. other algorithms on 8 sample workloads and averaged over 32 workloads on the 24-core system with 4-MCs

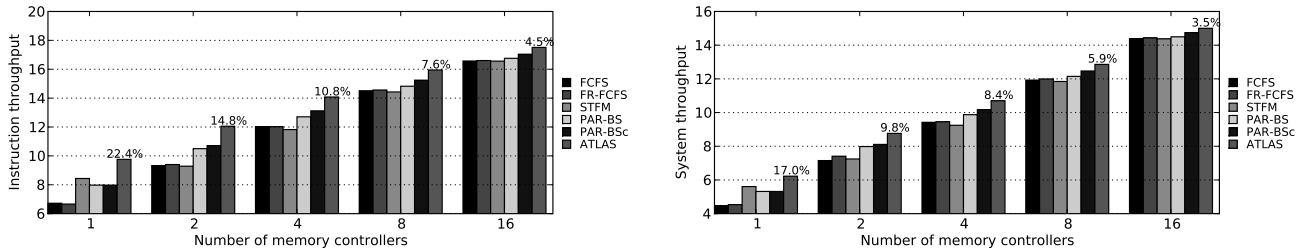


Figure 6. Performance of ATLAS vs. other algorithms on the 24-core system with varying number of memory controllers

riences lighter load per controller at the expense of increased system cost due to increased pin count. ATLAS provides the highest instruction/system throughput in all cases. However, ATLAS's performance advantage over PAR-BS increases when memory bandwidth becomes more scarce, i.e. MCs are more heavily loaded: ATLAS's instruction/system throughput improvement over PAR-BS is 14.8%/9.8% on a system with two MCs and 7.6%/5.9% on a costly system with 8 MCs. On a single-MC system, ATLAS performs 22.4%/17.0% better than STFM, the best previous algorithm for that system. We conclude that ATLAS is more effective than previous algorithms at utilizing the scarce memory bandwidth.

Analysis Several observations are in order from the above results:

First, as expected, FCFS and FR-FCFS policies, which are thread-unaware and prone to starvation, perform significantly worse than PAR-BS and ATLAS, which are thread-aware, starvation-free, and aimed to optimize system performance. Even though FR-FCFS maximizes DRAM throughput by exploiting row-buffer locality, its performance improvement over FCFS is small (0.3%), because always prioritizing row-hit requests over others leads to threads with high row-buffer locality to deny service to threads with low row-buffer locality, leading to degradations in system throughput.

Second, STFM, provides lower system performance than FR-FCFS or FCFS in multiple-MC systems even though it performs better than all algorithms but ATLAS in the single-MC system. This is because each MC implements the STFM algorithm by itself without any coordination. As a result, each MC computes a different slowdown value for each thread and aims to balance the slowdowns of threads locally, instead of all controllers computing a single slowdown value for each thread and consistently trying to balance thread slowdowns. Unfortunately, the slowdown a thread experiences in one controller is usually significantly different from that it experiences in another, which causes the MCs to make conflicting thread prioritization decisions. Section 4 shows that it is difficult to design a coordinated version of STFM as it incurs high coordination costs due to the amount of information that needs to be exchanged between controllers every DRAM cycle. Note that with a single controller, STFM outperforms FCFS, FR-FCFS, and PAR-BS by 25.2%, 23.6%, 5.4% on average, proving that the algorithm is effective for a single MC but not scalable to multiple MCs.

Third, the idealized coordinated version of PAR-BS (called PAR-BSc), which we have developed, outperforms the uncoordinated PAR-BS by 3.3%/3.0% on average in terms of instruction/system throughput because it ensures consistent thread prioritization across all controllers. However, both PAR-BS and PAR-BSc suffer from three shortcomings. First, they perform request batching at too fine a granularity compared to ATLAS (see Section 4). Second, their shortest stall-time-first heuristic used for thread ranking becomes increasingly inaccurate with a large number of threads (see Section 4). Third, since batches are formed based on request counts, batches are not balanced across controllers due to request count imbalance among controllers. This causes some controllers to opportunistically service requests while others are obeying thread ranking, leading to conflicting thread prioritization decisions among controllers.

