Data Structure Engineering for Byte-Addressable Non-Volatile Memory

Ismail Oukid (TU Dresden & SAP SE), Wolfgang Lehner (TU Dresden)

SIGMOD Tutorial, Chicago, IL, May 14th, 2017

Tutorial Overview

Part 1: Motivation & Challenges

1. Motivation
2. NVRAM Programming Challenges
3. NVRAM Programming Models

Part 2: Data Structure Engineering for NVRAM

1. Persistent Memory Management
2. Data Structure Design
3. Fail-Safety Testing
4. NVRAM Performance Emulation

Tutorial Overview

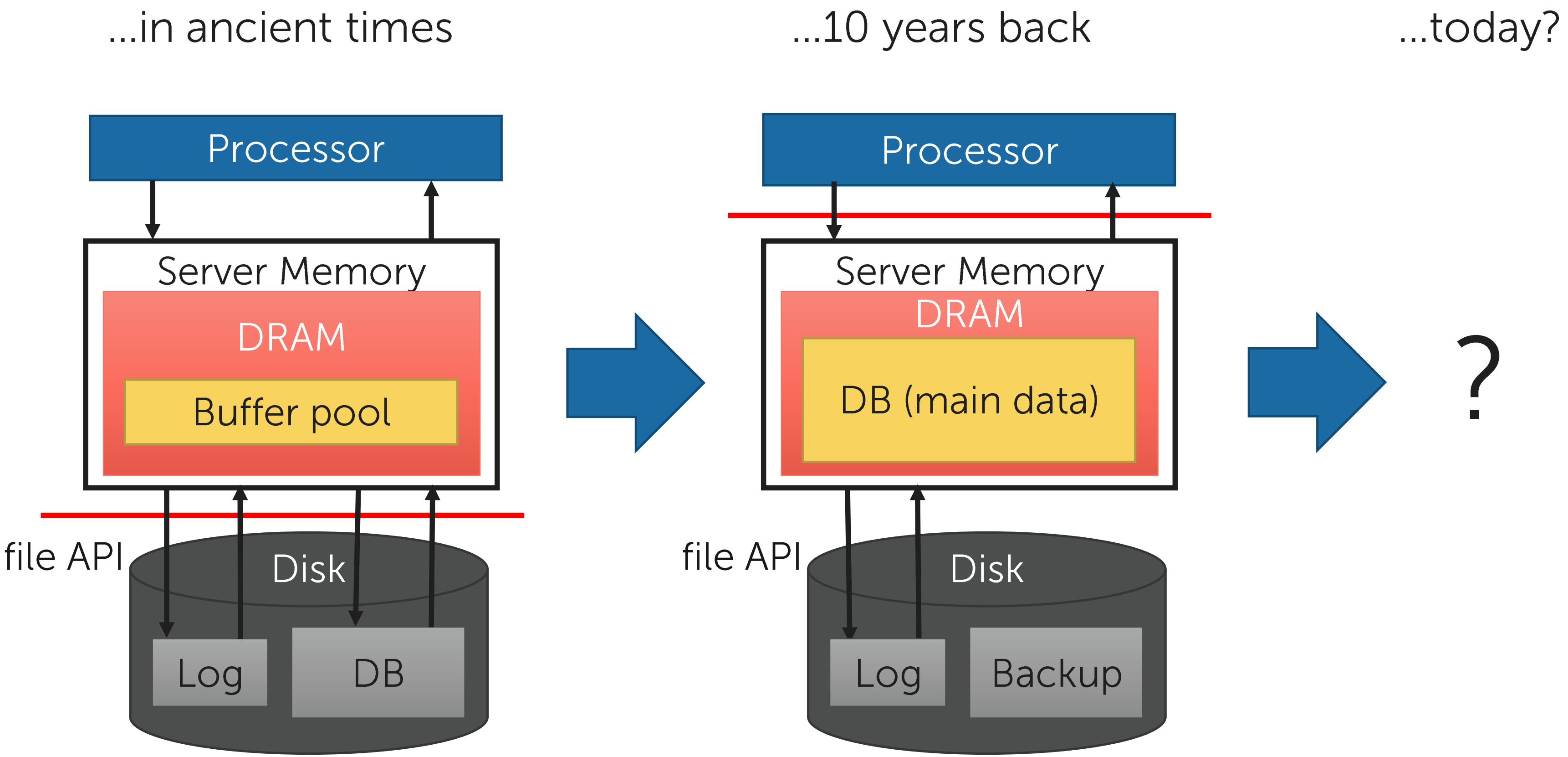
Part 1: Motivation & Challenges

1. Motivation
2. NVRAM Programming Challenges
3. NVRAM Programming Models

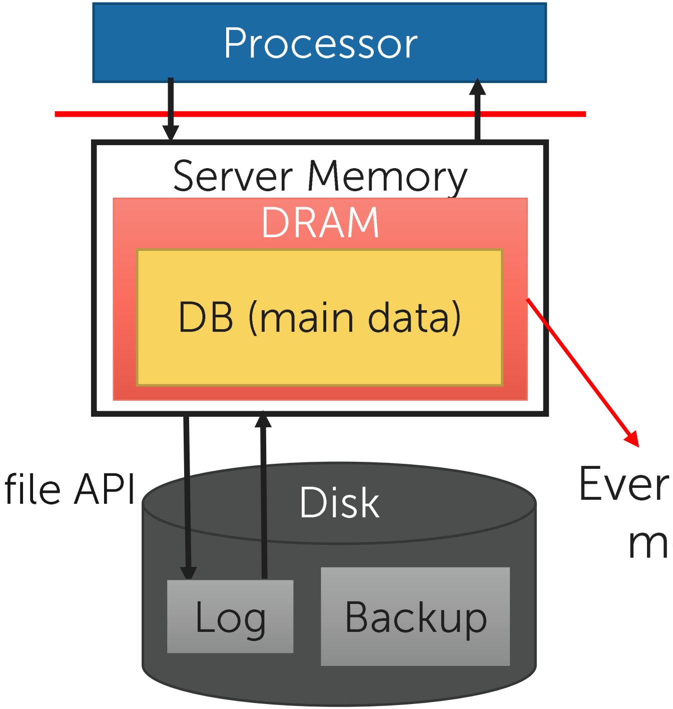
Part 2: Data Structure Engineering for NVRAM

1. Persistent Memory Management
2. Data Structure Design
3. Fail-Safety Testing
4. NVRAM Performance Emulation

From Disk to Main Memory



From Disk to Main Memory



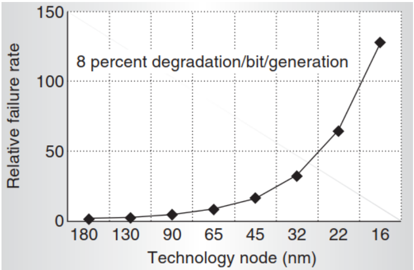
Intrinsically hard to further increase DRAM's density

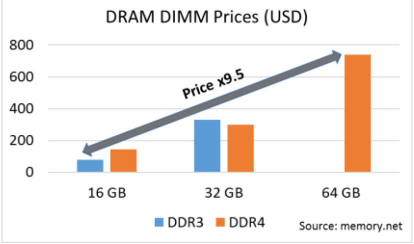
Cost per GB does not scale
→ 9,5x price for 4x capacity

