Rethinking Applications in the NVM Era

Amitabha Roy
ex- Intel Research

NVM = Non Volatile Memory

- Like DRAM, but retains its contents across reboots
- Past: Non Volatile DIMMs
    - Memory DIMM + Ultra-capacitor + Flash
    - Contents dumped on power fail, restored on startup
    - DRAM style access and performance but non-volatile
- Future: New types of non volatile memory media
    - Memristor, Phase Change Memory, Crossbar resistive memory, 3DXPoint
    - 3DXPoint DIMMS (Intel and Micron), demoed at Sapphire NOW 2017
    - Non Volatile without extra machinery - practical

Software Design

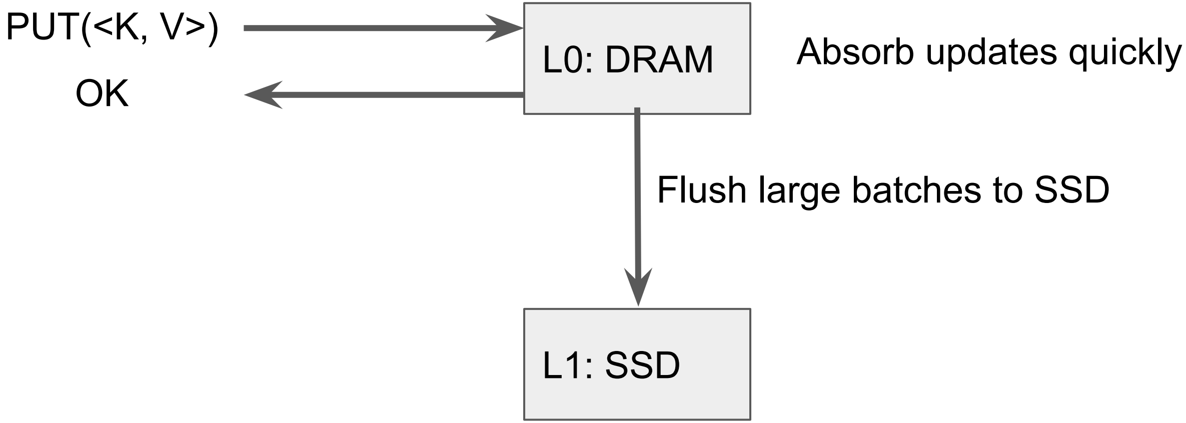
- New level in the storage hierarchy

Disk/SSD	NVM	DRAM
Persistent	**Persistent**	Volatile
Block oriented	**Byte oriented**	Byte oriented
Slow	**Fast**	Fast

⇒ Fundamental breakthroughs in how we design systems

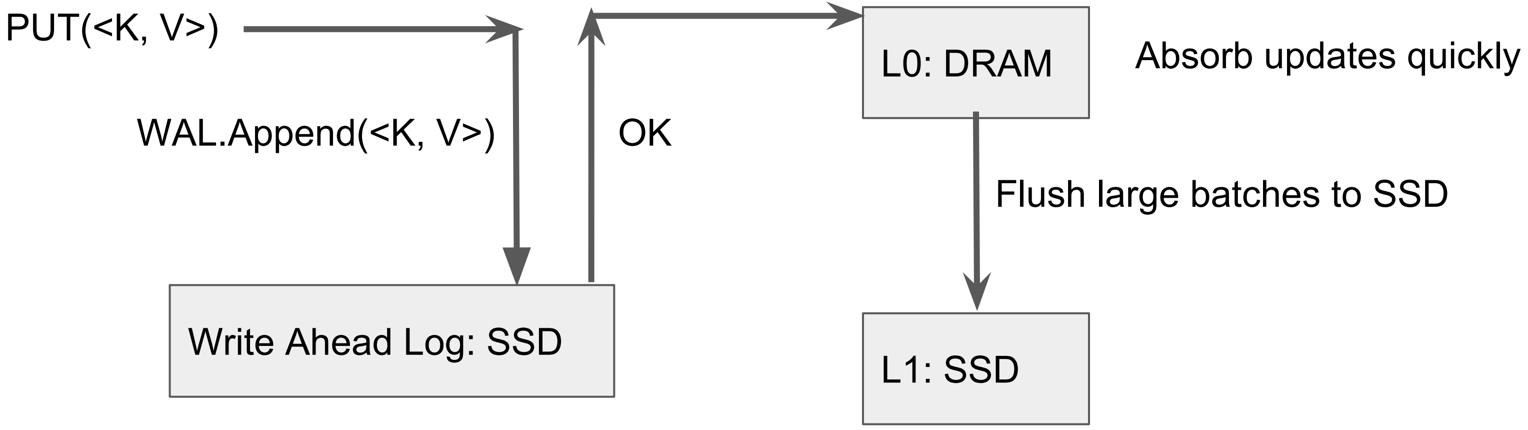
Use Case: Rocksdb

- Rocksdb - open source persistent key value store
- Optimized for Flash SSDs
- persistent map<key:string, value:string>
- Two levels (LSM tree) - sorted by key



Use Case: Rocksdb

- Problem: Lose all data in DRAM on power fail
- Durability guarantee requires a write ahead log
- Solution: synchronously append to a write ahead log



PUT(<K, V>)

WAL.Append(<K, V>)

