LAMS: A Latency-Aware Memory Scheduling Policy for Modern DRAM Systems

Wenjie Liu[§], Ping Huang[†‡], Tang Kun[†‡], Tao Lu[‡], Ke Zhou[§✉], Chunhua Li[§] and Xubin He[†‡]
[§]*Wuhan National Lab for Optoelectronics, Huazhong University of Science and Technology, China*
[†]*Department of Computer and Information Sciences, Temple University, USA*
[‡]*Department of Electrical and Computer Engineering, Virginia Commonwealth University, USA*
{*lwj0012, k.zhou, li.chunhua*}*@hust.edu.cn*, {*templestorager, xubin.he*}*@temple.edu*, {*tangk2, lut2*}*@vcu.edu*

***Abstract*—This paper introduces a new memory scheduling policy called LAMS, which is inspired by a recently proposed memory architecture and targets for future high capacity memory systems. As memory capacity increases, the bit-lines connected to memory row buffers become much longer, dramatically lengthening memory access latency, due to increased parasitic capacitance. Recent study has proposed to partition long bit-lines into near and far (relative to the row buffer) segments via inserting isolation transistors such that access to near segment can be accomplished much faster, while access to far segment remains nearly the same. However, how to effectively leverage the new memory architecture still remains unexplored. We suggest to take advantage of this new memory architecture via performing latency-aware memory scheduling for pending requests to explore their performance potentials. In this scheduling policy, each memory request is classified to one of the following three categories, *row-buffer hit*, *near-buffer*, and *far-buffer*. Based on the classification, it issues requests in the order of *row-buffer hit* → *near-buffer* → *far-buffer*. In doing so, it avoids long-latency requests blocking short-latency memory requests, reducing total memory queuing time in the memory controller and improving overall memory performance. Our evaluation results on a simulated memory system show that comparing with the commonly used FR-FCFS scheduler, our LAMS improves performance and energy efficiency by up to 20.6% and 34%, respectively. Even comparing with the four competitive schedulers chosen from memory scheduling champion (MSC), LAMS still improves performance and energy efficiency by up to 6.1% and 23.4%, respectively.**

1. Introduction

DRAM is an essential component of all computing systems. Memory performance is crucial to the overall system performance as it provides a buffer stage to bridge the huge gap between the performance required by the processor and the relatively stagnant operations of disk system and flash storage system. As cloud computing and web services become more and more prevalent, applications are becoming increasingly memory hungry, which imposes a lot of pressure on memory systems. With the advancement of memory technologies, DRAM capacity and bandwidth have dramatically improved across generations during the past decades [2], [3]. However, the memory latency remains almost constant across generations, exacerbating the performance gap between the memory system and the processor. This is known as "memory wall" problem. In fact, memory capacity increase can hurt memory latency. The primary reason is that as capacity increases, the bit-lines connected to the row buffer become longer in length and the associated parasitic capacitance also increases dramatically, making accessing DRAM cells more time-consuming [1], [3]. Extensive recent research efforts have been invested on mitigating the "memory wall" problem via reducing memory latency. Existing memory latencies reduction techniques focus on reducing latencies caused by DRAM refresh [4], [5], taking advantage of DRAM characteristic [6], [7], [8], [9], such as internal parallelized architecture, and passing cross-layer hints [10].

In this paper, we suggest a straightforward and effective method to improve memory performance based on the a recently proposed memory architecture, called *Tiered Latency (TL-DRAM)*. Based on the observations that the time spent on driving the bitlines dominates the memory access latency, *TL-DRAM* splits each bit-line into two distinct segments, *near* and *far*, by inserting an isolation transistor [1]. Data in *near* segment can be obtained quickly since short lines have smaller parasitic capacitance, while data in *far* segment takes slightly longer time to visit due to isolation capacitors. Therefore, depending on memory request addresses, the memory request latency exhibits asymmetries. We believe taking advantage of such memory access asymmetries can improve performance. Motivated by this observation, we suggest a latency-aware memory scheduling method (called *LAMS*) which prioritizes scheduling memory requests visiting *near* segments over memory requests visiting *far* segments so as to reduce the total request waiting time in the memory queue and as a result to improve overall performance. In this scheduling method, pending memory requests are classified into three types, namely, *row-buffer hit*, *near-buffer/near-segment*, and *far-buffer/far-segment*. *Row-buffer hit* requests are requests that address the currently opened rows, *near-buffer* requests are the requests that access the rows closer to row buffers, while *far-buffer* requests access rows far away from row buffers. Based on this classification, in every scheduling cycle, *LAMS* first issues the *Row-buffer hit* requests as they have the shortest latency, as it is also done in the commonly used *First Ready First Come First Service (FR-FCFS)* [11] memory scheduler. It then

978-1-5090-5252-3/16/$31.00

issues *near-buffer* requests, and finally *far-buffer* requests. Moreover, to avoid scheduling starvation, *LAMS* enforces a maximally allowed scheduling delay for requests. Once the scheduling delay is reached, the requests in question are given chances for scheduling in the order of their arriving time. To the best of our knowledge, our work is the first to consider the potential of leveraging new memory architecture at memory scheduling level to improve overall memory performance.