Ever-increasing need for more main memory

Core count increasing faster than DIMM capacity

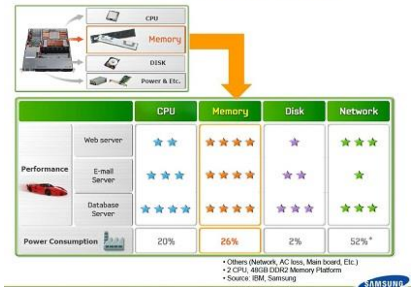
DRAM is hitting its scalability limits





Source: memory.net

Importance of Memory in Power


CPU
Memory
DISK
Power & Etc.
SAMSUNG

		CPU	Memory	Disk	Network
Performance	Web server	★★	★★★★	★	★★★
	E-mail Server	★★★	★★★★	★★	★
	Database Server	★★★★	★★★★	★★★	★★★
Power Consumption		20%	26%	2%	52%*

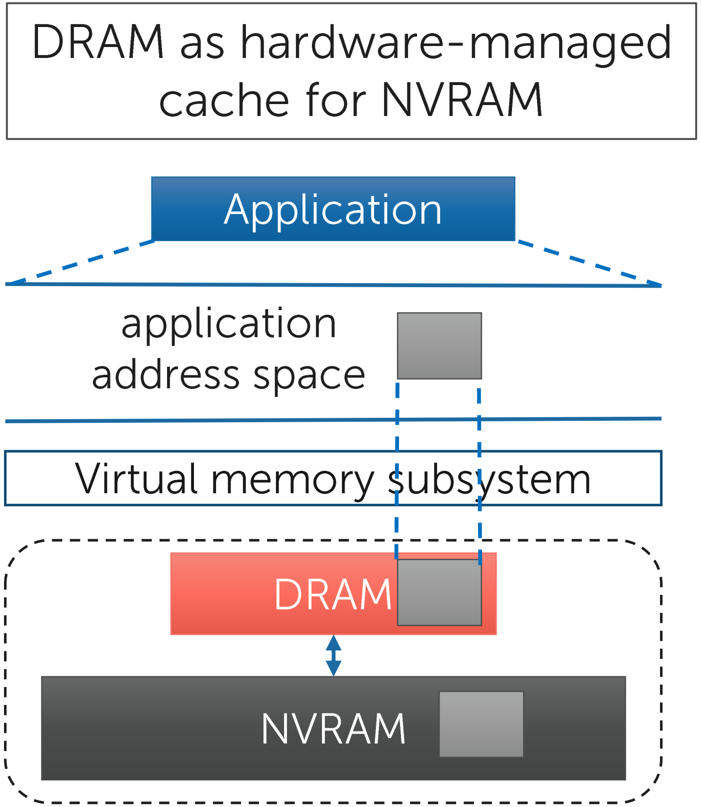
- Others (Network, AC loss, Main board, Etc.)
- 2 CPU, 48GB DDR2 Memory Platform
- Source: IBM, Samsung

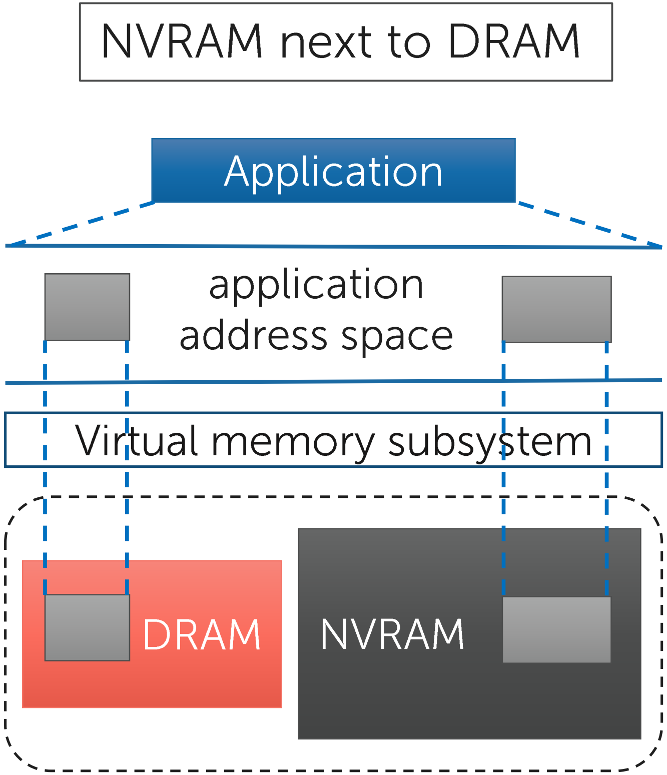
NVRAM for Database Systems?

1) Price	Cheaper than DRAM???
2) Capacity/error	Higher capacity (3 TB per socket for first-gen 3D XPoint)
3) Energy consumption	Significantly more energy efficient
4) Non-volatile	May serve as disk replacement
5) Byte adressable	Directly work on persistent version
6) Higher/asymmetric latency	Writes noticeably slower than reads

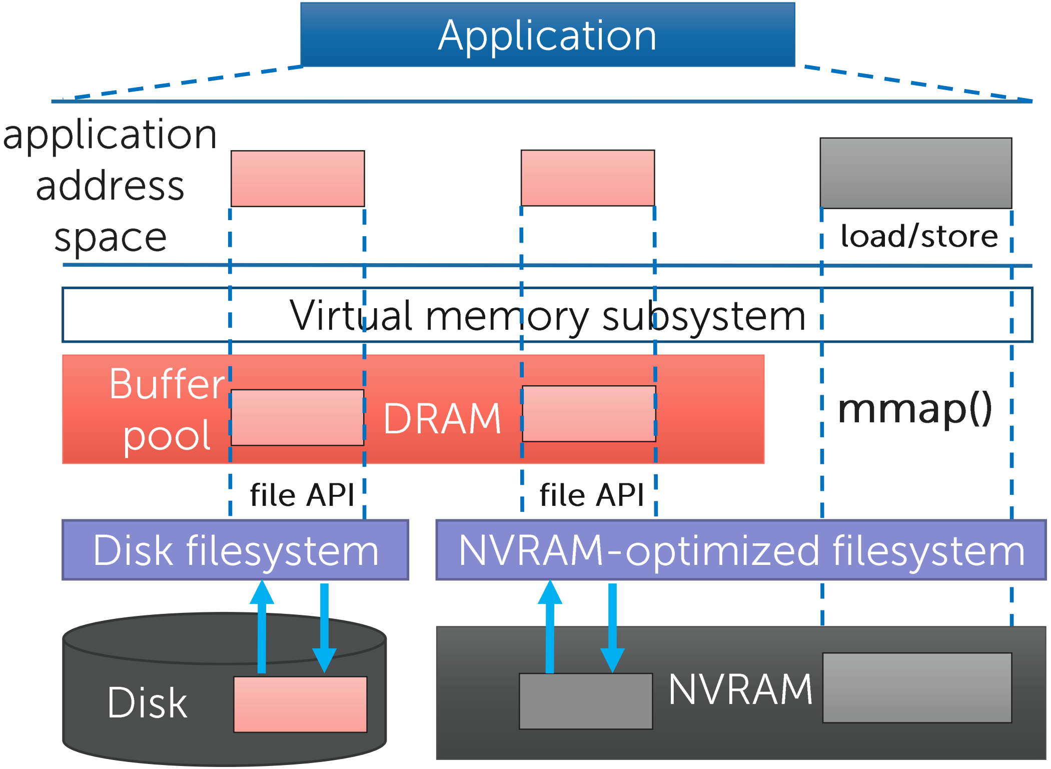
NVRAM as a promising technology

NVRAM as Transient Main Memory


DRAM as hardware-managed cache for NVRAM
Application
application address space
Virtual memory subsystem
DRAM
NVRAM



NVRAM as Persistent Main Memory



- SNIA recommends to access NVRAM via file mmap
- An NVRAM-optimized filesystem provides zero-copy mmap, bypassing the OS page cache

→ Several filesystem proposals: NOVA, PMFS, SCMFS, etc.

