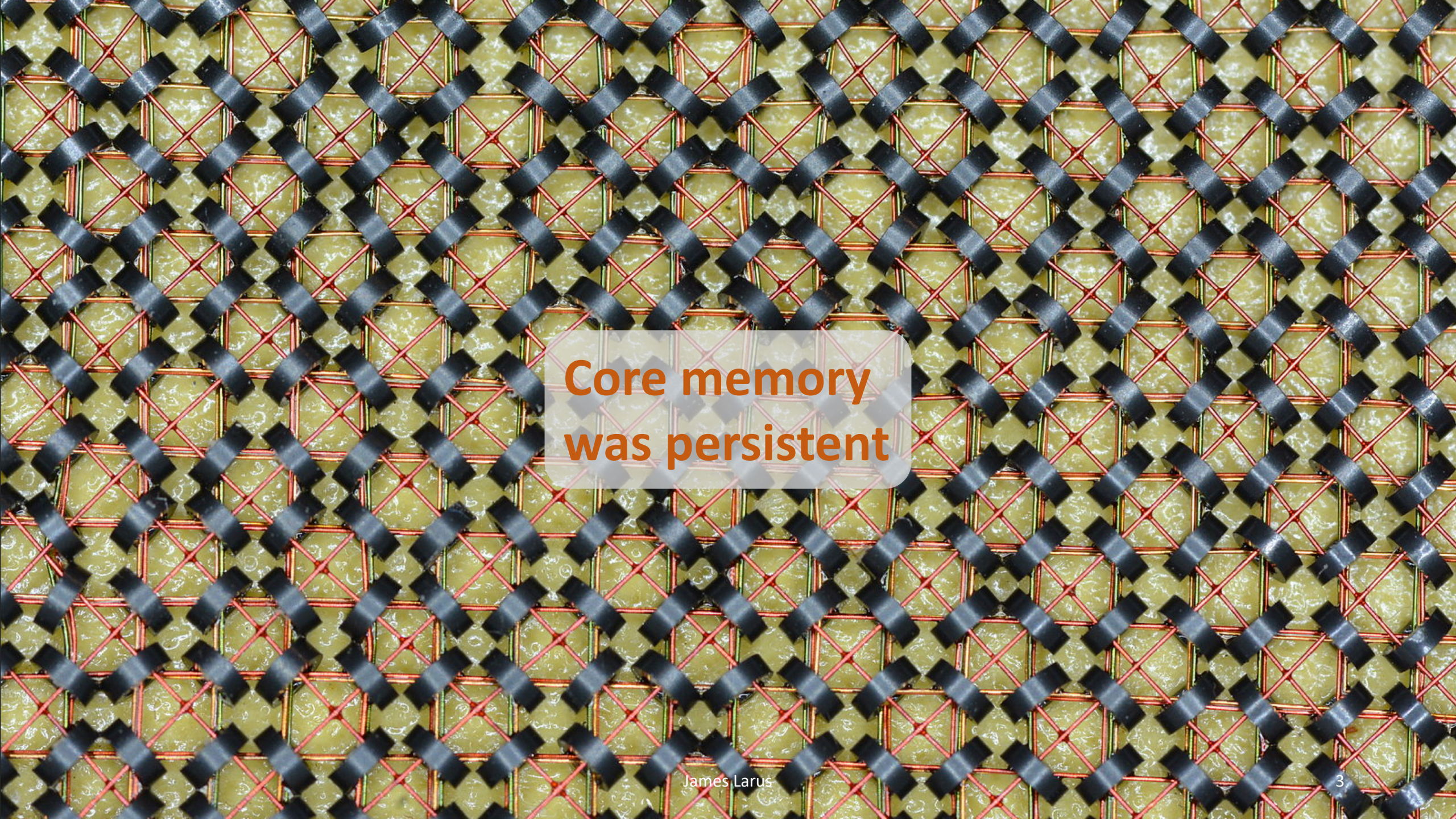

Ecole polytechnique fédérale de Lausanne
EPFL School of Computer and Communication Science (IC)
Programming Non-Volatile Memory
James Larus
Professor and Dean, IC
October 23, 2018

Joint Work

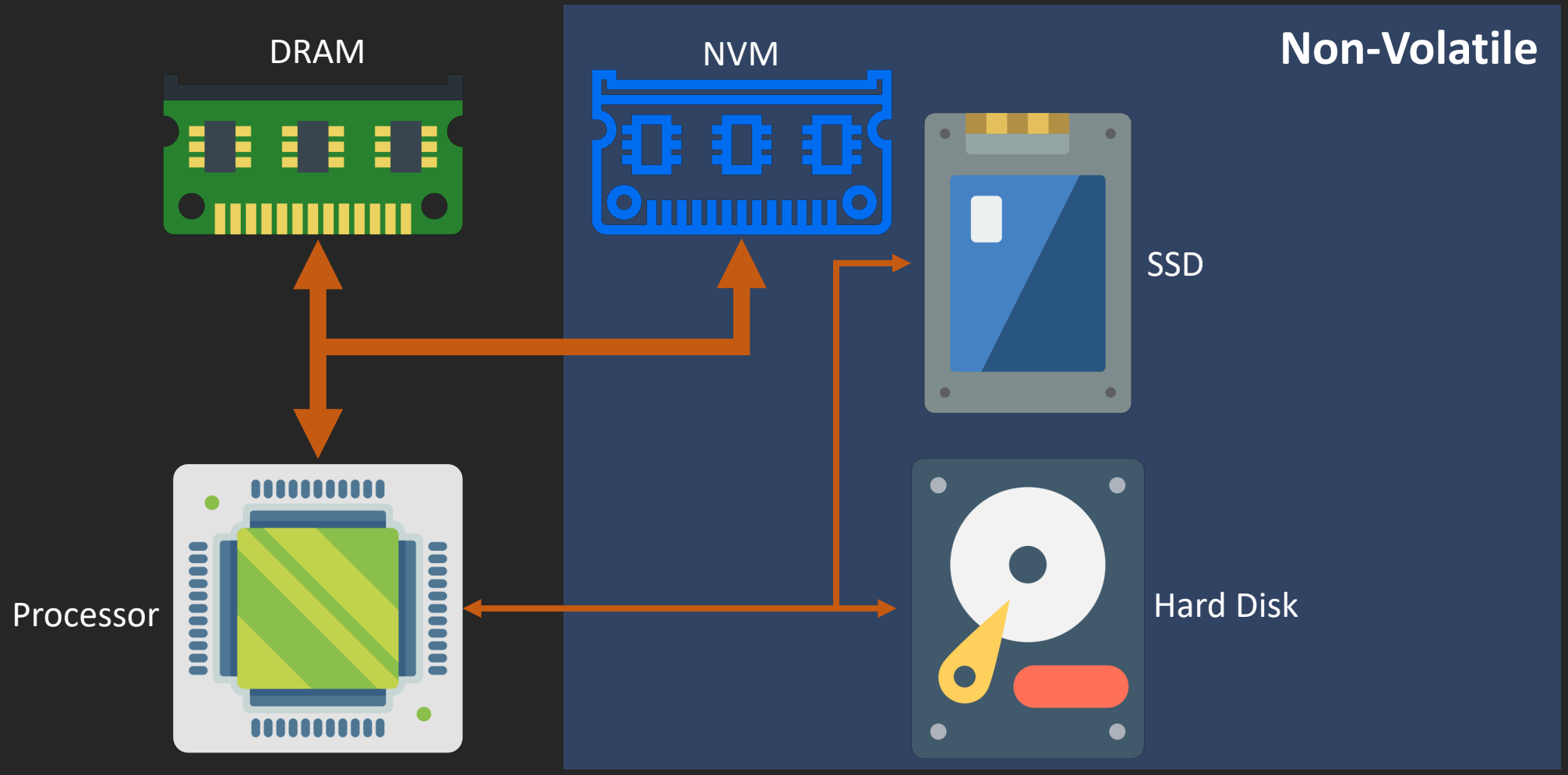
Nachshon Cohen, post doc

David Aksun, PhD student

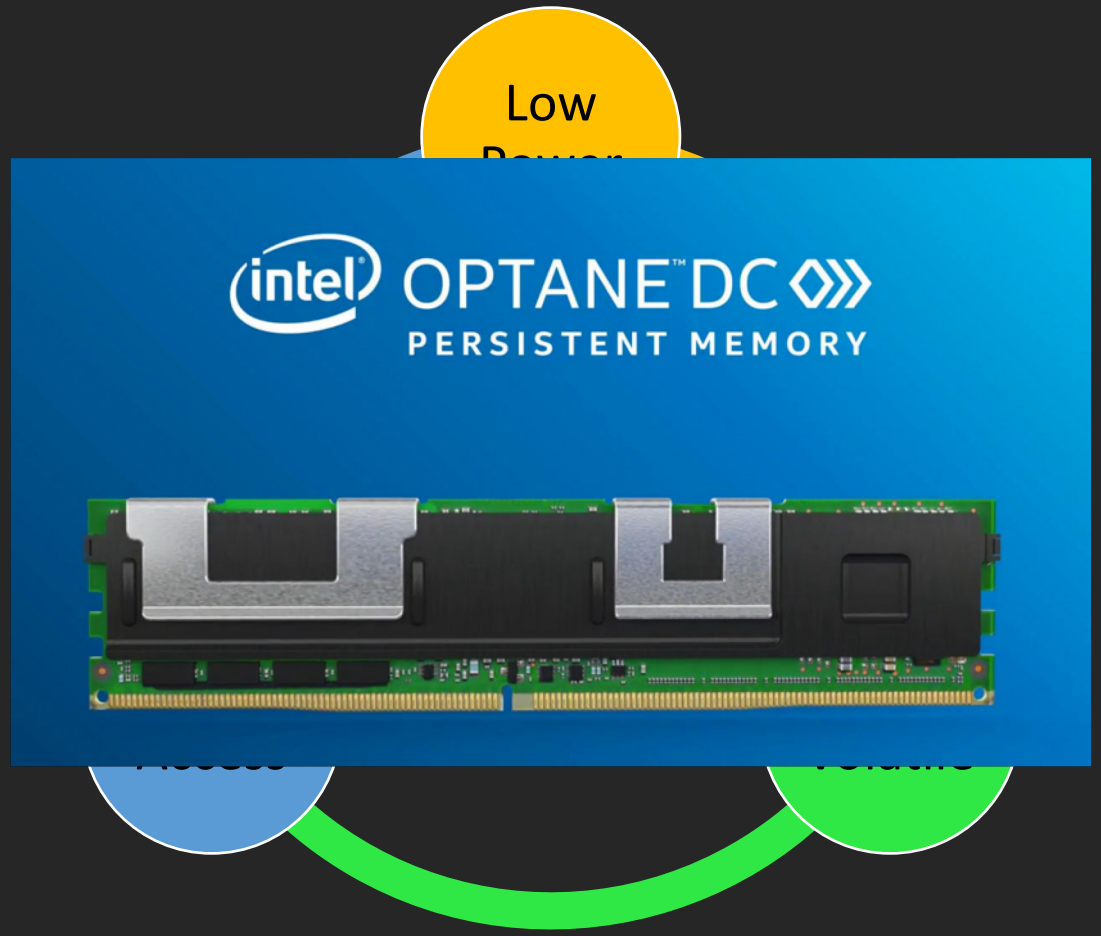

Core memory
was persistent

Background on NVM

Non-Volatile Memory (NVM)