In summary, our work makes the following two main contributions:

- We devise a new memory scheduling policy that leverages the potentials provided by modern new memory architecture so that it can make informed scheduling decisions and therefore improve memory performance. Our scheduler is much more simpler that other prediction-based schedulers for achieving the same level of performance improvements. As memory technology advances, it is important to optimize upper level components in order to exploit the full potential.
- We conduct extensive evaluations to testify the efficacy of the new memory scheduling policy. Evaluation results show that it improves performance and reduces memory energy by an impressive degree, compared with four state-of-the-art memory schedulers from memory scheduling championship for the same workloads.

This rest of this paper is organized as follows. In Section 2, we give the background knowledge of our work. We proceed to elaborate on the details of our latency-aware memory scheduler in Section 3. To verify our idea, we conduct extensive experiments in Section 4. Following that, related work is given in Section 5 with some further discussion in Section 6. Finally, we conclude the paper in Section 7.

2. Background

2.1. DRAM Basics

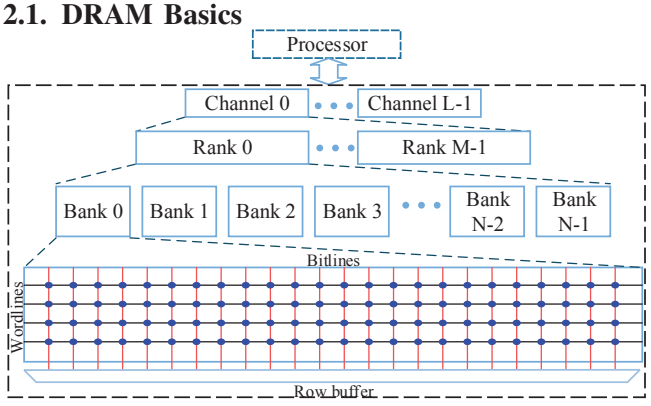


Figure 1: An overview of memory organization. Memory systems adopt a hierarchical structure comprising of *channel*, *rank*, *bank*, and *subarray*. The memory address is interpreted as a combination of bit components with each component representing a memory layer.

Modern memory systems are commonly built using dynamic random access memory (DRAM) technology. A DRAM cell consists of a capacitor and an access transistor and it is the basic storage unit in DRAM technology. As depicted in Figure 1, a DRAM chip consists of hundreds of thousands of cells which are organized in a hierarchical manner, namely, *subarray*, *bank*, *rank*, and *channel*, from the bottom to upper level. The memory address is decoded to channel, rank, bank and subarray and the decoded information is used to traverse the memory hierarchy to locate the corresponding data. A DRAM channel contains multiple ranks each of which has multiple banks (typically 4, 8, 16 banks). Each bank has a number of subarrays. A subarray is composed of cells organized as 2-dimensional arrays with rows and columns. DRAM cells in the same row are connected with each other via a word line and cells in the same column are connected via a bit-line. On each bit-line, there is a sense-amplifier which amplifies the weak charge signal represented by charges stored in DRAM cells on the bit-line. As all the cells on a bit-line share the same sense amplifier, at any time, only one of the DRAM cells is connected to the sense amplifier. The collection of bit-line sense amplifiers in a subarray forms a *row buffer*. When the voltage of a word line is raised to V_{dd}, all cells in the row controlled by the word line are connected the row buffer. The process is called memory row activation and the corresponding row is said to be in open state. To access a data block from memory, the row address (i.e., word line) is first asserted to put the whole row content to the row buffer. Then the column address is used to drive out the data block to on memory bus. Memory accessing is destructive, meaning once a row is activated it has to be written back before activating another row. Row buffer content is restored to the original row via *precharing*, known as page close.

When a memory request is received, depending on the request type, the memory controller creates a series of memory micro-operations which are specific to the deployed memory technology to satisfy the request. DRAM specifications have defined dozens of timing parameters that need to be observed by the micro-operations for system correctness. Basically, a memory service process follows three steps. First, an *ACT* command is issued to put the data of the requested rows into *row buffer* within a time window of t_{RCD} (called *row-column delay*). Second, the memory controller issues a column command to read or write the corresponding data block. After a period of time (t_{CL} for read and t_{CWL} for write), the data block is sent on the memory bus and it arrives at the processor after t_{BL} time. Third, the memory controller issues a *precharge* command which takes t_{RP} to prepare the row ready for the next activation. The row activation time t_{RAS} is defined as the sum of $t_{RCD} + t_{CL}/t_{CWL} + t_{BL}$. The row cycle time t_{RC} is defined as the sum of $t_{RAS} + t_{RP}$. The interval between two consecutive row activations should be at least t_{RC}. Lee et al [1] studied the various memory timing parameters. As activation and precharge operations are dependent on bit-lines' parasitic capacitance, shortening the length of bit-lines can reduce the parasitic capacitance and thus the corresponding t_{RCD} and t_{RP}. Splitting a bit-line into two segments by adding an isolation transistor creates a near row-buffer segment which has shorter wire length and less parasitic

capacitance [1], resulting in reduced memory access latency in the segment.

2.2. Memory Scheduling

Memory controllers in modern DRAM systems are becoming increasingly sophisticated to improve performance [10], [12], [13], [14]. One important functionality of memory controller is memory scheduling where a wide range of optimizations can be implemented, including improving row access locality [11], leveraging bank-level parallelism [15], [16], avoiding interference [17], [18], [19], etc. Without a proper memory scheduler, the disarrayed memory requests could result in low row buffer hit rate and poor bank-level parallelism, leading to bad system performance. Especially, unmanaged memory request queue may lead to low-performance and is prone to starvation problem when multiple competing threads share the memory controller in general-purpose multicore/multi-threaded systems. To avoid memory system performance degradation, a decent memory scheduler is needed.