OK

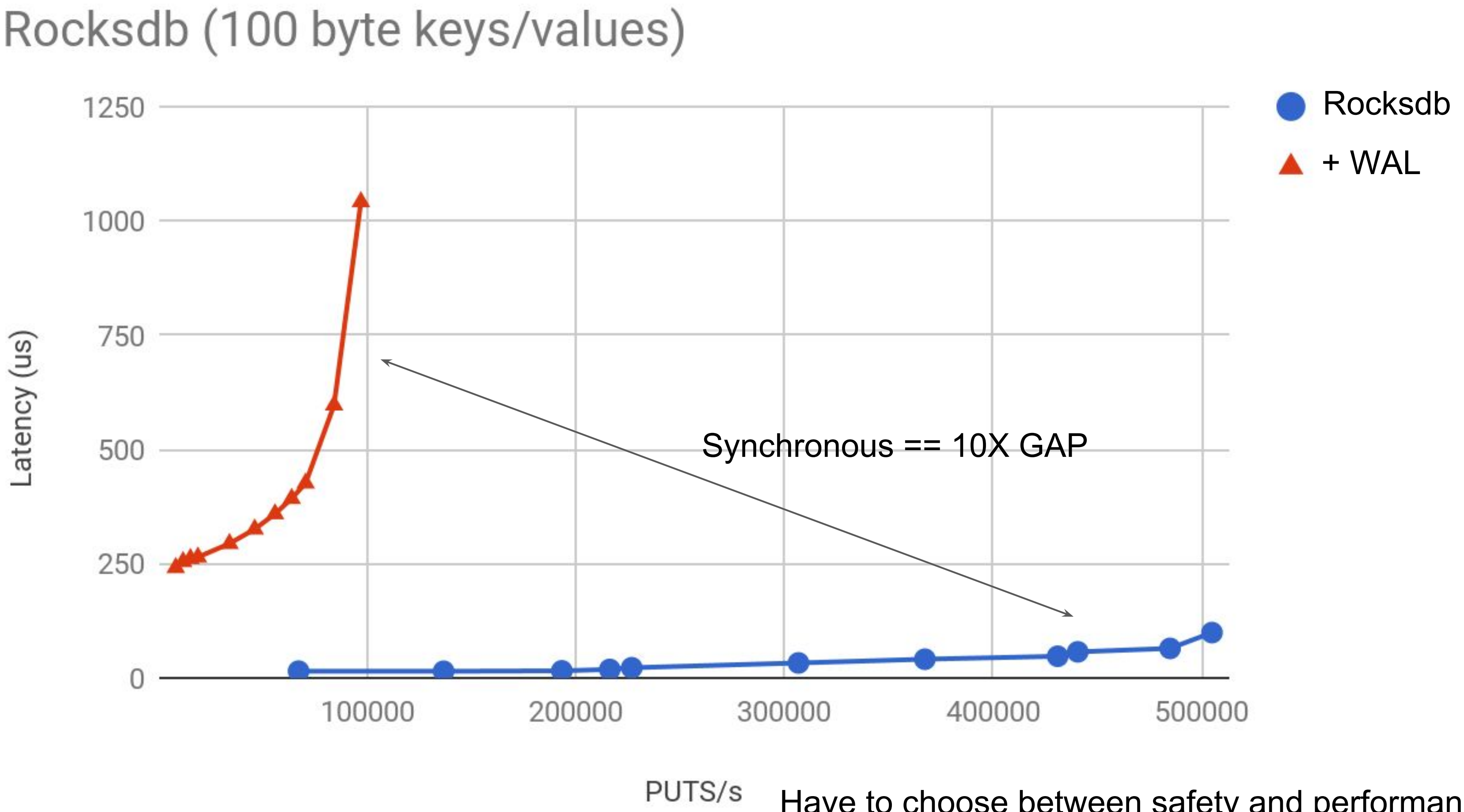
Write Ahead Log: SSD

L0: DRAM

Absorb updates quickly

Flush large batches to SSD

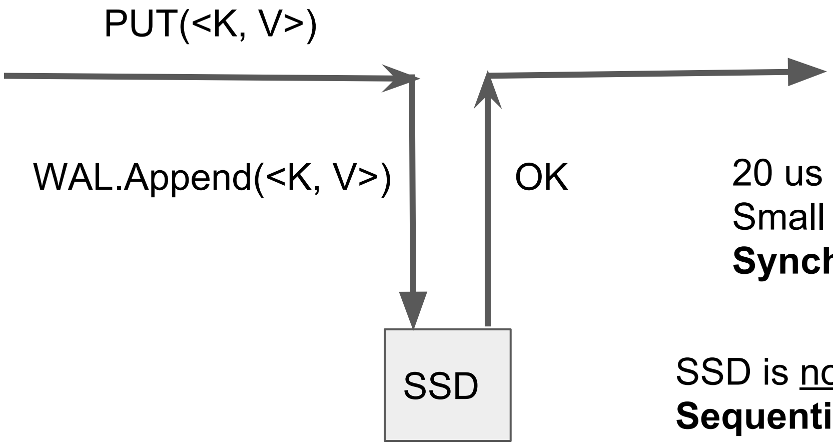
L1: SSD



Synchronous == 10X GAP

Have to choose between safety and performance

Rocksdb WAL Flow



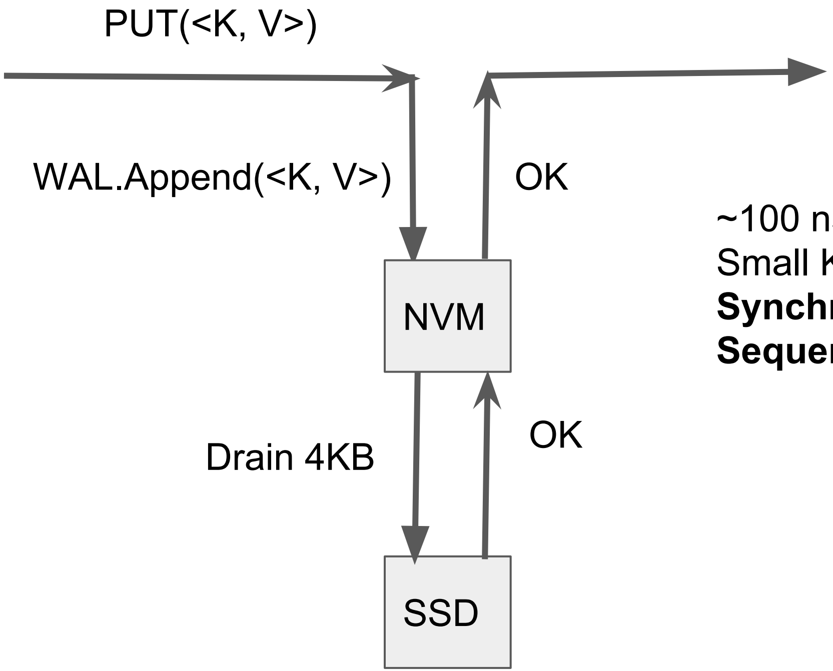
20 us round trip to SSD
Small KV pairs ~ 100 bytes
Synchronous writes => 5 MB/s

SSD is not the problem.
Sequential SSD BW => 1 GB/s

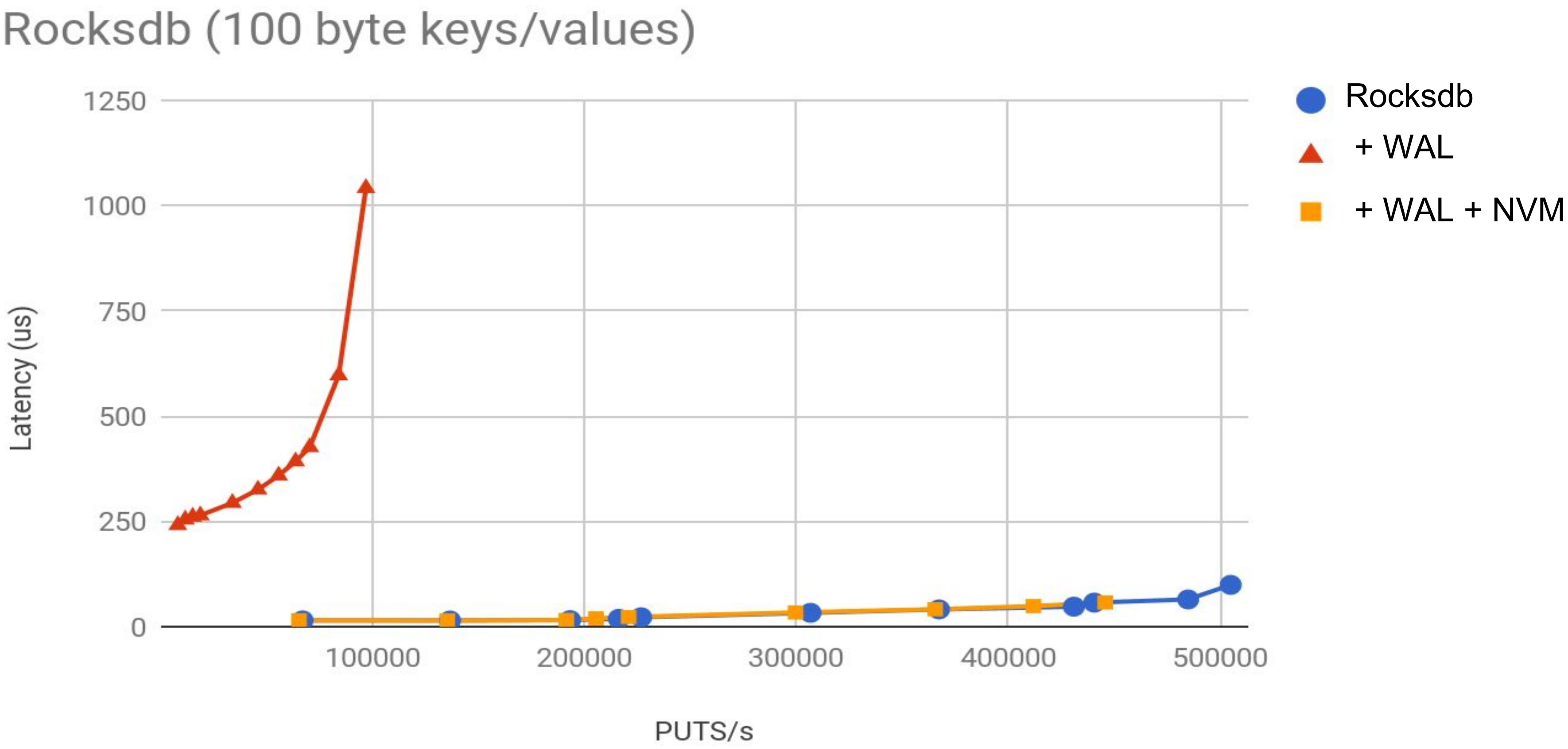
Problem: Persistence is block oriented
Most efficient path to SSD is 4KB units not 100 bytes
Have to pay fixed latency cost for only 100 byte IO