7.1. Scalability with Cores and Memory Controllers

Figure 7 compares the performance of ATLAS with PAR-BS and the idealistic PAR-BSc when core count is varied from 4 to 32. We have results for FCFS, FR-FCFS, and STFM for all configurations, but do not show these due to space constraints. PAR-BS provides significantly better system performance than any of these algorithms for all configurations. The percentages on top of each configuration show the performance improvement of ATLAS over the implementable PAR-BS. Each graph presents system performance results averaged over 32 workloads on each core configuration with varying number of memory controllers; we did not tune ATLAS parameters in any configuration. We make several major observations. First, ATLAS outperforms all other algorithms for all configurations. Second, ATLAS's performance improvement over PAR-BS increases as the number of cores increases. For example, if the MC count is fixed to 4, ATLAS outperforms PAR-BS by 1.1%, 3.5%, 4.0%, 8.4%, 10.8% respectively in the 4, 8, 16, 24, and 32-core systems. Hence, ATLAS's benefits are likely to become more significant as core counts increase with each technology generation.

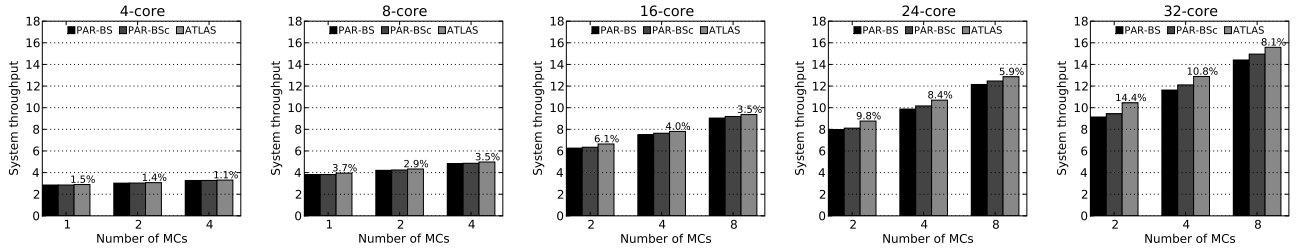


Figure 7. ATLAS performance with varying number of cores and memory controllers

7.2. Effect of Memory-Intensity in the Workload

Figure 8 compares ATLAS's performance to other scheduling algorithms on the 24-core 4-MC system as the number of memory-intensive benchmarks in each workload is varied between 0 and 24, which varies the load on the memory controllers. All benchmarks in an intensity class were selected randomly, and experiments were performed for 32 different workload mixes for each memory-intensity configuration.

Three major conclusions can be made from the results. First, ATLAS provides the highest system performance compared to the best previous algorithm (both PAR-BS and PAR-BSc) for all different types of workload mixes: respectively 3.5%, 6.1%, 8.4%, 7.7%, and 4.4% higher performance than PAR-BS for workload mixes with 0, 6, 12, 18, and 24 memory-intensive workloads. Second, ATLAS's performance improvement is highest when the workload mix is more heterogeneous in terms of memory intensity, e.g. when 12 or 18 of the 24 benchmarks in the workload are memory intensive. This is because scheduling decisions have more potential to improve system performance when workloads are more varied: ATLAS assigns the memory-intensive threads lower rankings and prevents the memory episodes of the other threads from being stuck behind longer ones. Therefore the majority of the threads are quickly serviced and returned to their compute episodes. Third, when a workload consists of all memory-intensive or all memory non-intensive threads, ATLAS's performance improvement is less pronounced because 1) the disparity between the memory episode lengths of threads is lower, 2) scheduling decisions have less of an impact on system performance due to too light or too heavy load. However, ATLAS is still able to distinguish and prioritize threads that are of lower memory-intensity and, hence, still outperforms previous scheduling algorithms in such workload mixes. We conclude that ATLAS is effective for a very wide variety of workloads but provides the highest improvements in more heterogeneous workload mixes, which will likely be the common composition of workloads run on future many-core systems.

