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
Non-volatile memory is here
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?
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]
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
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

```java
Book bk1 = new Book(3); // transient allocation
Book bk2 = new Book(8);
PJJavaStore library = PJJavaStore.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?

**Common programming idiom:**

```c
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
```
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

A failure may lead to a dangling persistent pointer:

tmp = nvm_malloc()          // allocate a persistent location

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

Crash
Required Ordering Constraints

Requires initialization to be ordered before publication:

```c
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

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

Persistent Region API

User Program

Compile

Instrumented User Program

Execute

Persistent Data

Consistency API

Runtime library

Recover
Persistent Region API

**Basic program structure**

```c
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)
```

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

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

---

Thread 1

Thread 2

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
Computing a Consistent State

Thread operations

<table>
<thead>
<tr>
<th>Time</th>
<th>X = Y = Z = 0</th>
<th>X = 1</th>
<th>X = 2</th>
<th>Y = X</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Transient hash table
Computing a Consistent State
Advancing the Consistent State

Thread operations

Transient hash table

Time

X = Y = Z = 0

X = Y = X

Y = X
Pruning the Log Structure

![Diagram showing the sequence of operations and states involving threads T1 and T2, with transient hash table and thread operations labeled.]
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

![Bar chart showing speedup for different workloads:]

<table>
<thead>
<tr>
<th>Workload</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fillrandomsync</td>
<td>518</td>
</tr>
<tr>
<td>Fillseqsync</td>
<td>460</td>
</tr>
<tr>
<td>Mtest</td>
<td>287</td>
</tr>
<tr>
<td>MT-bench</td>
<td>168</td>
</tr>
</tbody>
</table>
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**

- Transient: +60%
- Atlas: +20%
- Atlas-no-flush
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