SDC

STORAGE DEVELOPER CONFERENCE

SNIA ■ SANTA CLARA, 2016

In-memory Persistence: Opportunities and Challenges

Dhruva Chakrabarti

Hewlett Packard Labs

Cheap DRAM = emergence of in-memory systems

256GB of DRAM is common, and 1-2TB DRAM is affordable!

- Applications have moved from disk optimized to completely in-memory system
  - Memcached : Facebook stores 25% of all data in DRAM (RAMCloud 2009)
  - SAP HANA: in-memory database on TBs of RAM
  - Spark: In-memory processing with query latency in seconds

- But persistent storage is still needed for durability and availability

HPE DL580 servers
Upto 3TB of DRAM

Non-volatile memory is here

HPE Unveils Persistent Memory NVDIMMs For ProLiant Gen9 Servers

By Paul Alcorn MARCH 31, 2016 9:01 AM - Source: Toms IT Pro

TAGS : Storage HPE

HPE announced that its new ProLiant Gen9 server portfolio will be the industry's first server platform to provide native support for NVDIMMs with its HPE Persistent Memory products. Persistent memory is a hot topic as the storage and memory industry reach for the next level of performance beyond flash.

SSDs provide incredible performance in comparison to disk-based storage, and one of the attractive SSD attributes is its persistence. This means that any data written to the device will survive when power is removed from the device, such as during a power loss or a simple power cycle. There are other types of memory that are exponentially faster than flash, such as DRAM, but they lose data when the power source is removed.

THENEXTPLATFORM

HOME COMPUTE STORE CONNECT CONTROL CODE ANALYZE HPC ENTERPRI

INTEL REVEALS PLANS FOR OPTANE 3D XPOINT MEMORY

August 18, 2015 Timothy Prickett Morgan

INNOVATION

Inside The Machine: Hewlett Packard Labs mission to remake computing

Hewlett Packard Labs reveals the progress it's making in its attempt to reinvent computing for the era of big data.

By Nick Heath | December 8, 2015, 6:57 AM PST

SDC 16

Benefits of non-volatile memory

- Three key features of non-volatile memory (NVM):

1. Persistence = data retention even after power off

2. low latency = read/write in < 100ns (DRAM like)

3. byte addressability = use load/store instructions

Why should we care?

SDC 16

Use case 1: Making filesystems faster

- NVM devices are 10x-100x faster than flash → faster filesystem operations by just using NVM

- Filesystems can be made NVM-aware for further improvements. E.g., ext4-with-DAX
  - Page cache is used to buffer read and writes
  - In the presence of NVM, remove page cache and write directly to NVM devices. Reduces extra copies.

- Research file systems: Intel PMFS [Eurosys 2014], Microsoft BPFS [SOSP 2009]

SDC 16

Use case 2: Improving performance of databases

- Databases have high overheads to maintain consistency in the presence of failures
  - Transaction implementation require logging and concurrency control mechanisms

- In the presence of NVM, transaction commit protocols can leverage byte addressable persistence
  - Variety of new approaches: in-place updates with logging, copy-on-write updates with no logging, etc. [Arulraj et. al. Sigmod'15]

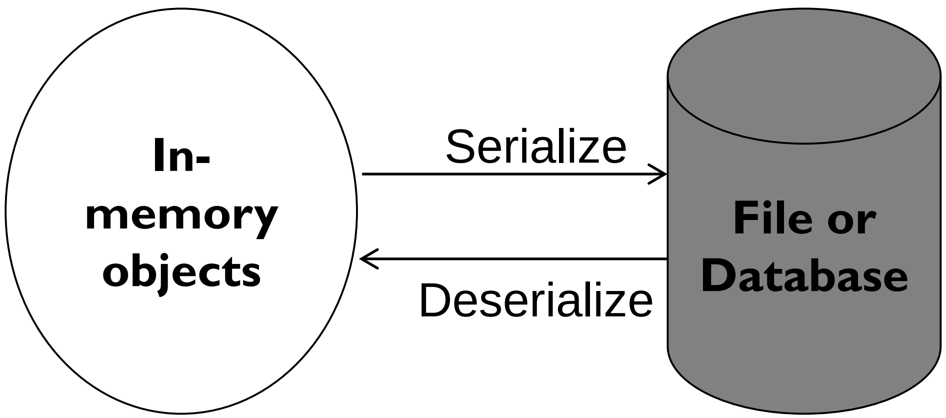
- NVM can improve database performance by more than 5x in some cases

Use case 3: Persistence for everyone

- Object level persistence
  - Single format of data
  - Load/store access
  - No translation layer
  - Possibly part of programming languages

- A new class of applications with persistence as a first class citizen, not an after-thought