NVM Characteristics

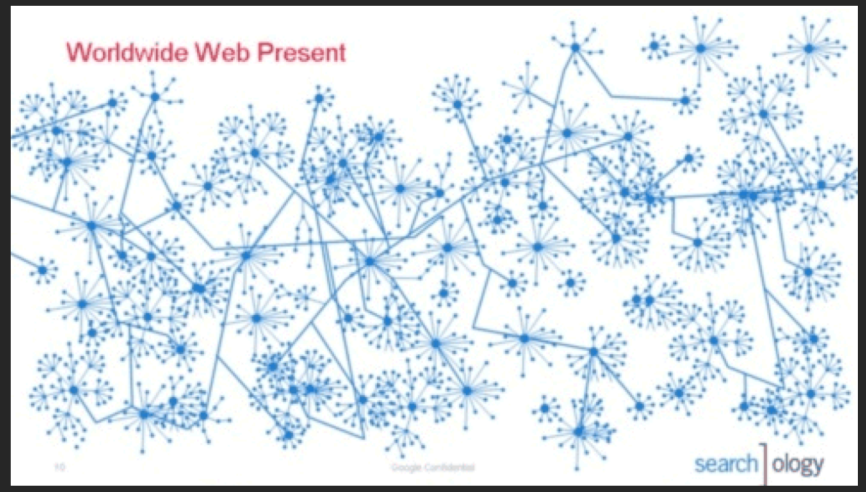


Only consumes power during memory access

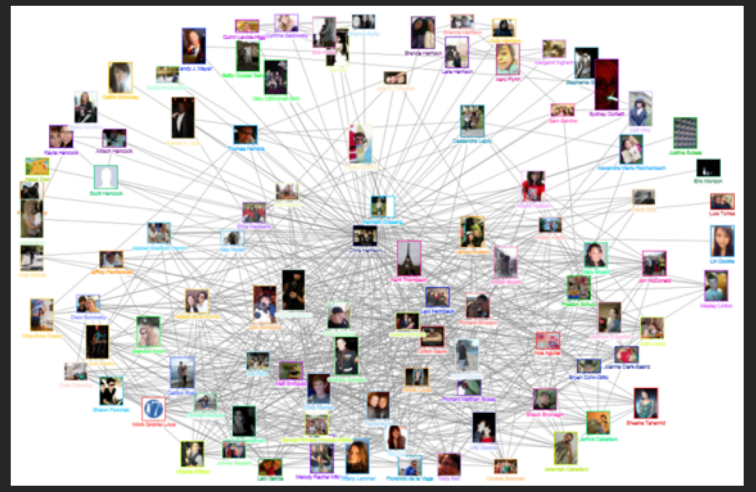
Retains data without power

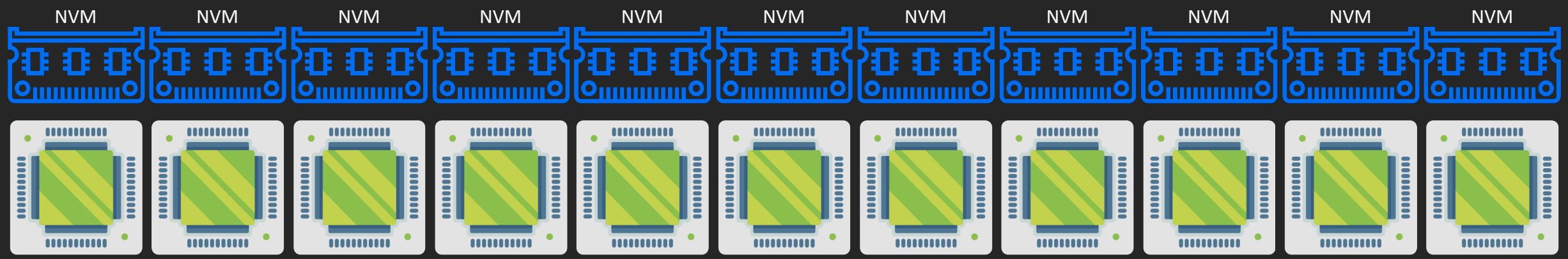
Direct byte-granularity data access

NVM Use Case



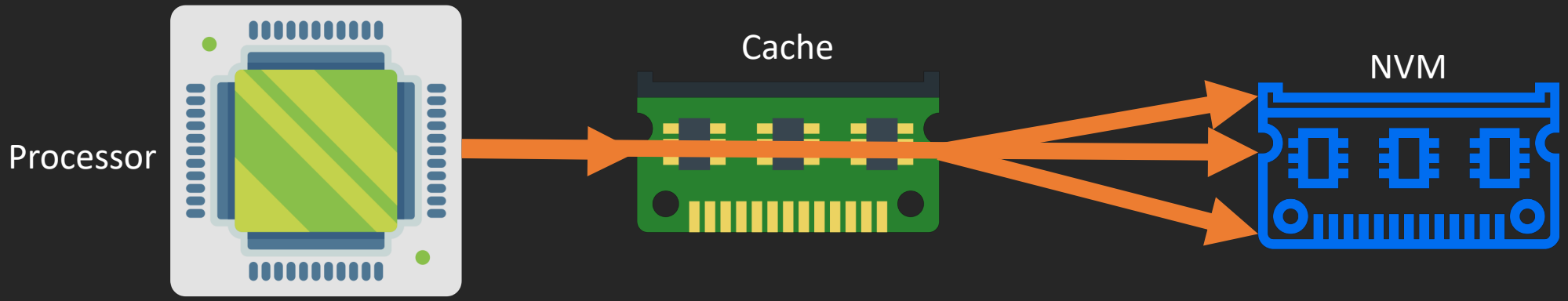
In-memory DB





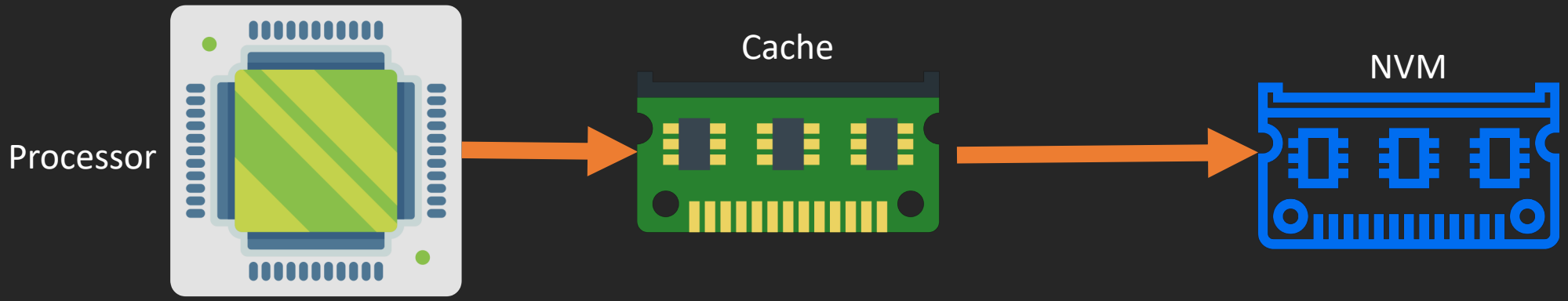
How Not to Program NVM

```
Transfer(Acct from, Acct to, Money amount) {
   from->balance -= amount;
   to->balance += amount;
}
```



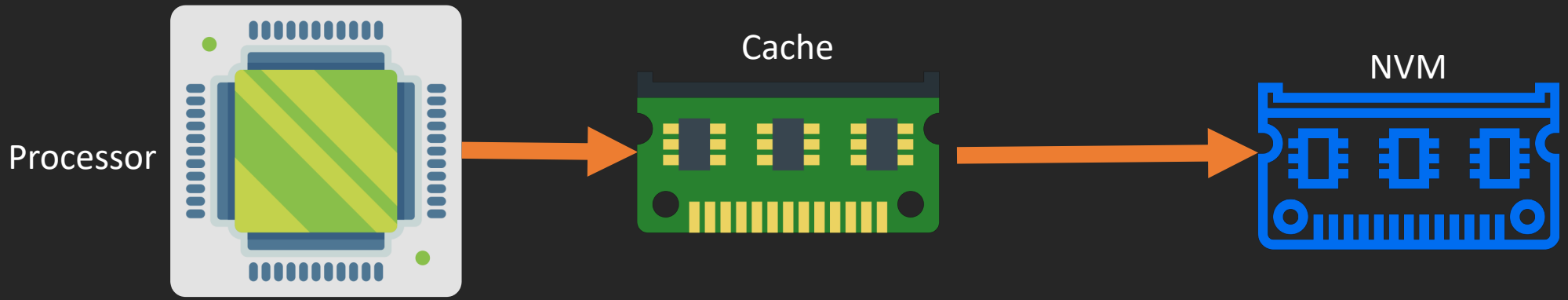
Programming NVM, version 1

```
Transfer(Acct from, Acct to, Mone      unt) {
  from->balance -= amount;
  to->balance += amount;
  clflush(&from->balance); clflush(&to->balance);
  sfence();
}
```

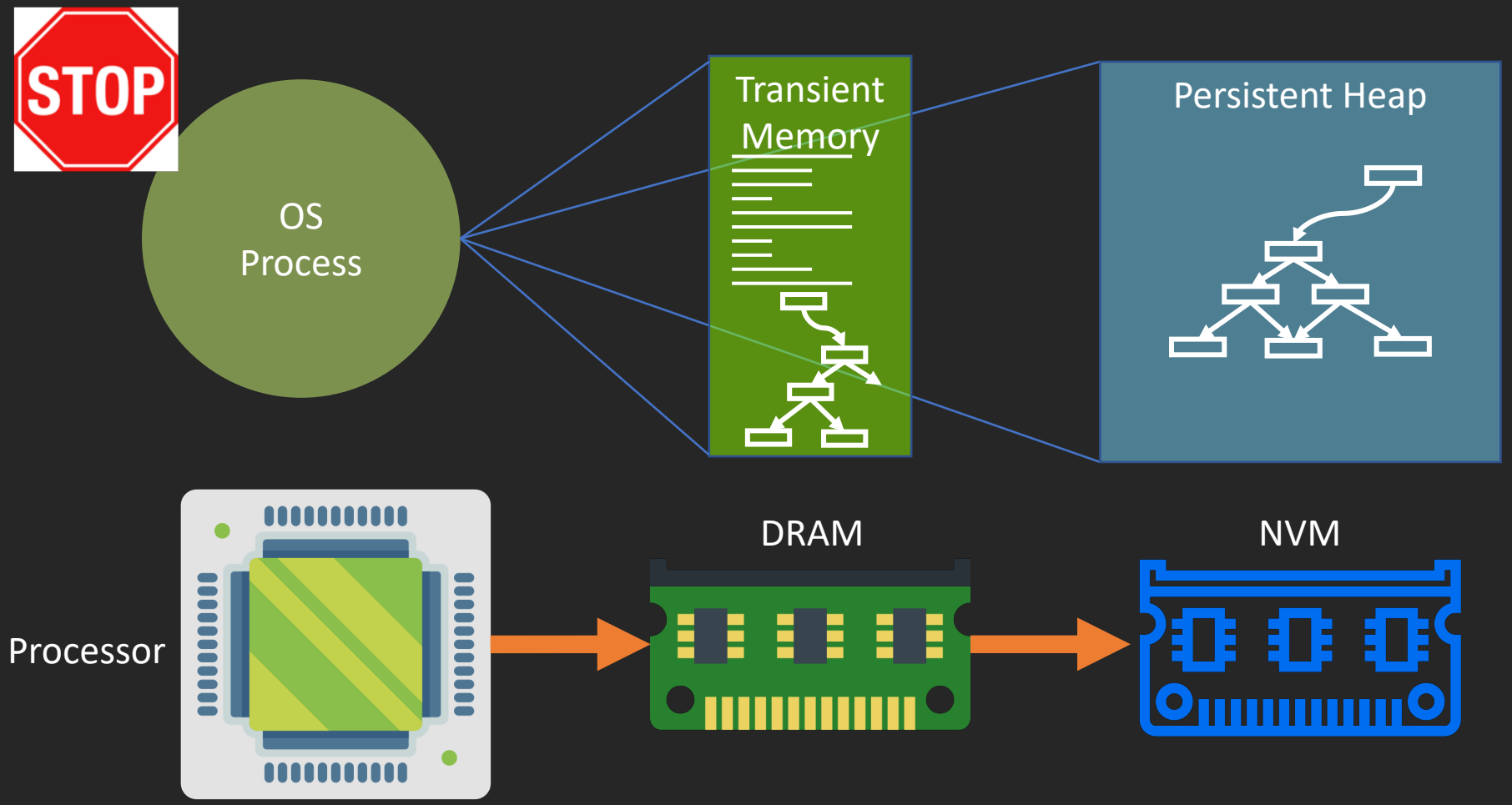


Programming NVM, version 2

```
Transfer(Acct from, Acct to, Money amount) {
  atomic {
    from->balance -= amount;
    to->balance += amount;
    clflush(&from->balance); clflush(&to->balance);
    sfence();
  }
}
```

