Exploring Storage Class Memory with Key Value Stores

Katelin Bailey, Peter Hornyack, Luis Ceze, Steve Gribble, Hank Levy
University of Washington
core memory

flash

battery-backed DRAM

flash-backed DRAM

FeRAM

memristors

phase change memory
core memory
flash
battery-backed DRAM
flash-backed DRAM
FeRAM
memristors
phase change memory
Where does SCM (e.g., PCM) fit in?
Where does SCM (e.g., PCM) fit in?

- ✓ Fast
- ✓ Byte-addressable
- ✓ Non-Volatile
- ✓ Dense
- ✓ Cheap
Where does SCM (e.g., PCM) fit in?

✓ Fast
✓ Byte-addressable
✓ Non-Volatile
✓ Dense
✓ Cheap
Where does SCM (e.g., PCM) fit in?

✓ Fast
✓ Byte-addressable
✓ Non-Volatile
✓ Dense
✓ Cheap

- Latency
- Write asymmetry
- Wear-out
Where does SCM (e.g., PCM) fit in?

✓ Fast
✓ Byte-addressable
✓ Non-Volatile
✓ Dense
✓ Cheap

- Latency
- Write asymmetry
- Wear-out
Where does SCM (e.g., PCM) fit in?

✓ Fast
✓ Byte-addressable
✓ Non-Volatile
✓ Dense
✓ Cheap

- Latency
- Write asymmetry
- Wear-out
Why we can’t use existing systems on SCM
Why we can’t use existing systems on SCM

- Can’t just drop in an in-memory system
  - in-place updates to durable media may be interrupted without completion
  - interrupted writes could leave **bad data** or **missing references**

- Can’t just drop in a storage system
  - **overhead** from assuming block structure or from file system
Goals for our storage system

- CPU
- SCM
- DRAM
Design a storage system from scratch to:

- Utilize DRAM to offset SCM costs, without hurting durability
- Merge large object storage (e.g. file system) with small object storage from working memory
- Provide an abstraction for durability, consistency and recoverability
- Scale well enough to handle manycore systems
Our Solution: Echo
Our Solution: Echo

We chose a key value store interface...

• It provides a very simple interface to put, get and delete

• It’s size-agnostic and can handle large and small data

• Additional transaction support handles consistency issues

• Scales well with multiple reader/writers

• Can build more complicated interfaces on top
Echo is not a distributed key value store

- The entire store (worker/master) lives on a single node
- It does not shard the keyspace between worker threads
- Utilizes SCM/DRAM hybrid memory
- Treats large files and small data objects exactly the same
- Explicitly keeps versions of each committed value
Where do we see it being used?

- At the application level (e.g. text editor)
- At the system/storage level (e.g. file system)
- As a backing store for distributed systems
- At a combination of levels...
- .... wherever you want!
Where do we see it being used?

- At the application level (e.g. text editor)
- At the system/storage level (e.g. file system)
- As a backing store for distributed systems
- At a combination of levels...
- .... wherever you want!
• SCM and Other Technology Trends

• **Storage System Design**

• Echo: Implementation Insights

• Evaluation

• Summary
Echo: Storage System Goals

1. Optimize for SCM (and deal with SCM challenges)

2. Unify fine-grained accesses to small data with coarse-grained accesses to large files

3. Provide an abstraction for consistency and recovery

4. Continue to perform well as number of cores grows (scaling)

5. Provide historical versioning
The Echo Hybrid Architecture

```
put(key, value)
Local store
Worker thread
get(key, snapshot)
Local store
Worker thread
put(key, value)
Local store
Worker thread
```

- **DRAM**
- **Commit queue**
- **Master store**
- **Master threads**
- **SCM**

`put(key, value)`
`get(key, snapshot)`

`Local store`

Worker thread

`Commit queue`

Master store

Master threads
Echo: Consistency Model

We use *snapshot isolation*:

- Trades serializability for increased concurrency.
- Optimistic concurrency control, optimal for low-conflict situations
- Widely used in database and key value systems.
Echo Workflow: Put / Commit

- put(key, value)
- get(key, snapshot)
- put(key, value)
Echo Workflow: Put / Commit

Local store

Master store

Commit queue

Master threads

DRAM

Local Store

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>key1</td>
<td>&quot;steve&quot;</td>
</tr>
<tr>
<td>key2</td>
<td>&quot;hank&quot;</td>
</tr>
<tr>
<td>key3</td>
<td>&quot;luis&quot;</td>
</tr>
</tbody>
</table>