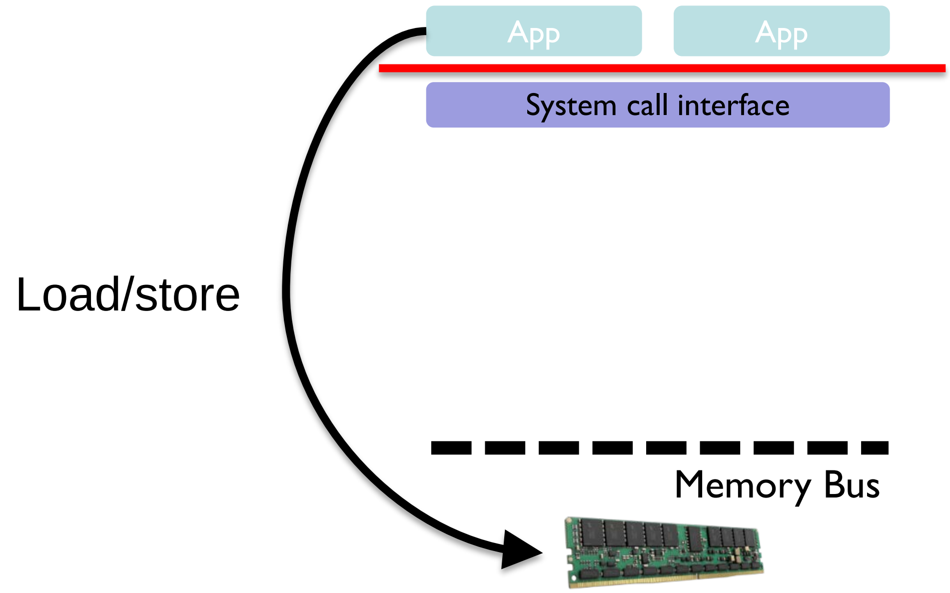
Traditional approach to durability



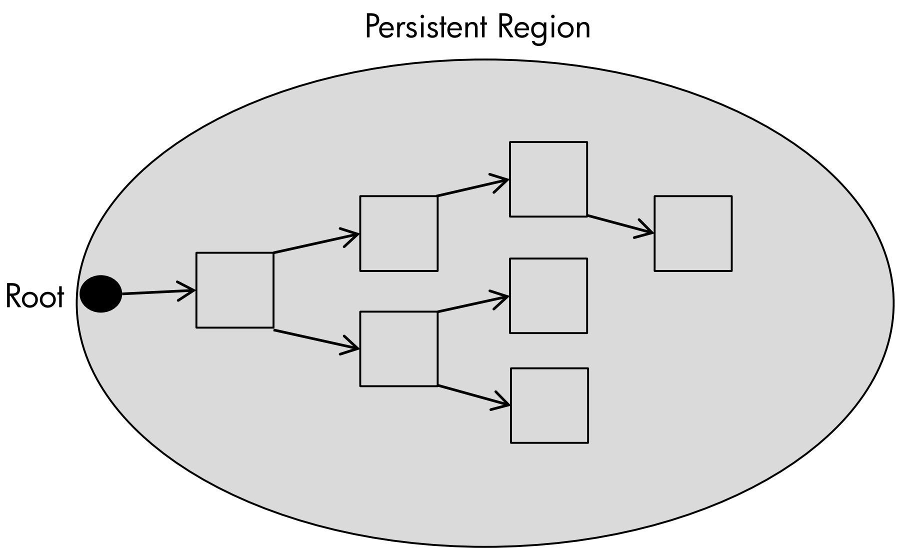
- Separate object and persistent formats
- Translation code
- Programmability and performance issues

User Persistent Memory

- Challenge: consistency



Overall abstraction: Object-level persistence



Outside data is considered transient

Programming principles: Orthogonal Persistence

(Atkinson et al., SIGMOD Record ‘96)

- Persistence independence
  - Transient and persistent data are manipulated uniformly
- Data type orthogonality
  - Persistence should not depend on the type of data
- Persistence by reachability
  - All data reachable from a persistent root are persistent

SDC 16

An Example: PJava

- Any object transitively reachable from the persistent map is made persistent
- Small amount of additional code required
- Compiler and standard libraries are unchanged

```
Book bk1 = new Book(3);  // transient allocation
Book bk2 = new Book(8);
PJavaStore library = PJavaStore.getStore();   // root location of the persistent region
library.newPRoot(“Book_main”, bk1);  // make allocated book persistent
```

The E persistent programming language

(Richardson et al., TOPLAS '93)

- Extension of C++
- Supports transparent persistence
- Database type (or db type) introduced
- Persistent storage class introduced
- Persistence by allocation, not reachability

Example:

```
persistent dbint total_count = 0;
```

Summary of past research

- Orthogonal persistence was a major consideration
- Object oriented databases: a number of flavors were tried
- Some flavors of transactional support provided
- The underlying storage technology (hard disk) was abstracted away
- Performance was an issue since no real persistent memory was available

Should we re-think persistent programming principles for the new hardware?

What are some primary challenges?

<u>Common programming idiom:</u>

```
tmp = nvm_malloc()              // allocate a persistent location

init(tmp)                       // initialize the allocated location

persistent_ptr  = tmp           // publish it, assigning a persistent pointer to the location
```

Consistency management

Due to compiler/hardware reordering:

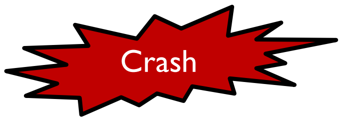
```
tmp = nvm_malloc()          // allocate a persistent location
persistent_ptr  = tmp       // publish it, assigning a persistent pointer to the location
init(tmp)                   // initialize the allocated location
```

Need for Ordering Constraints

<u>A failure may lead to a dangling persistent pointer:</u>

```
tmp = nvm_malloc()            // allocate a persistent location

persistent_ptr  = tmp         // publish it, assigning a persistent pointer to the location
```