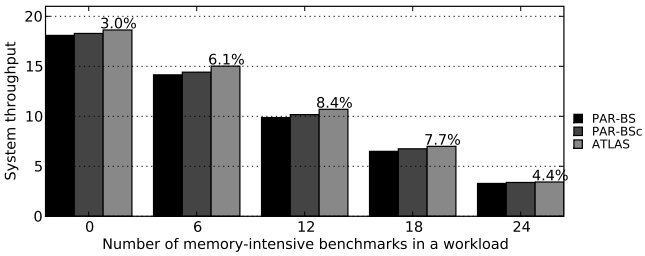


Figure 8. ATLAS performance on the 24-core 4-MC system when number of memory-intensive benchmarks in each workload is varied

7.3. Analysis of ATLAS

Effect of Coordination in ATLAS Figure 9 compares the performance of an uncoordinated version of ATLAS where each controller performs local LAS-based thread ranking based on local attained service values of each thread to the performance of (coordinated) ATLAS for 32 workloads on the baseline system. The results show that coordination provides 1.2%/0.8% instruction/system throughput improvement in ATLAS, suggesting that it is beneficial to have a LAS-based ranking that is consistent across controllers. However, we note that the uncoordinated version of ATLAS still provides respectively 7.5% and 4.5% higher performance than uncoordinated PAR-BS and ideally coordinated PAR-BSc. If the hardware overhead required for coordination is not desirable in a system, ATLAS can be implemented in each controller independently, which still yields significant system performance improvements over previous mechanisms.

ATLAS does not benefit from coordination as much as PAR-BS does because ATLAS uses a very long time quantum (10M cycles) compared to PAR-BS's short batches ($\sim$2K cycles). A long time quantum balances the fluctuation of load (and hence attained service) across different MCs and therefore the MCs are more likely than in PAR-BS to independently form a similar ranking of threads without coordinating. On the other hand, short batches in PAR-BS are vulnerable to short-term load imbalance of threads across multiple MCs: because threads' accesses to memory are bursty, one MC might have higher load for one thread while another has low load, leading to very different rankings without coordination. We conclude that long-time quanta reduce the need for coordination in ATLAS, but ATLAS still benefits from coordination.

ATLAS is a scheduling algorithm specifically designed to operate with minimal coordination among multiple memory controllers. Therefore, it is unsurprising that ATLAS shows a reduced benefit from coordination. This should not be used as a basis to discredit coordination for scheduling algorithms in general. Particularly, Figure 5 shows that an idealized coordinated version of PAR-BS provides 3.3%/3.0% gain in instruction/system throughput over uncoordinated PAR-BS.

Effect of Coordination Overhead We varied the time it takes to perform coordination (as described in Rule 2) between the 4 controllers in our baseline system from 100 to 10,000 cycles. We found that the system performance changed only negligibly with coordination overhead. This is due to two reasons: 1) coordination latency is negligible compared to the 10M-cycle quantum size, 2) the consistent ranking in the previous interval continues to be used until coordination completes.

Effect of Quantum Length Figure 10(a) shows the performance of ATLAS as quantum size is varied from 1K to 25M cycles, assuming there is no coordination overhead among controllers. A longer quantum has three advantages: 1) it reduces the coordination overhead among controllers, 2) it increases the probability that large memory episodes fit within a quantum, thereby enabling the prioritization of shorter episodes over longer ones, 3) it ensures thread ranking stays stable for a long time, thereby taking into account long-term memory-intensity behavior of threads in scheduling. On the other hand, a short quantum can capture more fine-grained changes in the phase behavior of workloads by more frequently changing the LAS-based ranking. Results show that system performance increases with quantum size because taking into account long-term thread behavior provides more opportunity to improve system throughput, until the quantum size becomes too large (25M cycles). Very small quantum sizes lead to very low performance because thread ranking changes too frequently, leading to degradations in both bank-level parallelism and ability to exploit long-term thread behavior.