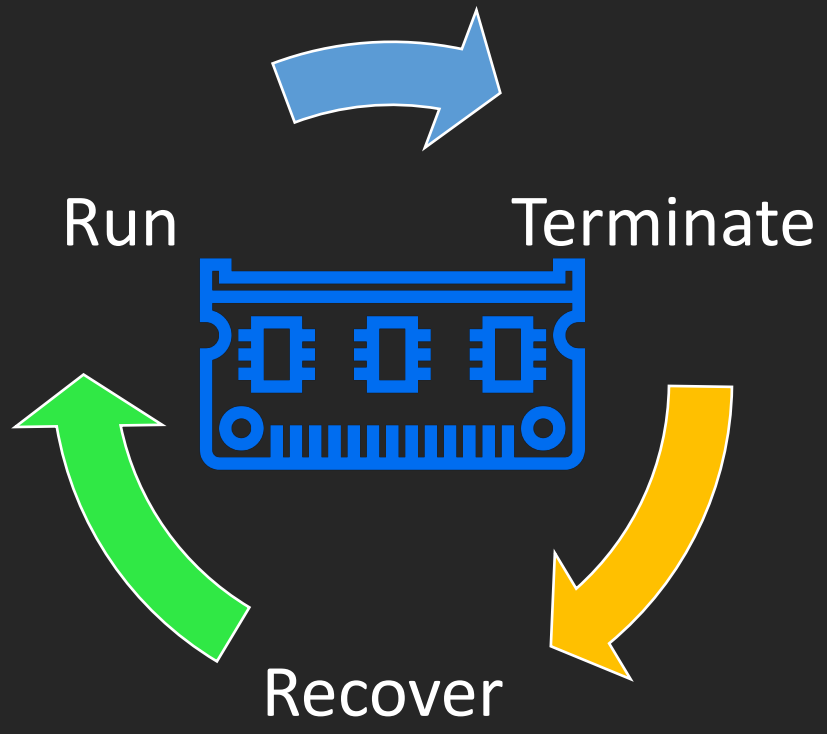


NVM Usage


STOP
OS
Process
Transient
Memory
Persistent Heap
Processor
DRAM
NVM

NVM Lifecycle

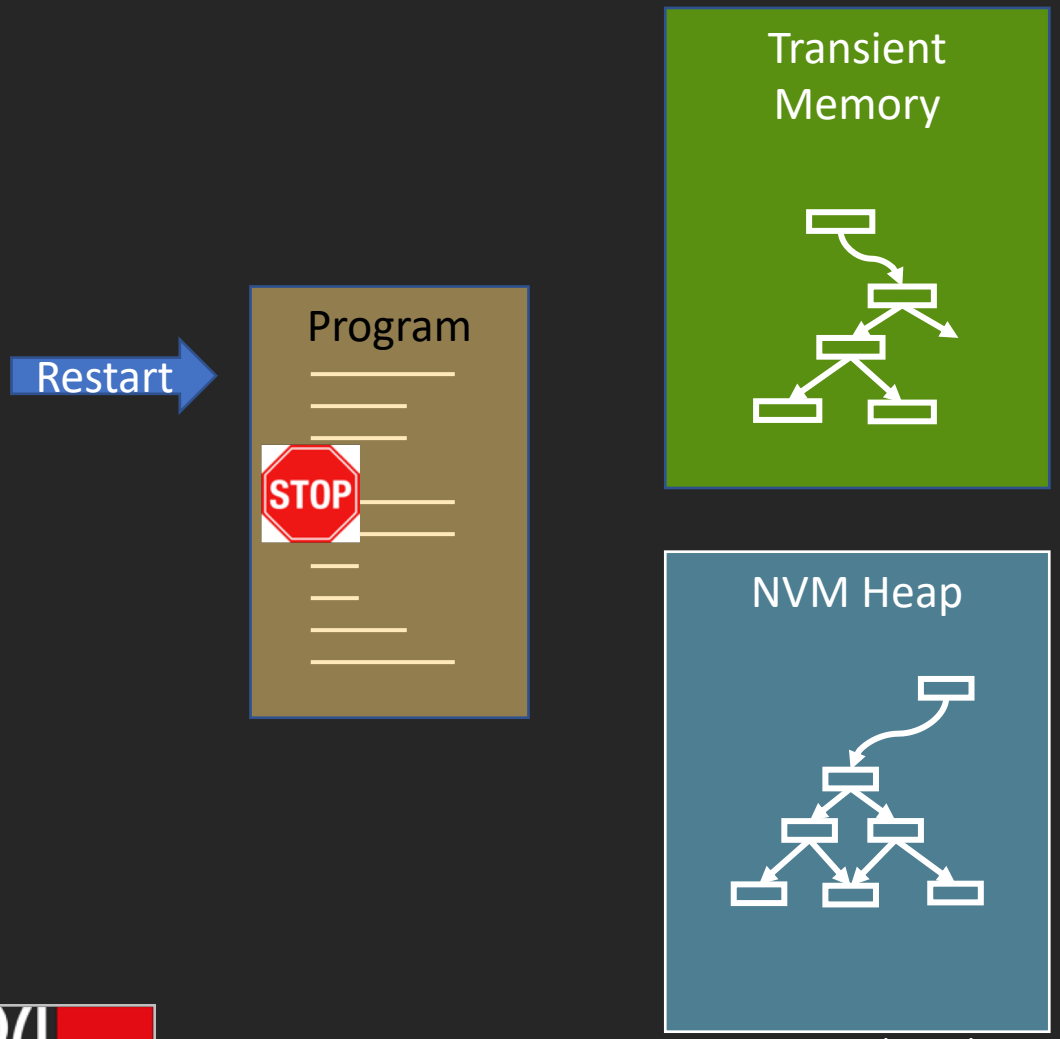
1. Processes accesses NVM with individual load and store instructions



2. NVM must record a consistent memory state before termination, whether planned or unexpected

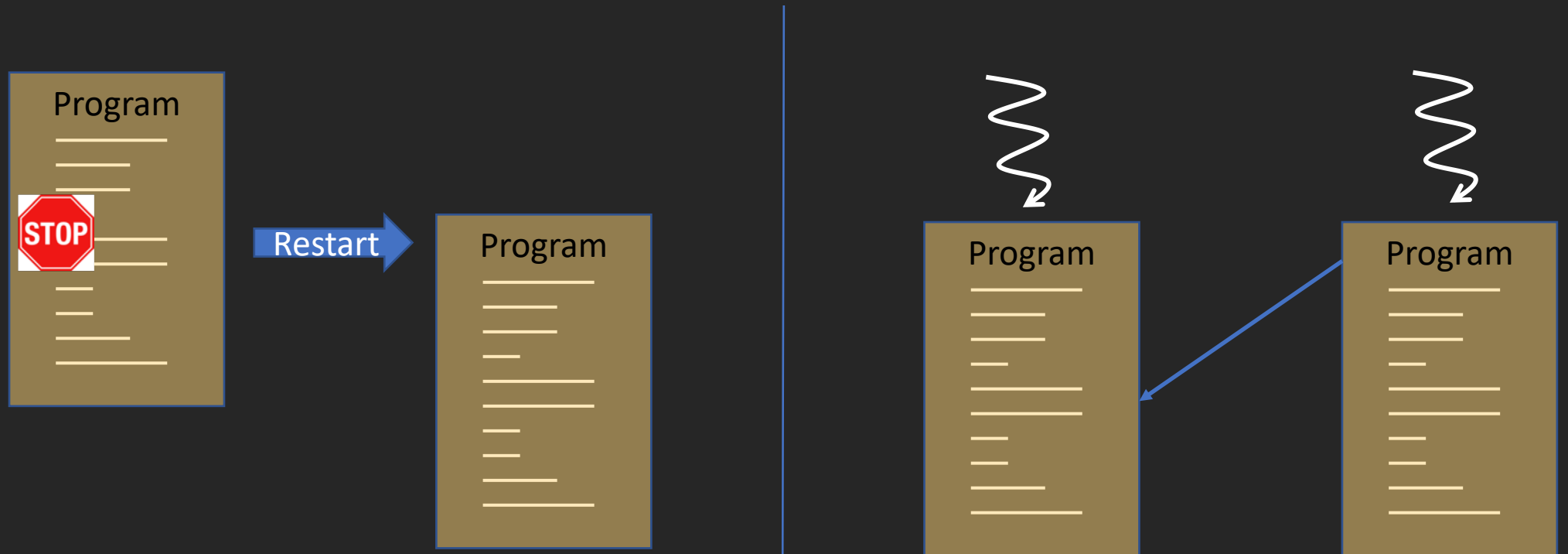
3. Need to ensure that NVM state is consistent in the environment in which execution restarts.

When is NVM “Consistent”?



- NVM is consistent when restarted program produces same output as some execution of the program without a premature termination
- In general, error recovery is difficult if memory state does not satisfy program invariants
- Programs transition between invariant-satisfying states, but states in between are inconsistent (and impossible to recover)
- NVM should not be left in one of these inconsistent states

Similar to Concurrency

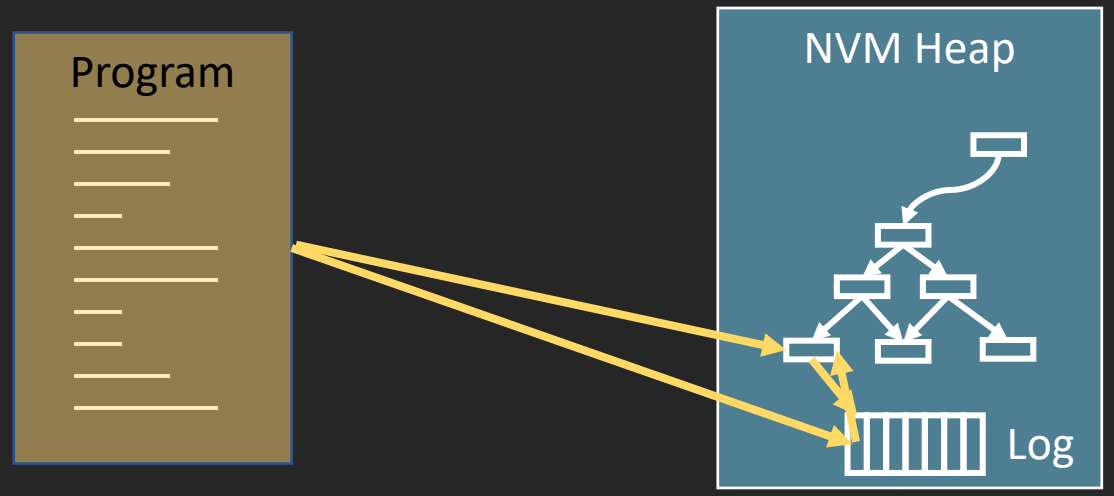


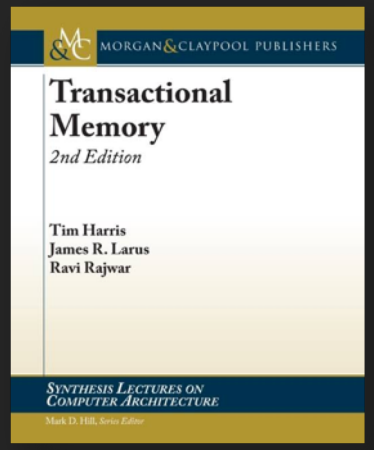
Key difference: execution stops and only part of state is preserved. Some parallelization optimizations (such as privatization) can cause errors.

Durable Transactions

```
Transfer(Acct from, Acct to, Money amount) {
   atomic {
      from->balance -= amount;
      to->balance += amount;
   } //clflush(&from->balance); clflush(&to->balance); sfence();
}
```

Software Transactional Memory





Undo log – save contents of overwritten memory locations, so they can be restored if transaction fails

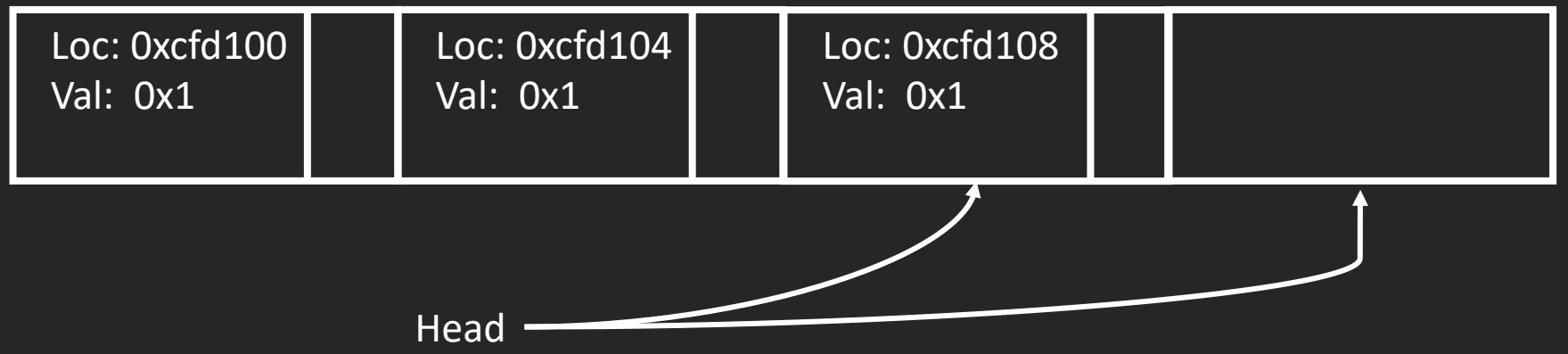
Redo log – save updated values, so they can applied to memory if transaction succeeds

