NV-Heaps:
Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories

Joel Coburn

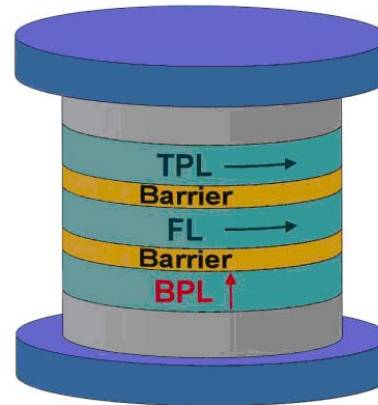
Work done at UCSD with Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, Steven Swanson

NVSL
Non-volatile Systems Laboratory

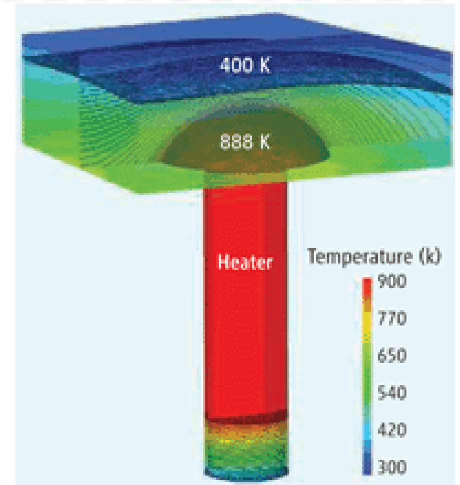
UCSDCSE
Computer Science and Engineering

Emerging Non-volatile Memories

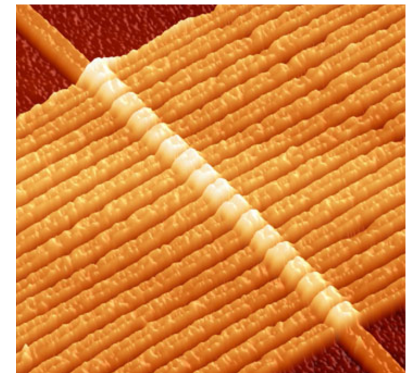
- Device characteristics
  - As fast as DRAM
  - As dense as flash
  - Non-volatile
  - Reliable
- Applications
  - DRAM replacements
  - Fast storage

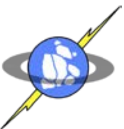


Spin-torque MRAM

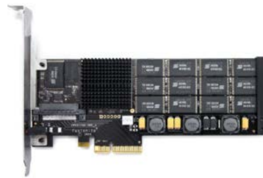


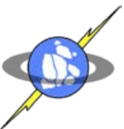
Phase change memory


Memristor



The Future of Storage

Hard Drives	PCIe-Flash 2007	PCIe-NVM 2013?	DDR-NVM 2016?
		 	
Lat.: 7.1ms	68us	8.2us	1.5us
1x	104x	865x	4733x = 2.5x/yr
BW: 2.6MB/s	250MB/s	1.6GB/s	14GB/s
1x	96x	669x	5384x = 2.6x/yr



*Random 4KB reads from user space

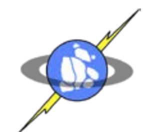
Overhead of Software

Log Request Latency (us)