Effect of History Weight (α) A large α ensures attained service values from prior quanta are retained longer and persistently affect the ranking for future quanta, causing TotalAS to change very slowly over time. Therefore, a large α allows ATLAS to better distinguish memory-intensive threads since it does not easily forget that they

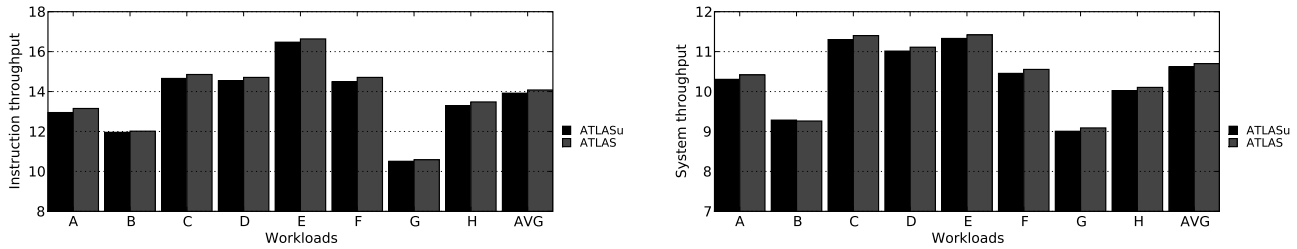
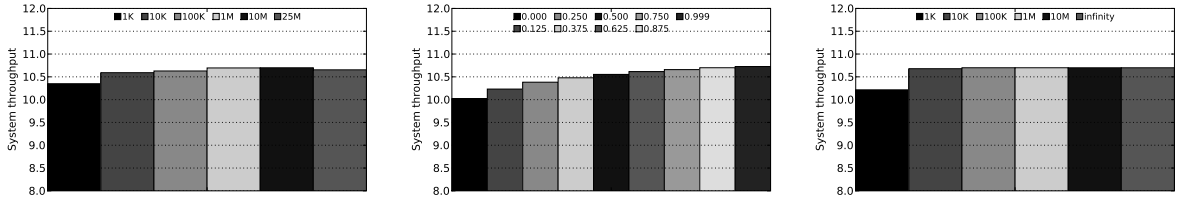


Figure 9. Performance of ATLAS with and without coordination



(a) Quantum length (b) History weight (c) Starvation threshold

Figure 10. ATLAS performance with varied ATLAS parameters

have amassed large amounts of attained service in the past. On the other hand, too large an α degrades the adaptivity of ATLAS to phase behavior for the exact same reasons. Even when a thread transitions into a memory-intensive phase and starts to accumulate significant amounts of attained service, its ranking will still remain high for subsequent quanta since its historical attained service is perceived to be low.

Figure 10(b) shows that system performance increases as α increases. In our benchmarks, we do not see degradation with very large α values because we found that per-thread memory intensity behavior stays relatively constant over long time periods. However, using a too-high α value can make the memory scheduling algorithm vulnerable to performance attacks by malicious programs that intentionally deny service to others by exploiting the fact that the algorithm is not adaptive and unfairly favors threads that have been memory non-intensive in the past. To prevent this, we choose $\alpha = 0.875$ which balances long-term attained service and the last quantum's attained service in our evaluations.

Effect of Starvation Threshold (T) Figure 10(c) shows system performance as the starvation threshold is varied from one thousand to infinite cycles. When T is too low, system performance degrades significantly because scheduling becomes similar to first-come-first-serve since many requests are forced to be serviced due to threshold violations and therefore the advantages of LAS-based ranking order is lost. When T becomes too large, the strict starvation-freedom guarantee is lost, but system performance stays very similar to our default threshold, $T = 100,000$. This is because LAS-based ranking by itself provides a way of reducing starvation because starved threads become highly-ranked since their attained service is smaller. However, we still use a starvation threshold, which can be configured by system software, to strictly guarantee non-starvation.