End of background

Outline

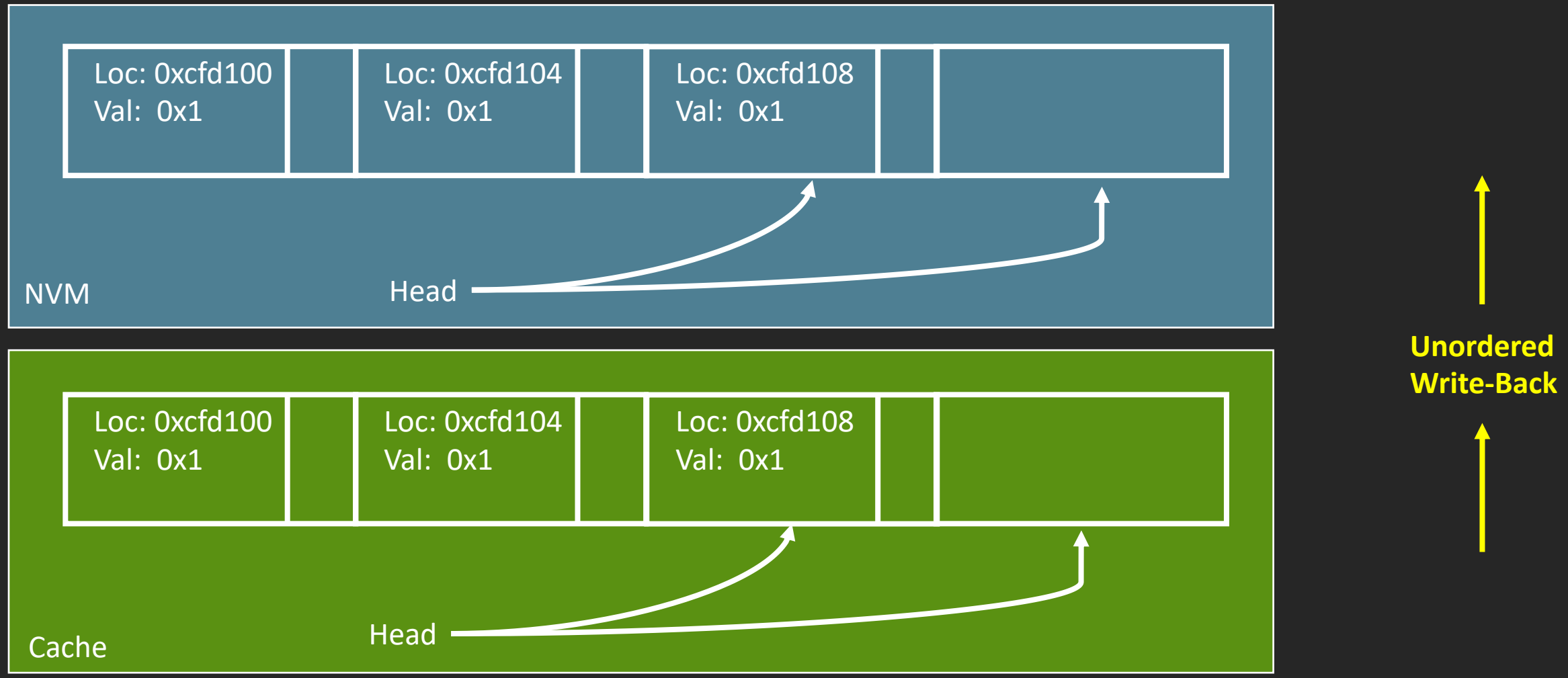
- Efficient Logging
- Checkpointing with InCLL
- NVM Recovery

Efficient Logging

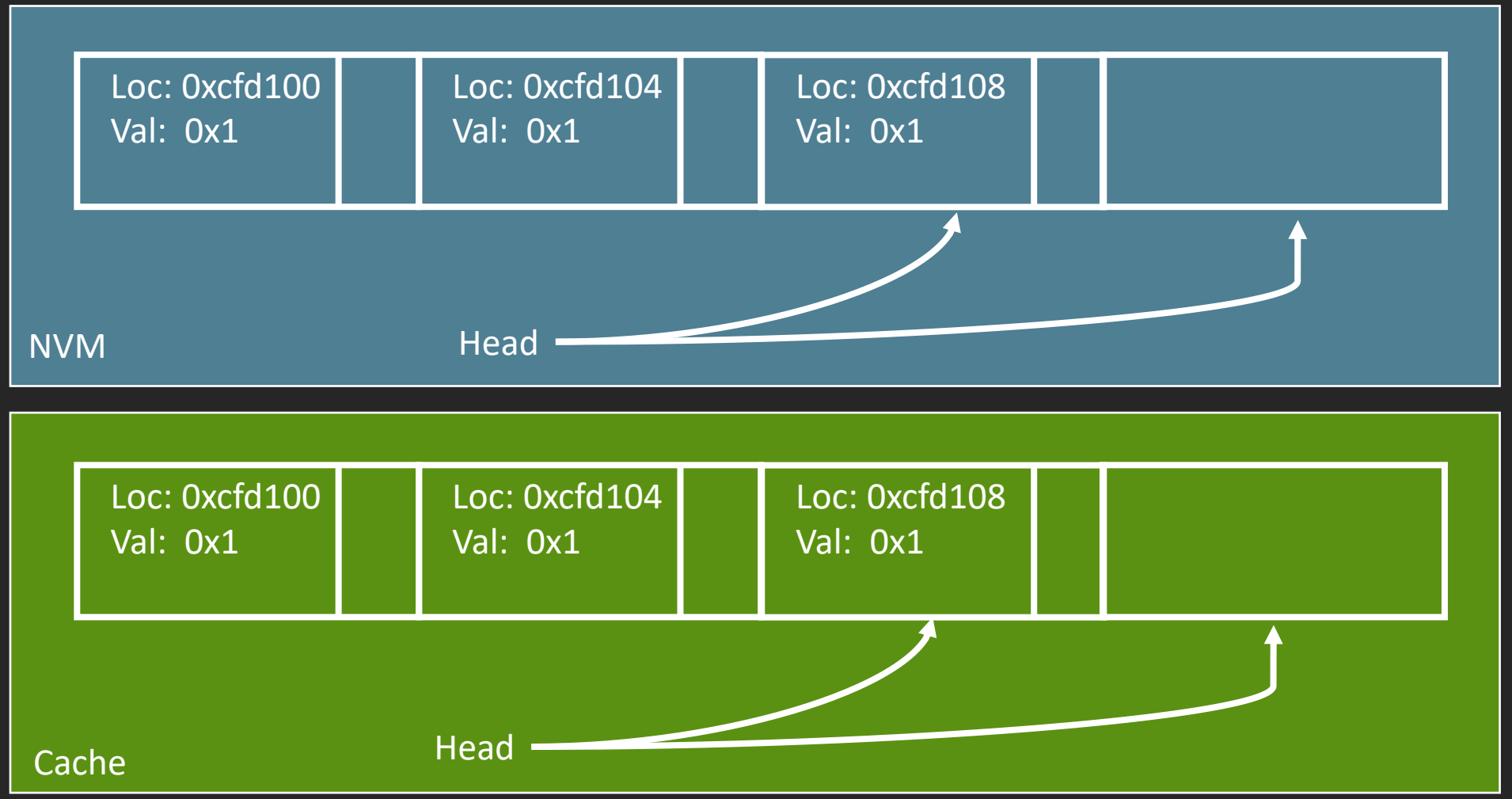


Cohen, Friedman, Larus. **Efficient Logging in Non-Volatile Memory by Exploiting Coherency Protocols**. OOPSLA 2017.

NVM, Caches, and Logging



NVM, Caches, and Logging



Cache Line Flush

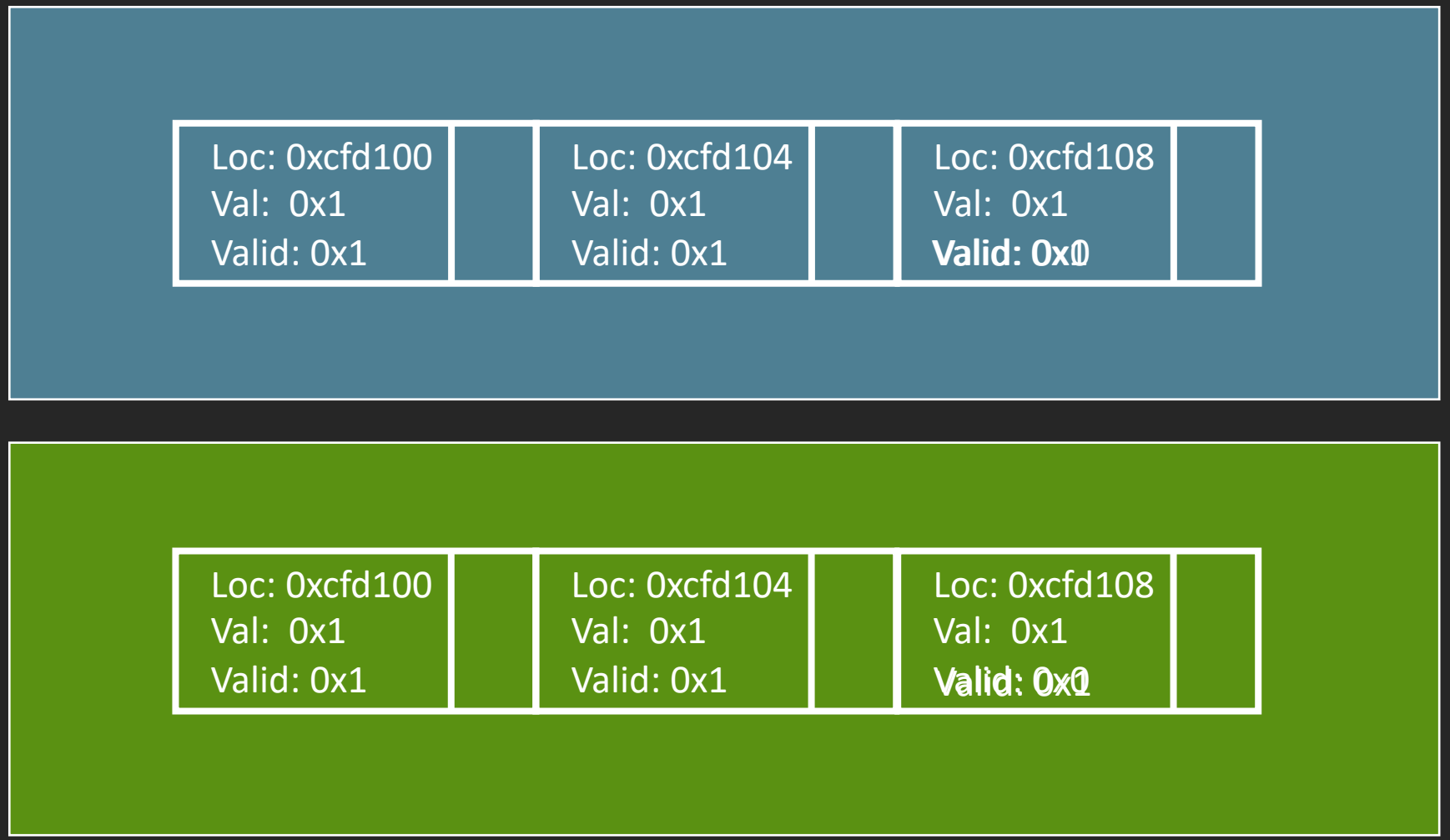
70-200 ns latency x2

NVM Consistency Model

- Assumption 1: If store instruction S_1 becomes visible to other threads before instruction S_2, then the value written by S_1 reaches the cache before the value written by S_2
- Assumption 2: A cache line is transferred from the cache to the NVM atomically

- **Persistent ordering**
  - $\dfrac{S_1 <_{hb} cflush(a(S_1)) <_{hb} sfence(a(S_2)) <_{hb} S_2}{S_1 <_{hb} S_2}$ (explicit flush)
  - $\dfrac{S_1 <_{hb} S_2 \wedge \mathrm{c}(\mathrm{S}_1) = \mathrm{c}(\mathrm{S}_2)}{\mathrm{S}_1 <_{\mathrm{hb}} \mathrm{S}_2}$ (granularity)

Validity Bit



Assume: log entry fits in cache line (64 bytes) and at least one bit (of 512) is unused

Log entry is valid if bit is set (unset)

Requires only 1 cache flush

Cache Line Flush

What If Validity Bit Does Not Fit?