Rocksdb WAL Flow

Solution: Use byte oriented persistent memory



~100 ns round trip to NVDIMM
Small KV pairs ~ 100 bytes
Synchronous writes => 1GB/s
Sequential SSD BW => 1 GB/s



NVM removes the need for a safety vs performance choice

NVM = No more synchronous logging pain for KV stores, FS, Databases...

Software Engineering for NVM

- Building software for NVM has high payoffs
  - Make everything go much faster
- Not as simple as writing code for data in DRAM
  - Even though NVM looks exactly like DRAM for access
- Writing **correct** code to maintain persistent data structures is difficult
  - Part 2 of this talk
- Getting it wrong has high cost
  - Persistence = errors do not go away with reboot
  - No more ctrl+alt+del to fix problems
- Software engineering aids to deal with persistent memory
  - Part 3 of this talk

Example: Building an NVM log

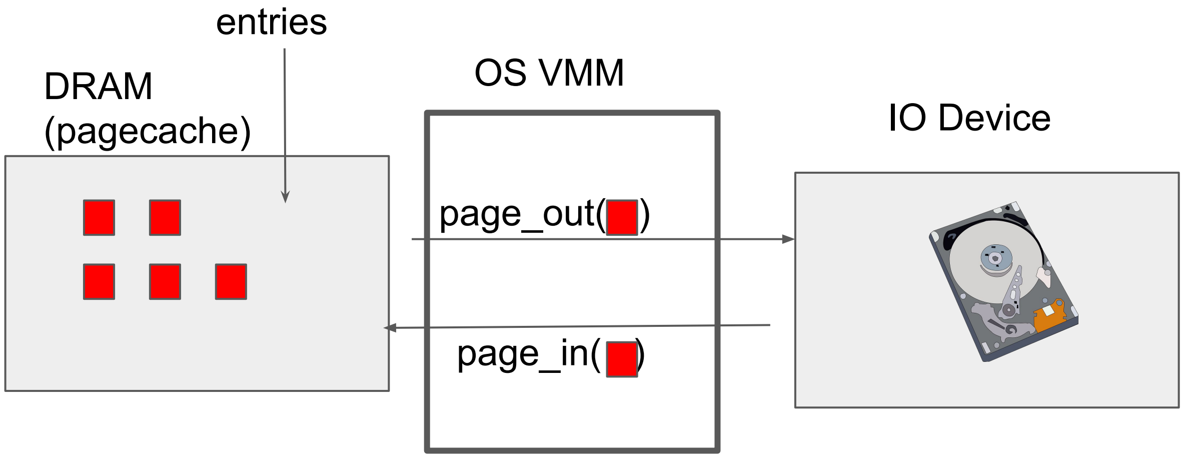
- Like the one we need for RocksDB
- Start from DRAM version

```
int * entries;
int tail;

void append(int value) {
        tail++;
        entries[tail] = value;
}
```

Making it Persistent

Persistent devices are block oriented
Hide block interface behind mmap abstraction

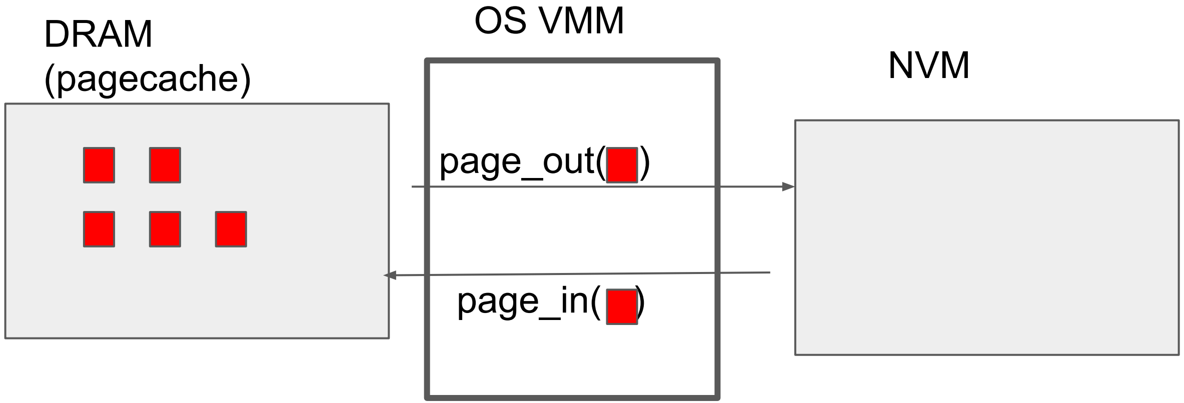
```
…
entries = mmap(fd, ...);
...

int * entries;
int tail;

void append(int value) {
        tail++;
        entries[tail] = value;
}
```



Persistent Data Structures Tomorrow

Does not work for NVM

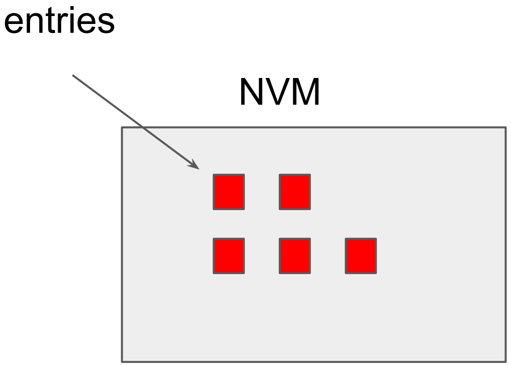
Wasteful copying - NVM is byte oriented and directly addressable



Direct Access (DAX)

```
# Most Linux filesystems support for NVM
# mount -t ramfs -o dax,size=128m ext2 /nvm
```

```
fd = open(“/nvm/log”, …);
int *entries = mmap(fd, ...);
int tail;

void append(int value) {
        tail++;
        entries[tail] = value;
}
```