Existing works mainly take advantage of either the source of memory requests or the application characteristic to make scheduling decisions to improve memory performance. First Ready First Come First Served (FR-FCFS) [11] scheduler gives priority to requests that address the currently opened row, as doing so can shorten memory latency. However, as memory capacity increases, it becomes more difficult to leverage row buffer locality. When a memory system contains more banks, the amount of requests visiting a bank is reduced, decreasing the possibility of row buffer hit. Memory scheduler can also learn application characteristics for a period of time and then based on the learned information it makes informed scheduling decisions for further requests [12], [17]. The commonality of the existing works on memory schedulers is that they all "look up" to get information about request patterns or application characteristics. We propose to "look down" at the underlying memory architecture to exploit potentials at the architectural level. Our proposal is more lightweight and orthogonal to those existing memory scheduler works.

3. Latency-aware Memory Scheduling

In this section, we discuss the details of latency-aware memory scheduling built upon the tiered latency memory architecture [1], which provides asymmetric memory latencies for memory requests depending on their addresses, i.e., decoded row number from the address.

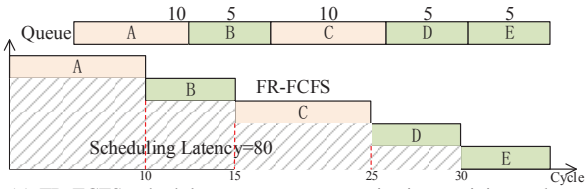
3.1. Variable Memory Latencies

As discussed in Section 2.1, tiered latency memory architecture introduces an isolation transistor on each bit-line to create so-called near and far segments. The position of the isolation transistor is referenced to determine whether a row is in near segment or far segment. The isolation transistor is turned off when requests access the near segment. Due to reduced bit-line parasitic capacitance, near segment takes less time to drive the bit-lines, resulting in smaller t_{RCD} and t_{RC} time. Therefore, memory requests going to the near segment can be accomplished faster. Compared with conventional memory architecture, both t_{RCD} and t_{RC} are significantly reduced in the near segment. In the far segment, despite the latency of an additional isolation transistor, t_{RCD} is still reduced, while t_{RC} is slightly increased. For example, in a conventional memory architecture, the t_{RCD} and t_{RC} of bit-lines with 512 cells are $15ns$ and $52.5ns$, respectively. When bit-lines are split into a 128-cell near segment and a 384-cell far segment, the t_{RCD} goes down to $9.3ns$ and $13.2ns$ and the t_{RC} becomes $27.8ns$ and $64.1ns$ in the near and far segment, respectively. As memory capacity increases, long bit-lines can also be partitioned into multiple segments to create multiple latency tiers. Unfortunately, existing schedulers are agnostic to such latency asymmetry.

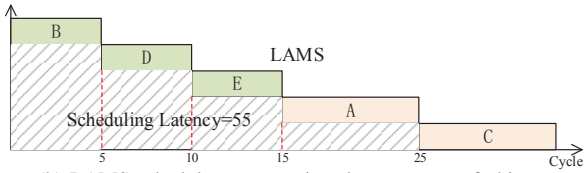
3.2. Latency-Aware Scheduling

Given the memory latency asymmetry, it is beneficial to make the memory scheduler be aware of differentiated request service time in order to improve overall memory performance. Knowing the service latency of each pending request, a memory scheduler can choose to prioritize scheduling low-latency requests over long-latency ones, similar in spirit to the shortest job first scheduling policy. The difference is that conventional shortest job first scheduling method may not have available information to estimate each job's service time, e.g., a CPU thread's execution time, while the memory scheduler can easily infer the latency of memory requests from their row addresses, as memory latency is related to the relative distance between the target row and the row buffer.

To enable latency aware scheduling, the memory scheduler classifies pending requests in the memory queue into three categories. 1) *row buffer hit* requests, which hit the currently opened row; 2) *near-segment* requests, which access the rows in the near segment; and 3) *far-segment* requests, which access the rows in the far segment. In scheduling cycles, the memory scheduler first tries to schedule *row buffer hit* requests, as they have the shortest service time and doing so reduces the overall row switches. After that, it tries to schedule *near-segment* requests if there are any. *Far-segment* requests are scheduled lastly. Please note that *near-segment* and *far-segment* requests are static, while *row buffer hit* requests dynamically change as the referenced currently opened row changes over time. The latency-aware memory scheduling (*LAMS*) policy can reduce the total queuing time (a.k.a., scheduling delay) relative to convention FR-FCFS scheduler, as it is demonstrated in Figure 2. In this figure, we assume there are five memory requests put in arriving-time order on the top of the figure and they are awaiting in the memory queue. Request A, C access data in the *far-segment* (with 10-cycle latency), and request C, D, E access data in the *near-segment* (with 5-cycle latency). We assume there are no row buffer locality among those five requests, because both *FR-FCFS* and *LAMS* schedule row-buffer hit requests first and after scheduling row-buffer hit requests the situation degenerates to the one depicted in the figure. As can be seen from the figure, *LAMS* reduces the total scheduling delay from 80 cycles to 55 cycles, an amount of more than 31% reduction, which translates into performance improvement, as we will see in Section 4.



(a) FR-FCFS schedules memory requests in time-arriving order.



(b) LAMS schedules requests in a latency-aware fashion.