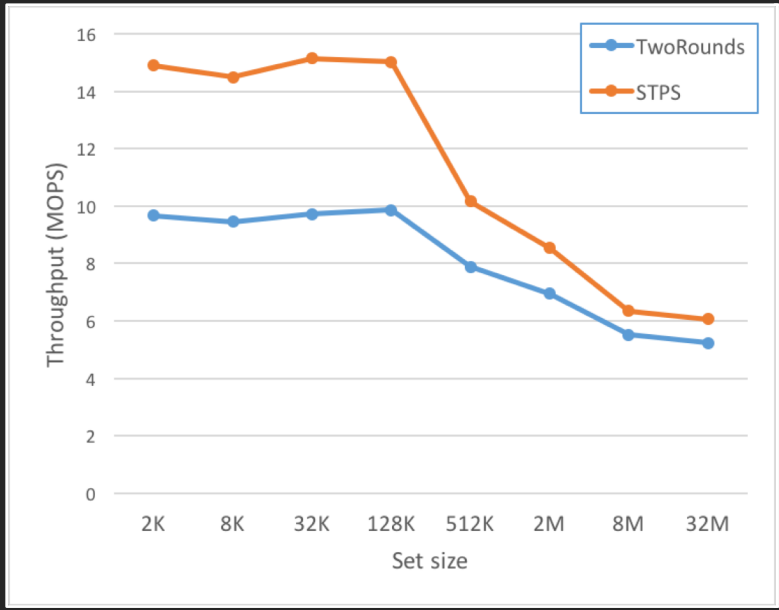
- Randomization
  - Initialize log memory with 64-bit random value
  - Write log entry in order
  - If last word is not random value, then entry is valid (with high probability)

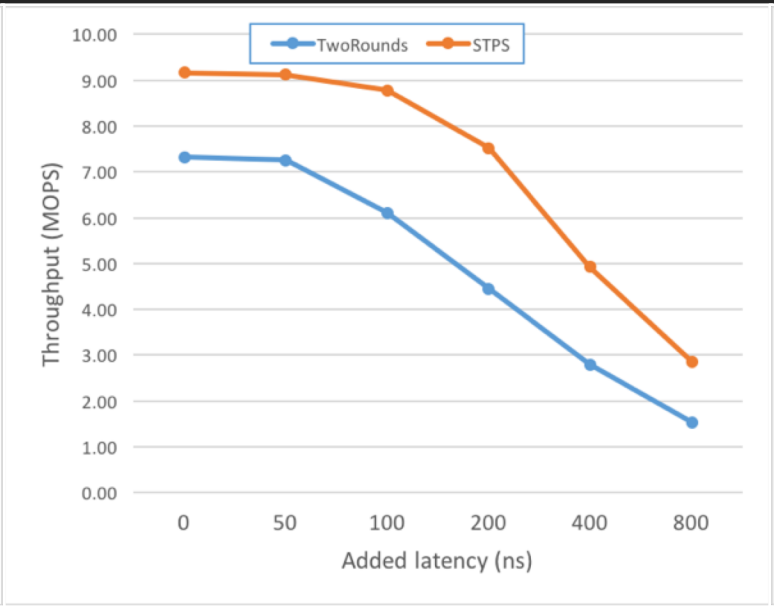
- Flexible validity bit
  - Find first bit different between old cache line and new value
  - If bit exists, use it as validity bit (otherwise, doesn't matter)
  - Store bit position in external table

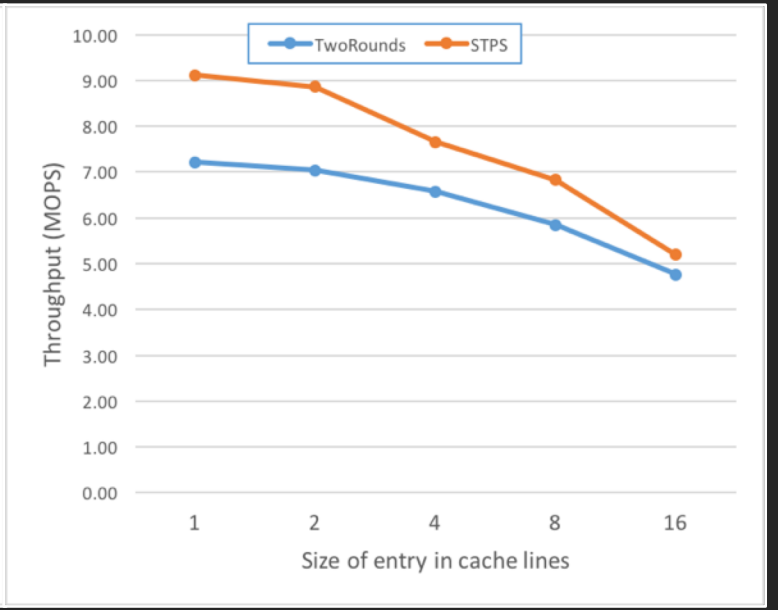
- Almost always can find validity bit
  - x64 address have 15 unused bit at top and typically 2/3 unused bits at bottom

- More details and examples in paper

Performance







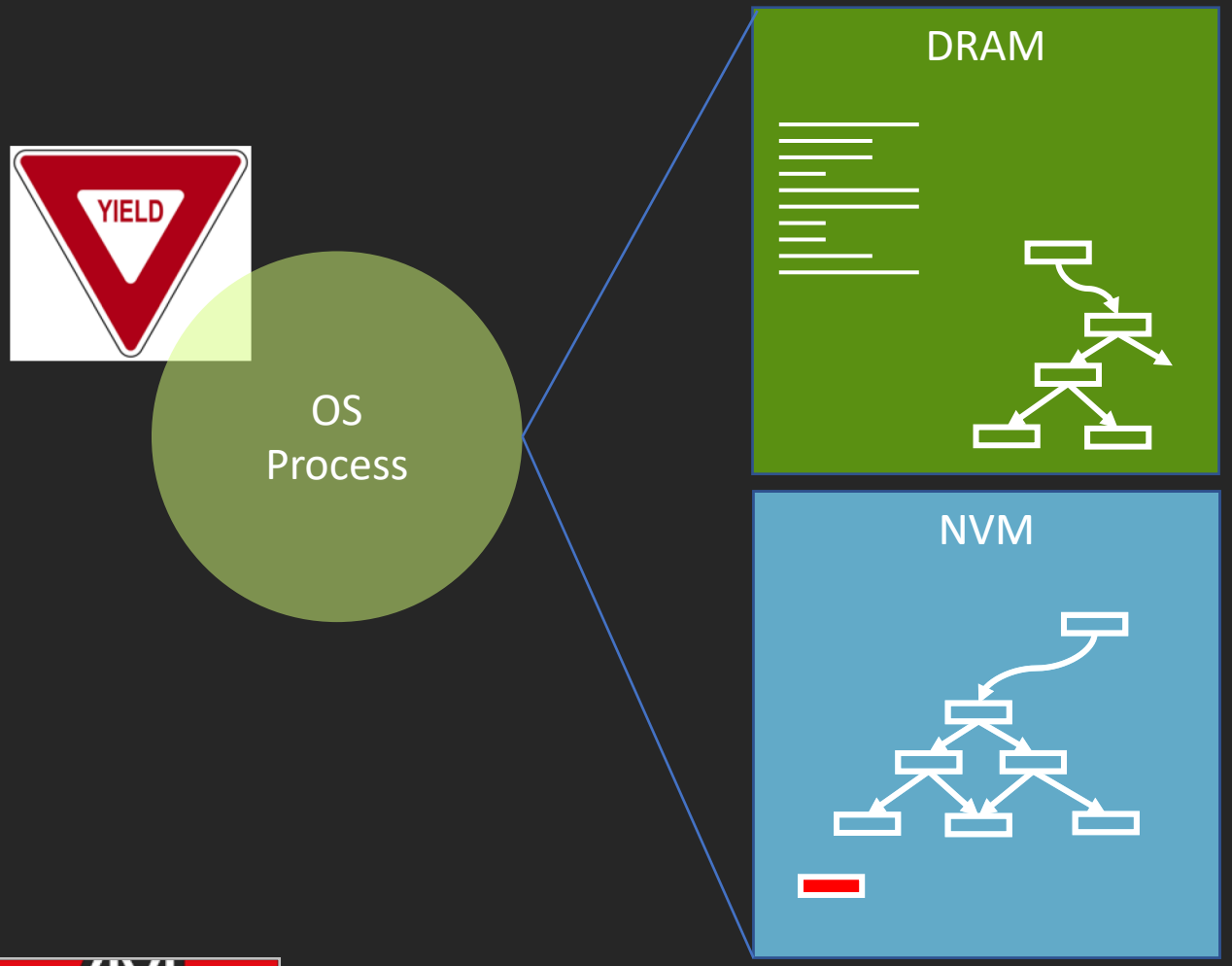
Single-Trip Persistent Set (STPS)
Two cache flushes (TwoRounds)

YCSB write heavy benchmark (50% writes)

Outline

- Efficient Logging
- Checkpointing with InCLL
- NVM Recovery

Checkpointing



Checkpointing used in high-performance computing (HPC)

Problems:

- Long pause while copying heap
- Recovery time proportional to checkpoint interval

Fine-Grain Checkpointing of Masstree

- Masstree is cache-efficient combination of trie and B+ tree
  - Mao, Kohler, Morris. **Cache craftiness for fast multicore key-value storage**. EuroSys '12, 2012
- Used in Silo in-memory DB, key-value stores, etc.

- Make Masstree persistent by storing data structure in NVM

Cohen, Aksun, Larus. **Fine-Grain Checkpointing with In Cache Line Logging**. Submitted for Publication.

Epochs

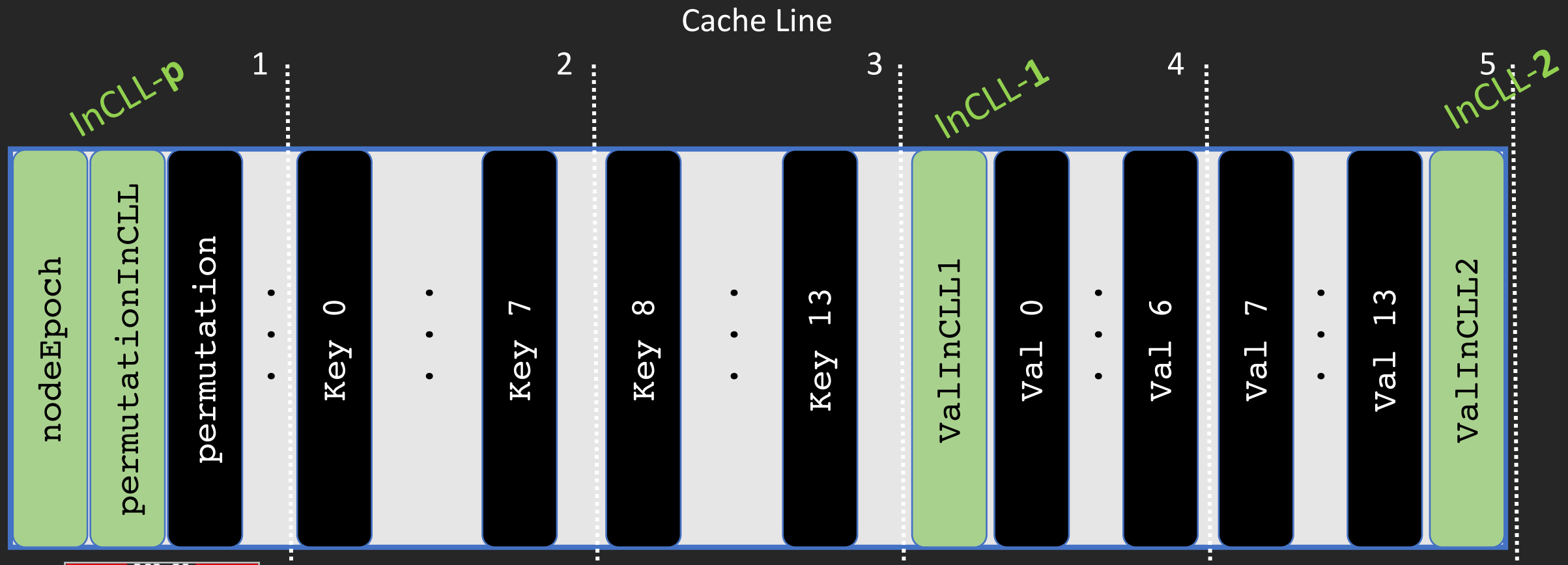
Faisal Nawab, et al., **Dalí: A Periodically Persistent Hash Map, DISC** 2017.