Required Ordering Constraints

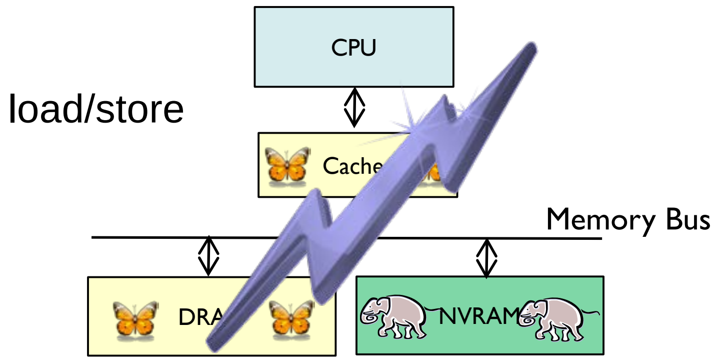
Requires initialization to be ordered before publication:

```
tmp = nvm_malloc()            // allocate a persistent location

init(tmp)                     // initialize the allocated location

persistent_ptr  = tmp         // publish it, assigning a persistent pointer to the location
```

Architectural model



- Challenges
  - Fail-stop fault model
  - Only data in NVRAM survives
  - Transient data does not survive

Programming model and principles (Atlas)

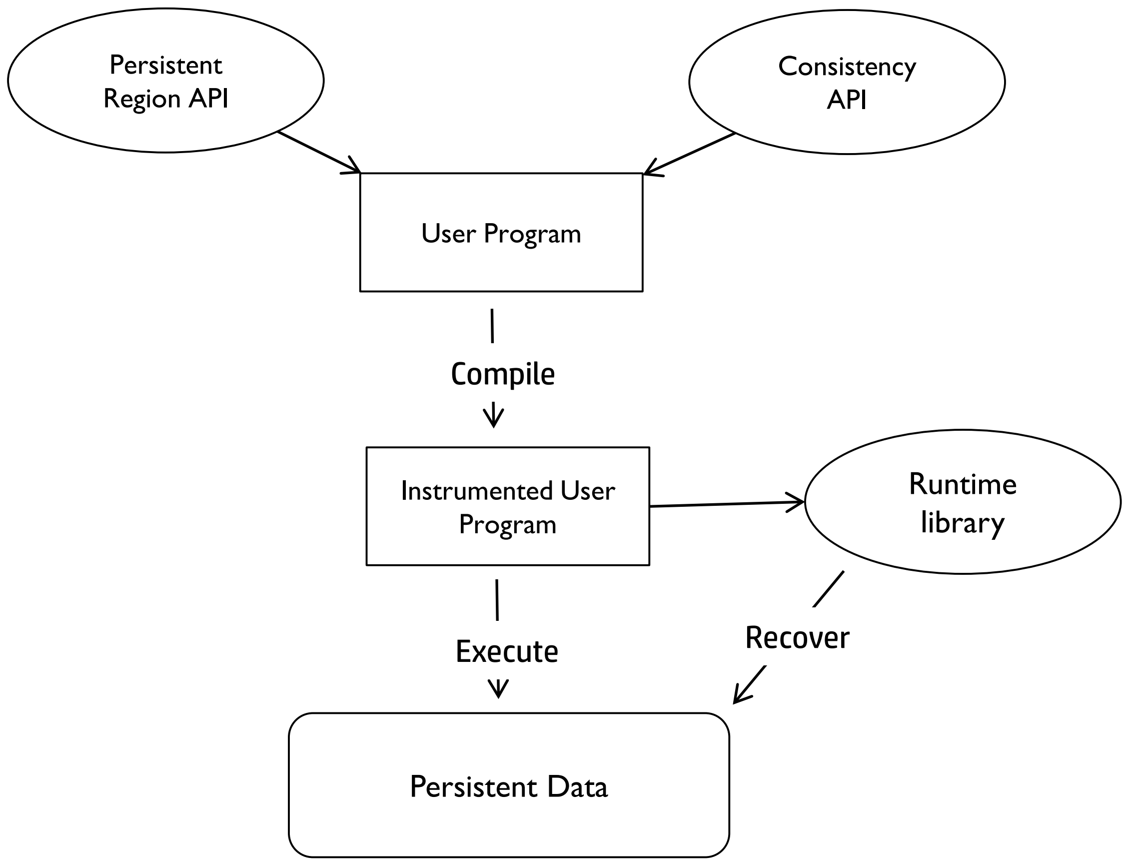
- Persist and reuse (instant restart)
- One format of data
- Fine grained updates: directly byte-addressed with CPU loads and stores
- Preserve existing programming models as much as possible

Eliminating failure-induced inconsistencies

- Support for failure-atomic updates

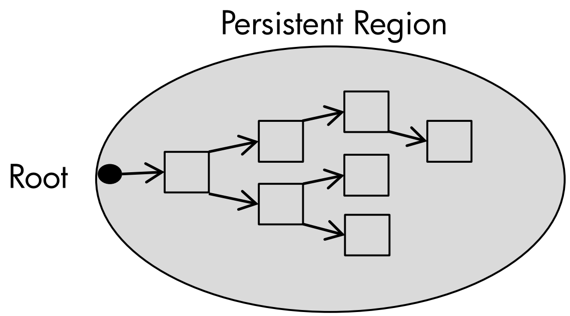
```
transfer(int from, int to, double amount) {
   __durable {
       accounts[from] -= amount;
       accounts[to]   += amount;
   }
}
```

