Multiple Sub-Row Buffers in DRAM: Unlocking Performance and Energy Improvement Opportunities

Nagendra Gulur  
Texas Instruments (India)  
nagendra@ti.com

Mahesh Mehendale  
Texas Instruments (India)  
m-mehendale@ti.com

R Manikantan  
Indian Institute of Science  
rmani@csa.iisc.ernet.in

R Govindarajan  
Indian Institute of Science  
govind@serc.iisc.ernet.in

ABSTRACT
The twin demands of energy-efficiency and higher performance on DRAM are highly emphasized in multicore architectures. A variety of schemes have been proposed to address either the latency or the energy consumption of DRAMs. These schemes typically require non-trivial hardware changes and end up improving latency at the cost of energy or vice-versa.

One specific DRAM performance problem in multicores is that interleaved accesses from different cores can potentially degrade row-buffer locality. In this paper, based on the temporal and spatial locality characteristics of memory accesses, we propose a reorganization of the existing single large row-buffer in a DRAM bank into multiple sub-row buffers (MSRB). This reorganization not only improves row hit rates, and hence the average memory latency, but also brings down the energy consumed by the DRAM. The first major contribution of this work is proposing such a reorganization without requiring any significant changes to the existing widely accepted DRAM specifications. Our proposed reorganization improves weighted speedup by 35.8%, 14.5% and 21.6% in quad, eight and sixteen core workloads along with a 42%, 28% and 31% reduction in DRAM energy.

The proposed MSRB organization enables opportunities for the management of multiple row-buffers at the memory controller level. As the memory controller is aware of the behaviour of individual cores it allows us to implement coordinated buffer allocation schemes for different cores that take into account program behaviour. We demonstrate two such schemes, namely Fairness Oriented Allocation and Performance Oriented Allocation, which show the flexibility that memory controllers can now exploit in our MSRB organization to improve overall performance and/or fairness.

Further, the MSRB organization enables additional opportunities for DRAM intra-bank parallelism and selective early precharging of the LRU row-buffer to further improve memory access latencies. These two optimizations together provide an additional 5.9% performance improvement.

Categories and Subject Descriptors
C.1.2 [PROCESSOR ARCHITECTURES]: [Multiple Data Stream Architectures (Multiprocessors)]

Keywords
DRAM, Memory Performance, Multi-Core Architecture

1. INTRODUCTION
With the widening gap between processor and memory performance, the memory performance can impact the overall performance of a multicore system in a significant way. Further, the energy consumption of DRAM memory accounts for a non-trivial (greater than 30%) part of the system energy [10], [11]. The importance of energy efficiency and performance of DRAM is emphasized by current trends in high-performance computing which achieve performance scaling via multicores. In both server-class and personal computers, multicore configurations are becoming widespread with several cores sharing DRAM memory that is accessed via one or more memory controllers. Thus a recent research focus has been on improving the performance and energy efficiency of DRAM design which has traditionally been architected to address high density and low cost.

A key component of DRAM that impacts both performance and energy is the row-buffer. When an access request is made to a bank in DRAM, it fetches a large row of data (typically 8KB – 16KB, across all devices accessed in parallel) into the row-buffer. This operation is known as row-activate. In the presence of spatial locality, future requests to the same row hit in the row-buffer. In these cases, the row-buffer provides the data, which reduces both access latency and the energy consumed as the row-activate operation is eliminated. A request to a different row in the same bank replaces the row-buffer with the contents of the new row, after the current row-buffer is written back (precharge).

The spatial locality exploited in the row-buffer is greatly reduced when accesses from multiple cores get interleaved at the DRAM [12]. A number of memory access re-ordering schemes [1, 2, 3, 25, 4], varying in complexity from the simple FR-FCFS (First Ready-First Come First Served) [5] to the more complex reinforcement learning based approach [25], have been proposed to improve row-buffer hits. Further address re-mapping in hardware or software [8] and memory access re-ordering schemes that are prefetch-aware have also been proposed [9]. While these schemes are effective in improving performance, they do not address the issue of energy reduction directly. Further, they require non-trivial modifications and complex scheduling policies to be implemented at the memory controller.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2012 ACM 978-1-4503-1316-2/12/06 ...$10.00.
One significant contributor to the high energy consumption inside DRAMs is the frequent row Activate and Precharge operations to move data back and forth between the row-buffer and the DRAM core. Common methods proposed to reduce DRAM energy include smaller row-buffers [12], storage re-organization [13], and exploiting opportunities for power-down modes of operation [14]. However, these methods generally incur performance degradation and also introduce hardware complexity. For instance, smaller row-buffers reduce the energy consumption at the expense of lower hit rates and a small performance loss.

Though there is a lack of spatial locality, multi-programmed workloads do exhibit considerable temporal locality among memory accesses, at the level of DRAM pages/rows [15]. The observed temporal and spatial locality characteristics make a strong case for having multiple smaller row-buffers per bank instead of a single row-buffer. Such a configuration has been known to provide benefits in the context of Phase Change Memory [15]. However, such a row-buffer reorganization and its impact on performance and energy reduction has not been studied in the context of DRAMs. This is important as the challenge is to accomplish this reorganization under a fairly rigid JEDEC standard [17]. Further, the proposed organization enables a set of optimization opportunities which have not been explored thus far and are relevant specifically for DRAMs.

Our first contribution is to propose a practical design to incorporate multiple sub-row buffers (MSRBs) in DRAMs with minimal changes to the existing DRAM specifications. A study of multiple narrow row-buffers in the context of DRAM shows that it can significantly improve both performance and energy in multi-cores. The performance gains (in terms of weighted speedup) over the baseline are 35.8%, 14.5% and 21% respectively for quad, eight and sixteen cores respectively. This gain in performance is achieved along with an energy reduction of 42%, 28% and 31%. We refer to this organization as MSRB.