10000
1000
100
10
1

Disk
Flash
Fast NVM

File System
OS
Hardware



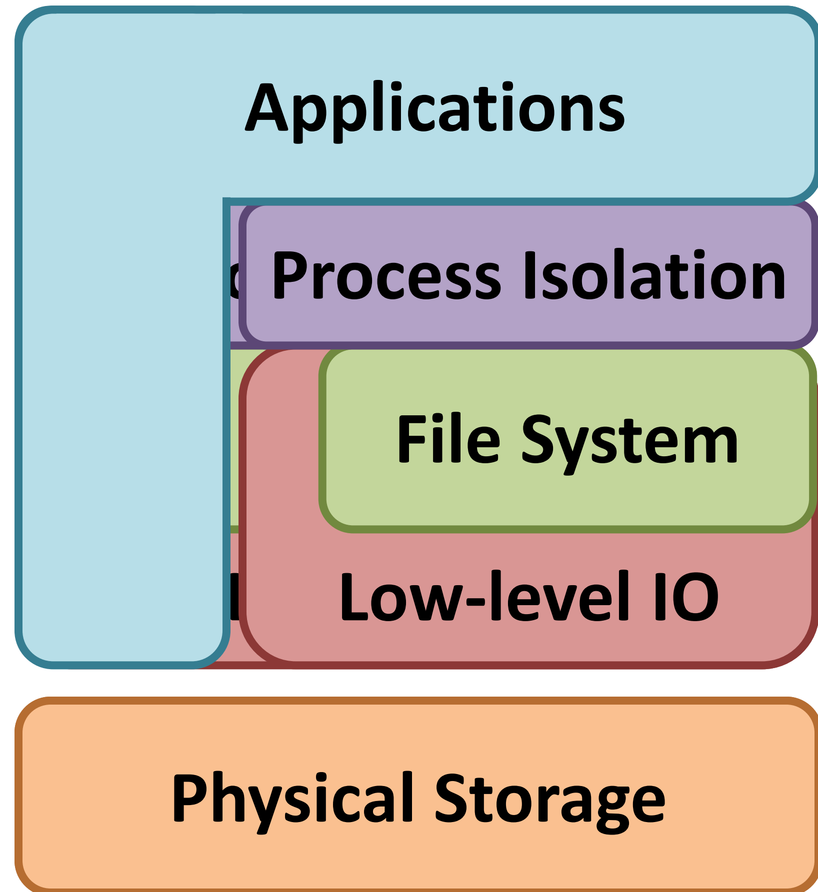
Redefining Persistence for the Programmer

Old Way: Treat it like disk

- Build the OS!
- Read/Write via the OS
- Use a file system

New Way: Treat it like Memory

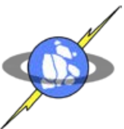
- Avoid NVRAM "disk"
- Create classes
- Use pointers/references
- Leverage strong types



Familiar way to build persistent data structures

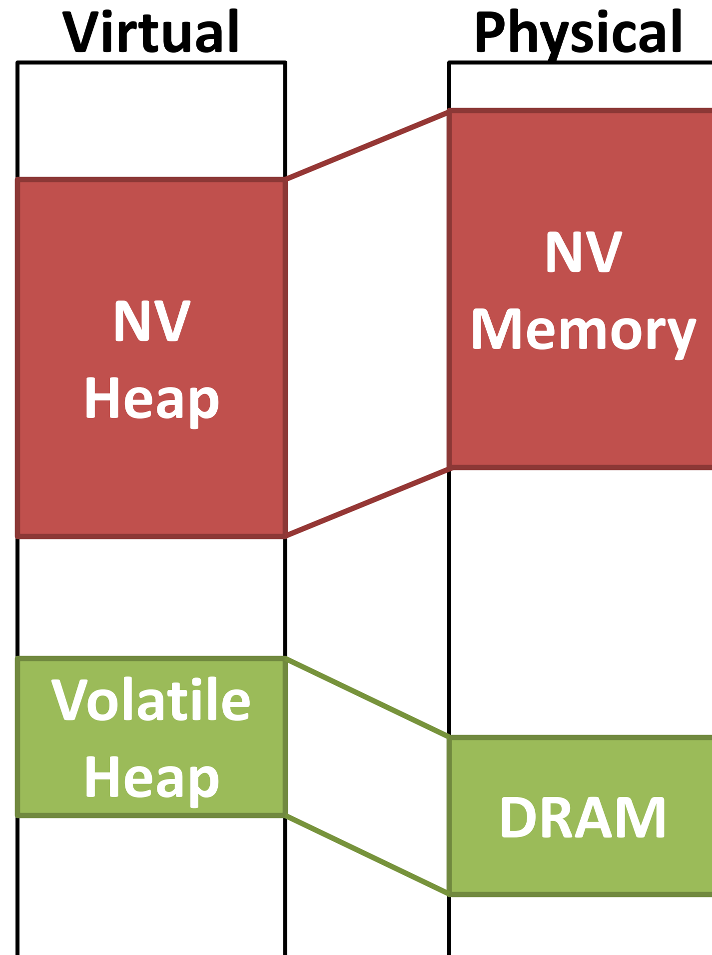
Overview

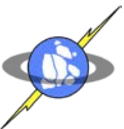
- Motivation
- Requirements for NV-Heaps
- System Implementation
- Benchmark performance
- Conclusion