Use model



Persistent
Region API
Consistency
API
User Program
Compile
Instrumented User
Program
Runtime
library
Execute
Recover
Persistent Data

Persistent Region API



Outside data is considered transient

Basic program structure

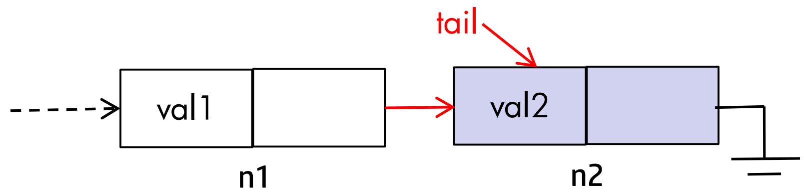
```
pr = find_or_create_persistent_region(name);
persistent_data = get_root_pointer(pr);
if (persistent_data is absent) {

      // initialize persistent_data
      p_addr = pmalloc(pr, size);
      set_root_pointer(pr, p_addr);
}
else {
   use_persistent_data()
}
```

Consistency API

- lock/unlock
  - Outermost critical sections are failure-atomic (FASE)
  - Transparent compatibility with existing threaded software
- durable {}
  - Durable sections (no isolation)

A multithreaded persistent queue using Atlas




```
pr = find_or_create_persistent_heap("queue");
q = get_root(pr);
if (q is absent) initialize q and call set_root(pr, q)
```

Thread 1

```
enqueue(val) {
    pmalloc()
    init_node()
    L.lock()
    attach_node()
    move_tail()
    L.unlock()
}
```

Thread 2

```
dequeue() {
    L.lock()
    elem = read_head()
    move_head()
    L.unlock()
    return elem
}
```

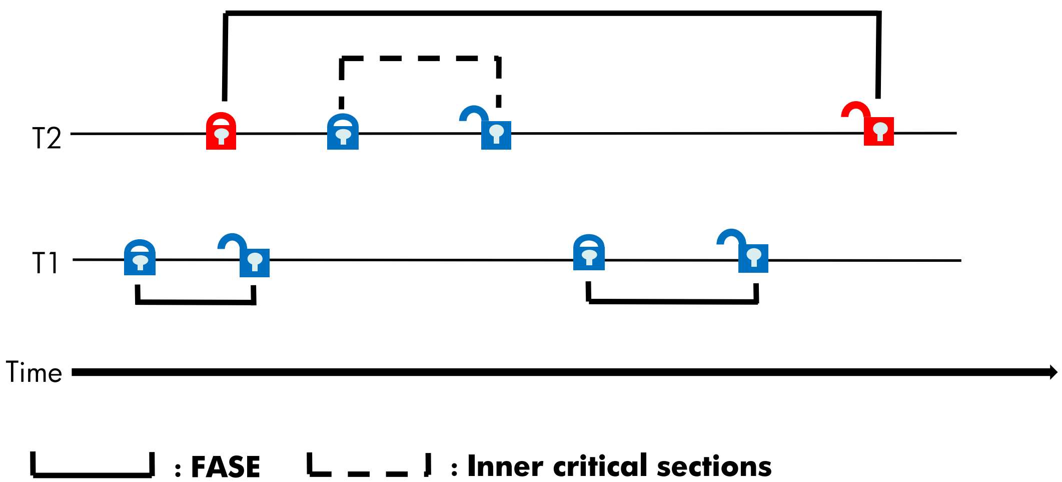
Consistent section

Current implementation

- Compiler support built on top of LLVM
- Library support for C/C++ environment on single node
- First-class persistence support for lock-based programs
- Compiler/runtime generate cache line flush/invalidate
- Persistent logs automatically written at runtime
- If there is a failure, recovery initiated automatically before program restart

Code available at https://github.com/HewlettPackard/Atlas

Notion of a Failure-Atomic SEction (FASE) (OOPSLA 2014)