The necessary controls for implementing MSRB are incorporated at the memory controller. While this allows the DRAM design changes to be minimal with no changes to the pin interface, it also opens up opportunities for further optimizations to effectively utilize the row-buffers to achieve performance and/or fairness goals. This is because unlike the DRAM, an on-chip memory controller can observe the behavior of various cores and hence can manage the allocation of row buffers to suit the observed behavior. We demonstrate two such buffer allocation strategies - Fairness Oriented Allocation and Performance Oriented Allocation. Fairness-oriented Allocation allocates dedicated row-buffers to cores that suffer the most interference thereby improving both performance and fairness. This allocation scheme improves fairness by 43%. Performance-oriented Allocation takes into account the differing memory bandwidth requirements of the various cores and tries to allocate row buffers to cores in line with their demand. Results show that this scheme further improves performance.

Our third contribution is to examine the additional optimizations enabled by our multiple row-buffer design. On a row-buffer miss, it is essential to write back the currently open row, before the newly requested data can be brought into the row-buffer. This precharge latency is typically in the critical path of memory requests. In a multiple row-buffer configuration, it is possible to do an eager precharge of the least recently used row-buffer. This proactive approach, which we refer to as Early Precharge, hides the precharge latency that will be experienced by a future request that misses in the row-buffer. This optimization improves performance by an additional 1.8% over MSRB design. A second optimization enabled by the new design is the ability to simultaneously service row-hit requests from one row-buffer while a different row-buffer is being activated or precharged. This introduces Intra-Bank Parallelism and improves performance by an additional 4.7% over MSRB. The Early Precharge and Intra-bank Parallelism optimizations taken together yield an additional performance improvement of 5.9%.

Last, we compare our row-buffer reorganization with two best-in-class memory controller schedulers (Thread Cluster Memory (TCM) scheduling [1] and Parallelism Aware Batch Scheduling (PARBS) [2]) and demonstrate that our design produces far greater system throughput than what is achieved via scheduler optimizations alone. Further, we compare our results against a hypothetical DRAM device which is highly banked (32-banks and 256-columns per bank structure). We observe that our proposed MSRB organization is more effective, in terms of both performance and energy, than a highly banked DRAM with just one small row-buffer per bank.

2. BACKGROUND AND MOTIVATION

In this section, we provide the necessary background and the required motivation for our work.

2.1 Background

We consider the popular JEDEC-style ([17]) DRAM as the baseline architecture throughout the paper. DRAM devices are packaged as Dual In-line Memory Modules (DIMMs) which are interfaced typically to an on-chip memory controller (refer Figure 1). DIMMs contain one or more ranks. A rank is a collection of DRAM devices that operate in parallel. Each DRAM device typically serves up a few bits at the specified (row, column) location. Operating together, the devices in a rank match the data bus width. For example, an x16 device supplies 16 bits of data and such devices making up a rank can supply data needed to match the 64-bit interface of the memory controller. Each device in a rank is organized into a number (4, 8 or 16) of logically independent banks. Each bank consists of multiple rows (also called pages) of data. Banks within the same rank can operate in parallel and this provides for some degree of memory level parallelism. Figure 1 shows a diagrammatic description of this organization.

A typical DRAM read request has to first Activate the corresponding row by bringing the row data to the row-buffer. This is followed by a column read/write that reads/writes the selected words from/to the row-buffer. Finally Precharge writes back the row-buffer to the appropriate row. Precharge is needed even on rows that were only read, since row activation depletes the charge in the corresponding row in the DRAM device. Each bank is equipped with its own row buffer and logic to perform row Activate (termed RAS), column read/write (termed CAS) and Precharge (termed PRE) operations. An open page policy delays precharging until just before the next row activate has to be performed, while the closed page policy eagerly precharges the row soon after the first read/write operation to this row. The memory controller is responsible for efficient scheduling of requests, as well as for the implementation of the DRAM access protocol. Each cycle, the controller selects a valid command by examining the set of pending requests and issues it to the memory.

2.2 Motivation

A typical DRAM bank is equipped with a large row buffer comprising 1024 to 2048 columns. This large buffer size is an artifact of the original use-model it was intended for, viz., exploiting spatial locality while serving requests from a single core processor. In the multicore scenario wherein requests from multiple workloads are
interleaved by the memory controller, there is insufficient reuse of the open row buffer. Further, modern programs have large working sets and even single-core programs could access multiple rows in succession in a bank. These scenarios favor the closed page policy (which precharges and closes the page immediately after the first use) as the de-facto policy, incurring high access time and energy by bringing in a large row for a single word or cacheline. The above observations indicate a potential benefit in using multiple small row-buffers for each bank in the DRAM. Figure 2 shows cumulative stack distance histograms of row hit rates achieved in the top 1, 2, and 4 (Most Recently Used) positions for a representative subset of SPEC 2006 CPU workloads with 1024-columns wide row buffers, while Figure 3 shows the same data for 256-columns wide row buffers. A detailed description of the experimental methodology, simulated configuration, and workloads is included in Section 5.

The graph in Figure 2 shows that, on the average, the hit rate more than doubles as we go from one row-buffer to even just two buffers. This is especially observed in memory intensive programs, e.g. milc, gromacs, soplex, and calculix. Two (or more) row buffers appear to work significantly better at capturing temporal/spatial locality in programs.

The graph in Figure 3 shows the crucial observation for our motivation: small buffers do nearly as well as their larger counterparts in exploiting spatial locality. On average, we observed that small buffers captured over 90% of the hits seen with larger ones. This study reveals that multiple small row-buffers per bank help improve temporal locality without significant loss in spatial locality. Incorporating this into DRAMs within the existing DRAM standards pose certain challenges. At the same time, the MSRB also offers a few additional opportunities such as implementing different row-buffer management policies for performance and fairness, and optimizations such as Early Precharging and Intra-Bank Parallel accesses. We address these topics in the following sections.