7.4. Effect of ATLAS on Fairness

We evaluate the fairness of ATLAS using two separate metrics: maximum slowdown and harmonic speedup [13]. The maximum slowdown of a workload is defined as the maximum of the inverse-speedups (i.e., slowdowns) of all comprising threads. For two algorithms that provide similar performance, the one that leads to a smaller maximum slowdown is more desirable. The harmonic speedup of a workload is defined as the harmonic mean of speedups of all comprising threads (higher harmonic-speedup correlates with higher fairness in providing speedup [29]). Figures 11 and 12 show that maximum slowdown increases by 20.3% and harmonic speedup decreases by 10.5%.

It is important to note that ATLAS is unfair to memory-intensive threads that are likely to be less affected by additional delay than non-intensive ones. While memory-intensive threads suffer with regard to fairness, overall system throughput increases significantly. For threads that require fairness guarantees, system software can ensure fairness by appropriately configuring the thread weights that ATLAS supports (Section 3.2).

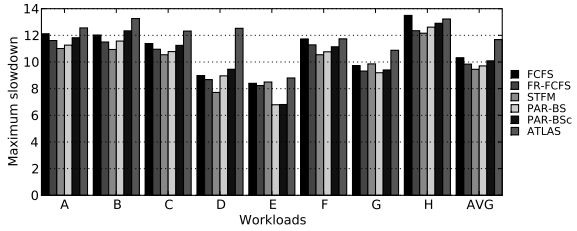


Figure 11. Max. slowdown of ATLAS vs. others on 24-core 4-MC system

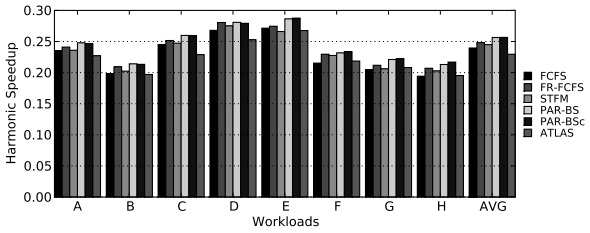


Figure 12. Harmonic speedup of ATLAS vs. others on 24-core 4-MC system

7.5. Sensitivity to System Parameters

Effect of Memory Address Mapping All results presented so far assume a cache-block-interleaved memory addressing scheme, where logically consecutive cache blocks are mapped to consecutive MCs. Although this addressing scheme is less likely to exploit row-buffer locality, it takes advantage of the higher memory bandwidth by maximizing bank-level parallelism. As a comparison, Figure 5 shows performance averaged over 32 workloads when rows (2 KB chunks), instead of blocks (64-byte chunks), are interleaved across MCs, thereby trading off bank-level parallelism in favor of row-buffer locality. Two conclusions are in order: 1) block-interleaving and row-interleaving perform very similarly in our benchmark set with PAR-BS, 2) ATLAS provides 3.5% performance improvement over PAR-BS, when row-interleaving is used, versus 8.4% when block-interleaving is used. ATLAS's performance benefit is higher with block-interleaving than with row-interleaving because ATLAS is able to exploit the higher bank-level parallelism provided by interleaved blocks better than PAR-BS for each thread due to the long quantum sizes. Since bank-level parallelism of each thread is low to begin with when row-interleaving is used, the potential for ATLAS to exploit bank-level parallelism for each thread is lower. We conclude that ATLAS is beneficial regardless of the memory address mapping scheme.