Figure 2: Comparison of scheduling delay between FR-FCFS and LAMS schedulers. The shaded area represents scheduling latency. LAMS reduces scheduling latency from 80 cycles to 55 cycles.

TABLE 1: Simulation Parameters

Processor	Single core/two cores/four cores			
Memory Controller	64/64-entry read/write request queue, writes are scheduled in batches			
DRAM	DDR3, 1 channel, 2 rank per channel			
		near	far	baseline
	t_{RCD}	27	38	44
	t_{RC}	76	177	156

3.3. Starvation Avoidance

To prevent starvation of memory requests addressing far-segment and ensure fairness is not compromised in multi-programmed environment, we enforce a maximum scheduling delay of far-segment requests. When there are any far-segment requests that have been delayed over the maximum value, the memory scheduler schedules those requests in the next scheduling cycle, even though there may be row-buffer hit requests or near-segment requests. Though the maximum value of scheduling delay is a configurable parameter, we set it to be 200 cycles in our evaluations and noticed that it was seldom surpassed.

3.4. Latency-Aware Scheduling Algorithm

Algorithm 1 summarizes the algorithm of latency-aware scheduling. At every scheduling cycle, it first checks whether there are any requests which have been waiting for a long time, exceeding the set threshold. If there exist such requests, it puts them into a FIFO issue buffer. Next, it schedules row-buffer hit requests if there are any of them. After that, it goes to check near-segment requests and puts them into the FIFO issue buffer. If there are neither row buffer hit requests nor near-segment requests, it schedules far-segment requests.

4. Evaluation

In this section, we conduct experiments to evaluate LAMS. We first introduce our evaluation methodology including experimental testbed, related simulation parameters, and workloads. Then we present and discuss the evaluation results of various metrics.

4.1. Experimental Methodology

To evaluate LAMS, we implement our idea in the cycle-accurate USIMM [20] memory simulator, which takes memory traces as input and output various metrics at the end

Algorithm 1 Latency-aware scheduling algorithm

Input: $Global\_Cycle$: global time; $Request\_Queue$: pending requests; $Starve\_Threshold$: maximum scheduling delay;

```
while Request_Queue is not empty do
    Global_Cycle ++;
    for Req in Request_Queue do
        Req.waiting + +;
    end for
    if exist Req.waiting > Starve_Threshold then
        insert_request_to_FIFO_buffer(Req);
        continue;
    end if
    if exist any row buffer hit request Req then
        insert_request_to_FIFO_buffer(Req);
        continue;
    end if
    if exist any near segment request Req then
        insert_request_to_FIFO_buffer(Req);
        continue;
    end if
    if exist any far segment request Req then
        insert_request_to_FIFO_buffer(Req);
        continue;
    end if
end while
```

TABLE 2: Workloads Denotations

Workload	Benchmarks
MTc	MT-canneal
bl-bl-fr-fr	black,black,freq,freq
c1-c1	comm1,comm1
c1-c1-c2-c2	comm1,comm1,comm2,comm2
c2	comm2
fa-fa-fe-fe	face,face,ferret,ferret
fl-sw-c2-c2	fluid,swapt,comm2,comm2
st-st-st-st	stream,stream,stream,stream

of simulation. As each memory request is performed on rank level and its efficiency does not rely on the number of ranks in the memory system, for simplicity, we configure the memory system to consist of one channel and two rank in the channel. We adopt the parameters of t_{RCD} and t_{RC} from the original paper [1] and translate them into cycles according to the configured memory frequency. The main simulation parameters are summarized in Table 1. Other parameters not listed in Table 1 use the default values from USIMM.

For our workloads, we use the memory request traces provided by USIMM, which include 8 representative benchmarks from PARSEC suite and two server-class transaction-processing benchmarks. The benchmarks are grouped to eight combinations of workloads, for both single core and multi core configurations. Workloads names are listed in Table 2. A workload that contains multiple benchmarks (e.g., 4) will be executed on the same number of cores (4 cores). We plan to see the comparisons between our scheduler and the baseline *FR-FCFS*, other four competitive memory schedulers from the memory scheduling championship (MSC), denoted as *CPP* [21] , *RLDP* [22], *RDAF* [23], and *TFMRR* [24], respectively. We implement our proposal on each of the four schedulers and to see its performance improvement over them. In the figure, the names of those four schedulers with latency-aware scheduling are denoted as *CPP with LAMS*, *RLDP with LAMS*, *RDAF with LAMS*,

and *TFMRR with LAMS*, respectively. We compare different schedulers in terms of performance, queuing time, access latency, fairness, and energy consumption.

4.2. Performance Comparsion

We use the normalized instruction per cycle (IPC) relative to baseline FR-FCFS scheduler to compare the performance of different memory schedulers. Figure 3 shows the comparison results. From the figure, we obtain two main observations. First, the four memory schedulers from the memory scheduling championship consistently outperform the baseline FR-FCFS scheduler for all the workloads. The *RDAF* scheduler achieves the most performance improvement of 15.2% for workload *MTc*. The average performance improvements of the *CPP*, *RLDP*, *RDAF*, and *TFMRR* relative to the FR-FCFS scheduler are 10%, 8%, 10%, and 7%, respectively. Second, with LAMS, the performance is further improved. Our scheduler further improves the performance by up to 6.1% (*RLDP* scheduler for *C2* workload), with an average of 2.7% across all the workloads, resulting in a maximum improvement of 20.6% relative to the FR-FCFS scheduler. This set of results have demonstrated that latency-aware scheduling algorithm can further improve memory performance upon existing schedulers.