3. MULTIPLE SUB-ROW BUFFER ORGANIZATION

In this section, we describe our MSRB organization for DRAM. Specifically, we replace the one large row-buffer in each bank by four small row-buffers, each one-fourth the size of the original. This organization consists of three components, namely (i) Sub-row activation, (ii) Row-buffer selection, and (iii) Row-Buffer allocation. We describe the first two components in detail below while the row-buffer allocation is deferred to Section 4. Figure 4 provides an overview of a DRAM bank organized to support multiple small row buffers.

3.1 Sub-Row Activation

In order to select the appropriate sub-row, the DRAM needs to have access to both the row address and a part of the column address. Once both are available, the DRAM logic decodes and activates both row_select_i and sub_row_select_j lines and fetches the selected columns to a row buffer. While row_select lines run across the entire length of the row, they need only participate in the sub-row decoding and as such have very little load on them. At the selected sub-row, the traditional wordline is activated to access the entire sub-row. Our implementation of sub-row activation is similar to that described in [12].

To perform sub-row activation, the DRAM needs both the row address (from the RAS command) and a few address bits from the column access command (CAS), since the sub-row selection is dependent on a few column address bits: 2 bits if the size of each sub-row is 1/4-th the size of the row. In a traditional DRAM interface, the RAS command is issued first and the CAS command follows it a few cycles later. This is done in order to multiplex row and column addresses onto the same set of pins. In the JEDEC standard, a timing parameter termed tRCD specifies the gap between the two commands that has to be met [17], and is typically of the order of 10-15 nano-seconds. With such tRCD delay, a naive scheme which
waits for the column address would introduce intolerably high latency to sub-row activation. In order to avoid this, we discuss at least three alternatives that could get the sub-row select address bits to the DRAM without incurring the RAS-to-CAS delay:

- Expanding the address pins by additional sub-row-select pins would address the sub-row selection problem in a straightforward way. Typically, this is an additional 2 to 3 pins depending on the size of the sub-row relative to the full row. Though simple, given the slow growth in pin-count, we do not consider this a feasible option.

- Issue RAS and CAS commands in back to back cycles. The DRAM is expected to latch the addresses issued in these two commands and use them at appropriate times internally. This scheme is termed Posted-RAS in [12] and Posted-CAS in [24]. There is a 1-cycle delay incurred in this scheme. We chose not to use this scheme due to this 1-cycle latency addition to an already large DRAM access latency.

- Modern DRAMs support double-data-rate transfers on the data pins (hence termed DDR). That is, both the DRAM and the memory controller have the capability to transmit/receive data at twice the bus clock. One could extend this capability to address pins as well. A RAS command issued on the rising edge of the bus clock followed by the sub-row-select command issued on the falling of that bus clock serves to transfer all the necessary address bits in one clock cycle and thus incurs no latency in sub-row activation. We term this scheme Double-Address-Rate. This is the scheme we assume in our detailed simulations. A similar proposal for fast signalling of address bits appears in [26].

### 3.2 Row-Buffer Selection

The introduction of multiple row-buffers necessitates a few additional changes in the overall memory organization: i) the memory controller needs to remember the sub-rows in each bank that are currently available in the row-buffers. This is required so that no RAS command is issued for a sub-row that is already available in a row-buffer. ii) when a new sub-row is activated, allocation of a row buffer that will store the newly fetched sub-row data. iii) ensure precharge for the sub-row that is being replaced. Responsibilities (i) and (iii) are necessarily handled by the memory controller even in the case of a single row-buffer. We feel it is natural to let the memory controller handle these responsibilities even for multiple row-buffers. Further, as the controller maintains the book-keeping information regarding open sub-rows, responsibility (ii) can also be handled by the controller. This decision helps in keeping the DRAM logic simple. Essentially, the controller has to maintain cache-like metadata for these buffers: valid bits, row and sub-row tags, and recency bits. We observe that this decision of letting the memory controller manage row-buffer usage not only keeps DRAM logic simple but has several additional benefits, including:

- Enforcement of different row-buffer allocation policies (for instance, the controller could enforce specialized fairness oriented or performance oriented row-buffer allocation policies to suit the memory access characteristics of workloads)

- Holistic management of the pool of row-buffers available across all the DRAM banks in all the DRAM ranks.

In each of the three DRAM access operations (Activate, Precharge, Column Access), a row buffer is accessed. Therefore we look at each operation and discuss signaling and timing issues involved in specifying row-buffer selection information:

- Activate: Row buffer selection is not in the timing critical path for this operation since the DRAM has to first activate the sub-row and start discharging column data onto bit lines. Thus, the row buffer selection should only be ready by the time bit lines have been driven from the storage cells. Thus a simple mechanism such as signaling the row buffer specification bits in the cycle following the RAS command suffices. We call this Posted-Buffer-Selection. This requires no additional pins. Alternatively, the buffer selection bits could be driven along with the sub-row selection bits using the Double-Address-Rate scheme.

- Precharge: In this case, we assume that the row-buffer selection is timing critical. For this operation, the controller needs to specify both the sub-row selection bits as well as the row-buffer selection bits. We propose to use the double-address-rate scheme to accomplish this transfer without adding latency.

- Column Access: We assume that row-buffer selection is timing critical to column accesses as well and use the Double-Address-Rate mechanism to issue these bits quickly.

The Double-Address-Rate scheme thus takes care of signalling both the sub-row selection bits as well as the row-buffer selection bits without incurring additional latency.