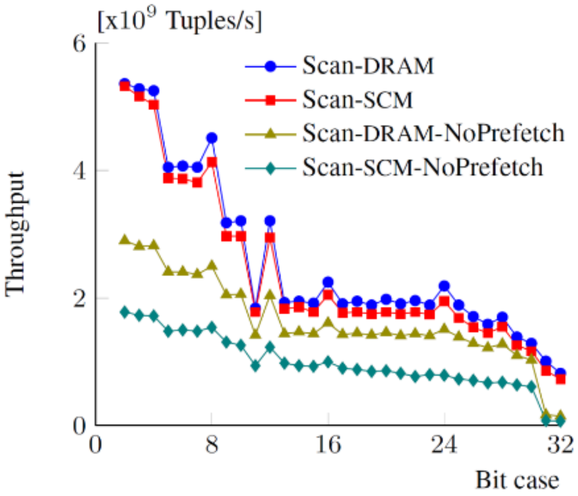
→ Linux ext4 and xfs already provide Direct Access support

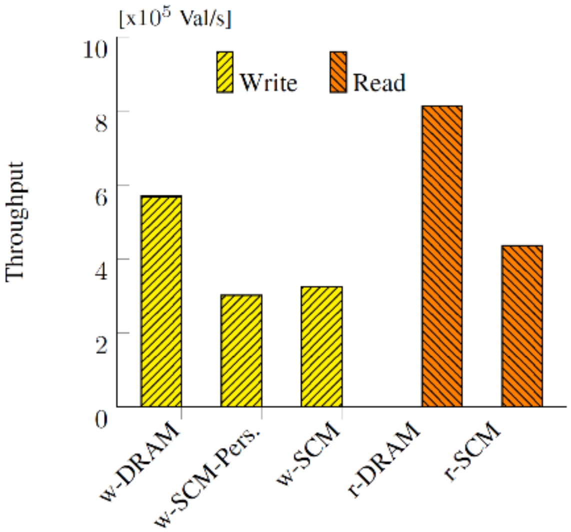
NVRAM may become a universal memory

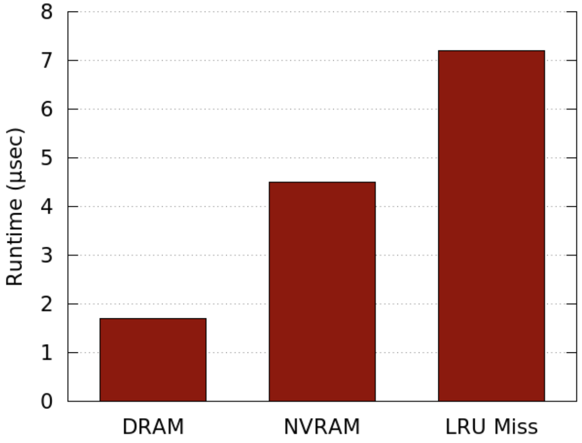
NVRAM Performance Implications

sequential vs. random access pattern

DRAM as NVRAM cache







Balance of DRAM and NVRAM required

Tutorial Overview

Part 1: Motivation & Challenges

1. Motivation
2. NVRAM Programming Challenges
3. NVRAM Programming Models

Part 2: Data Structure Engineering for NVRAM

1. Persistent Memory Management
2. Data Structure Design
3. Fail-Safety Testing
4. NVRAM Performance Emulation

Data Durability

Little control over when data is persisted

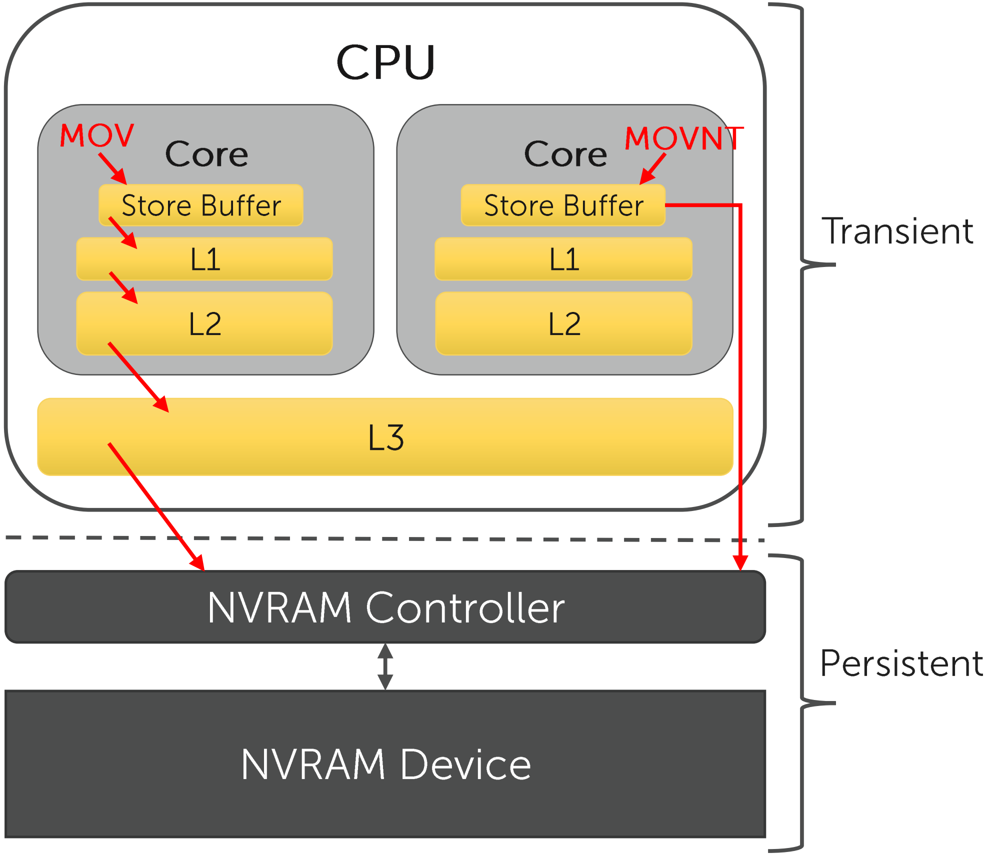
- CPU Cache eviction policy
- Memory reordering