Master Store

<table>
<thead>
<tr>
<th>snapshot 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Echo Workflow: Put / Commit

Local store

Commit queue

Master store

DRAM

Worker thread

Local Store

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>key1</td>
<td>&quot;steve&quot;</td>
</tr>
<tr>
<td>key2</td>
<td>&quot;hank&quot;</td>
</tr>
<tr>
<td>key3</td>
<td>&quot;luis&quot;</td>
</tr>
</tbody>
</table>

Worker thread

Commit queue

Master threads

SCM
Echo Workflow: Put / Commit

Local store
put(key, value) → Worker thread
get(key, snapshot) ← Worker thread
put(key, value) ← Worker thread

Local store

Commit queue

Master thread

Master store

Local Store
key3   “pete”

Master Store

<table>
<thead>
<tr>
<th></th>
<th>snapshot 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>key1</td>
<td>“steve”</td>
</tr>
<tr>
<td>key2</td>
<td>“hank”</td>
</tr>
<tr>
<td>key3</td>
<td>“luis”</td>
</tr>
</tbody>
</table>

DRAM

SCM
Echo Workflow: Put / Commit

**Local Store**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Master Store**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>snapshot 1</td>
<td>snapshot 2</td>
</tr>
<tr>
<td>key1</td>
<td>“steve”</td>
</tr>
<tr>
<td>key2</td>
<td>“hank”</td>
</tr>
<tr>
<td>key3</td>
<td>“luis”</td>
</tr>
</tbody>
</table>

**DRAM**

**SCM**

worker thread

put(key, value)

get(key, snapshot)

Commit queue

worker thread

Local store

Local store

Local store

Master store

Master store

Master store
Echo Workflow: Get

Local store

Worker thread

get(key, snapshot)

Local store

Worker thread

Local store

Worker thread

put(key, value)

Commit queue

Master store

Master threads

DRAM

Local Store

key4  “dan”

Master Store

<table>
<thead>
<tr>
<th>snapshot 1</th>
<th>snapshot2</th>
</tr>
</thead>
<tbody>
<tr>
<td>key1</td>
<td>“steve”</td>
</tr>
<tr>
<td>key2</td>
<td>“hank”</td>
</tr>
<tr>
<td>key3</td>
<td>“luis”</td>
</tr>
<tr>
<td></td>
<td>“pete”</td>
</tr>
</tbody>
</table>
Echo Workflow: Get

Local store

Commit queue

Master store

Master threads

DRAM

Local Store

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>key4</td>
<td>“dan”</td>
</tr>
</tbody>
</table>

Master Store

<table>
<thead>
<tr>
<th>key</th>
<th>snapshot 1</th>
<th>snapshot 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>key1</td>
<td>“steve”</td>
<td></td>
</tr>
<tr>
<td>key2</td>
<td>“hank”</td>
<td></td>
</tr>
<tr>
<td>key3</td>
<td>“luis”</td>
<td>“pete”</td>
</tr>
</tbody>
</table>
Echo Workflow: Get

Local store

Commit queue

Master store

Worker thread

put(key, value)

Local store

get(key, snapshot)

Local store

Master threads

Worker thread

put(key, value)

Local store

DRAM

SCM

Local Store

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>key4</td>
<td>&quot;dan&quot;</td>
</tr>
</tbody>
</table>

Master Store

<table>
<thead>
<tr>
<th>key</th>
<th>snapshot 1</th>
<th>snapshot 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>key1</td>
<td>&quot;steve&quot;</td>
<td></td>
</tr>
<tr>
<td>key2</td>
<td>&quot;hank&quot;</td>
<td></td>
</tr>
<tr>
<td>key3</td>
<td>&quot;luis&quot;</td>
<td>&quot;pete&quot;</td>
</tr>
</tbody>
</table>
Echo: Conflict Detection

• Conflict detection on every commit.

• Checks **window of potential conflict** defined by:
  
  • The snapshot a transaction read from.
  
  • The snapshot at which the transaction commits.

• If anyone wrote to one or more of the **same keys** in between, there’s a conflict, and the commit aborts.