### 3.3 Row-Buffer Allocation

Our MSRB organization requires the memory controller to decide which of the buffers to allocate for a new row activation. For our default configuration, we employ the commonly used Least Recently Used (LRU) policy in the memory controller to make this allocation decision. We defer a more detailed discussion of alternative allocation policies to Section 4.

### 3.4 Sources of Energy Reduction

Energy savings in MSRB are obtained as a combination of:

1. Reduction in the energy consumed by each Activate and Precharge operation due to smaller rows: Since fewer capacitors have to be charged and discharged, and fewer bits have to be latched in the sense amps, the energy consumed reduces.

2. Reduction in the number of Activate and Precharge operations due to fewer row misses: Multiple row-buffers offer higher data retentivity in the buffers thereby reducing the number of times that rows are activated and precharged. Every additional row hit saves the energy that would have been expended in precharging one row and activating another.

Our scheme results in additional energy reduction due to the fact that more row hits lead to fewer total memory cycles, thereby saving additional background power.

### 3.5 Area Impact

Area overheads comprise the MSRB re-organization overhead in DRAM as well as the book-keeping overhead inside the memory controller, and we discuss each below.

#### 3.5.1 DRAM Area Overhead

In the following discussion, we assume an MSRB organization comprising 4 small row-buffers per bank. Our estimate of the area
overhead in MSRB includes: additional decoders for sub-row selection, running additional sub-row selection lines (wires), additional decoders for row-buffer selection, additional multiplexers to control data routing between selected sub-row & selected row-buffer, and additional wiring for data routing.

- **Sub_row_select** lines and AND gates: Since the size of each small row buffer is 1/4-th the size of the full row buffer, we need to run 4 sub_row_select lines and add (4 × number of rows) AND gates. Each gate adds about 6 additional transistors to the decode logic. As in [12], this was implemented using hierarchical word lines [28] and modeled analytically in CACTI [27]. The CACTI model is set up for exploring the highest density implementation as is the case with commodity DRAMs. With these, we obtain an area overhead of 4.9% to support splitting each row into 4 sub-rows.

- Row-buffer selection demultiplexers, and buffer_select_n lines: Since each operation has to access one of the 4 available row-buffers, additional decode circuitry is added to decode two buffer selection bits and drive the appropriate buffer selection lines. While the buffers themselves are sense amps that have a much larger transistor size, the decoder logic transistors are of a smaller transistor size and thus do not significantly increase area. We lay out the 4 small buffers in a 2 × 2 configuration allowing for efficient wiring of buffer_select_n lines. Modeling these overheads in CACTI, we obtain an area overhead of 1.9%.

- Note that in our design, the total storage capacity of the row-buffers (4 buffers of one-fourth the size compared to a single large buffer) does not increase. Thus our design is buffer-capacity-neutral.

Thus the total area overhead of the proposed re-organization is 6.8% per DRAM bank.

### 3.5.2 Area Overhead in the Memory Controller

The memory controller has to maintain certain metadata — tags, valid bits, dirty bits, and recency bits — which constitutes the overhead. This overhead is less than 16 bytes for the 4 row-buffers in a bank. For a 4 GB RAM organized as 4 ranks and 8 banks, this overhead would be 4 × 8 × 16 = 512 bytes. For the baseline with one row buffer per bank, the memory controller still incurs one-fourth of this overhead (128 bytes) since it has to maintain this state information anyway. We consider the additional storage overhead negligible.

### 4. UNLOCKING PERFORMANCE, ENERGY AND FAIRNESS OPPORTUNITIES

Our MSRB organization opens up the design space for row-buffer allocation and management — policies for allocation of these resources across the cores for performance and/or fairness benefits. As discussed in Section 3.3, the row-buffer allocation decision is done at the memory controller. While a detailed exploration of allocation policies is outside the scope of this paper, we present two simple schemes below to illustrate the flexibility that row-buffer reorganization facilitates. The first - Fairness oriented Buffer Allocation - improves fairness via judicious buffer allocation. The Performance oriented Buffer Allocation improves performance of programs with high miss rates. In addition, we discuss a pair of scheduling and hardware optimizations, namely: Early Precharge and Intra Bank Parallelism that are enabled by MSRB. The net effect of improving performance (via increased row hits) is to also reduce energy consumption. All of these optimizations are implemented at the memory controller.

#### 4.1 Fairness oriented Buffer Allocation

Here, the intuition is that in a typical multicore workload, some cores stand to benefit a lot more from higher row-buffer hit rates than others. Since the interleaving of requests from multiple cores causes cores with lower arrival rates to suffer greater row-buffer misses, this allocation scheme counters the disproportionate increase in miss rate by allocating dedicated buffers for such cores. The scheme works by maintaining a per-core basis, the actual row-buffer hit rate as well as an estimate of the hit rate had the core been running alone (referred to as standalone hit rate). The standalone hit rate is obtained by keeping a shadow row buffer (one per core) in the memory controller which would be updated only with the requests from the given core. The difference between the standalone hit rate and the shared (actual) hit rate provides an estimate of the loss suffered by the core due to interference. If this difference exceeds a threshold (in our case, it is set to 0.5), we classify the core as suffering unfairness. The scheduler then attempts to allocate dedicated buffers to cores suffering unfairness. For instance, if one of 4 cores in a quad-core configuration is unfairly suffering, then the scheduler dedicates one of the 4 row-buffers to this core, while the other 3 row-buffers are made available to all the cores.