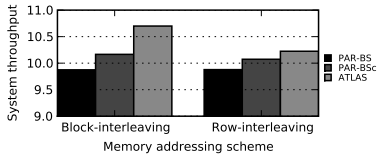


Figure 13. Performance of ATLAS with cache-block-interleaving vs. row-interleaving on the 24-core system with 4 MCs

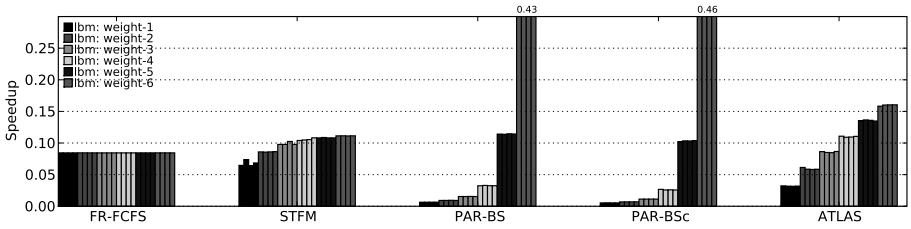


Figure 14. Evaluation of ATLAS vs. PAR-BS and STFM with different thread weights

	L2 Cache Size				t_{CL}				
	512 KB	1 MB	2 MB	4 MB	5ns	7.5ns	15ns	22.5ns	30ns
System Throughput Improvement over PAR-BS	8.4%	7.1%	6.0%	6.4%	4.4%	5.9%	8.4%	8.8%	10.2%

Table 5. Sensitivity of system performance of ATLAS to L2 cache size and main memory latency

Effect of Cache Size and Memory Latency Table 5 shows system performance improvement of ATLAS compared to PAR-BS as cache size and memory latency of the baseline system are independently varied. Results show that ATLAS significantly improves performance over a wide variety of cache sizes and memory latencies.

7.6. Evaluation of System Software Support

We evaluate how effectively ATLAS can enforce system-level thread weights to provide differentiated services. We run 24 copies of the lbm benchmark on our baseline 24-core 4-MC system, where there are six thread-weight classes each with 4 lbm copies. The priority classes respectively have weights 1, 2, 3, 4, 5, 6. Figure 14 shows the speedup experienced by each of the 24 lbm copies compared to when it is run alone. Since FCFS (not shown) and FR-FCFS are thread-unaware, they are unable to differentiate between lbm copies. With STFM, weights are not fully enforced–the weight is correlated with thread's speedup, but the relationship between weight and speedup is sub-linear. In fact, STFM has difficulty distinguishing between weights 4, 5, and 6 because controllers are not coordinated; a thread with higher weight can be better prioritized in one controller than in another. Both uncoordinated and idealized coordinated PAR-BS are able to enforce weights, but the relationship between weight and speedup is exponential. ATLAS provides a linear relationship between thread weight and speedup, which makes it easier to design system software algorithms because system software can reason about the speedup of a thread based on the weight it assigns to the thread.

8. Other Related Work

Memory Scheduling We have already qualitatively and/or quantitatively compared ATLAS to major previous scheduling algorithms (FCFS, FR-FCFS [59, 46, 45], FQM [38, 41], STFM [35], PAR-BS [36]) and shown that ATLAS provides significant performance and scalability benefits over them. Other scheduling algorithms [17, 55, 30, 56, 18, 37, 58, 50, 22] have been proposed to improve DRAM throughput in single-threaded, multi-threaded, or streaming systems. None of these works consider the existence of multiple competing threads sharing the memory controllers (as happens in a multi-core system).

Other Work on Memory Controllers Abts et al. [1] examined the placement of MCs in the on-chip network to improve performance predictability. Lee et al. [27] proposed adaptive prioritization policies between prefetches and demands in the memory controller for efficient bandwidth utilization. Several previous works [14, 43, 53, 57] examined different physical-to-bank address mapping mechanisms to improve performance obtained from the memory system. Other works [6, 37, 7, 8, 26, 12, 19, 3] have examined different memory controller design choices, such as row buffer open/closed policies and power management techniques. All of these mechanisms are orthogonal to memory scheduling and can be applied in conjunction with ATLAS.

