ECE 209AS S25 CA4 Report (Leaderboard Group: The Byte of 87)

Jacob Levinson
Computer Science
University of California, Los Angeles
Los Angeles, United States of America
jrlevinson@ucla.edu

Vincent Lin
Computer Science
University of California, Los Angeles
Los Angeles, United States of America
vinlin@ucla.edu

***Abstract*—Our scheduler for the Utah SImulated Memory Module (USIMM) competition builds upon the baseline by combining a first-ready first-come-first-serve (FR-FCFS) heuristic with auto-precharging on last hit as well as a timeout-based row closure mechanism to close idle rows. This aims for performance efficiency and improved DRAM utilization. Our best average EDP as submitted to the leaderboard is 0.378. Our total storage overhead still stays within the design constraint of the original USIMM competition.**

I. Design Methodology

A. Scheduling Algorithm

1) We continue using dynamic **write-drain** like in the baseline code. Every call to `schedule()` runs some bookkeeping logic to manage when a channel is currently in *write-drain mode*, dictated by the high and low write queue watermarks (`HI_WM` and `LO_WM` respectively). Write-drain mode determines whether we issue from the per-channel write queue or read queue. The watermarks are parameters that we sweep to find a well-performing configuration.
2) We traverse the appropriate queue to pick the oldest issuable row-hit, but only considering *open* rows in the first pass (**FR-FCFS**). If no candidate is found, we fall back to the original FCFS implementation of just choosing the oldest issuable row-hit regardless of open-ness. As covered in lecture, FR-FCFS aims to maximize row buffer hit rate, thereby maximizing DRAM throughput.
3) We then issue the chosen command. To enable our timeout mechanism, we manage an additional data structure `last_col_cycle` that keeps track of last column activity. The entry for the current (channel, rank, bank) is updated to the current `CYCLE_VAL`. Additionally, if this command is the *last* hit to this row, we **auto-precharge**. This saves cycles that would otherwise be used for explicit precharge.
4) Finally, we implement **timeouts** by scanning all banks and consulting the `last_col_cycle` state, closing rows that have not received column commands in `ROW_IDLE` many cycles as well as have no queue hits. Only one precharge is issued per idle cycle. `ROW_IDLE` is a new parameter we introduce and is another parameter we sweep to find a well-performing configuration.

In summary, this hybrid approach combines access locality, bank management, and dynamic mode switching to improve DRAM throughput and responsiveness, especially under mixed read/write workloads.

B. Parameter Sweep

We are interested in a configuration that *minimizes* the **energy-delay product (EDP)** metric (specifically, the average of the EDP reported for two benchmarks, `comm1`/`comm2` and `black`/`freq`).

As introduced before, we have three tunable parameters that influence the behavior of our algorithm:

- `HI_WM`, the write queue high water mark. We begin draining writes if write queue exceeds this value.
- `LO_WM`, the write queue low water mark. We end the write queue drain once the write queue has this many writes in it.
- `ROW_IDLE`, the number of cycles a row may stay open without hits (that is, the timeout interval).

We used shell scripts to sweep over combinations of these parameters to explore trends and discover the best-performing configuration. We observed that higher values tended to improve (decrease) EDP, so we continuously swept over larger parameter values. However, this trend eventually stopped—further increasing the parameters started to increase EDP again. We concluded that we found a sort of local minimum. Our smallest average EDP value observed was **0.378**, with configuration:

```
#define HI_WM 64
#define LO_WM 36
#define ROW_IDLE 72
```

II. Trade-offs

A. Dynamic Write Draining

Aggressive write draining as controlled by `HI_WM` and `LO_WM` minimizes write queue stalls to ensure forward progress amidst both read and write requests. However, this can delay critical reads.

B. Prioritizing Row-Hits (FR-FCFS Heuristic)

As covered in lecture, FR-FCFS reduces latency per row access and increases row buffer hit rate. However, it has the trade-off that it may be unfair in scenarios where multiple threads share the DRAM system. Because this heuristic continuously prioritizes open row accesses, there is potential *starvation* of requests that do not hit open rows.

C. Auto-Precharge on Last Hit

This optimizes row closure with minimal overhead, saving cycles that would otherwise be used for explicit precharge. However, incorrect prediction may lead to frequent row activations, which would introduce more overhead.

D. Timeout-Based Idle Row Closure

This mechanism ensures that unused rows are closed proactively, freeing banks for future accesses that may span different rows. However, this may incur premature row closures, sacrificing the advantage of open row accesses and reducing hit rate. This can happen by simply unlucky timing or specific workload patterns where same row accesses happen to be spaced out just beyond the chosen timeout interval, resulting in accesses just after closure. Furthermore, just continuously extending the interval in attempt to cover this case is not ideal either as it would undermine the purpose of the timeout mechanism in the first place, which is to dynamically balance the open and closed row states to accommodate different workload types.

There is also the runtime overhead of scanning *every* bank *every* cycle to consider timing out an open row. Depending on how this is implemented or parallelized at the hardware level, this could significantly bloat the latency of the memory controller.

III. Design Storage Overhead

A. Storage Overhead Derivations

The main difference from the baseline implementation in terms of storage is the new `last_col_cycle` data structure, which is a 3D matrix mapping banks to the CPU cycle number that bank was last accessed. This is needed such that our scheduler can "remember" the last column command per bank and implement the row idle timeout. It has declaration:

```
long long int last_col_cycle[MAX_NUM_CHANNELS
    ][MAX_NUM_RANKS][MAX_NUM_BANKS];
```

Thus, to calculate the overhead, it is just a matter of multiplying the dimensions by the data type size:

```
MAX_NUM_CHANNELS * MAX_NUM_RANKS *
    MAX_NUM_BANKS * sizeof(long long int)
```

We also consider the `drain_writes` array from the baseline implementation, which just stores binary 0/1's to keep track of whether a channel is currently in write-drain mode. It has declaration:

```
int drain_writes[MAX_NUM_CHANNELS];
```

We similarly calculate the size by multiplying dimensions by data type size:

```
MAX_NUM_CHANNELS * sizeof(int)
```

Note that we could even optimize this further by using a standard `bool` (one-byte `unsigned char`) instead of `int` since the values are just Boolean flags, but since the total space overhead is dominated by `last_col_cycle` anyway, it does not matter much.

B. Calculation For Given USIMM Setup

We plug in concrete numbers by referencing the macro constants defined in the framework header file `memory_controller.h`:

- `MAX_NUM_RANKS` $= 16$
- `MAX_NUM_CHANNELS` $= 16$
- `MAX_NUM_BANKS` $= 32$
- On most modern systems, `int` is 4 bytes, so we will use that here too.
- On most modern systems, `long long int` is 8 bytes, so we will use that here too.

Our overhead for this setup is thus:

- From `last_col_cycle`: $16*16*32*8 = 65536$ bytes (or exactly 64 KiB).
- From `drain_writes`: $16*4 = 64$ bytes.
- From miscellaneous local variables, constants, etc.: likely < 100 bytes.

We see that, while not required, our design happens to fit under the 68 KB hardware storage budget in the original USIMM competition.