For each core $c$ and bank $b$, the scheme computes the difference $d = (\text{standalone hit rate} - \text{shared hit rate})$. Higher values of this measure suggest higher benefits by dedicating buffers to such cores. At each bank, this metric is used to classify the core:

- **Type-1**: Core with $d \geq \text{threshold}$
- **Type-2**: Core with $d < \text{threshold}$

Since the classification is done per-bank, this scheme can inherently self-adjust to variations in bank utilization. The same core could be classified Type-1 in one bank while classified Type-2 in another. This classification is done periodically so as to adapt to program behavior changes. The controller then allocates dedicated row-buffers for Type-1 cores. In our implementation, we chose the below scheme:

- Only 1 Type-1 core in the workload: The Type-1 core gets one dedicated buffer, all cores can access remaining 3 buffers
- Two Type-1 cores in the workload: Each Type-1 core gets one dedicated buffer, all cores can access remaining 2 buffers
- Three or more Type-1 cores in the workload: It defaults to LRU scheme.

Whenever a core needs to activate a new row, the controller looks up the allocated buffers for that core and chooses the LRU buffer from amongst these to bring the new row into the bank.

#### 4.2 Performance oriented Buffer Allocation

This scheme works by dynamically adapting buffer allocations to the most demanding cores. The scheme estimates the needs of each core periodically on a per-bank basis by taking into account the
number of memory requests (i.e., misses from the last-level cache) from each core, and the row-buffer miss rates suffered by each core at each bank. For each core \( c \) and bank \( b \), it computes a rate product defined as: \( \text{rate product}[c][b] = \text{num_memory_requests}[c][b] \times \text{row_miss_rate}[c][b] \). Higher values of this measure suggest higher benefits by improving their hit rates. At each bank, rate products are used to classify cores into one of two types:

- Type-1: Core with \( \text{rate product} \geq \text{threshold} \)
- Type-2: Core with \( \text{rate product} < \text{threshold} \)

Several variations of this scheme are possible. We use a simple scheme which defaults to LRU allocation when the number of Type-1 workloads for a bank exceeds half the number of row-buffers per bank (2 in our case). Otherwise, the allocation scheme allows each Type-1 workload to have exclusive use of certain row-buffers (upto 2 in our case) and shared use of the remaining row-buffers by all workloads, similar to the previous scheme.

For both the above schemes, the storage overhead in the memory controller is negligibly small (of the order of \( R \times B \times M \times N \) bytes for \( R \) ranks, \( B \) banks per rank, \( M \) buffers per bank, and \( N \) cores).

### 4.3 Early Precharge Scheduling

Traditionally, memory controllers use either an open page or closed page policy for precharging rows. The policy essentially determines whether to precharge an open page eagerly (closed page) or lazily (open page). Eager policies work better in situations where it is highly likely that the next request would cause a row buffer eviction. Multiple row-buffers open up the possibility to precharge different row-buffers using different policies. We implemented selective early precharge scheduling wherein only the LRU row-buffer is precharged early while the rest of the row-buffers follow the open-page policy. The rationale for this is that the LRU row-buffer is most likely to be the candidate for eviction and is better off precharged early while the other row-buffers are more likely to see additional row-hits and therefore are kept open. Eagerly precharging the LRU buffer helps to reduce the latency of a subsequent row-buffer miss. In our implementation, the memory controller looks for idle cycles and inserts precharge operations for the LRU row-buffer in each bank. While this scheduling is orthogonal to the row-buffer allocation schemes described in the earlier section, we only implemented it over the baseline LRU policy for our experiments.

### 4.4 Intra-Bank Parallelism

Multiple row-buffers permit parallel operations within a bank; column accesses on one row-buffer could occur in parallel with an activate or precharge operation on another. While each individual memory access follows the standard DRAM access protocol, this optimization (abbreviated intraBP) allows pipelining of operations at each bank. It improves the efficiency of data bus utilization by allowing us to issue column accesses faster. Inside each DRAM bank, the necessary circuitry to support this parallelism is already available. The memory scheduler can easily incorporate this enhancement into its scheduling of sub-commands. Figure 5 shows an example timing diagram indicating this parallelism. This helps to hide some of the activate and precharge latencies via this optimization and it effectively translates to higher bandwidth and lower latency. In particular, programs that have low bank-level parallelism are greatly benefited by this feature since the scheduler is unable to keep multiple banks busy in parallel. Note that exploiting intraBP is not possible without multiple row-buffers.

![Figure 5: Example of intra-bank parallelism](image)

### Table 1: CMP configuration

<table>
<thead>
<tr>
<th>Processor</th>
<th>3.2 GHz OOO Alpha ISA</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1H Cache</td>
<td>32KB private, 64 byte blocks, Direct-mapped, 3 cycle hit latency</td>
</tr>
<tr>
<td>L1D Cache</td>
<td>32KB private, 64 byte blocks, 2-way set-associative, 3 cycle hit latency</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>For 1/4/8/16 cores: 1MB/4MB/8MB/16MB 4-way/8-way/16-way/32-way 32/128/256/512 MSHRs 64-byte blocks, 15 cycle hit latency</td>
</tr>
<tr>
<td>Controller</td>
<td>On-chip: 64-bit interface to DRAM 256-entry command queue FR_FCS scheduling [3], open-page policy Address-interleaving: rank-bank-row-column</td>
</tr>
<tr>
<td>Number of memory controllers for 1/4/8/16 cores: 1/1/2/4</td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td>DDR3-1600MHz, BL=8, CL-RCD-AR=9-9-9 a rank comprises 4 1GB x16 devices, each device has 8 banks, each bank has 65536 rows, 1024 columns</td>
</tr>
</tbody>
</table>

### 5. EXPERIMENTAL METHODOLOGY

#### 5.1 Simulation Setup

We evaluate our design using M5 [18] simulator integrated with a detailed in-house DRAM simulator. The DRAM simulator faithfully models both the memory controller as well as the DRAM with accurate timing. Each program in the workload is executed in fast-forward mode for 9 billion instructions, then in warm-up mode for 500 million instructions and finally, in detailed cycle-accurate mode for 250 million instructions. Multi-core simulations are run until all the programs complete 250 million instructions.\(^5\) As is the standard practice, programs that finish early continue to execute but the performance of only the first 250 million instructions is considered for each core.