Enforce order & durability of stores

- CLFLUSH, CLFLUSHOPT, CLWB
- MFENCE, SFENCE, LFENCE
- Non-temporal stores (MOVNT)

New primitives are being researched

- e.g., HOPS and its OFENCE and DFENCE barriers



Persistence Primitives

Persistence Primitive	Behavior	Ordering Constraints
CLFLUSH	evicts a cache line and writes its content to memory	Ordered with writes → contains implicit preceding and succeeding fences
CLFLUSHOPT	evicts a cache line and writes its content to memory	Ordered with SFENCE but not with writes. Enables better concurrency.
CLWB	writes back a cache line without invalidating it	Ordered with SFENCE but not with writes. Enables better concurrency.
MOVNT	write that bypasses the cache	NT writes can be reordered. Ordered with SFENCE, which drains NT writes from the store buffer directly to memory

Ordering Primitive	Guarantee
SFENCE	all preceding **store** instructions have been executed
MFENCE	all preceding **load and store** instructions have been executed
LFENCE	all preceding **load** instructions have been executed

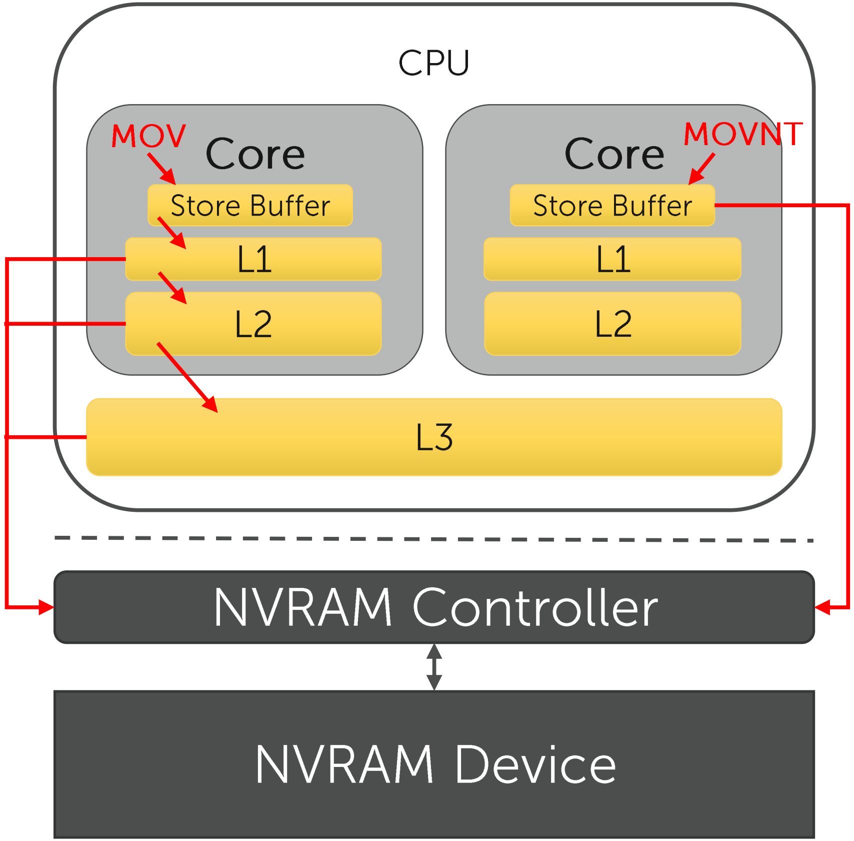
Source: Intel® 64 and IA-32 architectures software developer's manual

Data Durability

Ensure preceding writes made it to the store buffer → guarantee that the latest data is flushed

SFENCE + CLWB + SFENCE

Ensure CLWB finishes executing



SFENCE

Ensure the NT store buffer is drained to NVRAM

Data Durability: Example

Simplified array append operation

```
void push_back(int val){
    m_array[m_size] = val;
    sfence();
    clwb(&m_array[m_size]);
    sfence();
    m_size++;
    sfence();
    clwb(&m_size);
    sfence();
}
myArray.push_back(2017);
```

What is in NVRAM after the insertion?

m_size	m_array				
0					✗
1					✗ Corrupt!
0	2017				✗
1	2017				✓

Need to enforce write ordering and durability at cache-line granularity

Partial Writes

Persist = sfence + clwb + sfence

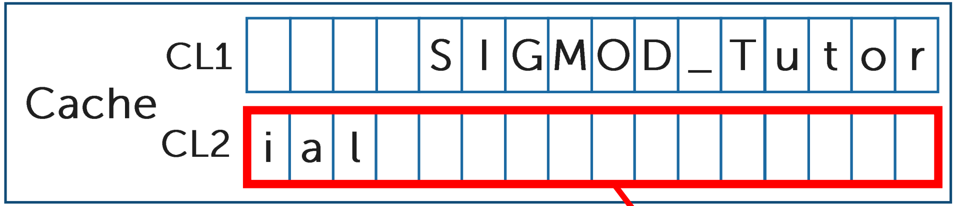
p-atomic store → executes in a one CPU cycle

Currently only 8-Byte stores are p-atomic on Intel x86

```
strcpy(ptr, “SIGMOD Tutorial”);
persist(ptr, 15);
flag = true;
persist(&flag);
```

What is in NVRAM?

1. “”
2. “SIGM”
3. “SIGMOD T”
4. “SIGMOD Tutor”
5. “SIGMOD Tutorial“
6. “\0\0\0\0\0\0\0\0\0\0\0\0ial”



Cache

CL1					S	I	G	M	O	D	_	T	u	t	o	r
CL2	i	a	l													

CL2 evicted before CL1, e.g., due to a context switch

Need software-built p-atomicity for writes > 8 bytes

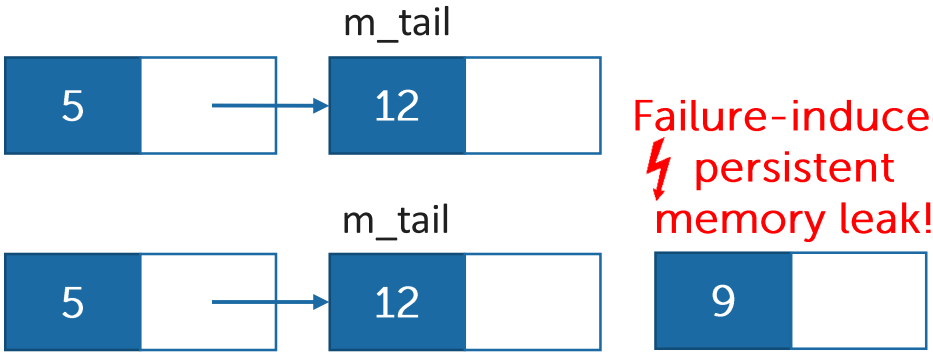
Persistent Memory Leaks

New class of memory leaks resulting from failures

Example: crash during a linked-list insertion

```
void append(int val){
    node *newNode = new node();
    newNode->value = val;
    persist(&(newNode->value));
  ⚡ m_tail->next = newNode;
    persist(m_tail);
    m_tail = newNode;
    persist(&m_tail);
}
List.append(9);
```