9. Conclusions

We introduced ATLAS, a fundamentally new approach to designing a high-performance memory scheduling algorithm for chip-multiprocessor (CMP) systems that is scalable to a very large number of memory controllers. Previous memory scheduling algorithms either provide low CMP system throughput and/or are designed for a single memory controller and do not scale well to multiple memory controllers, requiring significant fine-grained coordination among controllers, as we demonstrate in this paper. ATLAS tracks long-term memory intensity of threads and uses this information to make thread prioritization decisions at coarse-grained intervals, thereby reducing the need for frequent coordination. To maximize system throughput, ATLAS leverages the idea of least-attained-service (LAS) based scheduling from queueing theory to prioritize between different threads sharing the main memory system. We analyze the characteristics of a large number of workloads in terms of memory access behavior, and, based on this analysis, provide a theoretical basis for why LAS scheduling improves system throughput within the context of memory request scheduling. To our knowledge, ATLAS is the first memory scheduling algorithm that provides very high system throughput while requiring very little or no coordination between memory controllers.

Our extensive evaluations across a very wide variety of workloads and system configurations show that ATLAS provides the highest system throughput compared to five previous memory scheduling algorithms on systems with both single and multiple memory controllers. ATLAS's performance benefit over the best previous controllers is robust across 4 to 32-core systems with 1 to 16 memory controllers. ATLAS's performance benefit increases as the number of cores increases. We conclude that ATLAS can be a flexible, scalable, and high-performance memory scheduling substrate for multi-core systems.

Acknowledgements

Yoongu Kim and Dongsu Han are supported by Ph.D. fellowships from KFAS (Korea Foundation for Advanced Studies). This research was partially supported by CyLab at Carnegie Mellon University and the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

References

[1] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti. Achieving predictable performance through better memory controller placement in many-core CMPs. In *ISCA-36*, 2009.

[2] Advanced Micro Devices. AMD's six-core Opteron processors. `http://techreport.com/articles.x/17005`, 2009.

[3] N. Aggarwal, J. F. Cantin, M. H. Lipasti, and J. E. Smith. Power-efficient DRAM speculation. In *HPCA-14*, 2008.

[4] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of programs. In *VEE*, 2006.

[5] E. W. Biersack, B. Schroeder, and G. Urvoy-Keller. Scheduling in practice. *Performance Evaluation Review, Special Issue on "New Perspectives in Scheduling"*, 34(4), March 2007.

[6] F. Briggs, M. Cekleov, K. Creta, M. Khare, S. Kulick, A. Kumar, L. P. Looi, C. Natarajan, S. Radhakrishnan, and L. Rankin. Intel 870: A building block for cost-effective, scalable servers. *IEEE Micro*, 22(2):36–47, 2002.

[7] F. Briggs, S. Chittor, and K. Cheng. Micro-architecture techniques in the Intel E8870 scalable memory controller. In *WMPI-3*, 2004.

[8] P. Conway and B. Hughes. The AMD Opteron northbridge architecture. *IEEE Micro*, 27(2):10–21, 2007.

[9] M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM TON*, 5(6):835–846, 1997.

[10] M. E. Crovella, M. S. Taqqu, and A. Bestavros. Heavy-tailed probability distributions in the world wide web. In *A Practical Guide To Heavy Tails*, chapter 1, pages 1–23. Chapman & Hall, New York, 1998.

[11] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, 1989.

[12] B. Diniz, D. O. G. Neto, W. Meira Jr., and R. Bianchini. Limiting the power consumption of main memory. In *ISCA-34*, 2007.

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

[14] J. M. Frailong, W. Jalby, and J. Lenfant. XOR-Schemes: A flexible data organization in parallel memories. In *ICPP*, 1985.

[15] M. Harchol-Balter. Task assignment with unknown duration. *J. ACM*, 49(2):260–288, March 2002.