The baseline machine configuration used in our studies is shown in Table 1. L2 is the last level cache and is shared across all the cores. The baseline configuration (1 x 1024) has a single large 1024 columns wide row-buffer. Our MSRB configuration (4 x 256) uses 4 narrow row buffers, each 256 columns wide. MSRB is managed using a LRU buffer allocation policy unless specified otherwise. While the quad-core has one memory controller, the eight and sixteen cores have two and four memory controllers respectively.

#### 5.2 Power Estimation

We estimate the impact of our proposed row-buffer reorganization on DRAM power consumption using Micron’s PowerCalculation spreadsheet [19]. The spreadsheet models power consumption of a DRAM configuration by allowing the user to input DRAM configuration parameters and system usage values. We obtain the

\(^{5}\)Although we run only 250M Instructions per core in cycle-accurate mode, our 4, 8 and 16-core simulations each runs for a total of 1 Billion - 4 Billion instructions in cycle-accurate mode.
In Table 3.

The workloads are typically a mix of programs with varying levels of memory intensity. We use the L2 MPKI to identify memory intensive programs. All energy numbers reported in this paper refer to DRAM energy.

6. RESULTS

In this section, we evaluate the impact of MSRB on system performance and the energy benefits provided by it. Further we compare it with state-of-the-art memory access scheduling methods designed to improve row hit rate. A study on the performance improvements due to memory controller side buffer allocation is also presented.

6.1 Performance Benefits of MSRB

The performance of MSRB for the quad-core case is summarized in Figure 6. As can be seen from Figure 6(a), MSRB improves the weighted speedup by 35.8% over 1×1024. The performance gain in terms of harmonic speedup, as shown in Figure 6(b), is 27.5%. All the workloads show improved performance with MSRB. This shows the importance of focusing on the temporal locality (multiple row-buffers) at the cost of spatial locality (narrow row-buffers). MSRB achieves a significantly higher row-buffer hit rate of 0.6 (in Figure 6(c)) as compared to 0.2 observed in the baseline case. The observed gains in performance are typically in line with the improvement in row-buffer hit rate.

Figure 7 shows the performance improvement in terms of weighted speedup for 8 and 16 core workloads. MSRB provides 14.5% and 21.6% improvement in performance for 8 and 16 core workloads respectively over baseline. An interesting case-study is the row-buffer hit rates experienced by the individual programs in the workload E1. Figure 7(b) shows the hit rates experienced by the individual programs for baseline and MSRB. It can be seen that the row-buffer hit rate improves with MSRB for all the individual programs. Last, MSRB improved the performance in terms of IPC of single-core SPEC2000 and SPEC2006 workloads by 7.1% on an average (refer Figure 10 for IPC values obtained for a subset of benchmarks). Observe that programs such as 459.GemsfDFTD and 462.libquantum show considerably high gains (148% and 21%).

6.2 Energy Benefits of MSRB

Improved row-hits not only boosts the performance, but also translates into energy savings due to reduced number of activations and precharges. The smaller size of the row-buffers also reduces the energy required for activate and precharge operations. The energy consumption is computed using the methodology described in Section 5.2. Figure 8 shows the DRAM energy gains provided by MSRB for the quad-core workloads. On an average MSRB reduces the energy requirements by 40% compared to the baseline. It is interesting to note that high energy gains are obtained not only for workloads showing high performance gains but also for others due to the narrow row buffers used. Further, as expected improved hit rate not only translates to high performance gains, but also into significant savings in terms of energy consumed. The total activate power reduced on an average from 357mW to 127mW. For 8 core and 16 core workloads, the energy gains are 28% and 31% respectively over the baseline.

5.3 Workload and Metrics

We use multi-programmed workloads comprising programs from SPEC [23] 2000 and SPEC 2006 suites to evaluate our proposal. The workloads are typically a mix of programs with varying levels of memory intensity. The workload mix used in our studies is presented in Table 2.

We use weighted speedup [21] and harmonic speedup [22] to summarize performance. We report fairness using the ratio of minimum slowdown to maximum slowdown. These terms are defined in Table 3.

6

We use the L2 MPKI to identify memory intensive programs.

Table: Workloads

<table>
<thead>
<tr>
<th>Workloads</th>
<th>Weighted Speedup (WS)</th>
<th>Minimum slowdown</th>
<th>Harmonic Speedup (HS)</th>
<th>Maximum slowdown</th>
<th>Fairness</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1: (462, 459, 433, 454, 473, 450, 445)</td>
<td>∑IPCshared/∑IPCalone</td>
<td>min(1, IPCshared/IPCalone)</td>
<td>∑N(IPCshared/IPCalone)</td>
<td>max(1, IPCshared/IPCalone)</td>
<td></td>
</tr>
<tr>
<td>E2: (300, 454, 473, 179, 450, 435, 445, 450)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E3: (410, 464, 445, 435, 445, 458)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E4: (187, 172, 173, 410, 470, 435, 444, 177)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E5: (434, 435, 450, 453, 462, 471, 164, 186)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E6: (181, 473, 401, 172, 177, 176, 179, 435)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E7: (473, 459, 445, 454, 465, 171, 197)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E8: (429, 416, 433, 454, 464, 445, 458)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Workloads

Table 3: Performance and Fairness metrics

The performance of MSRB for the quad-core case is summarized in Figure 6. As can be seen from Figure 6(a), MSRB improves the weighted speedup by 35.8% over 1×1024. The performance gain in terms of harmonic speedup, as shown in Figure 6(b), is 27.5%. All the workloads show improved performance with MSRB. This shows the importance of focusing on the temporal locality (multiple row-buffers) at the cost of spatial locality (narrow row-buffers). MSRB achieves a significantly higher row-buffer hit rate of 0.6 (in Figure 6(c)) as compared to 0.2 observed in the baseline case. The observed gains in performance are typically in line with the improvement in row-buffer hit rate.