Expose NVMs as raw storage

- Map into virtual address space
- Access through loads and stores

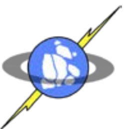




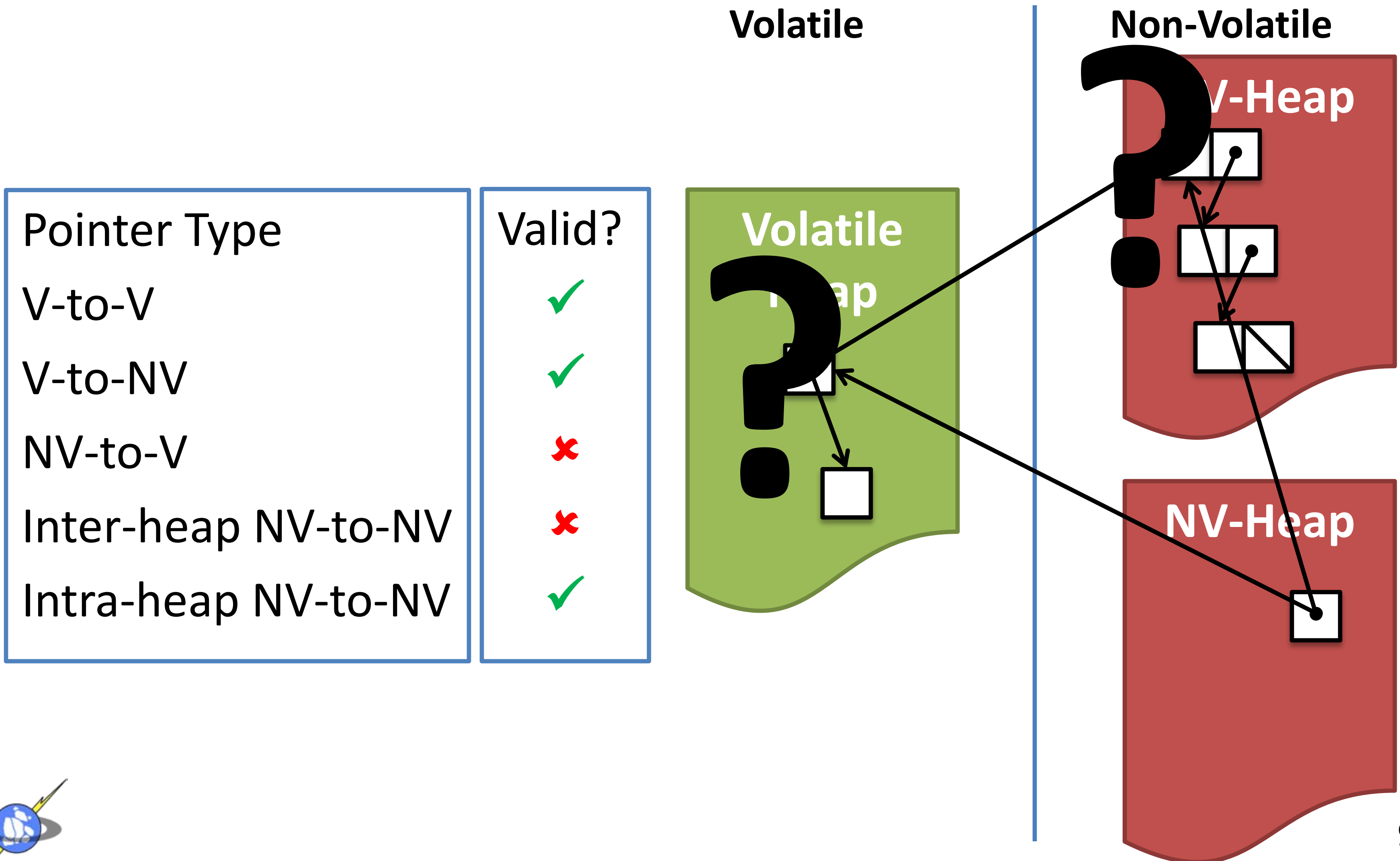
The Dangers of Direct Access

- All existing programming errors are still possible
  - Memory leaks
  - Multiple frees
  - Locking errors
- Programmers will get this stuff wrong

Rebooting/restarting won't help!