- Checkpoint with 64 ms epoch
  - Masstree uses this interval for allocating/reclaiming nodes
- Execute `wbinvd` instruction to flush entire cache (to NVM)
  - 430 – 550µs (< 1%)
- Failure during epoch causes execution to restart after previous epoch

- Need to undo changes written during a failed epoch
  - **In cache-line log (InCLL)**
  - Separate undo log for complicated, infrequent cases

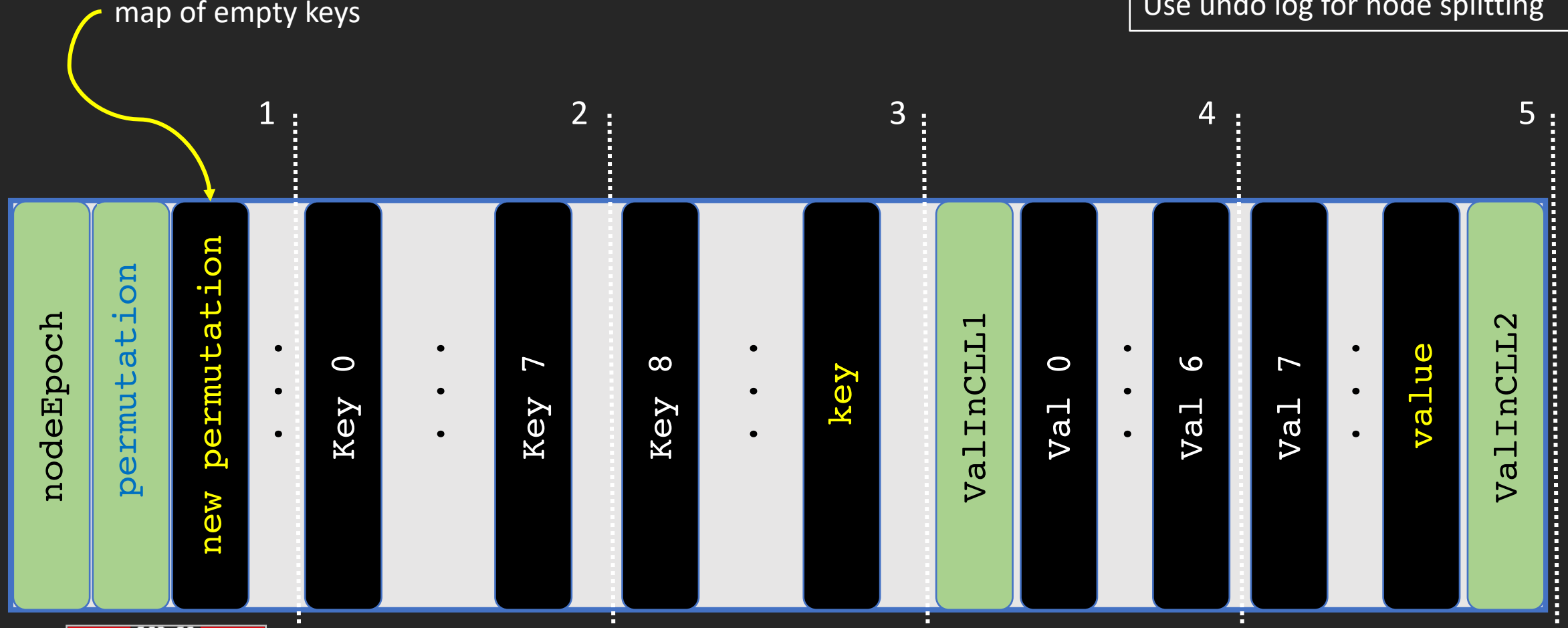
Masstree Leaf Node

14 keys, values
(1 fewer than standard)

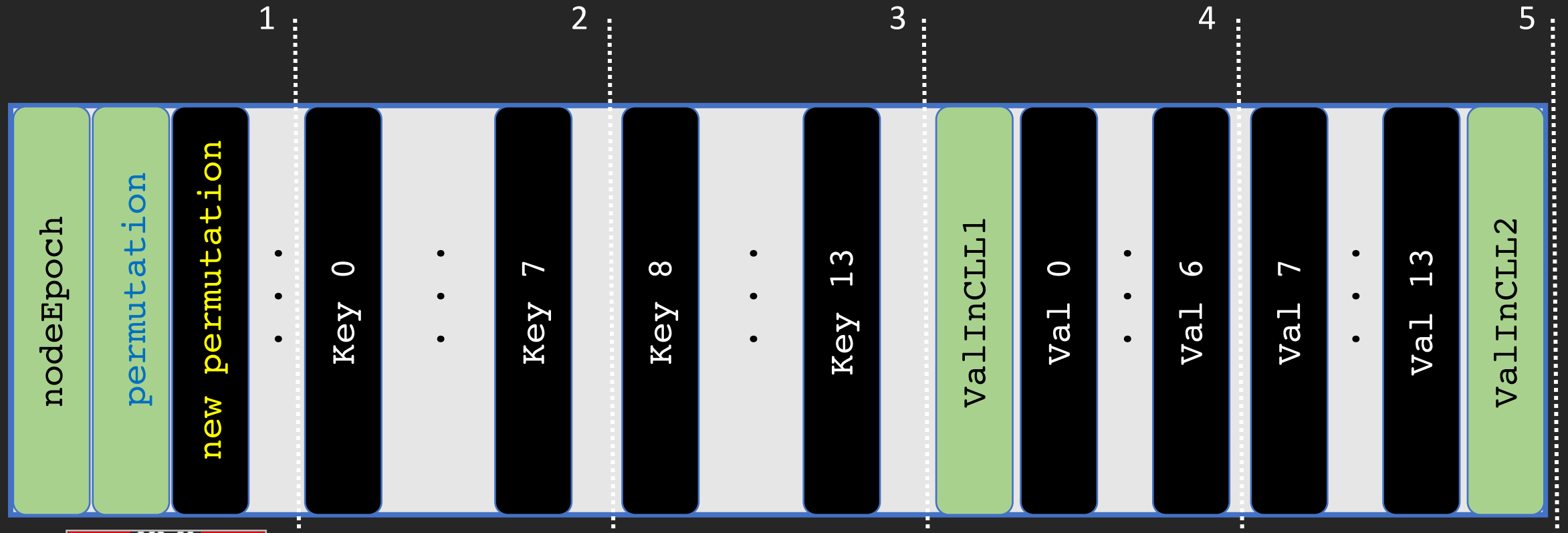


Insert (With Empty Value Slot)

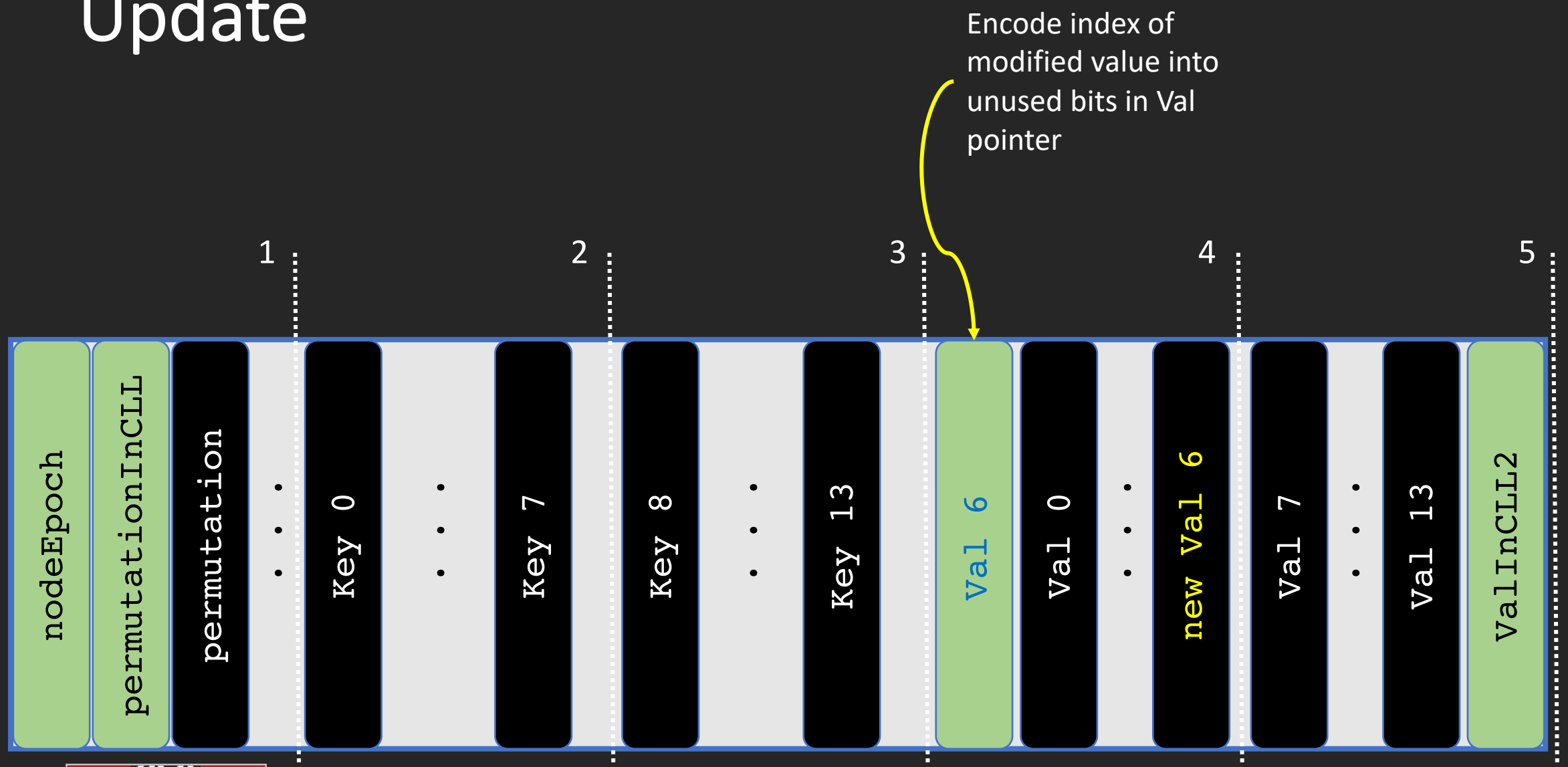
If no empty slot, then split node
Use undo log for node splitting



Delete


1
2
3
4
5
nodeEpoch
permutation
new permutation
. . .
Key 0
. . .
Key 7
Key 8
. . .
Key 13
ValInCLL1
Val 0
. . .
Val 6
Val 7
. . .
Val 13
ValInCLL2

Update


Encode index of modified value into unused bits in Val pointer
1
2
3
4
5
nodeEpoch
permutationInCLL
permutation
. . .
Key 0
. . .
Key 7
Key 8
. . .
Key 13
Val 6
Val 0
. . .
new Val 6
Val 7
. . .
Val 13
ValInCLL2

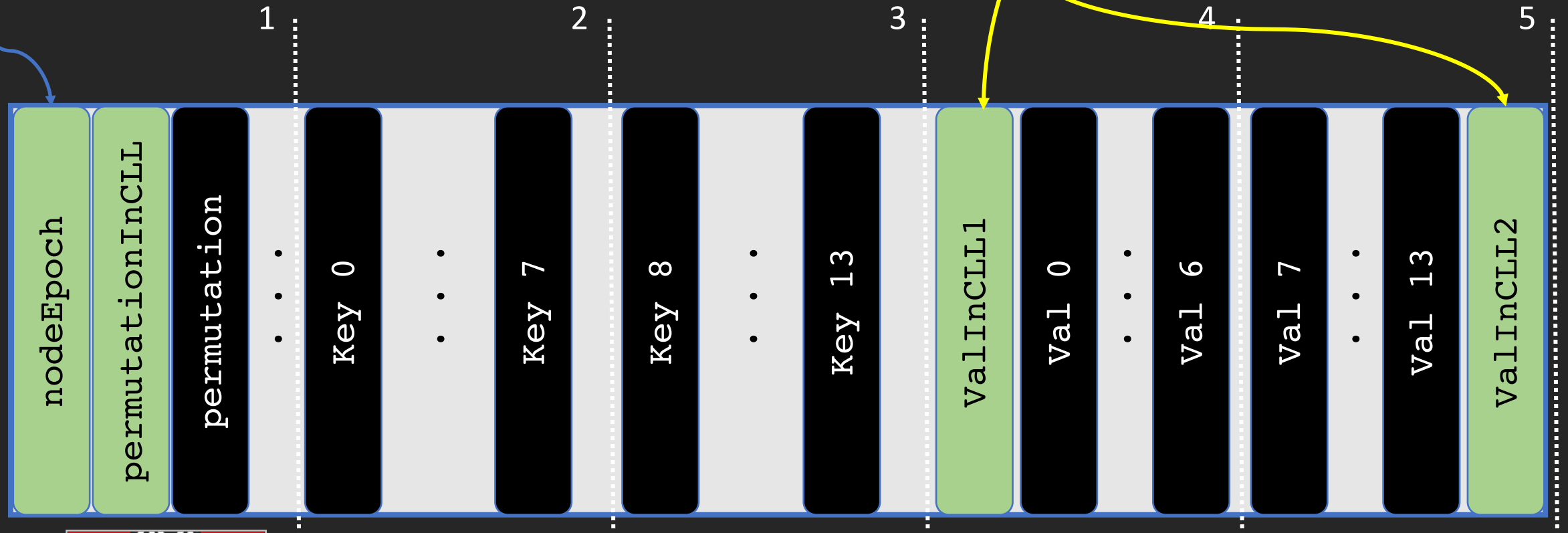
Sequences of Operations

Mixed sequences of insert delete require redo logging because InCLL can only hold one value