4.3. Queuing Time and Access Latency

In this section, we compare the queuing latency and access latency of the different schedulers. As *LAMS* schedules short-latency requests before long-latency requests, we expect it can reduce memory queuing time and overall access latency as well. Figure 4 shows the comparison results. From the figure, we can obtain similar conclusions in previous performance comparisons. First, for the most majority of workloads, LAMS can reduce both queuing time and access latency significantly relative to the FR-FCFS. With LAMS, the queuing time is reduced by up to 36.7% (*RDAF* scheduler for *c2* workload), with an average of 18.1%, relative to the FR-FCFS. The access latency is decreased by up to 30.5% (*TFMRR* scheduler for *fa-fa-fe-fe* workload), with an average of 19.4%, relative to the FR-FCFS. Second, LAMS further reduces the queuing time and access latency as well relative to the four schedulers, with an average of 1.9% and 4.8% reductions for queuing time and access latency, respectively. It is also interesting to observe that there exist some exceptions in which LAMS increases both queuing time and access latency. For example, with LAMS scheduler, the queuing time of *CPP-with LAMS* increases by 15% for *MTc* workload. The reason behind is the memory requests of *MTc* distributed largely on the far-segment which causes the longer queuing latency, and the different scheduling purposes between *CPP* and *LAMS* exacerbate the queuing latency. Nonetheless, compare the FR-FCFS, the *CPP* scheduler increases queuing time by 4.2%. So it could be because *CPP* cannot work well for *MTc* workload. Overall, due to the optimized scheduling mechanism, LAMS reduces both queuing time and access latency, which explains the performance improvement in Section 4.2

4.4. Near- and Far-Segment Request

In this section, we report the percentages of near-segment and far-segment requests for each workload. The percentages are workload specific. Intuitively, if a workload has more near-segment requests, then it benefits more from latency-aware scheduling. Figure 5 shows the statistics. From the figure, we know that each workload has a decent percentage of near-segment requests. There is an average of 50.9% near-segment requests across all the workloads. It implies that there is a good potential to partition bit-lines to near and far segments. The figure also partly explain why our scheduler improves performance for *MTc* workload the most, as it has the highest percentage (99.9%) near-segment requests.

4.5. Fairness Analysis

The USIMM simulator reports a metric of PFP, i.e., performance-fairness-product, which indicates the fairness among multi-programmed benchmarks. A smaller PFP value means the system has better fairness. Intuitively, prioritizing short-latency memory requests may compromise fairness for multi-programmed benchmarks, if one benchmark continuously issues requests to the near-segment. This section aims to demonstrate that our idea does not compromise fairness by comparing the PEP values. Figure 6 gives the comparison of PFP values for different schedulers. Two main conclusions are in order. First, compared with the FR-FCFS, our scheduler greatly reduces the PFP values, meaning it has better fairness. It reduces the PFP values by an average of 10.4% relative to the FR-FCFS scheduler. Second, it also reduces the PFP values by an average of 1.3% relative to the four schedulers. The results show that even though LAMS schedules memory asymmetrically, it does not compromise fairness, instead it improves fairness. One possible reason to explain that is we limit the maximum scheduling delay, so that long-latency requests are not starved.

4.6. Energy Comparison

In this section, we give a comparison of energy consumption between different schedulers. Figure 7 shows the comparison results. As it is shown, all the four schedulers have better energy efficiency than the FR-FCFS scheduler. *CPP* is the scheduler which reduces energy consumption most, achieving a 25% reduction for *fl-sw-c2-c2* workload. Across all the four schedulers, the average energy reduction is 15.3%. Our proposal further reduces the energy consumption compared with the four schedulers. It reduces a maximum of 23.4% energy consumption with *RLDP* scheduler for *C2* workload, with an average reduction of 11.1%. In summary, all evaluation results have shown that latency-aware scheduling has higher performance, better fairness, and is more energy efficient than other schedulers.

5. Related Work

There is a whole body of research work addressing the "memory wall" problem in computing systems from various aspects. DRAM refresh slows down memory performance and has particularly received a lot of research interest. *Refresh Pausing* [5] makes refresh operation interruptible and temporarily suspends on-gong refresh to give way to regular