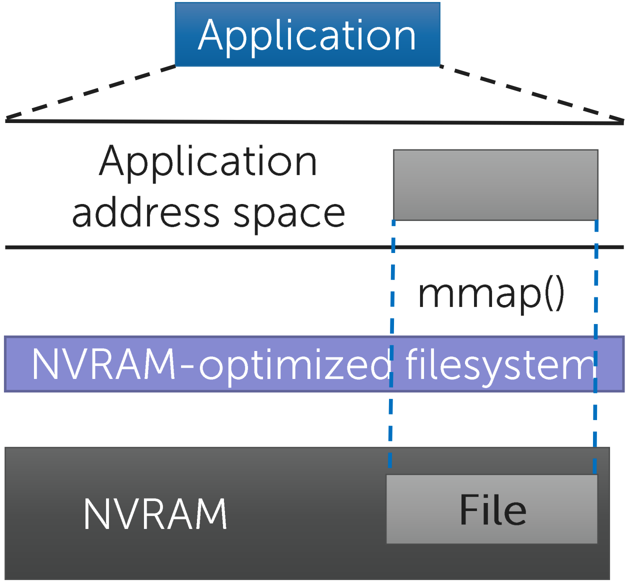
Persistent allocation



Failure-induced ⚡ persistent memory leak!

Avoiding memory leaks is a requirement

Data Recovery



Address space lost upon restart
→ stored virtual pointers become invalid

Filesystem provides a naming scheme

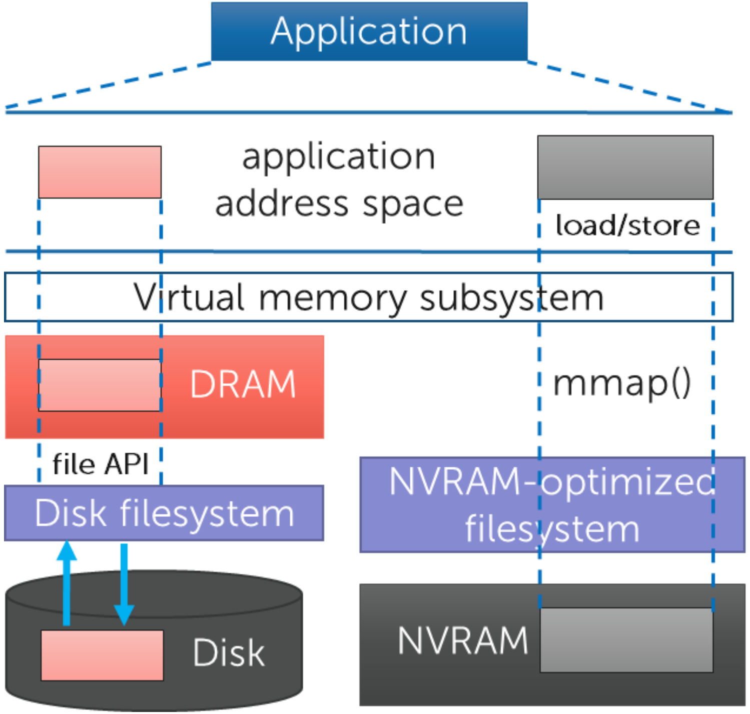
One file per object not realistic
→ How to recover objects?

Need persistent, recoverable NVRAM addressing scheme

Testing of NVRAM-Based Software

Traditional storage media accessed via DRAM → Data corruption risks minimized

Corruption happens first in DRAM → catch the corruption before it propagates to disk



NVRAM directly exposed to the user space → **more corruption risks**

Dangling pointer → Persistent data corruption

Missing or misplaced persistence primitives; wrong store order, etc.

Need testing and validation tools for NVRAM-based software

Summary

NVRAM programming challenges

- ➢Data durability
- ➢Partial writes
- ➢Persistent memory leaks
- ➢Data recovery
- ➢Testing of NVRAM-based software

Need new programming models that address these challenges

Tutorial Overview

Dresden Database Systems Group

Part 1: Motivation & Challenges

1. Motivation
2. NVRAM Programming Challenges
3. **NVRAM Programming Models**

Part 2: Data Structure Engineering for NVRAM

1. Persistent Memory Management
2. Data Structure Design
3. Fail-Safety Testing
4. NVRAM Performance Emulation

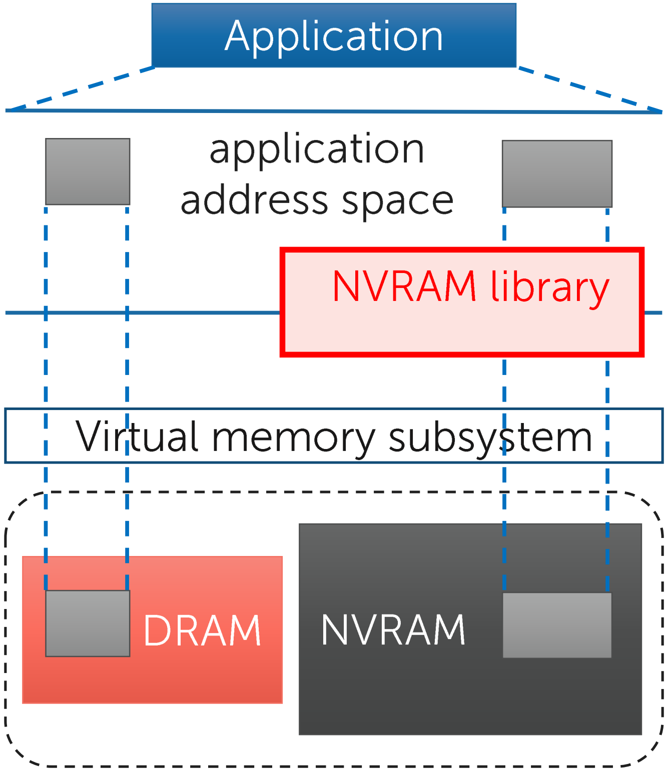
NVRAM Programming Models

We look at the following NVRAM programming challenges:

1. How to provide a recoverable addressing scheme?

2. How to avoid persistent memory leaks?

3. How to ensure data consistency?

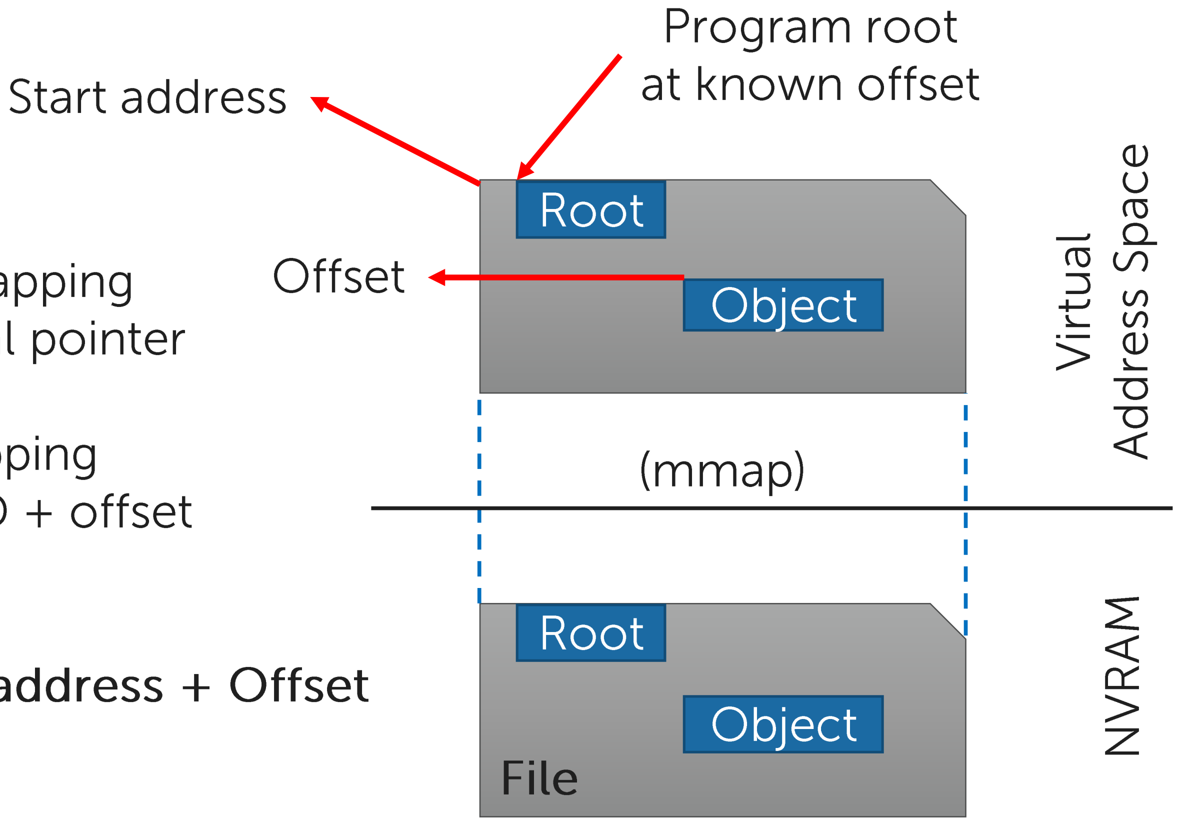


Recoverable Addressing Scheme

Two alternatives

- Fixed-address memory-mapping
  Persistent pointer → virtual pointer

- Unrestricted memory-mapping
  Persistent pointer → file ID + offset



Volatile pointer = File start address + Offset

Recoverable Addressing Scheme

Fixed-address memory-mapping

Pros:

- Familiar interface
- No runtime overhead

Cons:

- Fixed address is a security issue
- Can unmap existing mappings

Unrestricted memory-mapping

Pros:

- Safe, easy-to-implement, and portable approach

Cons:

- Potential overhead for converting to regular pointer

Unrestricted memory-mapping the safest way to go

Preventing Memory Leaks

```
pptr = allocate(size);
persist(&pptr);
```

→ Traditional interface has a "blind spot"

Three alternatives

- Reference passing

  → `allocate(PPtr &pptr, size_t allocSize)`

  `pptr` is owned by the data structure

- Transactional logging

  → Wrap operation involving allocation within fail-atomic transaction

  `BEGIN_TX {pptr = allocate(size); persist(&pptr);} END_TX`

- Offline garbage collection

  → Scan allocated blocks upon recovery to detect memory leaks

Preventing Memory Leaks

Reference Passing

Pros:

- Explicit memory management
- No runtime overhead

Cons:

- Data structure must be aware of memory leaks

Transactional Logging

Pros:

- Data structure can be leak-oblivious

Cons:

- Runtime overhead due to write-ahead log

Offline Garbage Collection

Pros:

- Catch existing memory leaks upon restart
- No runtime overhead

Cons:

- Restricts programming language
- Slow recovery

Reference passing closer to becoming the standard

Consistency Handling

Transactional Model

Provide durable transaction semantics for NVRAM programming

```
void push_back(int val){
    TXBEGIN {
        m_array[m_size] = val;
        m_size++;
    } TXEND
}
```

At least 4 writes

Lightweight Primitives

Provide basic functionality, e.g., memory allocation, leak avoidance etc.

```
void push_back(int val){
      m_array[m_size] = val;
      persist(&m_array[m_size]);
      m_size++;
      persist(&m_size);
}
```

Only 2 writes

Consistency Handling

Transactional Model

Pros:

- Easy to use and to reason about

Cons:

- Overhead due to systematic logging
- Low-level optimizations not possible

Lightweight Primitives

Pros:

- Low-level optimizations possible

Cons:

- Programmer must reason about the application state
  → Harder to use and error prone

High Performance → Lightweight Primitives

Existing NVRAM Libraries

PPtr → Persistent Pointer

Approach	Consistency Handling	Addressing Scheme	Leak Prevention	Compiler support	Source
Mnemosyne	Transactional & Lightweight primitives	**PPtr**: file offset **Recovery**: new mmap in reserved address space	Reference passing | Transactional logging	Yes	ASPLOS'11
NV-Heaps	Transactional	**PPtr**: file Id + offset **Recovery**: new mmap	Transactional logging	No	ASPLOS'11
Intel NVML	Transactional & Lightweight primitives	**PPtr**: file Id + offset **Recovery**: new mmap	Reference passing | Transactional logging	No	http://pmem.io/
Atlas	Transactional (sections determined by locks)	**PPtr**: volatile pointer **Recovery**: fixed mmap	Transactional logging	Yes	OOPSLA'14
REWIND	Transactional	Undefined, hints → **PPtr**: volatile pointer **Recovery**: fixed mmap	Transactional logging	Yes	VLDB'15
PAllocator	Lightweight primitives	**PPtr**: file Id + offset **Recovery**: new mmap	Reference passing	No	To appear

Recommended starting point: NVML → rich, open source, actively developed

Tutorial Overview

Part 1: Motivation & Challenges

1. Motivation
2. NVRAM Programming Challenges
3. NVRAM Programming Models

Part 2: Data Structure Engineering for NVRAM

1. Persistent Memory Management
2. Data Structure Design
3. Fail-Safety Testing
4. NVRAM Performance Emulation

Tutorial Overview

Part 1: Motivation & Challenges

1. Motivation
2. NVRAM Programming Challenges
3. NVRAM Programming Models

Part 2: Data Structure Engineering for NVRAM

1. Persistent Memory Management
2. Data Structure Design
3. Fail-Safety Testing
4. NVRAM Performance Emulation

Persistent Memory Allocation for NVRAM

We explore the following design dimensions

- Allocation strategies
- Pool structure (single file vs. multiple files)
- Concurrency Handling
- Garbage collection
- Persistent Fragmentation

Summary of existing persistent memory allocators

We assume wear-leveling will be handled by hardware

Allocation Strategies

Three main strategies
→ One file per allocation
→ Segregated-fit for small blocks (e.g., `< 4 KB`)
→ Best-fit for medium and large blocks (e.g., `[4 KB, 16 MB)`)