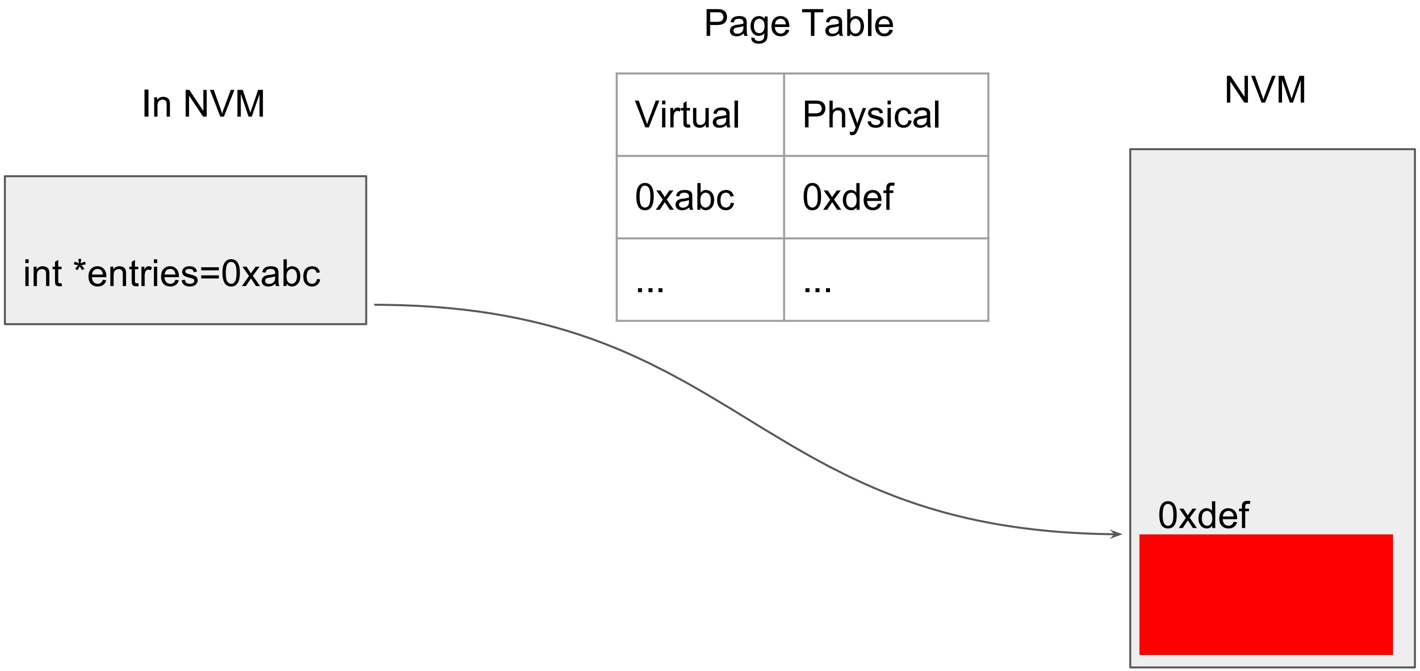
Tolerating Reboots

```
fd = open(“/nvm/log”, …);
int entries = mmap(fd, ...);
int tail;

void append(int value) {
        tail++;
        entries[tail] = value;
}
```

Persistent data structures live across reboots

Thinking about Persistence

void * area = mmap(.., fd, ..);



Page Table

Virtual	Physical
0xabc	0xdef
...	...

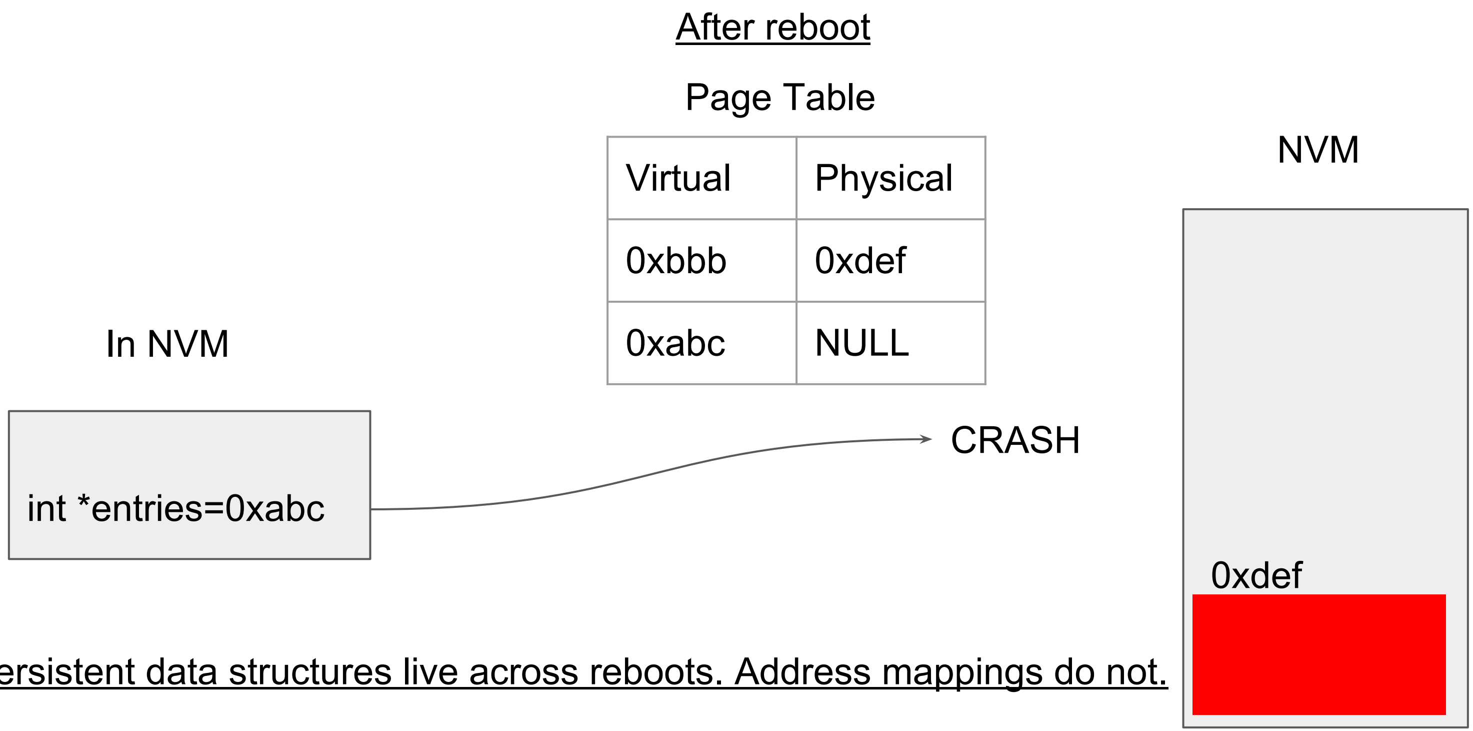
In NVM

int *entries=0xabc

NVM

0xdef

Thinking about Persistence



After reboot

Page Table

Virtual	Physical
0xbbb	0xdef
0xabc	NULL

NVM

In NVM

int *entries=0xabc

CRASH

0xdef

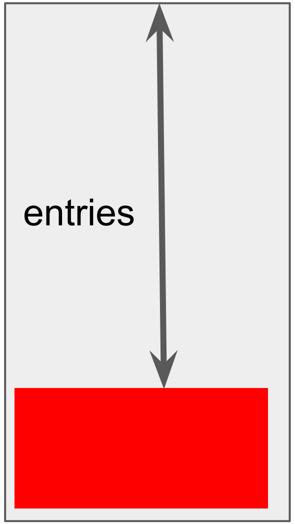
Persistent data structures live across reboots. Address mappings do not.

Persistent Pointers

Solution: Make pointers base relative. Base comes from mmap.