Figure 7 shows the performance improvement in terms of weighted speedup for 8 and 16 core workloads. MSRB provides 14.5% and 21.6% improvement in performance for 8 and 16 core workloads respectively over baseline. An interesting case-study is the row-buffer hit rates experienced by the individual programs in the workload E1. Figure 7(b) shows the hit rates experienced by the individual programs for baseline and MSRB. It can be seen that the row-buffer hit rate improves with MSRB for all the individual programs. Last, MSRB improved the performance in terms of IPC of single-core SPEC2000 and SPEC2006 workloads by 7.1% on an average (refer Figure 10 for IPC values obtained for a subset of benchmarks). Observe that programs such as 459.GemsfDFTD and 462.libquantum show considerably high gains (148% and 21%).

6.2 Energy Benefits of MSRB

Improved row-hits not only boosts the performance, but also translates into energy savings due to reduced number of activations and precharges. The smaller size of the row-buffers also reduces the energy required for activate and precharge operations. The energy consumption is computed using the methodology described in Section 5.2. Figure 8 shows the DRAM energy gains provided by MSRB for the quad-core workloads. On an average MSRB reduces the energy requirements by 40% compared to the baseline. It is interesting to note that high energy gains are obtained not only for workloads showing high performance gains but also for others due to the narrow row buffers used. Further, as expected improved hit rate not only translates to high performance gains, but also into significant savings in terms of energy consumed. The total activate power reduced on an average from 357mW to 127mW. For 8 core and 16 core workloads, the energy gains are 28% and 31% respectively over the baseline.

5.3 Workload and Metrics

We use multi-programmed workloads comprising programs from SPEC [23] 2000 and SPEC 2006 suites to evaluate our proposal. The workloads are typically a mix of programs with varying levels of memory intensity. The workload mix used in our studies is presented in Table 2.

We use weighted speedup [21] and harmonic speedup [22] to summarize performance. We report fairness using the ratio of minimum slowdown to maximum slowdown. These terms are defined in Table 3.

We use the L2 MPKI to identify memory intensive programs.

Table: Workloads

<table>
<thead>
<tr>
<th>Workloads</th>
<th>Weighted Speedup (WS)</th>
<th>Minimum slowdown</th>
<th>Harmonic Speedup (HS)</th>
<th>Maximum slowdown</th>
<th>Fairness</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1: (462, 459, 433, 454, 473, 450, 445)</td>
<td>∑IPCshared/∑IPCalone</td>
<td>min(1, IPCshared/IPCalone)</td>
<td>∑N(IPCshared/IPCalone)</td>
<td>max(1, IPCshared/IPCalone)</td>
<td></td>
</tr>
<tr>
<td>E2: (300, 454, 473, 179, 450, 435, 445, 450)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E3: (410, 464, 445, 435, 445, 458)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E4: (187, 172, 173, 410, 470, 435, 444, 177)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E5: (434, 435, 450, 453, 462, 471, 164, 186)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E6: (181, 473, 401, 172, 177, 176, 179, 435)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E7: (473, 459, 445, 454, 465, 171, 197)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E8: (429, 416, 433, 454, 464, 445, 458)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Workloads

Table 3: Performance and Fairness metrics
with each bank having a 256 column wide row-buffer. 32-Bank has the same number of row-buffers as our $4 \times 256$ MSRB configuration which has 8 DRAM banks and 4 row-buffers per bank. In addition, as our scheme is orthogonal to memory-access scheduling schemes to improve row-hit rates, it is possible to have multiple row-buffers in TCM and PARBS. We refer to these configurations of TCM and PARBS enhanced with MSRB (four 256 column wide row-buffers per bank) as TCM+MSRB and PARBS+MSRB respectively.

Figure 9(a) shows the performance in terms of weighted speedup (normalized to the baseline) for PARBS, TCM, 32-Bank, MSRB PARBS+MSRB and TCM+MSRB for quad-core workloads. For all the workloads, it can be seen that MSRB performs better than PARBS and TCM. On an average PARBS and TCM provide gains of only 8.5% and 10.5% over baseline, while MSRB improves the performance by 35.8%. The interesting thing to note is that MSRB can greatly aid the performance of TCM and PARBS, as can be seen from the significantly better speedups experienced by TCM+MSRB and PARBS+MSRB schemes. The observed trend in performance is also reflected in the row-buffer hit rates exhibited by the various schemes. As can be seen from Figure 9(b), MSRB is more effective in improving row-buffer hit rates compared to PARBS and TCM applied on a top of a single large row-buffer.

It is interesting to note that 32-Bank configuration gives only a 5.9% improvement in performance over baseline (weighted speedup). This is primarily because increasing the number of banks can exploit only a fraction of the temporal locality and the limitations of having only one row-buffer per bank shows up after that.

### 6.4 Sensitivity Study

We reorganized the baseline $1 \times 1024$ row-buffer into $4 \times 256$ MSRB in our study and it outperformed all other configurations. But are there other MSRB configurations that can yield even better
performance? For example how does the MSRB configuration with $2 \times 512$ perform? As row-buffer sizes below 256 resulted in noticeable losses even in the case of single-cores, we do not evaluate them in detail. Figure 11 shows the row-buffer hit rates experienced by quad-core workloads for MSRB configurations $2 \times 512$ and $4 \times 256$. $2 \times 512$ provides a row hit rate of 0.48 while $4 \times 256$ achieves 0.60. In terms of weighted-speedup, $2 \times 512$ provided a gain of 34.5% over the baseline configuration of $1 \times 1024$.