• “First committer wins” policy
Versioning

- Snapshot isolation requires you keep all versions read by someone

- So, **durable versions are created automatically** by the update mechanisms
  - keeping the versions is **cheap in execution time**
  - may require cleanup if we run out of space

- We expose the versions as **additional client functionality**.
• SCM and Other Technology Trends
• Storage System Design
  • **Echo: Implementation Insights**
• Evaluation
• Summary
Echo: Implementation Details

- The core data structure of a *store* is a hash table

- For each key, a *version table* is allocated that stores versions

- A *commit log (or queue)*, which is used to resolve conflicts and complete interrupted commits upon restart
The **asymmetry of read/write costs**

- Writes cost more than reads, and neither are as fast as DRAM

**Solution:**

- We use the local stores (in DRAM) to absorb write costs until commit time.
**CPU caching**, as it mitigates attempts at durability

- We lose durability for any data in the cache at power off.
- Likely creates inconsistency in the store on restart

**Solution:**

- Flush important metadata and state on commit, to ensure durability
- Flush cached SCM data on power fail with a super-capacitor
The *reachability problem*.

- Memory may be durably allocated in system structures, but lost before the pointer is returned or assigned.
- Persistent (growing) memory leaks

**Solution:**

- Echo attempts to note state in structures during each step
- `malloc()` would need to be redesigned
Echo: Implementation Details

Many more implementation details in the Echo paper:

- version collection
- conflict detection
- recovery methods
- accessing historical data
- ...

• SCM and Other Technology Trends

• Storage System Design

• Echo: Implementation Insights

• **Evaluation**

• Summary
Comparison Points

We want to achieve the durability of the block-oriented disk-reliant stores, with the performance of the top in-memory key value stores.

- We compare to Google’s LevelDB as an example of the durable store optimized for disk
- We compare to Masstree as an example of a highly optimized in-memory store
- We provide slightly different functionality from either of these stores.
Google’s LevelDB

• Basis for BigTable.

• Key value store explicitly designed for a block-oriented system.
  • Writes to large contiguous portions of memory, then flushes to disk.
    • Stands to lose anything not yet flushed.
    • Can force synchronous writes to be durable.

• Depends on a file system interface.

• We run LevelDB on three different systems: RAMDisk, SSD, Disk, analyzing its performance on each
Masstree

- Highly optimized for **speedy in-memory** access
- Based on a B-tree interface that allows for in-order range queries
- Does not provide multi-key transactions
  - Does make guarantees about multiple updates to the same key
- Provides a logging function for durability (optional)
Echo Evaluation: Setup

AMD Magny-Cours Opteron with **quad 1.9GHz 12-core** processors (48 cores total).

- 64GB of memory.
- Red Hat Enterprise Linux Server 6.3 (Santiago).
- The SSD used in our experiments was a 180GB high performance 520 series Intel Solid State Drive with SATA 3.0 Gb/s connection.
- The disk and ramdisk were 120GB and 4GB, respectively.
- The filesystems mounted on all three media were **ext4**.
Where are the SCM results?

We don’t have SCM...

• we need SCM to sit on the memory bus and have the access characteristics we expect from, e.g. PCM

• available modules simply don’t have those capabilities yet.

What we do:

• We use DRAM for most of our results

• We use architectural-level simulation to understand the impact of SCM

• We will have access to Intel emulators to run more accurate evaluations
Echo Evaluation: Goals

In order to better analyze next-generation stores (as exemplified by Echo) we evaluate...

1. **base latency** as a commitment to client

2. **scalability** across many cores

3. **durability** of valuable data
Echo: Base Latency

Single Op Latency (usecs) Osize=1024byte

- PUT
- UPDATE
- GET

Masstree
- PUT: 3.2
- UPDATE: 3.2
- GET: 2.6

Echo
- PUT: 5.8
- UPDATE: 5.2
- GET: 2.2
Echo: Base Latency

- **Single Op Latency (usecs) Osize=1024byte**
  - **PUT**
  - **UPDATE**
  - **GET**

<table>
<thead>
<tr>
<th></th>
<th>PUT</th>
<th>UPDATE</th>
<th>GET</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Masstree</strong></td>
<td>3.2</td>
<td>3.2</td>
<td>2.6</td>
</tr>
<tr>
<td><strong>Echo</strong></td>
<td>5.8</td>
<td>5.2</td>
<td>2.2</td>
</tr>
</tbody>
</table>

