Dalí: A Periodically Persistent Hash Map

Faisal Nawab*¹, Joseph Izraelevitz*², Terence Kelly*, Charles B. Morrey III*, Dhruva R. Chakrabarti*, and Michael L. Scott²

¹Department of Computer Science
University of California, Santa Barbara
nawab@cs.ucsb.edu

²Department of Computer Science
University of Rochester
{jhi1,scott}@cs.rochester.edu

The 31st Annual International Symposium on Distributed Computing (DISC)
October 17-19, 2017
Vienna, Austria

*Work done while at HPE Labs
Motivation

- Persistent non-volatile main memory (NVM) is coming
  - Byte addressable
  - Access times comparable to DRAM
  - Survives power outages

- Opportunity for fast durable storage
Motivation

• Persistent non-volatile main memory (NVM) is coming
  • Byte addressable
  • Access times comparable to DRAM
  • Survives power outages

• Opportunity for fast durable storage

• But caches and registers are expected to remain transient!
A practical concern

• How to ensure a meaningful persistent state in the wake of a crash?
Ordering persistent writes

• Writes to persistence *must* be ordered by the programmer, otherwise consistency can be violated.

![Diagram](ptr -> data=42)
Ordering persistent writes

• Writes to persistence must be ordered by the programmer, otherwise consistency can be violated.
Ordering persistent writes

• Writes to persistence must be ordered by the programmer, otherwise consistency can be violated.
Ordering persistent writes

- Writes to persistence **must** be ordered by the programmer, otherwise consistency can be violated.
Ordering persistent writes

• Writes to persistence *must* be ordered by the programmer, otherwise consistency can be violated.

• Because caches can leak, we need to persist the node *before writing the pointer.*
Ordering persistent writes

• Writes to persistence must be ordered by the programmer, otherwise consistency can be violated.

• We call this philosophy **Incremental Persistence**
Incremental Persistence

• Current ISA support is limited
  • CLFLUSH – write cache line to persistence
  • MFENCE – ensure prior flushes are persisted

1. \texttt{data\.st(42)}
   \texttt{clflush(&data)}
   \texttt{mfence()}

2. \texttt{ptr\.st(&data)}
   \texttt{clflush(&ptr)}

\texttt{ptr}

\texttt{data=42}
Incremental Persistence

- Current ISA support is limited
  - CLFLUSH – write cache line to persistence
  - MFENCE – ensure prior flushes are persisted

```
data.st(42)
clflush(&data)
mfence()
ptr.st(&data)
clflush(&ptr)
```

Enforce ordering, require round trip latency to main memory
The Problem

• Incremental persistence is slow
  • Multiple high latency ordering fences per operation
• Our work: *How can we reduce this overhead?*
Our Idea: Periodic Persistence

• A new *design philosophy* in contrast to incremental persistence.

• Instead of “flushing as you go,” *periodically ensure all cached data becomes persistent*.

• The recovery procedure avoids any stores which may not be persistent.
Contributions

• Present *periodic persistence* as an alternate design philosophy for durable data structures.
• Implement *Dalí*, a periodically persistent hash map.
• Prove Dalí correct under *buffered durable linearizability*.
• Demonstrate our Dalí prototype improves on state-of-the-art performance.

*Izraelevitz, Mendes, Scott, DISC ‘16*
Our Idea: Periodic Persistence

• A new *design philosophy* in contrast to incremental persistence.

• Instead of “flushing as you go,” *periodically ensure all cached data becomes persistent*.

• The recovery procedure avoids any stores which may not be persistent.
Periodic Persistence
Periodic Persistence

List becomes persistent using a global fence to persist cached data.
Periodic Persistence

Append nodes to the list, modifying head
Periodic Persistence
Periodic Persistence

Recovery procedure chooses the most recent list that is guaranteed to be persistent (might lose some recent operations)
Infrastructure

• *Global fence* pushes updates into persistence
  • If a global fence *begins* and *ends* between a store and a crash, the store will be persistent.
Infrastructure

- *Global fence* pushes updates into persistence
  - If a global fence **begins** and **ends** between a store and a crash, the store will be persistent.
    - We use a whole cache flush as the global fence.
Infrastructure

• *Global fence* pushes updates into persistence
  • If a global fence *begins* and *ends* between a store and a crash, the store will be persistent.
    • We use a whole cache flush as the global fence.
Infrastructure

• *Global fence* pushes updates into persistence
  • If a global fence *begins* and *ends* between a store and a crash, the store will be persistent.
    • We use a whole cache flush as the global fence.

Alle epoch 1 stores are persistent after epoch 2.
Infrastructure

- *Global fence* pushes updates into persistence
  - If a global fence *begins* and *ends* between a store and a crash, the store will be persistent.
    - We use a whole cache flush as the global fence.
Infrastructure

- **Global fence** pushes updates into persistence
  - If a global fence **begins** and **ends** between a store and a crash, the store will be persistent.
    - We use a whole cache flush as the global fence.
Infrastructure

• *Global fence* pushes updates into persistence
  
  • If a global fence **begins** and **ends** between a store and a crash, the store will be persistent.
  
  • We use a whole cache flush as the global fence.