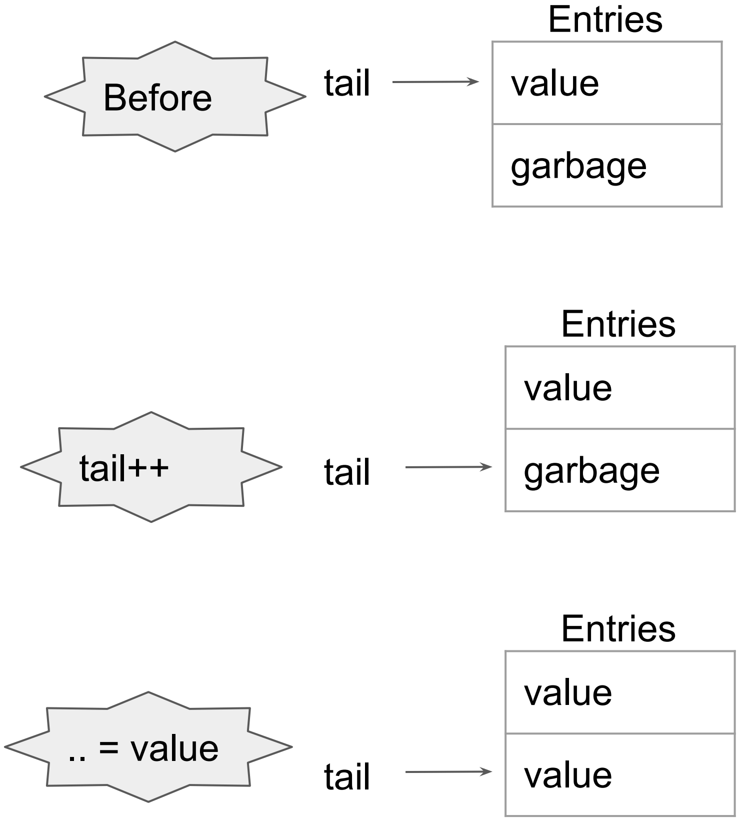
```
fd = open(“/nvm/log”, …);
nvm_base = mmap(fd, ...);
#define VA(off) ((off) + nvm_base)

offset_t entries;
int tail;

void append(int value) {
        tail++;
        VA(entries)[tail] = value;
}
```

nvm_base:



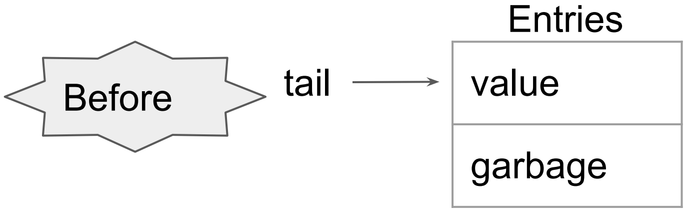
Power Failure

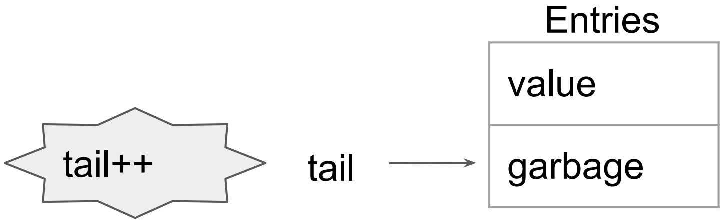
```
fd = open("/nvm/log", …);
nvm_base = mmap(fd, ...);
#define VA(off) ((off) + nvm_base)

offset_t entries;
int tail;

void append(int value) {
        tail++;
        VA(entries)[tail] = value;
}
```

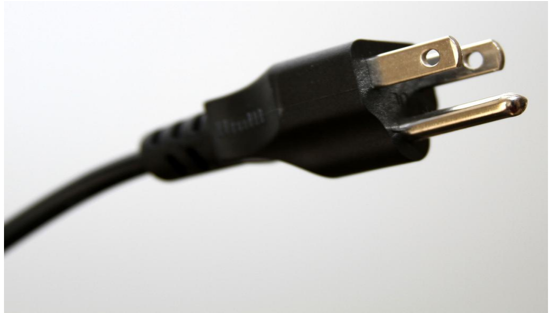


Power Failure

```
fd = open(“/nvm/log”, …);
nvm_base = mmap(fd, ...);
#define VA(off) ((off) + nvm_base)

offset_t entries;
int tail;

void append(int value) {
        tail++;
        VA(entries)[tail] = value;
}
```

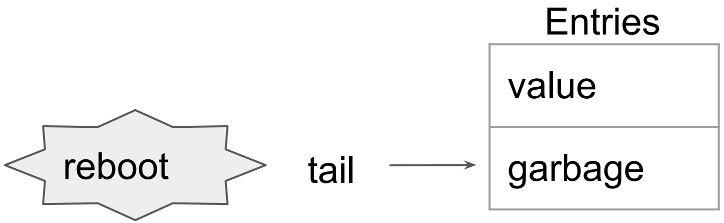






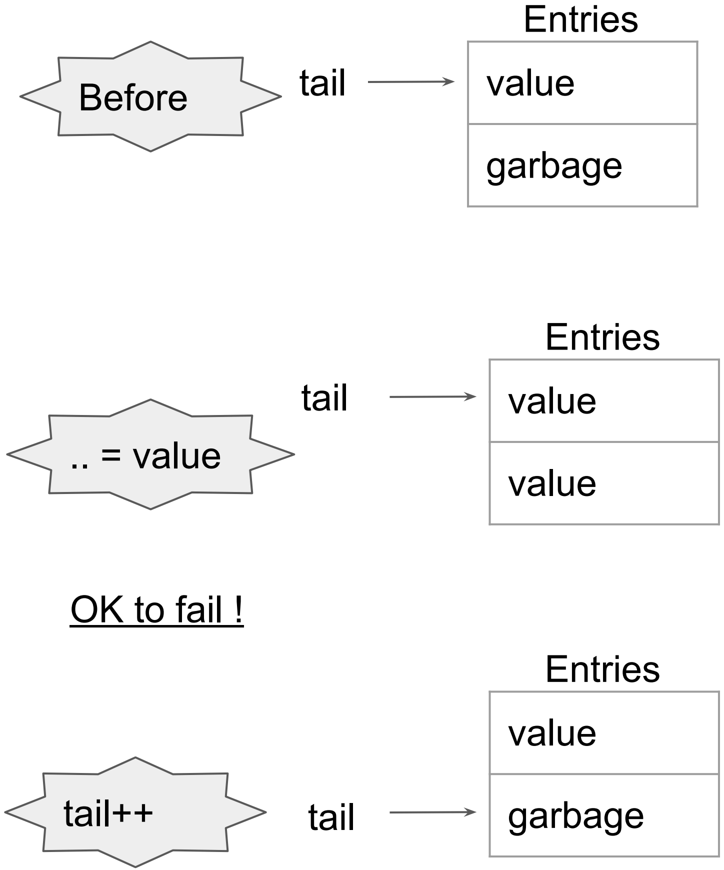
Reboot after Power Failure

```
fd = open(“/nvm/log”, …);
nvm_base = mmap(fd, ...);
#define VA(off) ((off) + nvm_base)

offset_t entries;
int tail;

void append(int value) {
        tail++;
        VA(entries)[tail] = value;
}
```



Ordering Matters

```
fd = open("/nvm/log", …);
nvm_base = mmap(fd, ...);
#define VA(off) ((off) + nvm_base)

offset_t entries;
int tail;

void append(int value) {
        VA(entries)[tail + 1] = value;
        tail++;
}
```

