Reinforcement Learning based DRAM scheduling

Pavan Holla
Ayush Gupta
Rangapriya Parathasarathy
Motivation

• DRAM bandwidth utilization is critical
Basic DRAM scheduling
Motivation

• Fixed access policies like FR-FCFS for common case behavior.
• Not efficient in adapting to dynamic workload behavior.
Motivation

Where can we optimize?

- Select load that results in maximum CPU throughput
- Decide write to read bus turnaround dynamically
- Switch between open row/close row adaptively
Objective

• Modify scheduling policy according to program behavior

• Use reinforcement learning

• Eliminate human involvement in scheduling decisions
Why Reinforcement Learning

- Maximize long term throughput
- Examples
  - If read traffic is heavy, prevent read to write bus turnaround
  - Learn that a closed page policy leads to faster reads
  - Learn that loads with many dependent instructions are high priority
References

Simulator: USIMM

- Reads in program traces.
- Models upto 16 OoO cores
- Cache misses are sent to the DDR controller
- Upto 4 channels, 2 ranks/channel, 8 banks/rank
- USIMM takes care of respecting timing constraints
RL Formulation
RL for writes

**States:**
- Number of Writes
- Number of Reads
- Time since last Read

**Actions:**
- Gate Writes
- Issue Writes
- Forced NOP
RL for writes

➢ Immediate reward of 1 for issuing a Write
➢ If memory is busy, then Forced NOP
➢ Discount the reward when we are stuck with a Forced NOP
Global RL policy

States:
- Number of Writes
- Number of Reads
- Number of Reads to open row
- ...
- ...

Actions:
- FCFS Read
- FR-FCFS Write
- FR-FCFS Read
- Precharge
- Forced NOP
Local RL Policy

Split DDR scheduler to 3 parts:

- Read RL Agent (Highest Priority)
- Write RL agent
- Adaptive Precharge module (Lowest Priority)
Local RL Policy

Why?

• State space does not explode, lower storage requirements
• Faster learning
• State-action-reward table interpretable for debugging
Write Policy

- Default write policy - Waiting for read queue to be empty for bus turnaround
- RL agent overlaps reads with writes
Some possible policies

- Too many writes in the queue, time to drain

- or Parallelize Precharge and Activates across banks
  - OK to pay WTR delay if reads and writes can be parallelized
Read RL agent

**States:**
- Expected best thread gain
- Expected FCFS gain
- Expected FR-FCFS gain
- Number of reads to active row

**Actions:**
- Issue FCFS
- Issue FR-FCFS
- Issue best thread
- Forced NOP
Read RL agent - Expected FCFS and FR-FCFS gain
Read RL agent - Number of reads to active row
Thread Cluster Memory Scheduling [2]

- Group threads into two clusters -
  - Latency sensitive cluster
  - Bandwidth sensitive cluster
- Prioritize latency sensitive cluster over bandwidth sensitive cluster to improve throughput
- Employ different algorithms within each cluster
Least intensive threads are always promptly serviced
allows them to quickly resume their computation => make large contributions to overall system throughput.
Bandwidth sensitive cluster

- higher niceness $\Rightarrow$ higher bank level parallelism $\Rightarrow$ more priority
- lower niceness $\Rightarrow$ high row buffer locality $\Rightarrow$ can cause interference to threads having more bank level parallelism
- prioritize based on niceness $\Rightarrow$ least nicest thread mostly deprioritized to avoid interference
Read RL agent - Expected best thread gain
Row buffer Management/Precharge Policy

• Open page/Close page
  • Popular choice

• Intel adaptive open page policy
  • dynamically decide open-page time interval

• Something less complicated than RL agent but better than rigid policies.
Our adaptive precharge policy

- “bad_precharge” bit for each row, only close the row if bad_precharge = 0
- bad_precharge = 1 if a row was speculatively closed but referenced again
- during normal operation, if a row is not immediately accessed again, bad_precharge = 0
Bad precharge

<table>
<thead>
<tr>
<th>bad_precharge bit</th>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Bank n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Candidate for precharge during an idle cycle
Results

We ran traces from the PARSEC benchmarks on our controller.

- Financial Analysis
- Computer Vision
- Animation
- Similarity Search
- Data Mining

RL Write agent improves over the Naive write agent by an average of 2%
Read RL agent compares well with FR-FCFS
Performs marginally better than FR-FCFS in 10/15 workloads
Performance improvement over write drain, FCFS and closed page policy
Issues Faced

- RL agent did not learn policy when the state space was large
  - Used neural networks to generalize state-action-reward table
  - Used smoothing functions to generalize state-action-reward table
  - Figured splitting the agent into parts is a good way to test.
- Tried to involve deep learning to generalize Reward table
  - Moved USIMM to C++ and integrated a deep learning framework
  - Current state space too small for deep networks, not image-like.
  - Convergence issues could arise with neural networks.
Conclusion

- RL may work well for DDR controllers
- State space formulation is key for RL
- Convergence of the Reward/Q table is important.
- Write policy is crucial for boost over conventional controllers.
Thank You