Failure in epoch 4 means stores in epochs 3, 4 are not guaranteed persistent *(the epochs “failed”)*
Infrastructure

- *Global fence* pushes updates into persistence
  - If a global fence begins and ends between a store and a crash, the store will be persistent.
    - We use a whole cache flush as the global fence.
Infrastructure

- **Global fence** pushes updates into persistence
  - If a global fence **begins** and **ends** between a store and a crash, the store will be persistent.
    - We use a whole cache flush as the global fence.
Periodic Persistence

Global fence divides list sections, epoch number determines which list to use.
Dalí

• Periodically persistent *hash map*
• Provably correct under *buffered durable linearizability*
Dalí

- Each bucket is an append only list
  - All updates add new node with key and new value
  - Deletes store a tombstone
  - Gets find most recent node that matches the key
  - Infrequently reclaim outdated key values
  - Lock the bucket for updates and reads

- How do we maintain the head?
Dalí Bucket Metadata

ptr0

ptr1

ptr2
Dalí Bucket Metadata in Epoch 3

As of epoch 3,

- **Committed (1)**
- **In-Flight (2)**
- **Active (3)**

State word: single 64-bit word
Now in Epoch 4, after a global fence

As of epoch 3,

- **Committed (1)**
- **In-Flight (2)**
- **Active (3)**

*State word*: single 64-bit word
Dalí Update in Epoch 4

As of epoch 3,

- **Committed (1)**
- **In-Flight (2)**
- **Active (3)**

State word: single 64-bit word
Dalí Update in Epoch 4

As of epoch 4, **Active** (4)

**Committed** (2)

**In-Flight** (3)

State word: single 64-bit word
As of epoch 4, **Active** (4)  
**Committed** (2)  
**In-Flight** (3)

State word: single 64-bit word

An update changes state word and adds the new node*.  
*We ensure both updates occur in the same epoch.
Dalí After a Crash in Epoch 4

As of epoch 4,

- **Active (4)**
- **Committed (2)**
- **In-Flight (3)**
Dalí After a Crash in Epoch 4

As of epoch 4,
- Active (4)
- Committed (2)
- In-Flight (3)

As of epoch 3,
- Active (3)
- Committed (1)
- In-Flight (2)
Dalí After a Crash in Epoch 4

As of epoch 4,

- **Active**: 4
- **Commited**: 2
- **In-Flight**: 3

As of epoch 2,

- **In-Flight**: 1
- **Active**: 2
- **Commited**: 0
Dalí After a Crash in Epoch 4

As of epoch 4,

**Active** (4)

**In-Flight** (3)

**Committed** (2)

As of epoch 2,

**In-Flight** (1)

**Active** (2)

**Committed** (0)

All state words will result in the correct dereference of the persistent suffix!
Dalí Rotations

- State word rotations for updates depend on
  - Presence/absence of recent crashes
  - Prior pointer assignments
- Details in the paper
Dalí Correctness

• Dalí is correct under buffered durable linearizability*. 

*Izraelevitz, Mendes, Scott, DISC ‘16*
Buffered Durable Linearizability*

• Data structure is linearizable in the absence of crashes
• After a crash, some operations may be lost, but no lost operation will have happened-before a preserved one.
  • The data structure’s state matches a consistent-cut across the happens-before order.

• For Dalí, this consistent cut is the end of the most recent successful epoch before the crash.
  • If the crash occurs in epoch F, the cut is consistent with the end of epoch F-2.
  • Proof of correctness in the paper.

*Izraelevitz, Mendes, Scott, DISC ‘16
Performance

• 4 x Intel Xeon E7-4890 v2 processors, each with 15 cores (60 hardware threads w/o hyperthreading).
• Keys are 8 bytes in length, and are drawn randomly from a uniform distribution of 100 million values. Values are 1000 bytes in length.
• We initialize the map with all keys in the key range, transactions consist of 3 gets/1 update.
Prototype Performance

![Graph showing throughput vs. number of threads for Dali, FOEDUS, IP, and Silo.](Image)

*Kimura, SIGMOD ’16  †Tu et al., SOSP’13
Future Work

• Generalizing periodic persistence
  • Applications to STM, other data structures

• Hardware support for global fences
Conclusion

• Present *periodic persistence* as an alternate design philosophy for durable data structures.
• Implement *Dalí*, a periodically persistent hash map.
• Prove Dalí correct under *buffered durable linearizability*.
• Demonstrate our Dalí prototype improves on state-of-the-art performance.
Backup Slides
Dalí Rotations

• State word rotations for updates depend on
  • Presence/absence of recent crashes
  • Prior pointer assignments

• They follow 3 rules
  • [C] The new committed pointer was written at least 2 epochs prior.
  • [F] The new in-flight pointer is the most recent node added in the previous epoch, or invalid otherwise.
  • [A] The new active pointer was last written before the new committed pointer, or was invalid.
Proof Sketch

• After a crash, some suffix of the list is contiguous and persistent,
  • Call the head \( v \).
  • At some point in time, a bucket pointer pointed to this suffix. Call this pointer \( ptr_v \).
• By [A], \( ptr_v \) will still point to \( v \) after a crash, since it has not yet been superseded by a more recent committed pointer.
• The state word after the crash will assign \( ptr_v \)'s to one of three roles.
  • In all cases, a reader seeing the associated state word will use \( v \) as the head.
Dalí Details

• **Transactions**: by ensuring all updates occur in a single epoch, we can guarantee they all become persistent at the same time.

• **Garbage Collection / Compression**: we remove shadowed nodes and consolidate existing values into big nodes. These are then linked into the list after the next periodic fence.

• **Optimization**: updates to the same key in the active epoch can overwrite the current node.
Dalí Bucket Metadata

ptr0

ptr1

ptr2

Update

ptr0

ptr1

ptr2
Dalí Bucket Metadata

ptr0

ptr1

ptr2

Crash

ptr0

ptr1

ptr2