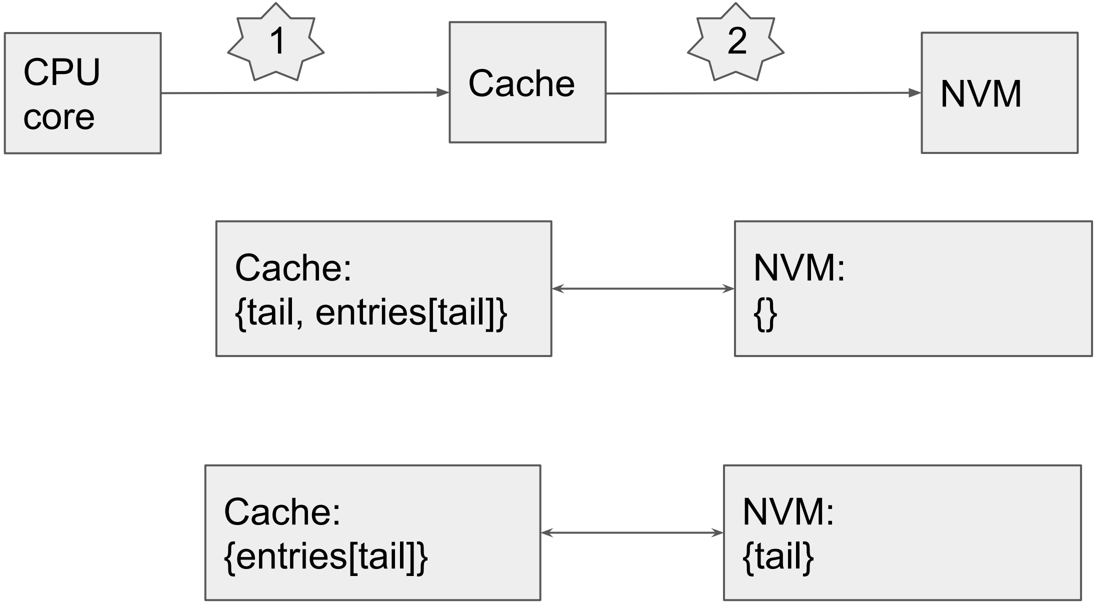


OK to fail !

The last piece: CPU caches

```
fd = open(“/nvm/log”, …);
nvm_base = mmap(fd, ...);
#define VA(off) ((off) + nvm_base)

offset_t entries;
int tail;

void append(int value) {
        VA(entries)[tail + 1] = value;
        tail++;
}
```

Transparent processor caches reorder your updates to NVM



CPU core

1

Cache

2

NVM

Cache:
{tail, entries[tail]}

NVM:
{}

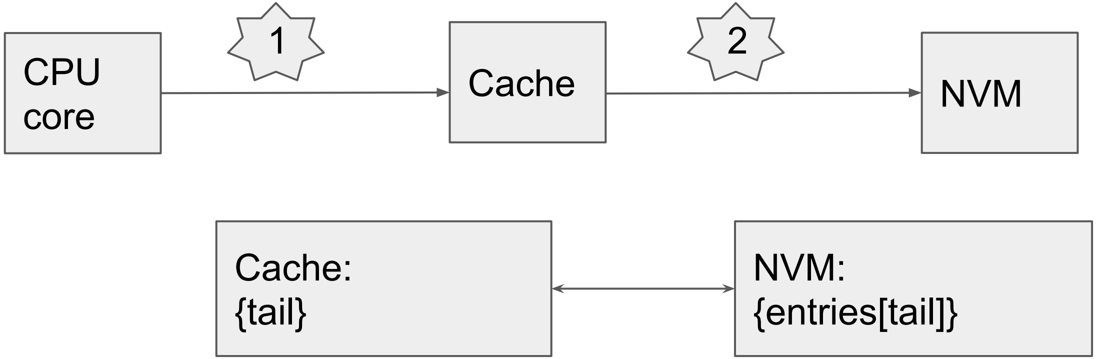
Cache:
{entries[tail]}

NVM:
{tail}

Explicit Cache Control

Use explicit instructions to control cache behavior

```
fd = open(“/nvm/log”, …);
nvm_base = mmap(fd, ...);
#define VA(off) ((off) + nvm_base)

offset_t entries;
int tail;

void append(int value) {
        tail++;
        sfence();
        clflush(&tail);
        VA(entries)[tail] = value;
        sfence();
        clflush(&VA(entries)[tail]);
}
```



Getting NVM Right

```
fd = open(“/nvm/log”, …);
nvm_base = mmap(fd, ...);
#define VA(off) ((off) + nvm_base)

offset_t entries;
int tail;

void append(int value) {
        tail++;
        sfence();
        clflush(&tail);
        VA(entries)[tail] = value;
        sfence();
        clflush(&VA(entries)[tail]);
}
```

COMPLEXITY !!!

Software Toolchains for NVM

- Correctly manipulating NVM can be difficult.
- Bugs and errors propagate past the lifetime of the program
  - Fixing errors with DRAM is easy - ctrl + alt + del
  - Your data structures will outlive your code
  - New reality for software engineering
- People will still do it (this talk encourages you to)
- Need automation to relieve software burden
  - Testing
  - Libraries

Software Testing for NVM

```
fd = open(“/nvm/log”, …);
nvm_base = mmap(fd, ...);
#define VA(off) ((off) + nvm_base)

offset_t entries;
int tail;

void append(int value) {
        tail++;
        sfence();
        clflush(&tail);
        VA(entries)[tail] = value;
        sfence();
        clflush(&VA(entries)[tail]);
}
```

```
TEST {
  append(42);
  ASSERT(entries[1] == 42);
}
```

Software Testing for NVM

```
fd = open(“/nvm/log”, …);
nvm_base = mmap(fd, ...);
#define VA(off) ((off) + nvm_base)

offset_t entries;
int tail;

void append(int value) {
        tail++;
        sfence();
        ~~clflush(&tail);~~ // BUG!!
        VA(entries)[tail] = value;
        sfence();
        clflush(&VA(entries)[tail]);
}
```

```
TEST {
  append(42);
  ASSERT(entries[1] == 42);
}
```

Thousands of executions…..
ASSERT nevers fires 😭😭

Software Testing for NVM

```
fd = open("/nvm/log", …);
nvm_base = mmap(fd, ...);
#define VA(off) ((off) + nvm_base)

offset_t entries;
int tail;

void append(int value) {
        tail++;
        sfence();
        ~~clflush(&tail);~~ // BUG!!
        VA(entries)[tail] = value;
        sfence();
        clflush(&VA(entries)[tail]);
}
```

```
TEST {
  append(42);
  REBOOT;
  ASSERT(entries[1] == 42);
}
```

Thousands of executions…..
ASSERT maybe fires 😭

YAT

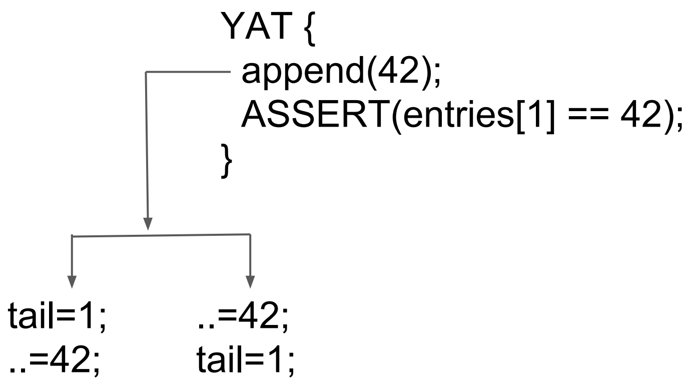
Automated testing tool for NVM software

Yat: A Validation Framework for Persistent Memory. Dulloor et al. USENIX 2014

Idea: Test power failure without really pulling the plug

1. Extract possible store orders to NVM

```
fd = open("/nvm/log", …);
nvm_base = mmap(fd, ...);
#define VA(off) ((off) + nvm_base)

offset_t entries;
int tail;

void append(int value) {
        tail++;
        sfence();
        ~~clflush(&tail);~~ // BUG!!
        VA(entries)[tail] = value;
        sfence();
        clflush(&VA(entries)[tail]);
}
```