One file per allocation not realistic…

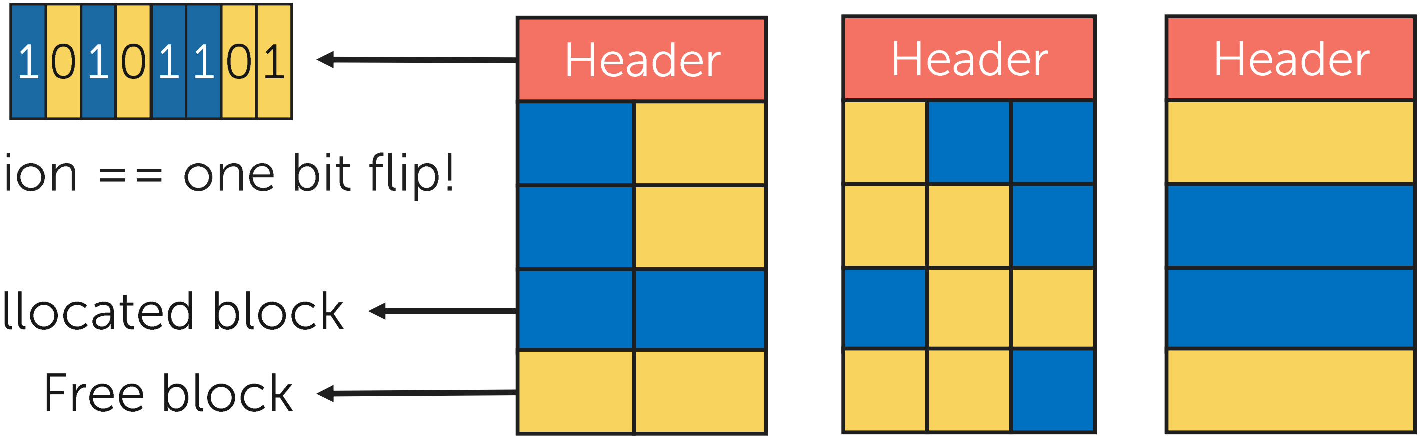
- Significant overhead and wasted memory for small blocks
- Filesystem might struggle to handle huge number of files

except for huge blocks!

→ Fragmentation handling pushed to filesystem

Segregated-Fit Allocation Strategy

Fixed-size memory chunk, e.g., 8 KB, divided into fixed-size blocks



Bitmap 1 0 1 0 1 1 0 1

One allocation == one bit flip!

Allocated block

Free block

Multiple class sizes

Reduced fragmentation with moderate number of class sizes

Not suitable for larger block allocations

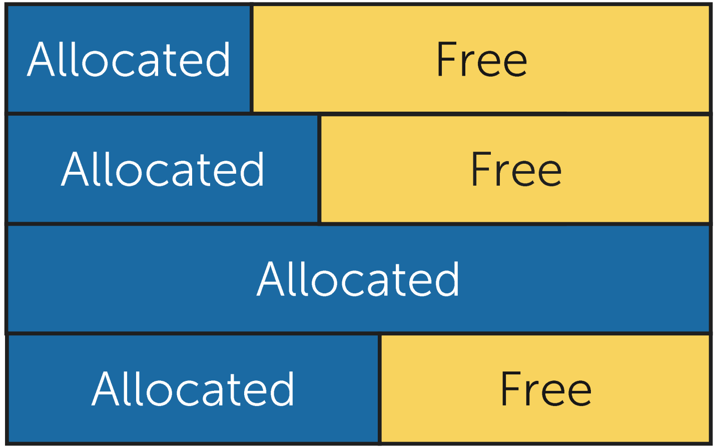
Best-Fit Allocation Strategy

Allocate multiple of a predetermined size (e.g., system page size)

Allocation Free blocks index sorted by block size

Coalescing Global block index sorted by block offset

Indexes can be transient and rebuilt during recovery


Allocated
Free
Allocated
Free
Allocated
Allocated
Free

Persistent memory pool

→ Suitable for large block allocation

→ Prone to fragmentation

Pool Structure: Single File Vs. Multiple Files

Pool as Single File

Pros

- 8-byte persistent pointers possible
- Easier to implement

Cons

- Hard to shrink
- Huge block allocation a problem
- Segregated-fit allocator must use best-fit allocator to create chunks

Pool as Multiple File

Pros

- Easier to grow and shrink
- Segregated-fit allocator can have dedicated files
- Easy, fragmentation-free huge allocation handling

Cons

- 16-byte persistent pointers

Multiple files better suited for database systems

Concurrency Handling

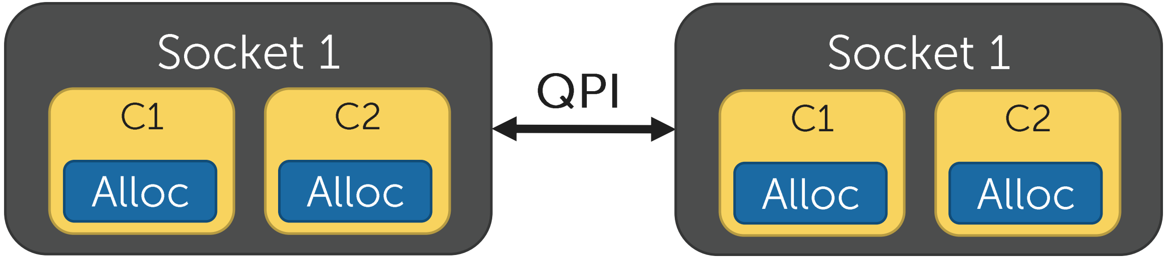
Thread-local allocation → One allocator object per thread

- The standard in general-purpose allocators
- Used for small block allocations
  → Local allocator requests chunks from global pool
- Need to be merged with global pool when thread terminates
- Does not scale under high concurrency
  → Frequent chunk requests to the global pool