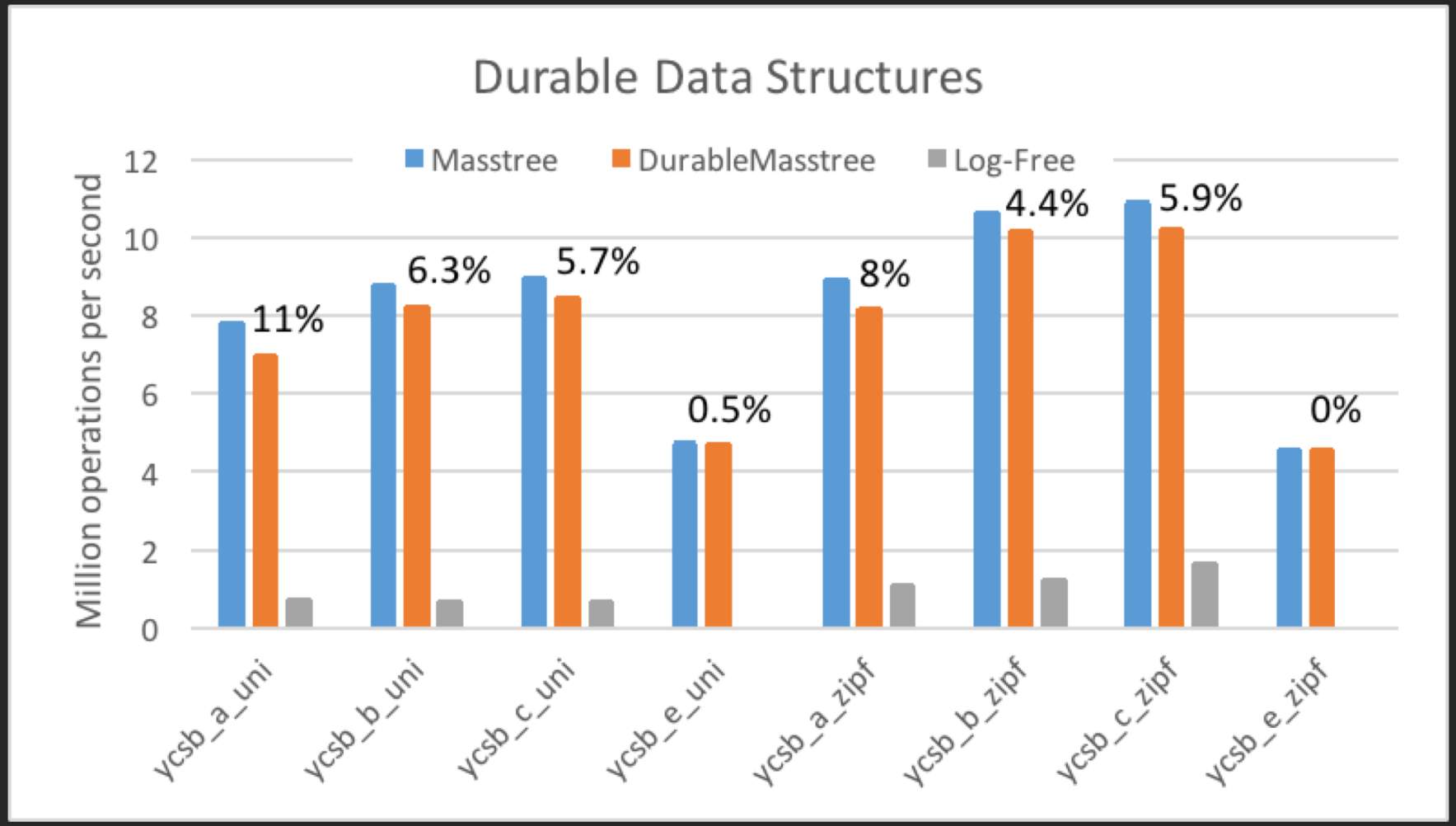
Epoch in which permutation was checkpointed

- Only copy permutation when Epoch differs (once per epoch)
- Allows lazy restoration

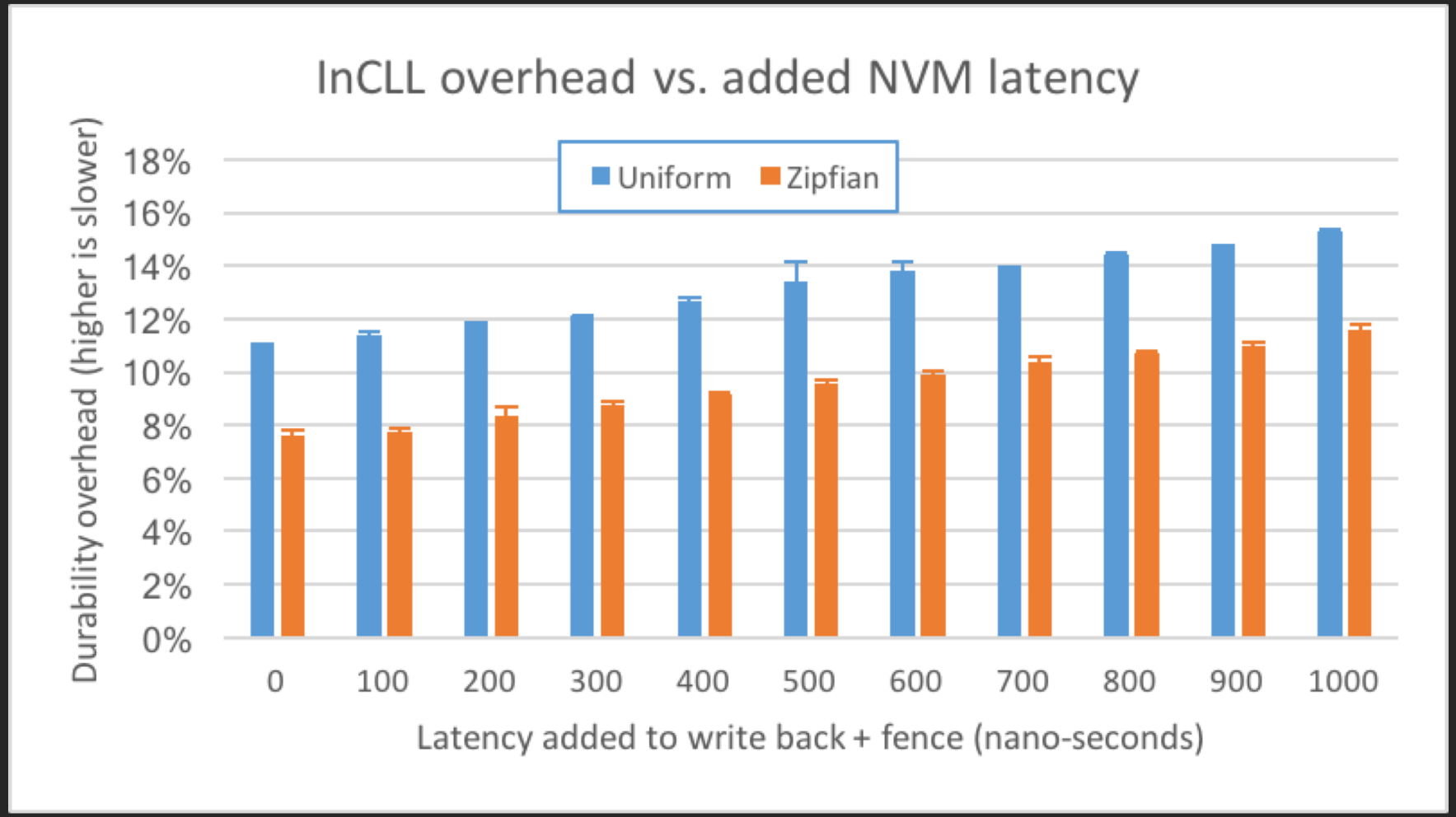
Also encode epoch in unused bits



Performance



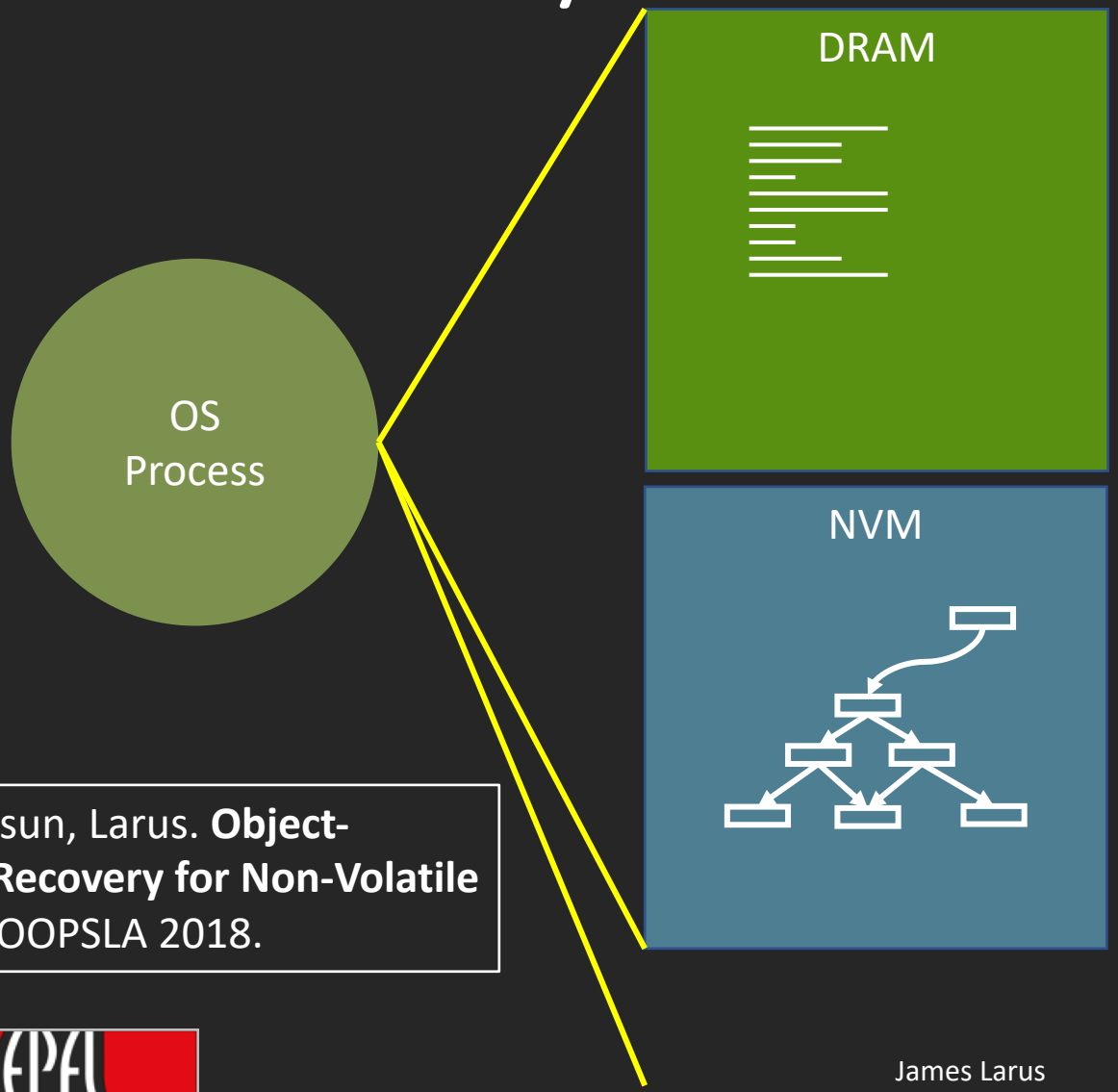
Performance (Added NVM Latency)



Outline

- Efficient Logging
- Checkpointing with InCLL
- NVM Recovery

NVM Recovery



Cohen, Aksun, Larus. **Object-Oriented Recovery for Non-Volatile Memory**. OOPSLA 2018.

Persistent objects point to methods

- Code may be loaded at different address because of ASLR, debugger, profiler, code changes, …

Persistent store may be mapped to different virtual address

Persistent data must be consistent in recovered process, as well as being consistent when original process fails

Potential Inconsistencies

- Transient objects pointed to from NVM
  - TCP sockets, locks, thread IDs, …
- Pointers between persistent objects if NVM mapped to different location
- Pointers to code and read-only data if text segment mapped differently

Previous Solutions

- Ignore the problem
  - If NVM maps to different location, quit…
    - Oops, there goes your data!
  - If text maps differently, continue…
    - Oops, there goes your data!
  - If you use an old lock, fail…
    - Oops, there goes your data!