- **LevelDB Disk**
  - 13105.4
  - 13382.3
  - 44.0
Echo: Base Latency

![Graph showing single op latency for PUT, UPDATE, and GET operations for different storage systems: Masstree, Echo, LevelDB SSD, and LevelDB Disk. The y-axis represents single op latency (usecs) for OpSize=1024byte, ranging from 1 to 1000000. The x-axis represents different operations and systems. The latency values are as follows:

- Masstree: PUT=3.2, UPDATE=3.2, GET=2.6
- Echo: PUT=5.8, UPDATE=5.2, GET=2.2
- LevelDB SSD: PUT=1248.1, UPDATE=1274.5, GET=44.3
- LevelDB Disk: PUT=13105.4, UPDATE=13382.3, GET=44.0]
## Echo: Base Latency

<table>
<thead>
<tr>
<th></th>
<th>PUT</th>
<th>UPDATE</th>
<th>GET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Masstree</td>
<td>3.2</td>
<td>3.2</td>
<td>2.6</td>
</tr>
<tr>
<td>Echo</td>
<td>5.8</td>
<td>5.2</td>
<td>2.2</td>
</tr>
<tr>
<td>LevelDB RAM</td>
<td>71.7</td>
<td>71.0</td>
<td>4.0</td>
</tr>
<tr>
<td>LevelDB SSD</td>
<td>1248.1</td>
<td>1274.5</td>
<td>44.3</td>
</tr>
<tr>
<td>LevelDB Disk</td>
<td>13105.4</td>
<td>13382.3</td>
<td>44.0</td>
</tr>
</tbody>
</table>
Echo: Scalability

Throughput: Aggregate Ops/Sec

- Blue plus: Echo
- Orange square: Masstree- not recoverable
- Green circle: Masstree- recoverable
- Red cross: LevelDB RAM

Number of Cores in System

0M 0 8 16 24 32 40 48
Echo: Impact of SCM Latency

- Results normalized to running on an all-DRAM system
Echo: Impact of SCM Latency

- Results normalized to running on an all-DRAM system
Echo: Impact of SCM Latency

- Results normalized to running on an all-DRAM system
Echo: FUSE Filesystem Case Study

• Each **file is a sequence of blocks**, to avoid having to read/write/modify huge chunks of data for each key
  
  • the key itself is a SHA1 hash of the block being stored.

• Data is committed (and versions created) at file close
  
  • Thus, Echo **recalls all saved versions** of the file, inherently

• EchoFS is functional, though slow: further optimization is needed
Echo: Zile Text Editor Case Study

- **Application metadata** such as the kill ring, buffers, and undo-redo logs are all easily added to Echo
  - In most cases this simplified the code

- Like EchoFS, each file is split into blocks and also stored in Echo, versions are stored on edits.
  - e-Zile recalls all saved edits and versions of the file
  - The modified e-Zile is functional, used for development!
• SCM and Other Technology Trends

• Storage System Design

• Echo: Implementation Insights

• Evaluation

• **Summary**
Echo is a viable storage system for SCM technologies that achieves our goals:

- Unifies large and small data accesses well
- As fast and scalable as highly optimized stores
- As durable and recoverable as disk-based stores
- Provides additional versioning at low cost
We have additional work in this area:

come find us to talk about other projects.

... SCM processes that outlive the kernel

... virtual memory protection and translation for SCM
Questions?

Contact at katelin@cs.washington.edu
http://www.cs.washington.edu/homes/katelin
Additional Results
Echo: Transaction Cost

![Graph showing single operation latency for puts (usecs) vs. number of puts in transaction for Echo PUT and LevelDB RAM PUT.](image-url)

- **Echo PUT**: Lower latency as the number of puts increases.
- **LevelDB RAM PUT**: Higher and more constant latency across various numbers of puts.
Echo: Varying Granularity

![Graph showing the relationship between Single Op Latency for Gets (usecs) and Value Size (bytes) for Echo GET, Masstree GET, and LevelDB RAM GET.](image)
Echo: Supporting Cache Flush

Value Latency per Op (useces)

- PUT
- UPDATE
- GET

Echo
Echo-FLUSH
LevelDB RAM
Echo: Impact of Cache Size

![Graph showing overhead for Hybrid Design and SCM Only with transaction size as a fraction of cache varying from 0.125 to 0.5. The overhead for Hybrid Design is generally lower than SCM Only.](image-url)