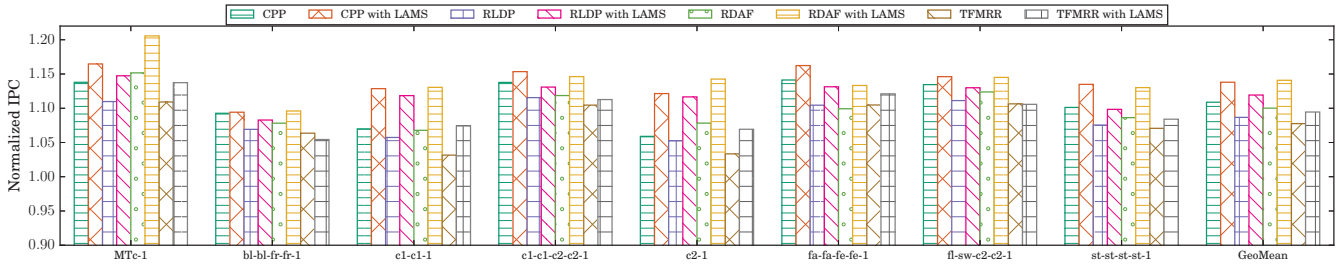
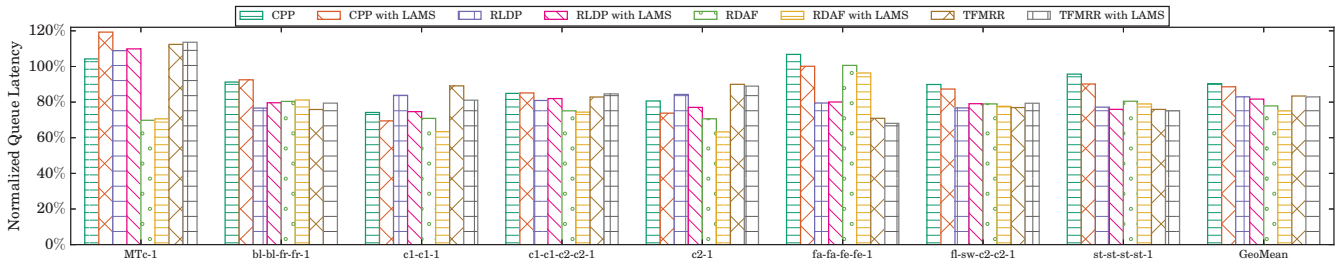
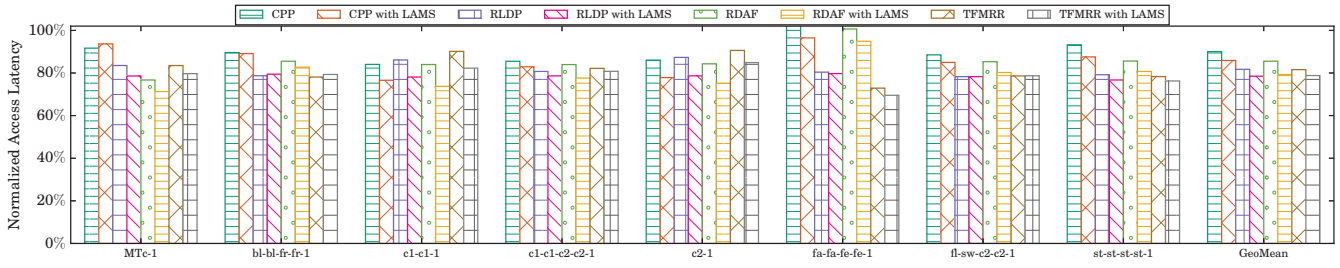


Figure 3: Performance comparison of different memory schedulers. It shows that with latency-aware scheduling the performance can be improved by up to 20.6%, with an average of 12.3% relative to the baseline FR-FCFS. It also further improves the memory performance by up to 6.1%, with an average of 2.7% compared with the four schedulers from MSC.



(a) Queuing time comparison across all the workloads.



(b) Access latency comparison across all the workloads.

Figure 4: Queuing time and access latency comparisons of different schedulers normalized to the FR-FCFS scheduler. LAMS reduces memory queuing time by an average of 18.1% compared with FR-FCFS, and achieves an average latency reduction of 1.9% compared with the four schedulers from MSC. In the meanwhile, the access latency is reduced by an average of 19.4% compared with FR-FCFS, with an average reduction of 4.8%, relative to the four schedulers from MSC.

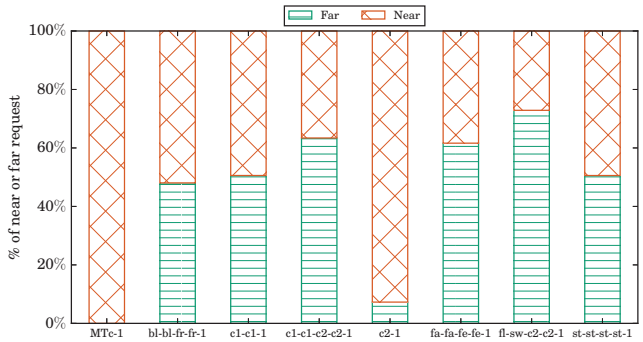


Figure 5: The percentages of near-segment and far-segment requests of all workloads. As it is shown, there is an average of 50.9% percentage near-segment requests across all the workloads, implying there is much improvement space for latency-aware scheduling to play its role.

memory requests. *Elastic Refresh* [4] allows to postpone up to eight refresh operations if there exist access conflicts between refresh operations and regular memory requests. Retention time aware refresh schemes [25] avoid unnecessary refreshes by leveraging retention-time disparities in DRAM cells. Parallelizing refresh and memory requests [7], [9] in different memory sources can also decrease refresh performance impacts.

Researchers have also conducted studies at memory architectural level to reduce memory latency. A memory system contains multiple levels of resources, e.g., banks, subarrays, providing rich internal parallelism that can be taken advantage of. Kim et al. [26] propose a memory architecture with subarray level parallelism and explore a number of ways to leverage the parallelism to improve performance. Jeong et al. [6] employ bank partitioning to assign separate banks to distinct applications so that accesses to the same bank may have better row buffer locality. Son et al. [27] suggest asymmetric bank organization in a memory system to improve performance. *Tiered latency DRAM* [1] splits bit-lines to near and far segments via adding isolation transistors on the bit-lines so that near segment has shorter wire length and less parasitic capacitance. Therefore, accesses in the near segment have shorter latency. In a similar way, *Row Buffer Decoupling* [3] also introduces isolation transistors on

