Reducing NVM Writes with Optimized Shadow Paging

Yuanjiang Ni, Jishen Zhao, Daniel Bittman, Ethan L. Miller
Center for Research in Storage Systems
University of California, Santa Cruz
Emerging Technology

Memory

- Byte-Addressable
- High speed
- Volatile
- Small capacity

BNVM

Storage

- Block-Addressable
- Slow
- Durable
- Large Capacity

Memory

Storage
New Storage Architecture

read()/write()  \[\downarrow\]  cache-line  \[\uparrow\]  load/store cache-line flush

DRAM

fsync(), etc  \[\downarrow\]  page  \[\uparrow\]  HDD/SSD

BNVM
Crash Consistency

XBEGIN
A.account -= 500,000
B.account += 500,000
XEND

Crash-consistency is a must!

A, B lost money! 😞
Opportunities

❖ Leverage byte-addressability
  ‣ e.g. Fine-grained logging.
Opportunities

❖ Leverage byte-addressability
  › e.g Fine-grained logging.

❖ Leverage virtual memory
  › Indirection is necessary for many techniques
  › Can we directly leverage virtual memory indirection?
Opportunities

❖ Leverage byte-addressability
  ‣ e.g Fine-grained logging.

❖ Leverage virtual memory
  ‣ Indirection is necessary for many techniques
  ‣ Can we directly leverage virtual memory indirection?

❖ Explore Hardware Support
  ‣ Intel proposes instructions such as clwb for especially persistent memory.
  ‣ Other HW supports?
Inefficiencies of Existing Approaches

- Extra writes to NVM are bad.
  - Performance
  - Endurance
Extra writes to NVM are bad.

- Performance
- Endurance

Logging
- Write the actual data twice

Inefficiencies of Existing Approaches
Extra writes to NVM are bad.
  - Performance
  - Endurance

Logging
  - Write the actual data twice

shadow paging
  - Copy unmodified data
Extra writes to NVM are bad.
  - Performance
  - Endurance

Logging
  - Write the actual data twice

shadow paging
  - Copy unmodified data

Our approach - OSP
Cache-line Level Mapping

Track modifications at cache line level?

Can’t simply reduce page size!
Cache-line Level Mapping

Two bits per cache line

- Committed Bit - Where is the old state?
- Updated Bit - has this cache line been updated?

Only required when pages are being actively updated!
TLB Extension

❖ Wider TLB entry
  › Committed bitmap
  › Updated bitmap
  › Additional PPN
TLB Extension

❖ Wider TLB entry
  ▶ Committed bitmap
  ▶ Updated bitmap
  ▶ Additional PPN

❖ Minimal impact on run-time performance.
  ▶ Require only few gate delays
  ▶ Done in parallel with cache access (e.g. VIPT caches)
TLB Extension

❖ Wider TLB entry
  › Committed bitmap
  › Updated bitmap
  › Additional PPN

❖ Minimal impact on run-time performance.
  › Require only few gate delays
  › Done in parallel with cache access (e.g. VIPT caches)

❖ Need not change the PTE
  › Additional information required only when pages are actively being updated.
Example

Wider TLB entry

<table>
<thead>
<tr>
<th>VPN</th>
<th>P₀</th>
<th>P₁</th>
<th>Committed</th>
<th>Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>P₀</td>
<td>P₁</td>
<td>1010</td>
<td>0000</td>
</tr>
</tbody>
</table>
Read the cache line 0

Read from $P$ (Committed_bit XOR updated_bit)

Wider TLB entry

<table>
<thead>
<tr>
<th>VPN</th>
<th>$P_0$</th>
<th>$P_1$</th>
<th>Committed</th>
<th>Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V$</td>
<td>$P_0$</td>
<td>$P_1$</td>
<td>1010</td>
<td>0000</td>
</tr>
</tbody>
</table>
Update the cache line 0

Wider TLB entry

<table>
<thead>
<tr>
<th>VPN</th>
<th>P₀</th>
<th>P₁</th>
<th>Committed</th>
<th>Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>V0</td>
<td>P₁</td>
<td>1010</td>
<td>1000</td>
</tr>
</tbody>
</table>
Update the cache line 1