We also simulated an eight-core system with one memory controller. In the case of a single memory controller, the performance gains provided by MSRB are further enhanced, resulting in performance improvement of 16% in terms of weighted-speedup over baseline.

6.5 Benefits of Early Precharge and Intra-Bank Parallelism

Figure 12 shows the performance in terms of weighted speedup (normalized to that of $1 \times 1024$) for MSRB, EarlyPrecharge, IntraBP and MSRB with both EarlyPrecharge and IntraBP for quad-core workloads. Also included is the performance of the baseline with a closed page policy. This scheme is equivalent to an EarlyPrecharge with a single row-buffer per bank. It can be observed that EarlyPrecharge with MSRB provides a gain of 40% over the baseline on an average. Enabling EarlyPrecharge improves performance by 30% in workload Q9 compared to MSRB. EarlyPrecharge sacrifices some of the row hits to reduce the latency of a future row-buffer miss. In cases where the reduced latency for a row-buffer miss is not sufficient to offset the loss of row-buffer hits, as in Q5, EarlyPrecharge shows a drop in performance compared to MSRB. In summary, EarlyPrecharge yields an additional performance improvement of 1.8% on top of MSRB.

IntraBP has a positive effect on every workload as to be expected. By utilizing the data bus cycles more efficiently, it achieves an average additional improvement of 4.7% on top of MSRB. While we do not present detailed results here due to lack of space, the average latency for each memory access reduces by 19% due to the increased parallelism. When EarlyPrecharge and IntraBP are both enabled, we observe an average additional improvement of 5.9%. It may also be observed that while EarlyPrecharge faired poorly in workload Q5, the combined optimization restores it to the baseline. Similarly, in workload Q9, EarlyPrecharge gains significantly and that gain is retained in the combined optimization.

6.6 Impact of Fairness-oriented and Performance-oriented Allocation

Figure 13(a) plots fairness of baseline, MSRB, and MSRB with Fairness oriented Allocation (MSRB+Fair) allocation schemes for quad-core workloads. As observed, MSRB improves fairness (over $1 \times 1024$ baseline) by 20% while MSRB+Fair improves fairness by an average of 43%. While detailed results are not included here, we observed that in several mixes, latency sensitive cores that suffered significant unfairness got a boost by allocating dedicated buffers to them. Performance oriented Allocation ensures more row-buffers for memory-intensive programs. Figure 13(b) plots the weighted speedup for the baseline, MSRB, and MSRB with Performance oriented Allocation (MSRB+Perf) allocation schemes for quad-core workloads. MSRB+Perf improves performance by 40.9% over the baseline. This corresponds to an additional improvement of 1.9% over MSRB. In programs with more memory intensive benchmarks, MSRB+Perf can improve performance by as much as 25%. Figure 13(c) shows the IPCs of the individual programs in workload Q9 with MSRB as well as MSRB+Perf. It can be seen that MSRB+Perf improves the performance of memory intensive programs 459.GemsFDTD and 462.libquantum without affecting the performance of programs like 445.gobmk and 410.bwaves which are relatively less memory intensive.

7. RELATED WORK

There is a large body of work on intelligent memory scheduling to improve row-buffer locality and bank-level parallelism ([1], [2], [3], [4], [9]). These methods generally require fairly sophisticated tracking of memory access patterns in the memory controller to drive scheduling decisions. Work on page coloring [6] and address-mapping techniques ([15], [17]) attempt to redistribute pages so as to improve performance ([8]) or reduce power ([6], [14]). Our proposed MSRB organization is orthogonal to all the above schemes and can complement them. Work on phase-change memories in [15] discusses multiple row buffers as a mechanism to render PCMs as a viable alternative to DRAMs. Though conceptually similar, we propose a practical implementation of this scheme in the context of the widely used DRAM memories without requiring major changes to the rigid JEDEC standard. Further, we illustrate other optimization opportunities enabled by the MSRB organization. The work in [16] explores the benefit of building a more full fledged SRAM cache in front of the DRAM array to catch more accesses in the cache. However, it necessitates a significant logic addition to the density-optimized DRAM design. Similarly, VCM [30] memory, introduced briefly in the 90’s by NEC, added a set of buffers shared across all the DRAM modules and introduced the notion of foreground and background operations. However it introduced significant changes to the DRAM access standard and the issue of buffer management has not been systematically addressed. Smaller row-buffers for energy-efficiency have received recent attention. Smaller row-buffers result in reduced energy consumption with minimal impact on performance [12]. Minirank [13], MCDIM [29] and Adaptive-Granularity [26] propose alternate data storage to reduce energy consumption while attempting to maintain performance. In contrast, our MSRB organization achieves performance improvement along with energy reduction and fairness improvements.

8. CONCLUSIONS

In this paper we have proposed a row-buffer reorganization for DRAMs which offers significant energy reduction while simultaneously improving performance. These dual benefits make this proposed DRAM architecture an attractive solution for today’s DRAM energy and performance issues. We discuss a feasible implementation of this architecture with minimal impact to existing DRAM standards. Our implementation opens up newer optimization opportunities such as Intra-Bank Parallelism and selective Early Precharge. Further, with MSRB different row-buffer allocation schemes can be tried to implement performance and fairness policies. We illustrated this flexibility using a pair of schemes - Fairness-oriented Allocation and Performance-oriented Allocation.
MSRB showed a performance improvement of 35.8% for quad-core workloads. This improvement was accompanied by an energy reduction of 43% in the DRAM. In comparison, state of the art memory access scheduling schemes TCM [1] and PARBS [2] were able to improve the baseline performance by only 10.5% and 8.5%. Further, the additional performance optimizations, namely Early Precharging and Intra-Bank Parallelism improved the system performance by an additional 5.9%.

9. REFERENCES