FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems

Jishen Zhao, Onur Mutlu, Yuan Xie
Penn State, CMU, UCSB, and HP Labs

MICRO - 2014
Persistent Applications:
Memory Access Behavior

**Memory Intensity:** Number of LLC misses per thousand instructions

**Write Intensity:** Portion of write misses (WR%) out of all cache misses

**Bank-Level Parallelism:** Avg. number of banks with outstanding memory requests, when at least one other outstanding request exists

**Row-Buffer Locality:** Avg. hit rate of the row-buffer across all banks

<table>
<thead>
<tr>
<th></th>
<th>MPKI</th>
<th>WR%</th>
<th>BLP</th>
<th>RBL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming</td>
<td>100/High</td>
<td>47%/Low</td>
<td>0.05/Low</td>
<td>96%/High</td>
</tr>
<tr>
<td>Random</td>
<td>100/High</td>
<td>46%/Low</td>
<td>6.3/High</td>
<td>0.4%/Low</td>
</tr>
<tr>
<td>KVStore</td>
<td>100/High</td>
<td>77%/High</td>
<td>0.05/Low</td>
<td>71%/High</td>
</tr>
<tr>
<td>Persistence Phase (KVStore)</td>
<td>675/High</td>
<td>92%/High</td>
<td>0.01/Low</td>
<td>97%/High</td>
</tr>
</tbody>
</table>

**Streaming and Random:** memory-intensive, *non-persistent applications*, performing streaming and random accesses to a large array, respectively. *(Two extreme cases of different BLP and RBL)*

**KVStore:** *Persistent application* that performs inserts and deletes to key-value pairs of an in-memory data structure.

**Persistence Phase:** When KVStore performs persistent writes.
Persistent Applications: Memory Access Behavior

- **High write intensity**: All 3 applications have the same memory intensity, however KVStore has much higher write intensity.
  
  Involves redo log updates that generate extra write traffic

- **Higher memory intensity with persistent writes**: KVStore in its persistence phase causes greatly higher memory traffic (MPKI = 675), and writes make up almost all (92%) the memory traffic.

- **Low BLP and high RBL with persistent writes**: KVStore has low BLP and high RBL. This write behavior similar to that of streaming’s reads: low BLP and high RBL.

<table>
<thead>
<tr>
<th></th>
<th>MPKI</th>
<th>WR%</th>
<th>BLP</th>
<th>RBL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming</td>
<td>100/High</td>
<td>47%/Low</td>
<td>0.05/Low</td>
<td>96%/High</td>
</tr>
<tr>
<td>Random</td>
<td>100/High</td>
<td>46%/Low</td>
<td>6.3/High</td>
<td>0.4%/Low</td>
</tr>
<tr>
<td>KVStore</td>
<td>100/High</td>
<td>77%/High</td>
<td>0.05/Low</td>
<td>71%/High</td>
</tr>
<tr>
<td>Persistence Phase (KVStore)</td>
<td>675/High</td>
<td>92%/High</td>
<td>0.01/Low</td>
<td>97%/High</td>
</tr>
</tbody>
</table>

*However, the memory bus can only service either reads or writes to any bank at any given time, because the bus can be driven in only one direction*
Inefficiency of Prior Memory Scheduling Schemes

Existing Memory Schedulers:

First Ready – First Come First Serve (FR-FCFS): Prioritizes memory requests that are row-buffer hits over others and, after that, older memory requests over others.

Thread Cluster Memory scheduling (TCM): Dynamically classifies applications into low and high memory-intensity clusters, and employs heterogeneous scheduling policies across the clusters.

However, they are designed based on main memory to used as working memory only (no persistency support), and has the fundamental assumptions:

1. **Reads are on the critical path of application execution whereas writes are usually not.** Therefore, memory scheduling schemes prioritize reads over writes.