Unlocked program points are thread-consistent

Outermost critical sections are failure-atomic

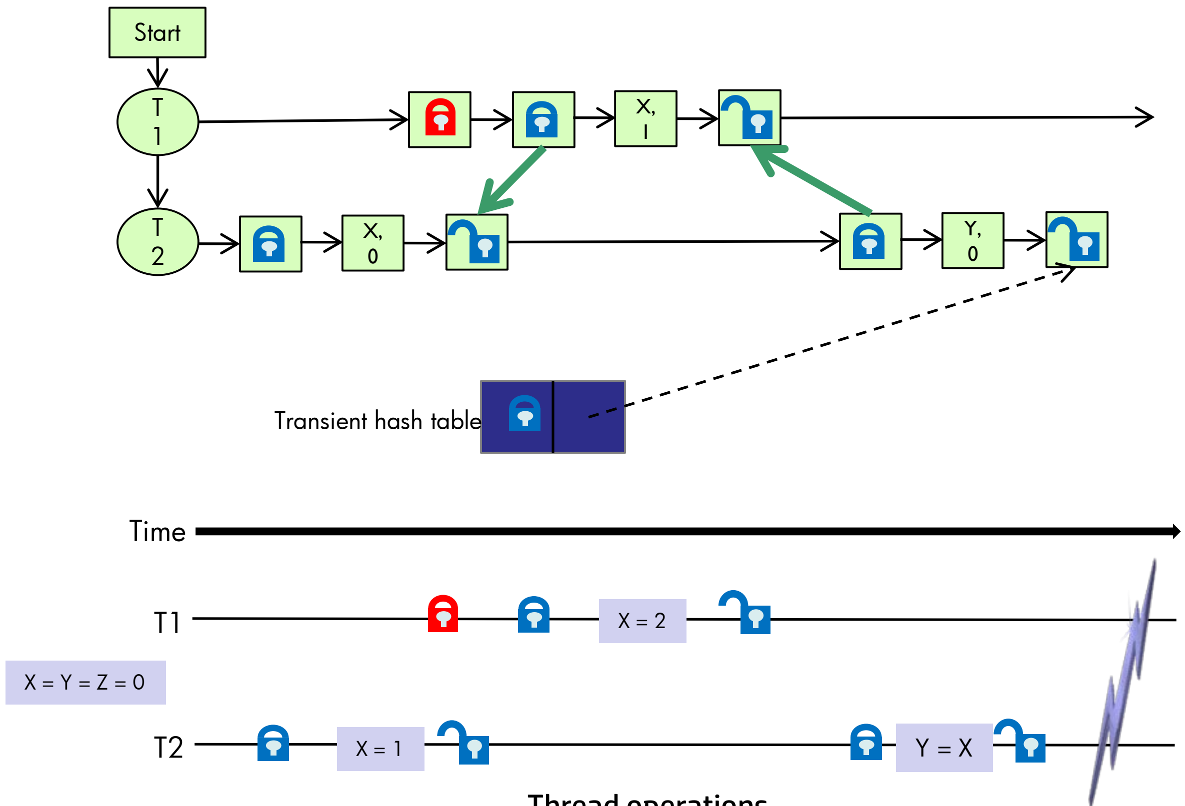
Property 1

If an update within a FASE is durable, all persistent updates within it are also durable

Other properties

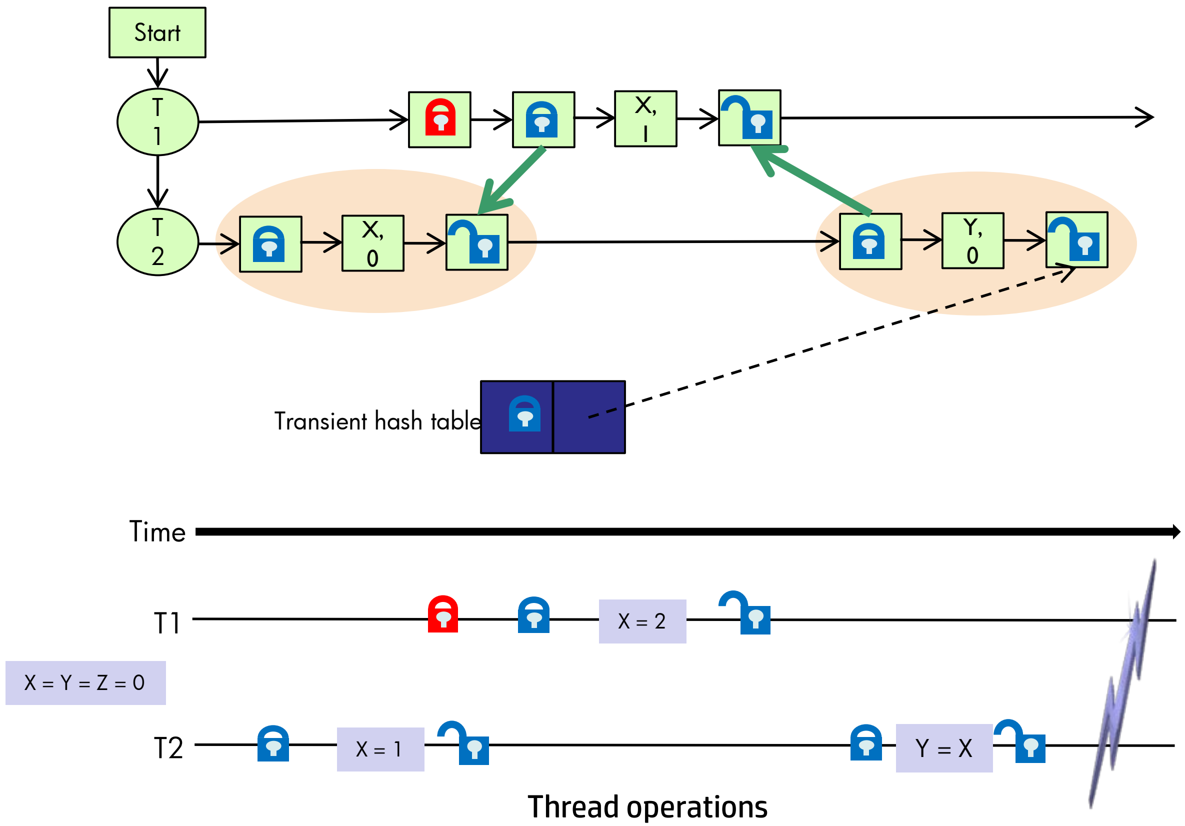
- FASEs may have to be persisted in order
- Handling updates outside FASEs