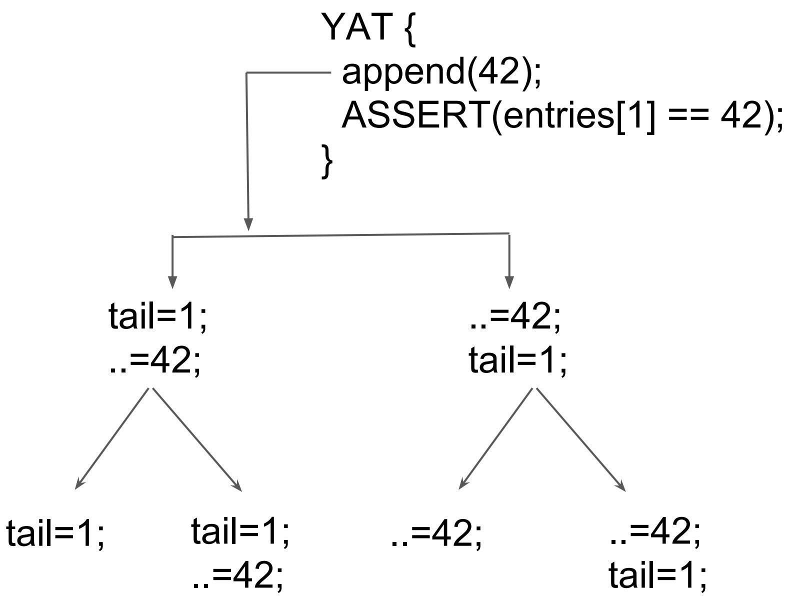
```
YAT {
  append(42);
  ASSERT(entries[1] == 42);
}
```

```
tail=1;        ..=42;
..=42;         tail=1;
```

Use a hypervisor or instrumentation via binary instrumentation (eg. PIN, Valgrind)
Use understanding of x86 memory ordering model

2. Consider All Possible Truncations

```
fd = open("/nvm/log", …);
nvm_base = mmap(fd, ...);
#define VA(off) ((off) + nvm_base)

offset_t entries;
int tail;

void append(int value) {
        tail++;
        sfence();
        ~~clflush(&tail);~~ // BUG!!
        VA(entries)[tail] = value;
        sfence();
        clflush(&VA(entries)[tail]);
}
```



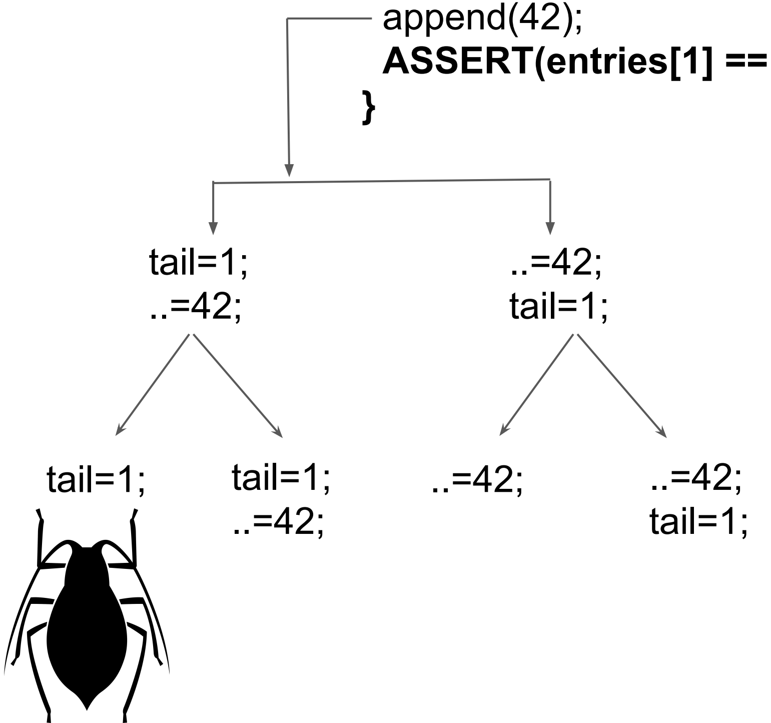
```
YAT {
  append(42);
  ASSERT(entries[1] == 42);
}
```

Each truncation is a simulated power failure!

2. Check Assertion for Each Truncation

```
fd = open(“/nvm/log”, …);
nvm_base = mmap(fd, ...);
#define VA(off) ((off) + nvm_base)

offset_t entries;
int tail;

void append(int value) {
        tail++;
        sfence();
        ~~clflush(&tail);~~ // BUG!!
        VA(entries)[tail] = value;
        sfence();
        clflush(&VA(entries)[tail]);
}
```

```
YAT {
  append(42);
  ASSERT(entries[1] == 42);
}
```



Non Volatile Memory Library

- Testing does not stop bugs - it only catches them after the fact
- Need to stop bugs at the source
- Make NVM look exactly DRAM to the programmer
- Automate the extra bits
- Enable complex datastructures - such as trees
    - Not as easy to reason about consistency like our toy example
    - Impossible except for ninja programmers
- Non Volatile Memory Library (NVML)

http://nvml.io

NVML

```
PMEMoid entries;
int tail;
void append(int v) {
        TX_BEGIN(...) {
                pmemobj_tx_add_range_direct(&tail, sizeof(int));
                tail++;
                int* array = pmemobj_direct(entries);
                pmemobj_tx_add_range_direct(&array[top], sizeof(T));
                array[tail] = v;
        } TX_END
```

Automation/Magic

1. Persistent pointers
2. **No need for sfence, clflush**
3. **Order of updates irrelevant**

Undo Log

```
TX_BEGIN(...) {
        ...
        pmemobj_tx_add_range_direct(&tail, ..);
        tail++;
        ...
        pmemobj_tx_add_range_direct(&array[tail], ..);
        array[tail] = v;
} TX_END
```

UNDO LOG

ADDRESS	CONTENT
&tail	10
&array[11]	GARBAGE

Undo Log

```
TX_BEGIN(...) {
        ...
        pmemobj_tx_add_range_direct(&tail, ..);
        tail++;
        ...
        pmemobj_tx_add_range_direct(&array[tail], ..);
        array[tail] = v;
} TX_END
```

```
If Hit TX_END - Success !
        sfence;
        foreach e in UNDO LOG:
                clflush e.address
        Delete UNDO LOG
```

UNDO LOG

ADDRESS	CONTENT
&tail	10
&array[11]	GARBAGE

Undo Log

UNDO LOG (in NVM)

ADDRESS	CONTENT
&tail	10
&array[11]	GARBAGE

```
TX_BEGIN(...) {
        ...
        pmemobj_tx_add_range_direct(&tail, ..);
        tail++;
        ...
        pmemobj_tx_add_range_direct(&array[tail], ..);
        array[tail] = v;
} TX_END
```

```
If Hit TX_END - Success !
        sfence;
        foreach e in UNDO LOG:
                clflush e.address
        Delete UNDO LOG
```

```
else Restart - Failed !
        foreach e in UNDO LOG:
                *e.address = e.content
                sfence
                clflush e.address
        Delete UNDO LOG
```

Undo Log == Failure Atomicity

```
TX_BEGIN(...) {
        ...
        pmemobj_tx_add_range_direct(&tail, ..);
        tail++;
        ...
        pmemobj_tx_add_range_direct(&array[tail], ..);
        array[tail] = v;
} TX_END
```