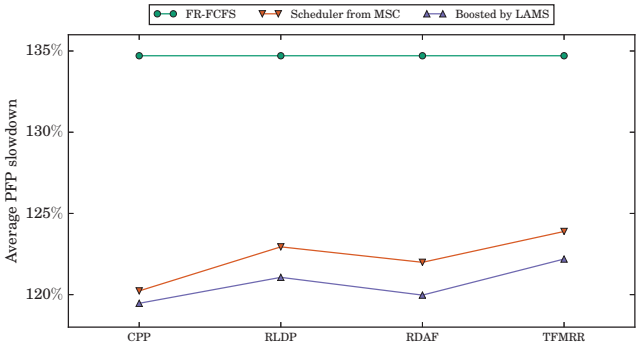


Figure 6: This figure compares the PFP (performance-fairness-product) values of different schedulers. The upper curve represents the average PFP values of the FR-FCFS. The middle curve denotes PFP values of the respective schedulers on x-axis. The bottom curve denotes the four schedulers with LAMS. It shows that LAMS can further reduce PFP values by an average of 1.3% across the four schedulers, providing better fairness.

bit-lines to decouple the row buffer from associated bit-lines so that the bit-lines can be precharged immediately after the row is activated. Therefore, the next row switch operation is obviated from the dominating bit-line precharge latency, greatly improving performance. It differs from tiered latency DRAM architecture in only where on the bit-lines isolation transistors are inserted. Seshadri et al. [2] propose a *Row Clone* memory architecture in which a row can be moved to anther row via row buffer very quick. New memory architectures offer new opportunities that do not exist in conventional memory architectures. Our proposed latency-aware memory scheduling is inspired by the potentials enabled by the tiered latency memory architecture.

An optimal memory scheduler can also improve memory performance given a specific memory architecture. There are a lot of researches aiming to design optimal memory schedulers. The community even held a memory scheduling championship in 2012. The classical memory scheduler is called FR-FCFS [11]. It schedules memory requests which hit the currently open row first and schedules the remaining requests in a first come first served manner, as row buffer hit results in shorter memory latency. FR-FCFS has become the most widely deployed scheduler and is the default scheduler in memory systems. Thread cluster memory (TCM) scheduling [17] groups threads with similar memory access behaviors into either the latency-sensitive (memory-non-intensive) or the bandwidth-sensitive (memory-intensive) cluster and each cluster employs different scheduling mechanisms to achieve both throughput and fairness. As ranking the memory access characteristics for each application leads to high hardware cost and complexity, BLISS [28] divide the applications into interference group (applications with a large number of consecutive requests served) and vulnerable-to-interference group (with higher priority), to achieve high performance and fairness with low cost. Parallelism-Aware Batch Scheduling [15] also tries to ensure throughput and fairness by scheduling memory requests to different banks simultaneously and scheduling requests in batches. ATLAS [19] periodically evaluates the service time that threads have received and prioritizes the threads which have the least attained service time so far. MORSE [14] is a reconfigurable memory scheduler that can achieve different goals based on the user's configuration. In order to maximize row buffer hit rate, TFMRR scheduler [24] groups requests with the same target row together to reduce average memory access latency and power consumption. RLDP scheduler [22] explores row buffer locality by turning write to read if there is a potential row buffer hit when the write queue is draining without damaging the row buffer locality. RDAF scheduler [23] tries to promote the applications with fewer requests to improve system throughput. Reinforcement Learn (RL) [12] leverages observed changes on system performance of scheduling decisions to guide the scheduling process to achieve a long-term performance improvement. CPP scheduler [21] estimates the execution phases of the threads based on learned information, and overlaps write with refresh in multi-rank configuration to maximize bandwidth utilization. As discussed, none of the existing schedulers takes underlying memory characteristics into account when making scheduling decisions, as our proposal does. Moreover, our scheduling policy is simpler to realize. Furthermore, our scheduler is orthogonal to existing schedulers and can be integrated with them to further improve memory performance, as we it is demonstrated in this work.

6. Further Discussion

Memory technology is developing fast and there are many new memory architectures [1], [2], [3], [27] emerging. It is required that upper layer components on data access path need to be optimized in order to leverage new potentials to the full. The original tiered latency paper has demonstrated two use cases, using the near segment as a managed cache for the far segment and exposing new segment to OS as well and mapping frequently accessed pages to near segment. Comparing with these two cases, our exploitation of the new memory architecture at memory scheduling level is much simpler, while still achieving similar performance improvement. To our knowledge, our work is the first to exploit latency asymmetry at the memory scheduling layer. Compared with existing prediction-based memory schedulers, our proposed scheduler is much more lightweight and less expensive to achieve performance improvement. Especially, prediction-based memory scheduling techniques tend to become ineffective in high capacity memory systems, as there are less memory requests going to each bank, making it hard to make accurate prediction to improve row buffer locality. Fortunately, our scheduling policy does not rely on prediction and therefore is free from the limitations.