Persistent Undo Log Structure

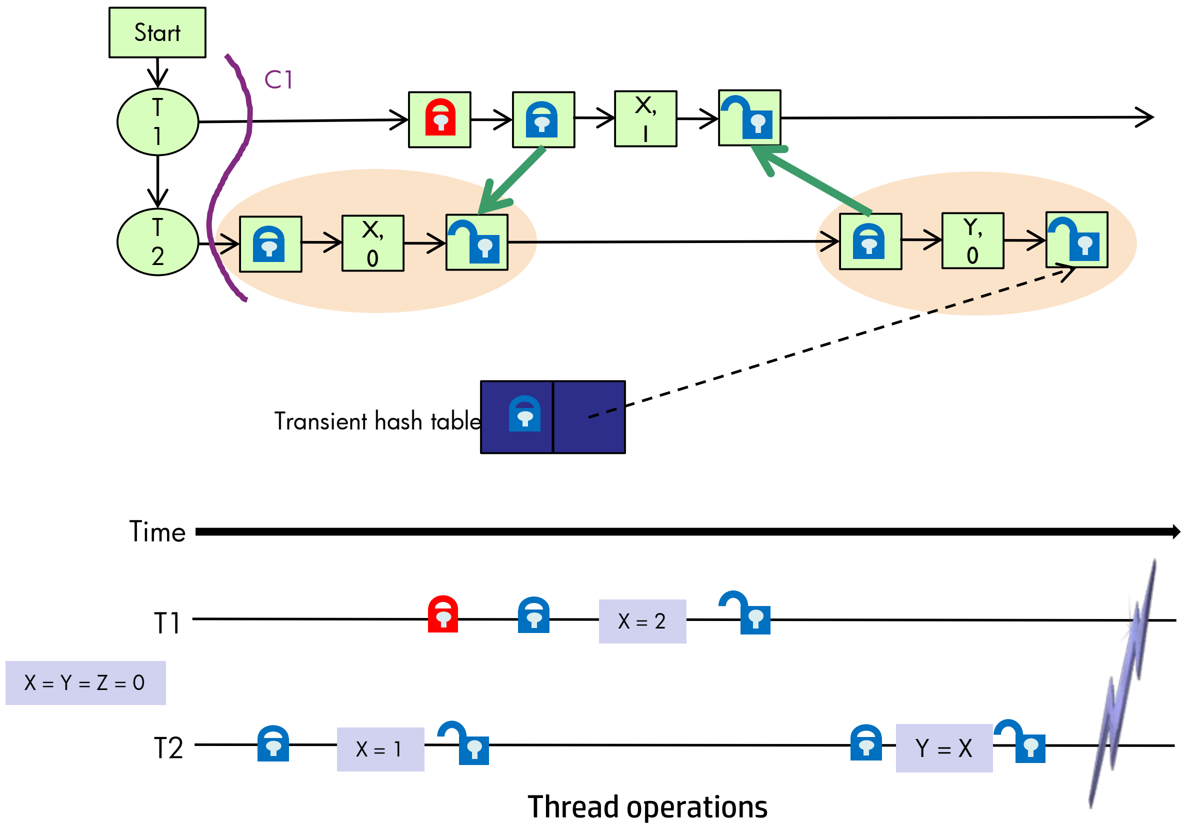


Thread operations

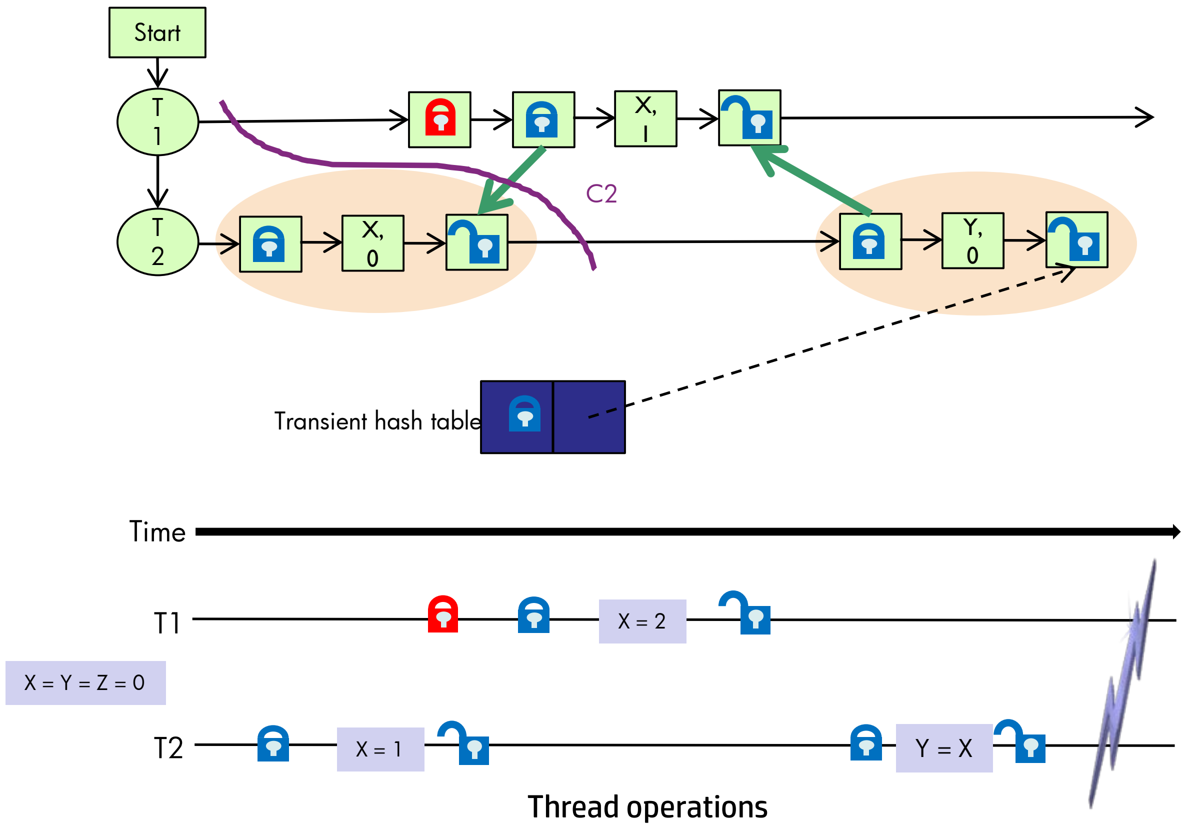
Computing a Consistent State



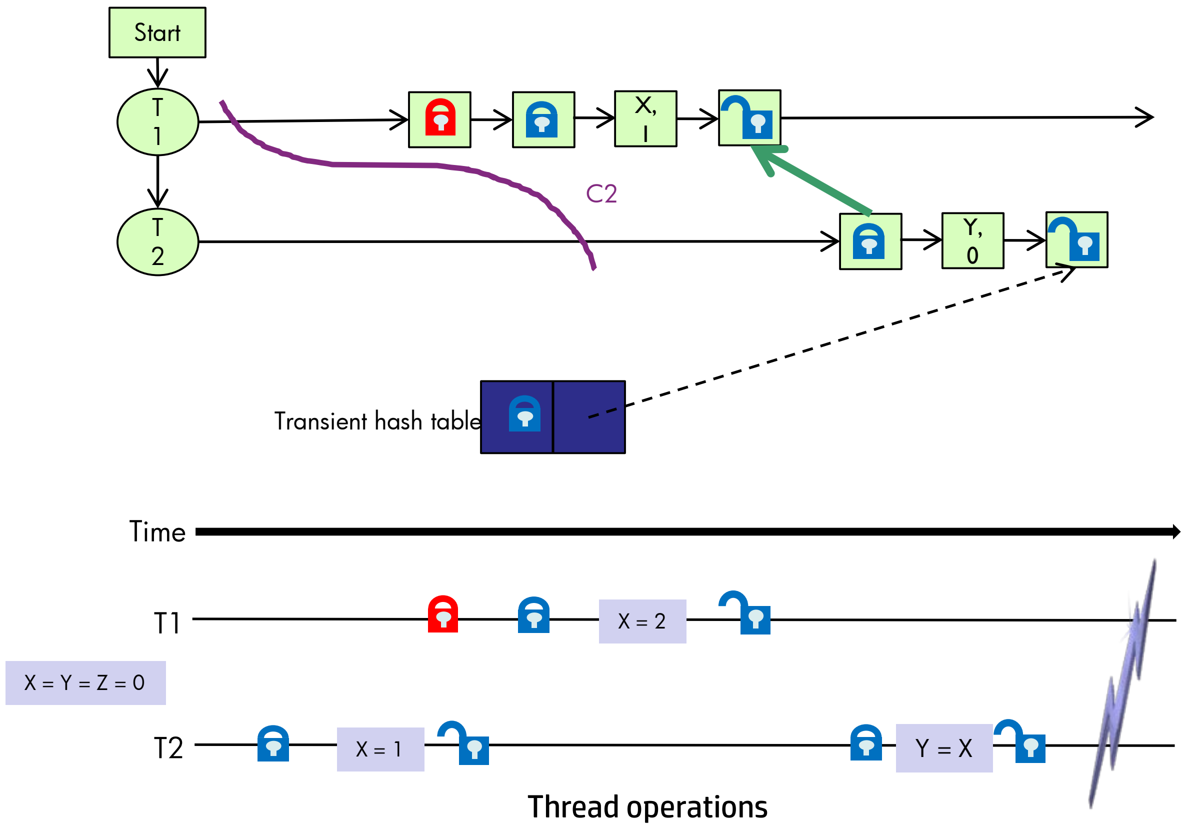
Computing a Consistent State



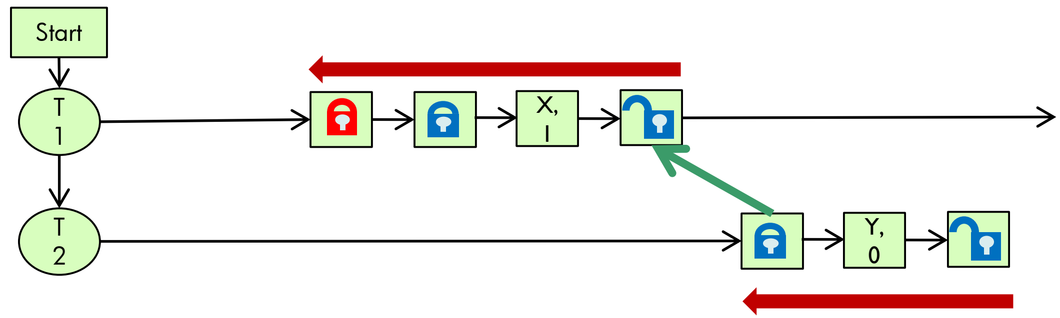
Advancing the Consistent State



Pruning the Log Structure



Recovery after a crash

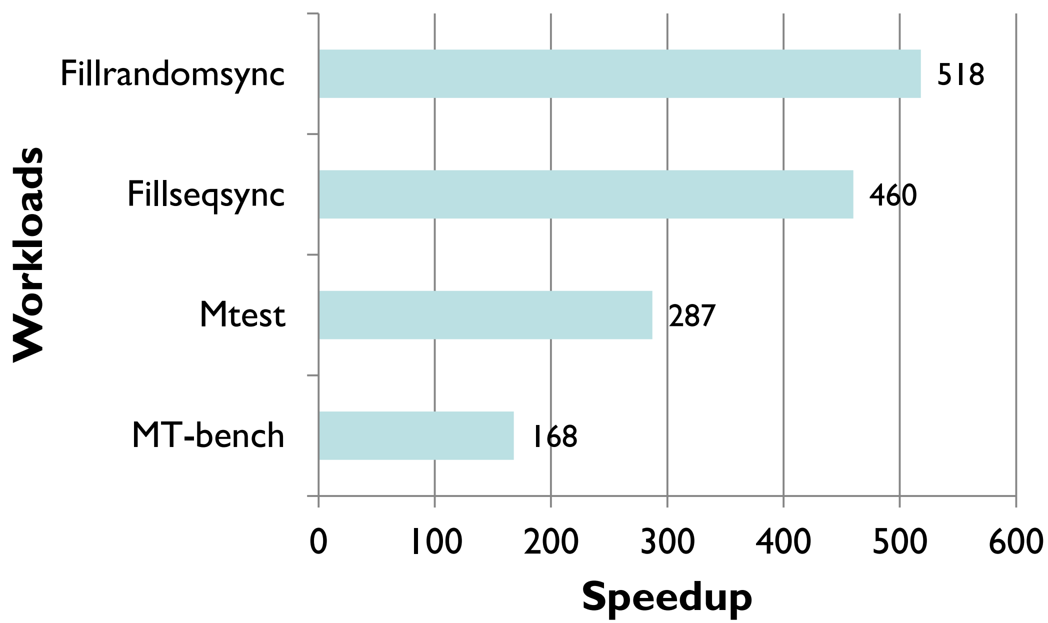


Optimizations