- Use offsets between persistent objects instead of addresses
  - Memory access becomes more expensive
  - Requires extensive code changes

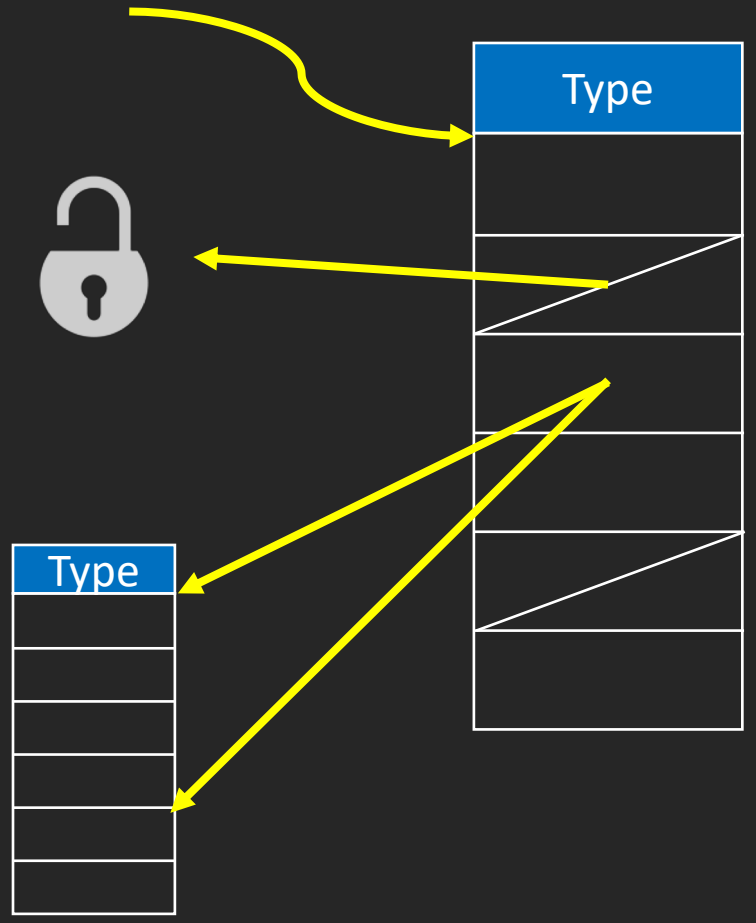
NVMReconstruction C++ Extension

```
struct kp_vt_struct {
  kp_kvstore *parent; // back-pointer to parent kvstore
  transient pthread_mutex_t *lock; // lock for this version table
   ...

  reconstructor(kp_vt_struct* o) {
    assert(kp_mutex_create("(*new_vt)->lock", &(o->lock)) == 0);
  }
}

void main() {
  kp_vt_struct *new_vt = pnew kp_vt_struct;
 ...
  pdelete new_vt;
}
```

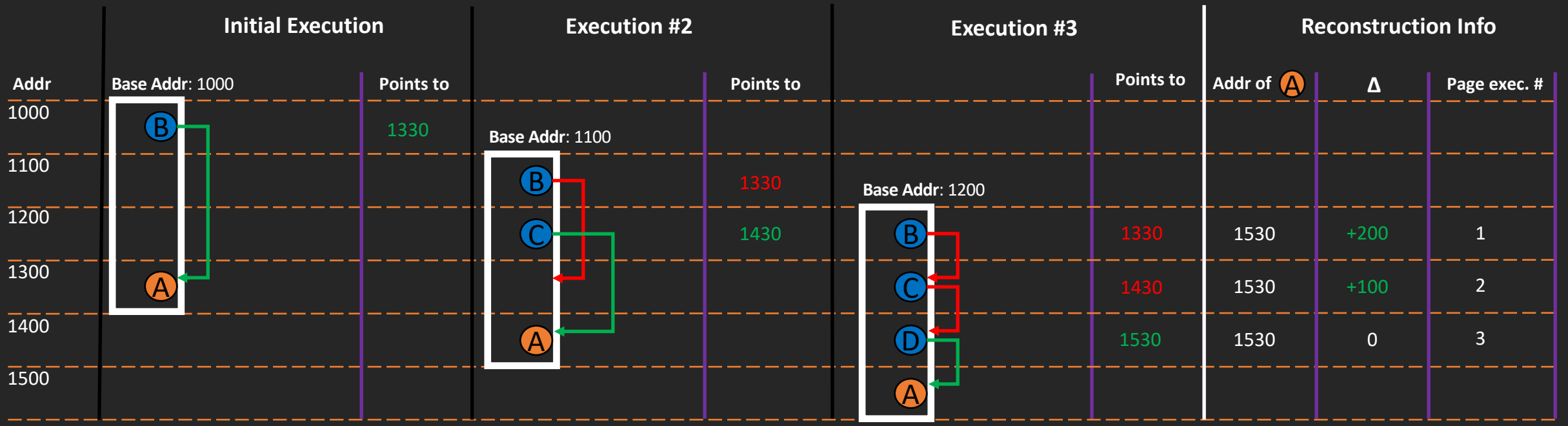
Reconstructing an Object



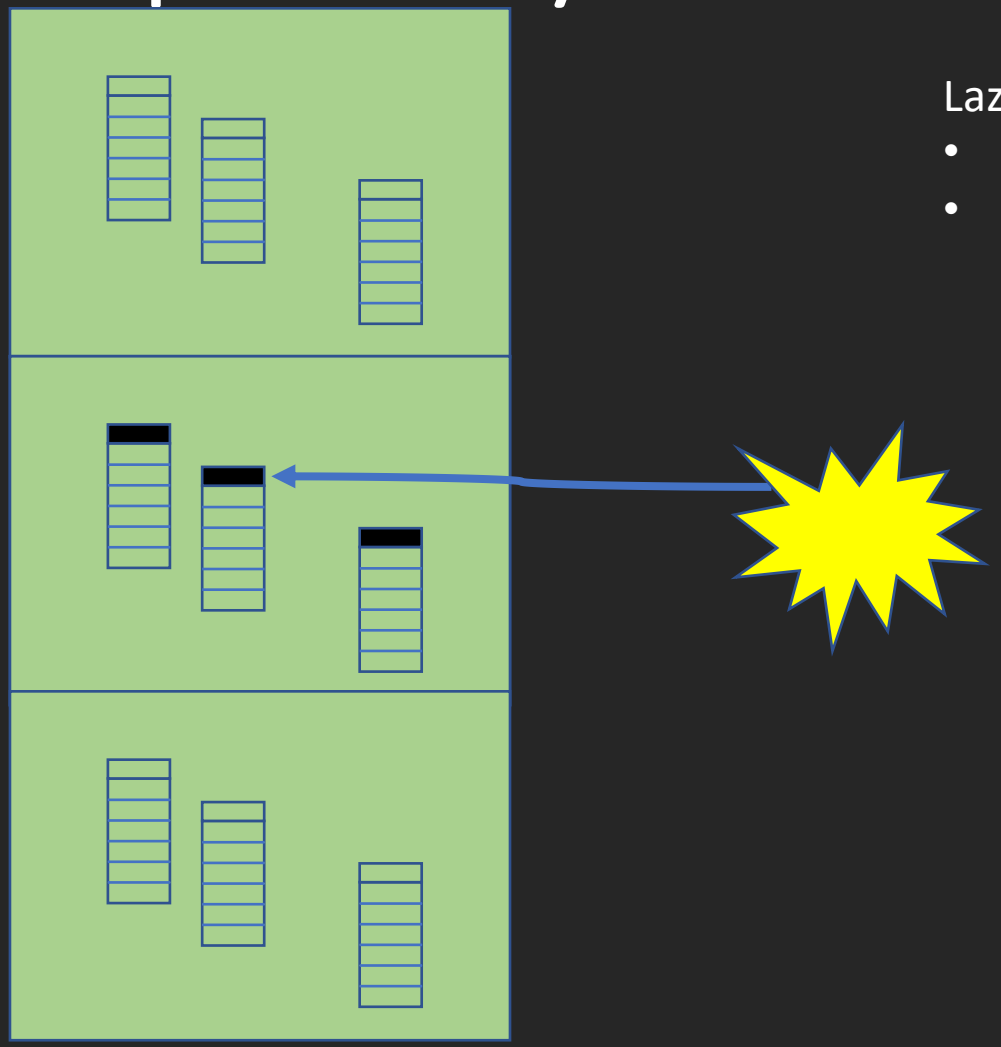
- Every persistent object has a type header
  - LLVM extension
- Zero transient fields
- Relocate pointers
  - Inter NVM
  - To code and read-only data
- Run reconstructor function

Similar to relocation in garbage collector, except…

Failure During Relocation



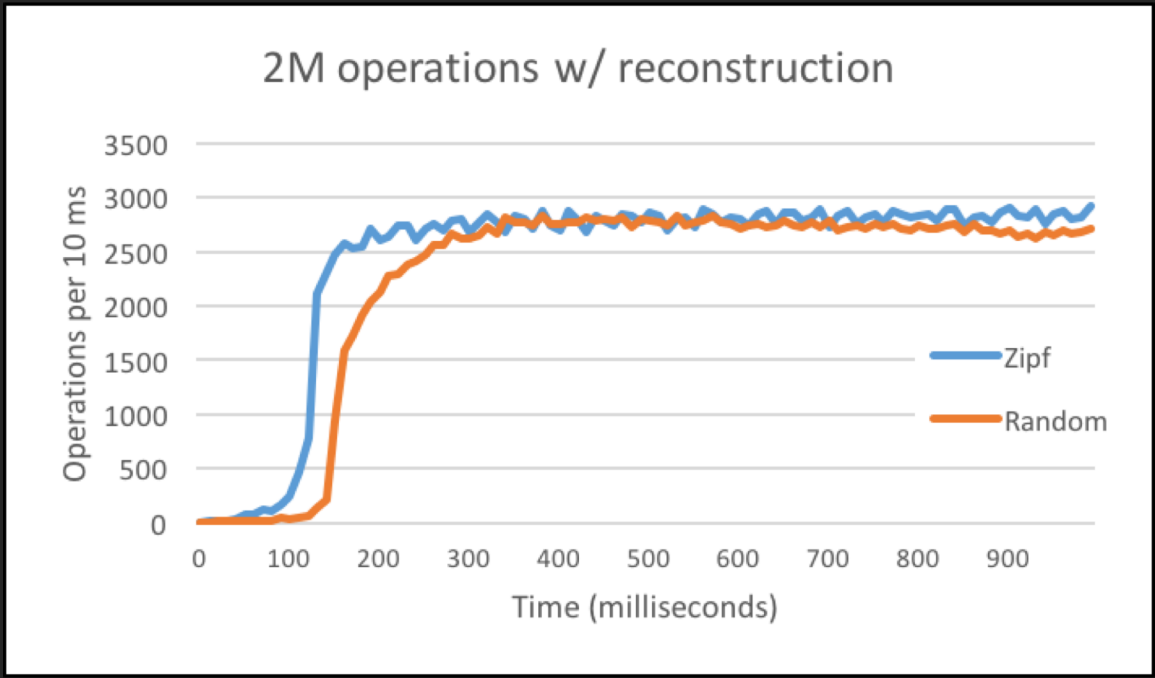
Startup Latency

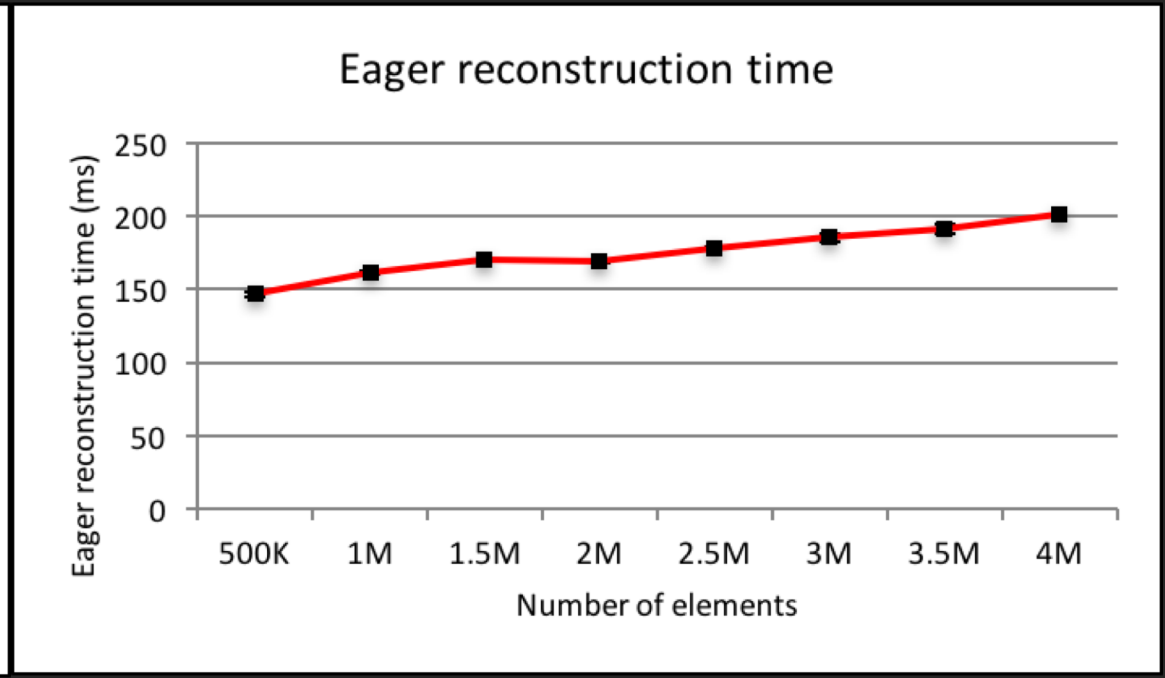


Lazy Reconstruction

- Use VM page protection to detect first access to object
- Reconstruct all objects on page

Performance Overhead





Key-value store implemented in Atlas. 1GB NVM heap. YCSB write-heavy workload (50% writes).

Conclusion

- NVM is persistent, directly accessible main memory
  - Well suited for very large in-memory data structures (DB, graphs, etc.)
- Programs must be aware of “NVMness” to allow recovery
- Caches in existing memory systems make consistency expensive
  - Key insight: memory consistently transfers entire cache line to NVM
- How much can we pack into a cache line?
  - More than you think
- But, don’t forget recovery!