Wider TLB entry

<table>
<thead>
<tr>
<th>VPN</th>
<th>P₀</th>
<th>P₁</th>
<th>Committed</th>
<th>Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>P₀</td>
<td>P₁</td>
<td>1010</td>
<td>1100</td>
</tr>
</tbody>
</table>

Widens TLB to extra entry

Writes go to P \( (\text{Committed}_\text{bit} \oplus 1) \)

And, set the \text{updated}_\text{bit}

And

\( 0 \rightarrow 1 \)
Commit

committed bitmap = (committed bitmap XOR updated bitmap)

And, clear the updated bitmap

Before

After

VPN | P₀ | P₁ | Committed | Updated
---|----|----|------------|--------
V   | P₀ | P₁ | 1010      | 1100

VPN | P₀ | P₁ | Committed | Updated
---|----|----|------------|--------
V   | P₀ | P₁ | 0110      | 0000
Abort

Clear the updated bitmap

<table>
<thead>
<tr>
<th>VPN</th>
<th>P₀</th>
<th>P₁</th>
<th>Committed</th>
<th>Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>P0</td>
<td>P1</td>
<td>1010</td>
<td>1100</td>
</tr>
</tbody>
</table>

Before

After P₀

<table>
<thead>
<tr>
<th>VPN</th>
<th>P₀</th>
<th>P₁</th>
<th>Committed</th>
<th>Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>P0</td>
<td>P1</td>
<td>1010</td>
<td>0000</td>
</tr>
</tbody>
</table>
Page Consolidation

- Double physical pages can waste memory space
- Reduce storage cost
  - Consolidating virtual pages that are not being actively updated.
  - Copy valid data into one page and free the other one.
  - TLB eviction identifies inactive virtual pages.
- Page Consolidation is not a per-transaction overhead.
Multi-page Atomicity

Consistent State Table

<table>
<thead>
<tr>
<th>VPN</th>
<th>Committed</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>....</td>
</tr>
<tr>
<td>V1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>....</td>
</tr>
<tr>
<td>V2</td>
<td></td>
</tr>
</tbody>
</table>

Can't atomically update separate locations in-place
Lightweight Journaling

Consistent State Table

<table>
<thead>
<tr>
<th>VPN</th>
<th>Committed</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>.......</td>
</tr>
<tr>
<td>V1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>.......</td>
</tr>
<tr>
<td>V2</td>
<td></td>
</tr>
</tbody>
</table>

Journaling

Completed

- V1: Bitmap1
- V2: Bitmap2
- TX-END

uncompleted

Lightweight and not a per-update overhead!
Experiment Setup

❖ Based on McSimA+
  ❖ 64-entry L1 DTLB
  ❖ Transactional workloads: array swap (SPS), hashtable (HT), RB-tree (RBT), B-tree (BT)
  ❖ *-uni : inserts/deletes in a uniformly random fashion
  ❖ *-zipf : inserts/deletes following Zipf distribution
  ❖ 1G ~ 4G footprint
  ❖ Metric: CPU flush
CPU Flashes

Reduces the number of CPU flushes by 1.6x on average
Nearly eliminate all of the consistency cost for workloads with locality
Discussion

❖ Limitations.
  ‣ Size of a transaction is limited by the TLB capacity
  ‣ Fallback path.
❖ TLB coherence for multi-threaded processes
  ‣ Overhead, correctness
❖ Work with virtual cache
Conclusion

❖ Use virtual memory system to implement efficient, transactional update
  ▶ avoid extra copies required by logging

❖ Keep two copies of each page being modified
  ▶ Track modifications at the cache line level
  ▶ Avoid the inefficiencies of traditional shadow paging

❖ Small changes to hardware: TLB extension

❖ Preliminary simulation shows great promise
Questions

Collaborators:

Yuanjiang Ni (yni6@ucsc.edu)
Jishen Zhao (jzhao@eng.ucsd.edu)
Daniel Bittman (dbittman@ucsc.edu)
Ethan Miller (elm@ucsc.edu)