- Log creation and pruning
- Cache line clean

Speeding up existing durable applications: MDB

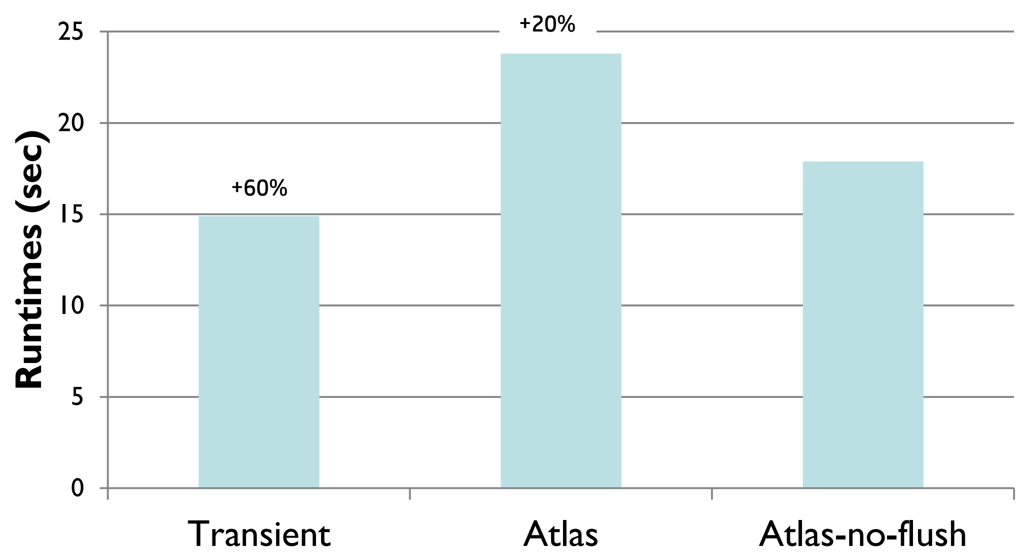
- Originally: disk-based key-value store
- Atlas-based:
  - Files replaced with persistent regions
  - Critical sections => no change



Adding durability to a transient Memcached

- Originally: transient distributed key-value cache
- Opportunity: instant restart from NVRAM
- Atlas-based (durable and consistent cache):
  - Hash table and helpers placed in a persistent region
  - Critical sections => no change

Comparison for 100,000 gets and puts



Related work

- Other NVMPL libraries
- SNIA NVM programming technical working group

Summary

- Non-volatile memory is coming
- Start writing code that can take advantage of NVRAM
- Reasonable programming frameworks already available

https://github.com/HewlettPackard/Atlas