2. **Applications are usually read-intensive, and memory controllers can delay writes without frequently filling up the write queues.** Therefore, optimizing the performance of writes is not as critical to performance in many workloads as the write queues are large enough for such read-intensive applications.
Inefficiency of Prior Memory Scheduling Schemes

The assumptions no longer hold when persistent applications use the same shared memory interface as non-persistent requests, because

i. Persistent writes, reads of in-flight persistent writes, and computations dependent on these writes to be serialized. So, *persistent writes are also on the critical execution path* and simply prioritizing read requests over persistent write requests can hurt system performance.

ii. Persistent applications are *write-intensive* as opposed to read-intensive.

Different memory access characteristics of persistent applications makes existing memory scheduler designs inefficient for two reasons:

i. the high write intensity causes frequent switching of the memory bus between reads and writes, causing *bus turnaround delays*

ii. the low write BLP causes underutilization of memory bandwidth while writes are being serviced, which delays any reads in the memory request buffer.
Naïve Extension of Existing Schedulers

Persistent applications’ writes behave similarly to streaming reads, so let's assign these persistent writes the same priority as read requests!

Simple but does not really work.

WorkLoad 1: KVStore performance improves since persistent writes gets the same priority as reads. But Streaming slows down greatly as its read requests are interfered much more frequently with the write requests of KVStore, and due to equal read and persistent write priorities, the memory bus has to be frequently switched between persistent writes and streaming reads, leading to high bus turnaround latencies.

WorkLoad 2: Random slows down the most with all of the four evaluated scheduling as it is more vulnerable to interference than the mostly-streaming KVStore due to its high BLP. FRFCFS-modified slightly improves KVStore’s performance while largely degrading random’s performance (same reason as WL1). TCM modified does not significantly affect either application’s performance.
Existing Schedulers:
1) Frequent entries into write drain mode due to high intensity of persistent writes.
2) Resulting frequent bus turnarounds between read and write requests that cause wasted bus cycles.
3) Memory bandwidth underutilization during write drain mode due to low BLP of persistent writes.

**FIRM:**
1) Use **persistent write striding** to ensure that persistent writes to memory have high BLP such that memory bandwidth is well-utilized during the write drain mode.
2) Use **persistence-aware memory scheduling** to minimize the frequency of write queue drains and bus turnarounds by scheduling the queued up reads and writes in a fair manner.
Components:

1) **Request Batching**: forms separate batches of read and write requests that go to the same row, to maximize row buffer locality,

2) **Source Categorization**: categorizes the request sources for effective scheduling by distinguishing various access patterns of applications

3) **Persistent Write Striding**: maximizes BLP of persistent requests

4) **Persistence-aware Memory Scheduling**: maximizes performance and fairness by appropriately adjusting the number of read and write batches to be serviced at a time.
Request Batching:
- The goal is to group together the set of requests to the same memory row from each source.
- A batch is considered to be formed when the next memory request in the request buffer of a source is to a different row.

Source Categorization:
- FIRM categorizes sources on an interval basis.
- At the end of an interval, each source is categorized based on its memory intensity, RBL, BLP, and persistence characteristics during the interval, predicting that it will exhibit similar behavior in the next interval.
- A persistent application is considered a persistent source when it is performing persistent writes. It may also be a non-intensive, a streaming, or a random source in other time periods.
Persistent Write Striding

Components:

1) **Request Batching**: forms separate batches of read and write requests that go to the same row, to maximize row buffer locality.

2) **Source Categorization**: categorizes the request sources for effective scheduling by distinguishing various access patterns of applications.

3) **Persistent Write Striding**: maximizes BLP of persistent requests.

4) **Persistence-aware Memory Scheduling**: maximizes performance and fairness by appropriately adjusting the number of read and write batches to be serviced at a time.
Persistence-aware Memory Scheduling

Components:

1) **Request Batching**: forms separate batches of read and write requests that go to the same row, to maximize row buffer locality.

2) **Source Categorization**: categorizes the request sources for effective scheduling by distinguishing various access patterns of applications.

3) **Persistent Write Striding**: maximizes BLP of persistent requests.

4) **Persistence-aware Memory Scheduling**: maximizes performance and fairness by appropriately adjusting the number of read and write batches to be serviced at a time.
Thank you