New Types of Bugs



Pointer Type	Valid?
V-to-V	✓
V-to-NV	✓
NV-to-V	✗
Inter-heap NV-to-NV	✗
Intra-heap NV-to-NV	✓

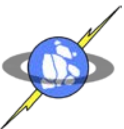
Existing Primitives are Error Prone

```
void Insert(Object * a, List<Object> * l)
{
...
}
```

Is **a** volatile? Is **l**?

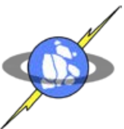
Are they in the same heap?

One wrong call causes permanent corruption



Memory Management, Locking, and NV Pointers

- Manual memory management and locking disciplines are well-known sources of errors
- Both rely on a program-wide invariant that is…
  - Not specified in the source
  - Not enforced by the system
- NV pointer safety relies on a similar invariant
- Programmers will get it wrong



How hard is it to get right?

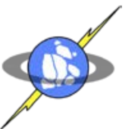
Example: **BPFS** [SOSP 09]

- Transactional file system for NVM on memory bus
- Carefully engineered NV data structure
- Exploits FS tree structure and limited operations
- Well worth the effort for a file system

Methodology does not scale for writing your average application!

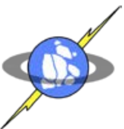
Persistent Object Systems

- Slow but safe
  - Database frontends (Java Persistence, C# LINQ)
  - Object-oriented databases (Objectstore [CACM 91], Texas [POS 92], Quickstore [SIGMOD 94], Thor [SIGMOD 96])
  - Transactional storage library (Stasis [OSDI 06])
  - Single-level stores (as400, Opal, etc.)
  - Orthogonally persistent Java [SIGMOD 96]
- Fast but unsafe
  - Recoverable Virtual Memory [SOSP 93]
  - Rio Vista (battery backed DRAM) [SOSP 97]
- Fast and safer
  - Mnemosyne (targets NVM) [ASPLOS 11]



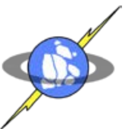
NV-Heaps: Safe Persistent Objects

1. **Safety**
   - Garbage collection
   - Pointer safety
   - Transactions
2. **Performance**
   - Approach raw NVM performance
3. **Scalability**
   - Operations are O(touched data) not O(storage size)
4. **Easy to use**
   - Familiar interface
   - Leverage existing file systems and tools
   - Intuitive separation between volatile and non-volatile data



Overview

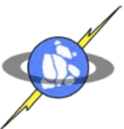
- Motivation
- Requirements for NV-Heaps
- System Implementation
- Benchmark performance
- Conclusion



Example Code – Linked List

```
class NVList : public NVObject {
   DECLARE_POINTER_TYPES(NVList);
public:
   DECLARE_MEMBER(int, value);
   DECLARE_PTR_MEMBER(NVList::NVPtr, next);
};
```

```
void remove(int k) {
   NVHeap * nv = NVHOpen("foo.nvheap");
   NVList::VPtr a = nv->GetRoot<NVList::NVPtr>();
   AtomicBegin {
      while (a->get_next() != NULL) {
         if (a->get_next()->get_value() == k) {
            a->set_next(a->get_next()->get_next());
         }
         a = a->get_next();
      }
   } AtomicEnd;
}
```



Implementation

NV-Heaps

Transaction Management

Garbage Collection
Pointer Safety

NVM Allocation

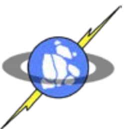
Locking,
logging,
and recovery

Reference counting
Pointer assignments
Pointer type enforcement
Reclamation

Memory mapping
Allocation and deallocation
Relocatability

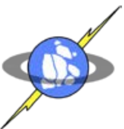
NVM Allocator

- Raw allocation and de-allocation
  - Per-thread free lists
  - Fixed-sized, write-ahead logging for atomicity and durability
  - Epoch barriers for consistency [SOSP 09] or combination of mfence and clflush
- Mapping
  - “Execute in place” support in Linux
  - Relative pointers for relocation