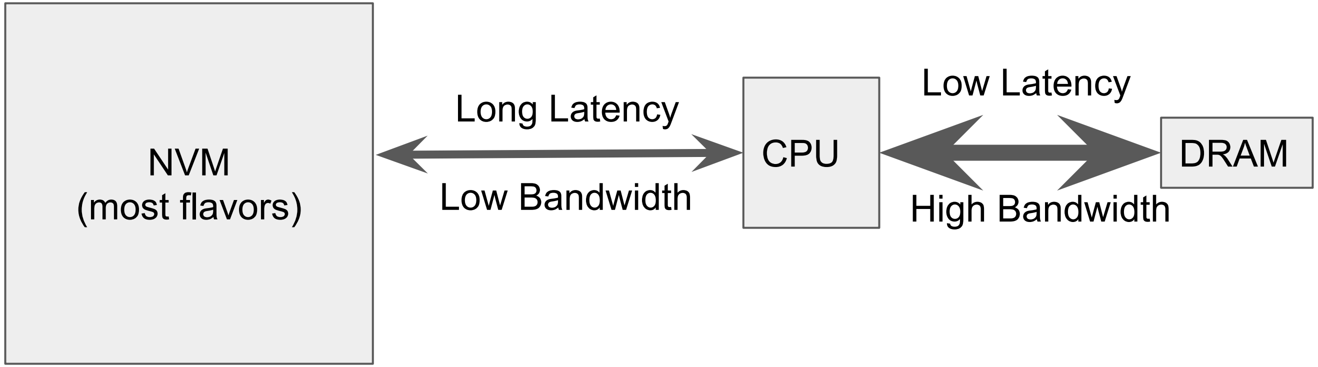
UNDO LOG (in NVM)

ADDRESS	CONTENT
&tail	10
&array[11]	GARBAGE

All or nothing semantics in the face of failure
Like **A**CID from DBMS world

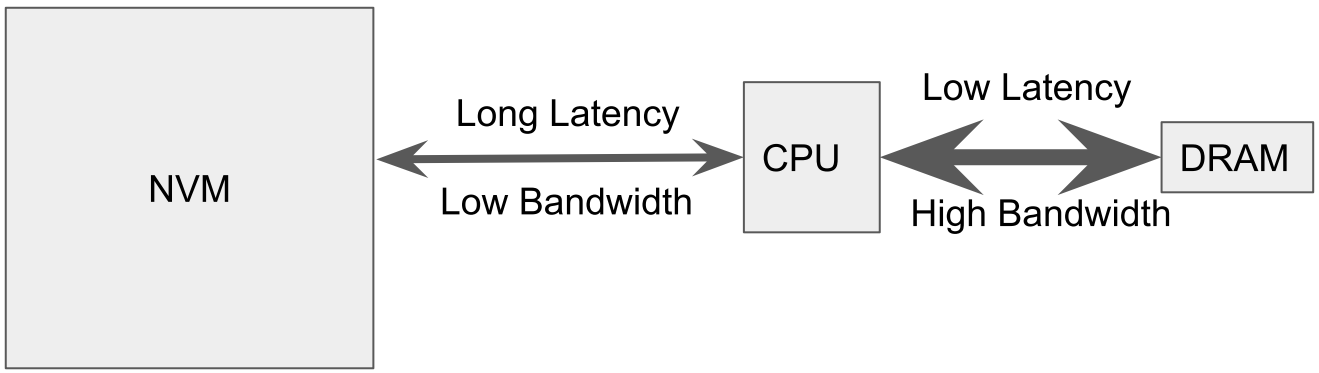
Profilers

NVM = Tiered performance of main memory



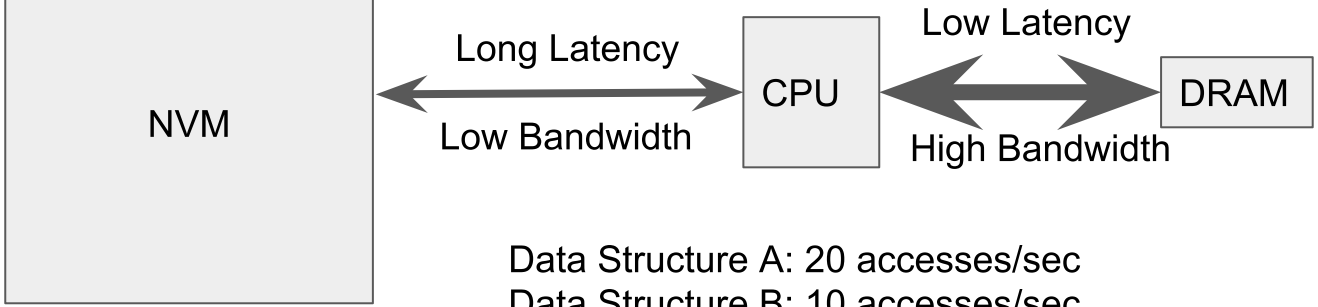
- Tiered memory => Different performance flavors of main memory
- NVM slower and more plentiful than DRAM

Performance and Placement



- Analytics - don’t care about persistence
- Use NVM as cheap, plentiful and slow memory
    - Surprising but projected use of ~~N~~VM
- Choice of where to place data structures

Performance and Placement



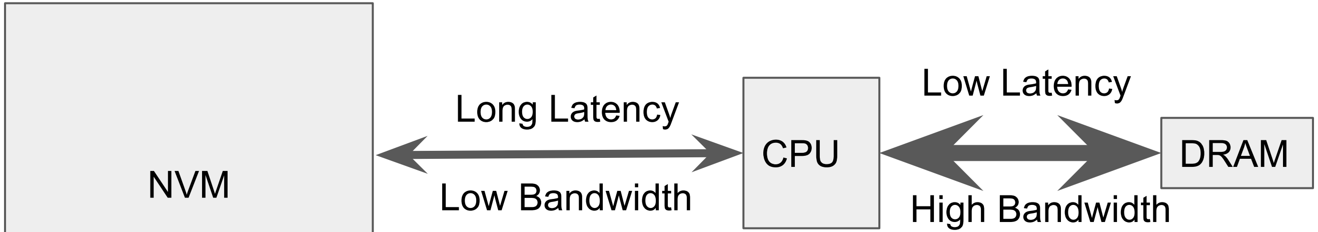
Data Structure A: 20 accesses/sec
Data Structure B: 10 accesses/sec

Storage caching wisdom(aka 5 minute rule): More frequent accesses to faster memory

Performance and Placement

Storage caching wisdom is wrong   Data Tiering in Heterogeneous Memory Systems. Eurosys 2016.

Memory access performance strongly governed by access pattern and not just access frequency



Data Structure A: 20 accesses/sec   - sequential scans
Data Structure B: 10 accesses/sec   - pointer chasing

More frequently accessed DS in slower memory!

Performance and Placement

- Little’s Law

  InFlightRequests = Bandwidth * Latency

- Can have larger bandwidth even with longer latency

  Bandwidth = InFlightRequests/Latency

- OOO CPU pipelines good at increasing InFlightRequests for scans
  - Prefetching
  - Non-Blocking Caches
- Scans should go to longer latency NVM even if more frequent
  - Pointer chasing needs high performance memory

X-Mem

- Guide data structure placement via profiling
- Beyond simple cache miss rate optimization
    - Eg. tools like vtune, gprof
- Need to determine access pattern (pointer or scan?)

Solution:

malloc(size, TAG) + map<Virtual Pages, TAG>

- TAG unique to datastructure: maps memory access to datastructure
- Access profiler determines best memory type for TAG
- malloc maps data structure blocks to pages from correct memory type

Example

- MemC3 hash table - An improved memcached

Name	Frequency	Type	Location
512B buckets	2%	Random	DRAM
8K Values	8%	Scans	NVM

Conclusion

- NVM adds a whole new dimension to software engineering
- Opportunities for fundamental breakthroughs
  - Solve system design problems in new ways
  - Eg. fixing synchronous logging in Rocksdb
- Challenges
  - Data structures outlive the code - can’t restart on a bug!
  - Persistent pointers, ordering, processor caches
  - Tiered main memory architecture
- Software engineering solutions
  - New ideas in testing, libraries, profilers
- What will you do with NVM ?