Though we demonstrate how to leverage latency asymmetry of the new memory architecture at the scheduling layer, we believe there are lot of other optimization possibilities at other layers as well, including operation system, processor level. For example, file system metadata can be put in the low-latency segment as the metadata is generally more access-intensive. Performance-sensitive data blocks can be put in the low-latency segment in order to

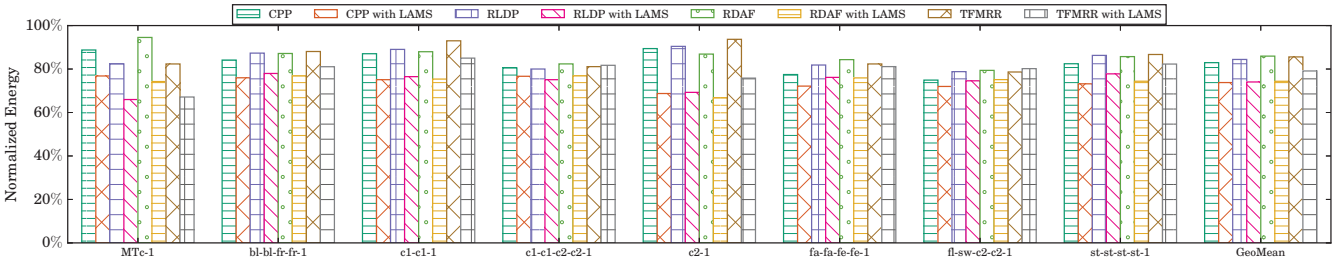


Figure 7: Energy comparison of different schedulers for all workloads. It shows that LAMS reduces energy by up to 34%, with an average of 24.7% compared with FR-FCFS, and reduces energy by up to 23.4%, with an average of 11.1% compare with the four schedulers from MSC.

avoid bottlenecks. The latency asymmetry information can also be made available to processors for it to make more informed decisions, e.g., on-chip cache replacement policies, reorder buffer (ROB) management. We will explore those possibilities in our future work.

7. Conclusion

In this paper, we propose to leverage the opportunities offered by the tiered latency memory architecture at the memory scheduling layer to improve memory performance and energy efficiency. The proposed latency-aware memory scheduling (LAMS) gives scheduling priority to short-latency memory requests (to near-segment) over long-latency memory requests (to far-segment), while still leveraging row-buffer locality at the same time. Doing so, it reduces the total request queuing time and thus improves performance, while also decreasing memory energy as the total execution time is shortened. Our evaluation results have shown that using LAMS improves performance, fairness, and energy efficiency, compared with the popular FR-FCFS scheduler and other four competitive memory schedulers from the memory scheduling championship (MSC).

8. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments that greatly improved our paper. The research is supported in part by the National Natural Science Foundation of China (NSFC) under grant Nos.61232004, 61502189, 61331010, and 61572012, and the U.S. National Science Foundation (NSF) under Grant Nos. CCF-1547804, CNS-1218960, and CNS-1320349. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] D. Lee *et al.*, "Tiered-latency dram: A low latency and low cost dram architecture," in *HPCA*, 2013.

[2] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.

[3] S. O *et al.*, "Row-buffer decoupling: A case for low-latency dram microarchitecture," in *ISCA*, 2014.

[4] J. Stuechelix *et al.*, "Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory," in *MICRO*, 2010.

[5] P. Nair, C.-C. Chou, and M. K. Qureshi, "A Case for Refresh Pausing in DRAM Memory Systems," in *HPCA*, 2013.

[6] M. K. Jeong *et al.*, "Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems," in *HPCA*, 2012.

[7] T. Zhang *et al.*, "CREAM: a Concurrent-Refresh-Aware DRAM Memory Architecture," in *HPCA*, 2014.

[8] S.-L. Lu, Y.-C. Lin, and C.-L. Yang, "Improving dram latency with dynamic asymmetric subarray," in *MICRO*, 2015.

[9] K. Chang *et al.*, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.

[10] J. Mukundan *et al.*, "Improving Memory Scheduling via Processor-Side Load Criticality Information," in *ISCA*, 2013.

[11] S. Rixner *et al.*, "Memory Access Scheduling," in *ISCA*.

[12] E. Ipek *et al.*, " Self-Optimizing Memory Controllers: A Reinforcement Learning Approach," in *ISCA*, 2008.

[13] R. Ausavarungnirun *et al.*, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.

[14] J. M. J. F. Martinez, "MORSE: Multi-objective Reconfigurable Self-optimizing Memory Scheduler," in *HPCA*, 2012.

[15] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *ISCA*.

[16] M. Xie *et al.*, "Improving system throughput and fairness simultaneously in shared memory cmp systems via dynamic bank partitioning," in *HPCA*, 2014.

[17] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.

[18] E. Ebrahimi *et al.*, "Parallel Application Memory Scheduling," in *MICRO*, 2011.

[19] Y. Kim *et al.*, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010.

[20] N. Chatterjee *et al.*, "USIMM: the Utah SImulated Memory Module, A Simulation Infrastructure for the JWAC Memory Scheduling Championship," University of Utah, Tech. Rep., 2012.

[21] Y. Ishii *et al.*, "High performance memory access scheduling using compute-phase prediction and writeback-refresh overlap," *JILP Memory Scheduling Championship*, 2012.

[22] Y.-S. Moon *et al.*, "The compact memory scheduling maximizing row buffer locality," *JILP Memory Scheduling Championship*, 2012.

[23] T. Ikeda *et al.*, "Request density aware fair memory scheduling," *JILP Memory Scheduling Championship*, 2012.

[24] K. Fang *et al.*, "Thread-fair memory request reordering," *JILP Memory Scheduling Championship*, 2012.

[25] J. Liu *et al.*, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.

[26] Y. Kim *et al.*, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.

[27] Y. H. Son *et al.*, "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," in *ISCA*, 2013.

[28] L. Subramanian *et al.*, "The blacklisting memory scheduler: Achieving high performance and fairness at low cost," in *ICCD*, 2014.