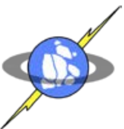
Garbage Collection + Pointer Safety

- Reference-counting
  - Per-object locks protect reference counts
  - Weak references for cycles
- Dynamic type system prevents dangerous NV pointers
  - Wide pointers allow run-time checks on assignments
  - A static type system is also possible



Challenge: Scalable locking

- NV-heaps require per-object locks
- Volatile locks don’t scale
  - Volatile storage rises with NV-heap size
- Non-volatile locks don’t scale
  - On recovery, all locks need to be released
  - Recovery time scales with NV-heap size



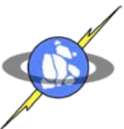
Generational Locks

Acquire the lock → **Lock** 4/5

Open the heap: Move to the next generation → **Generation** 4/5

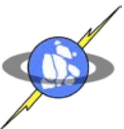
Lock > Generation → Unlocked / Locked

All locks released!



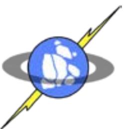
General, ACID Transactions

- Software transactional memory system
  - Object-based, undo logging
  - Eager conflict detection with locks and version numbers
- Logging
  - Per-thread NV write logs and V read logs
  - Using GC objects and pointers



Overview

- Motivation
- Requirements for NV-Heaps
- System Implementation
- Benchmark performance
- Conclusion



Comparison to Other Systems

13 to 1110X speedup over Stasis

2 to 643X speedup over BDB

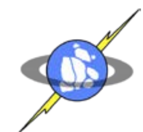
6.5X

Log Speedup Relative to Stasis RamDisk

10000
1000
100
10
1
0.1

Btree
SPS
Hash
6-Degrees
Average

Stasis RamDisk
BDB RamDisk
NV-Heaps PCM
NV-Heaps STTM
NV-Heaps DRAM



Layers of Safety

NV-Heaps

Transaction Management → *C-TX*

→ *TX*

Garbage Collection
Pointer Safety → *Safe*

NVM Allocation → *Base*

Price of Safety

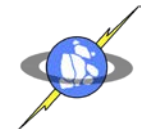
Speedup vs. Base

2.5
2
1.5
1
0.5
0

8 threads
4 threads
2 threads
1 thread

8.4X

Btree Base
Btree Safe
Btree TX
Btree C-TX
SPS Base
SPS Safe
SPS TX
SPS C-TX
Hash Base
Hash Safe
Hash TX
Hash C-TX
RBtree Base
RBtree Safe
RBtree TX
RBtree C-TX
6-Degrees Base
6-Degrees Safe
6-Degrees TX
6-Degrees C-TX
SSCA Base
SSCA Safe
SSCA TX
SSCA C-TX
Ave Base
Ave Safe
Ave TX
Ave C-TX



Application: Memcachedb

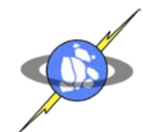
Operations/sec

120000
100000
80000
60000
40000
20000
0

8 to 28% slowdown

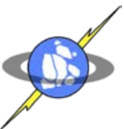
39X

Memcachedb | NV-heaps PCM | NV-heaps STTM | NV-heaps DRAM | Memcached



Looking Forward

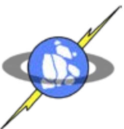
- Worse than DRAM, better than flash
  - How do we handle microsecond write times?
  - What does the new storage hierarchy look like?
- Hardware support for storage on the memory bus
  - Virtual memory overhead is high (TLB misses)
  - Costly memory fences and cacheline flushes
- What else do we need to guarantee safety?
  - Language support, program verification, application fsck, etc.
- Distributed storage using fast NVMs
  - Can we scale this abstraction to networked storage?



Conclusion

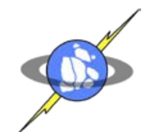
- NV-heaps give us robust non-volatile data structures in fast, non-volatile memory
  - Provide safe, easy to use, persistent objects
  - Very large application-level improvements

- Rethinking IO for NVMs is a major win!



Thank you!

Questions?



Thanks!

NVSL
Non-volatile Systems Laboratory