[16] M. Harchol-Balter and A. Downey. Exploiting process lifetime distributions for dynamic load balancing. In *SIGMETRICS*, 1996.

[17] S. I. Hong, S. A. McKee, M. H. Salinas, R. H. Klenke, J. H. Aylor, and W. A. Wulf. Access order and effective bandwidth for streams on a direct rambus memory. In *HPCA-5*, 1999.

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

[19] I. Hur and C. Lin. A comprehensive approach to DRAM power management. In *HPCA-14*, 2008.

[20] IBM. PowerXCell 8i Processor. `http://www.ibm.com/technology/resources/technology_cell_pdf_PowerXCell_PB_7May2008_pub.pdf`.

[21] Intel. Intel Core i7 Processor. `http://www.intel.com/products/processor/corei7/specifications.htm`.

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

[23] G. Irlam. Unix file size survey - 1993. Available at `http://www.base.com/gordoni/ufs93.html`, September 1994.

[24] ITRS. International Technology Roadmap for Semiconductors, 2008 Update. `http://www.itrs.net/Links/2008ITRS/Update/2008Tables_FOCUS_B.xls`.

[25] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *WMPI-2*, 2002.

[26] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *ASPLOS-IX*, 2000.

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

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

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

[30] S. A. McKee, W. A. Wulf, J. H. Aylor, M. H. Salinas, R. H. Klenke, S. I. Hong, and D. A. B. Weikle. Dynamic access ordering for streamed computations. *IEEE TC*, 49(11):1255–1271, Nov. 2000.

[31] Micron. 1Gb DDR2 SDRAM Component: MT47H128M8HQ-25. May 2007. `http://download.micron.com/pdf/datasheets/dram/ddr2/1GbDDR2.pdf`.

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

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

[34] O. Mutlu, H. Kim, and Y. N. Patt. Efficient runahead execution: Power-efficient memory latency tolerance. *IEEE Micro*, 26(1):10–20, 2006.

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

[36] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA-36*, 2008.

[37] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI-3*, 2004.

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

[39] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.

[40] V. Paxson and S. Floyd. Wide-area traffic: The failure of Poisson modeling. *IEEE/ACM TON*, pages 226–244, June 1995.

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

[42] I. A. Rai, G. Urvoy-Keller, and E. W. Biersack. Analysis of LAS scheduling for job size distributions with high variance. In *SIGMETRICS*, 2003.

[43] B. R. Rau. Pseudo-randomly interleaved memory. In *ISCA-18*, 1991.

[44] R. Righter and J. Shanthikumar. Scheduling multiclass single server queueing systems to stochastically maximize the number of successful departures. *Probability in the Engineering and Information Sciences*, 3:967–978, 1989.

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

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

[47] L. E. Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16:678–690, 1968.

[48] B. Schroeder and M. Harchol-Balter. Evaluation of task assignment policies for supercomputing servers: The case for load unbalancing and fairness. *Cluster Computing: The Journal of Networks, Software Tools, and Applications*, 7(2):151–161, April 2004.

[49] A. Shaikh, J. Rexford, and K. G. Shin. Load-sensitive routing of long-lived IP flows. In *SIGCOMM*, 1999.

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

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

[52] Sun Microsystems. OpenSPARC T1 Microarchitecture Specification. `http://opensparc-t1.sunsource.net/specs/OpenSPARCT1_Micro_Arch.pdf`.

[53] M. Valero, T. Lang, J. M. Llabería, M. Peiron, E. Ayguadé, and J. J. Navarra. Increasing the number of strides for conflict-free vector access. In *ISCA-19*, 1992.

[54] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[55] C. Zhang and S. A. McKee. Hardware-only stream prefetching and dynamic access ordering. In *ICS*, 2000.

[56] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee. The impulse memory controller. *IEEE TC*, 50(11):1117–1132, Nov. 2001.

[57] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *MICRO-33*, 2000.

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

[59] 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.