Concurrency Handling

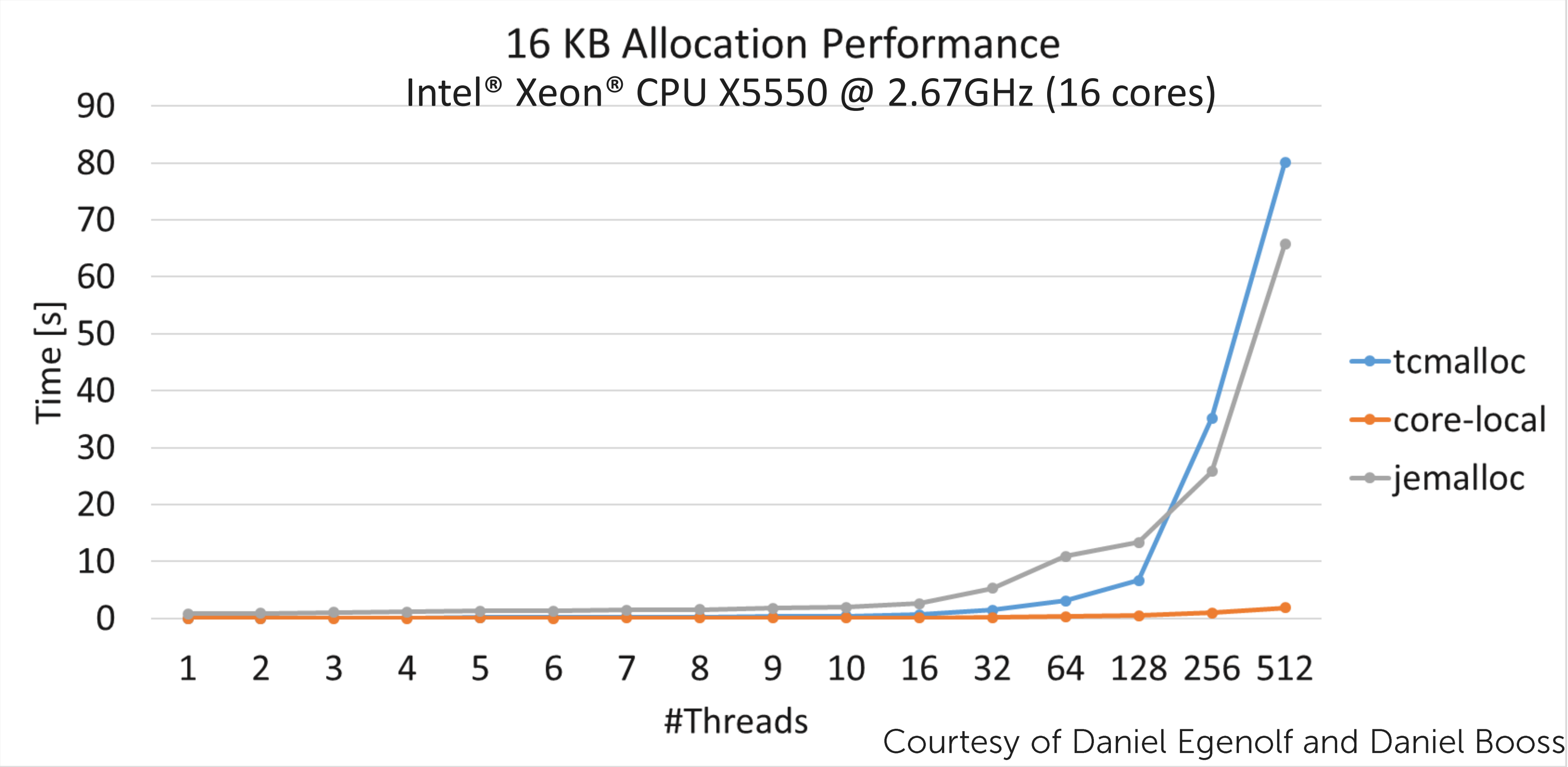
Core-local allocation → One allocator object per physical core

- Used in large-main-memory systems for both small and large blocks
  → Local allocators request large files from global pool
- Robust performance under high concurrency
  → Stable local allocators → Greedy



Core-local allocators better suited for database systems

Thread-local vs. Core-local



Courtesy of Daniel Egenolf and Daniel Booss

Garbage Collection

Reference counting

NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. ASPLOS'11

Deallocation calls the destructor, which might trigger recursive deallocations
→ Need to ensure fail-atomicity of recursive deallocations

Offline garbage collection

Makalu: Fast Recoverable Allocation of Non-volatile Memory. OOPSLA'16

1. Scan program object layout
2. Mark reachable blocks
3. Sweep unreached blocks

→ **Catch memory leaks that stem from programming errors**

Relax metadata persistence constraints → faster small-block allocations

Programming language constraints (e.g., no generic pointers)

Slow Recovery

Persistent Fragmentation

Restart is a last resort, but valid way of defragmenting volatile memory
→ does not apply to NVRAM

File system solutions do not apply to NVRAM

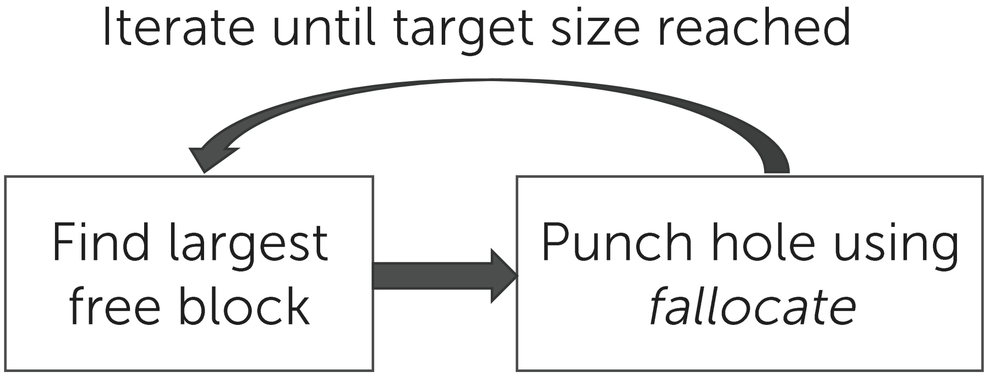
- File systems benefit from an additional indirection layer
- NVRAM is directly accessed with load/store instructions

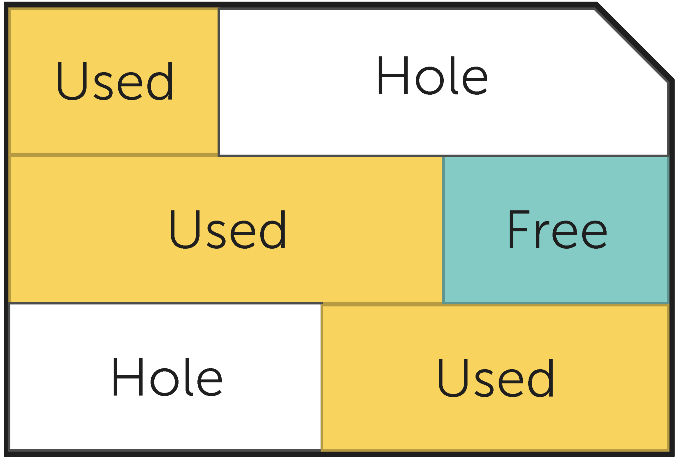
Need new defragmentation mechanisms

Defragmentation

Most file systems have support for sparse files

Defragmentation idea: Punch holes in free blocks


Iterate until target size reached
Find largest free block
Punch hole using *fallocate*



Must keep file size unchanged to maintain validity of offsets

Existing Persistent Memory Allocators

Allocator	Purpose	Pool structure	Allocation strategies	Concurrency handling	Garbage collection	Defragm-entation	Source
Mnemosyne	General	Multiple files	Segregated-fit + best-fit	Thread-local for small blocks	Yes	No	ASPLOS'11
NV-Heaps	General	Single file	Undefined	Thread-local	Yes	No	ASPLOS'11
nvm_malloc	General	Single file	Segregated-fit + best-fit	Thread-local for small blocks	No	No	ADMS'15
NVML	General	Single file	Segregated-fit + best-fit	Thread-local for small blocks	No	No	http://pmem.io/nvml/
Makalu	General	Single file	Segregated-fit + best-fit	Thread-local for small blocks	Yes (offline)	No	OOPSLA'16
PAllocator	Large systems	Multiple files	Segregated-fit + best-fit + file	Core-local	No	Yes	To appear

For completeness: NVMalloc and Walloc focus on wear-leveling

Salient differences in design decisions

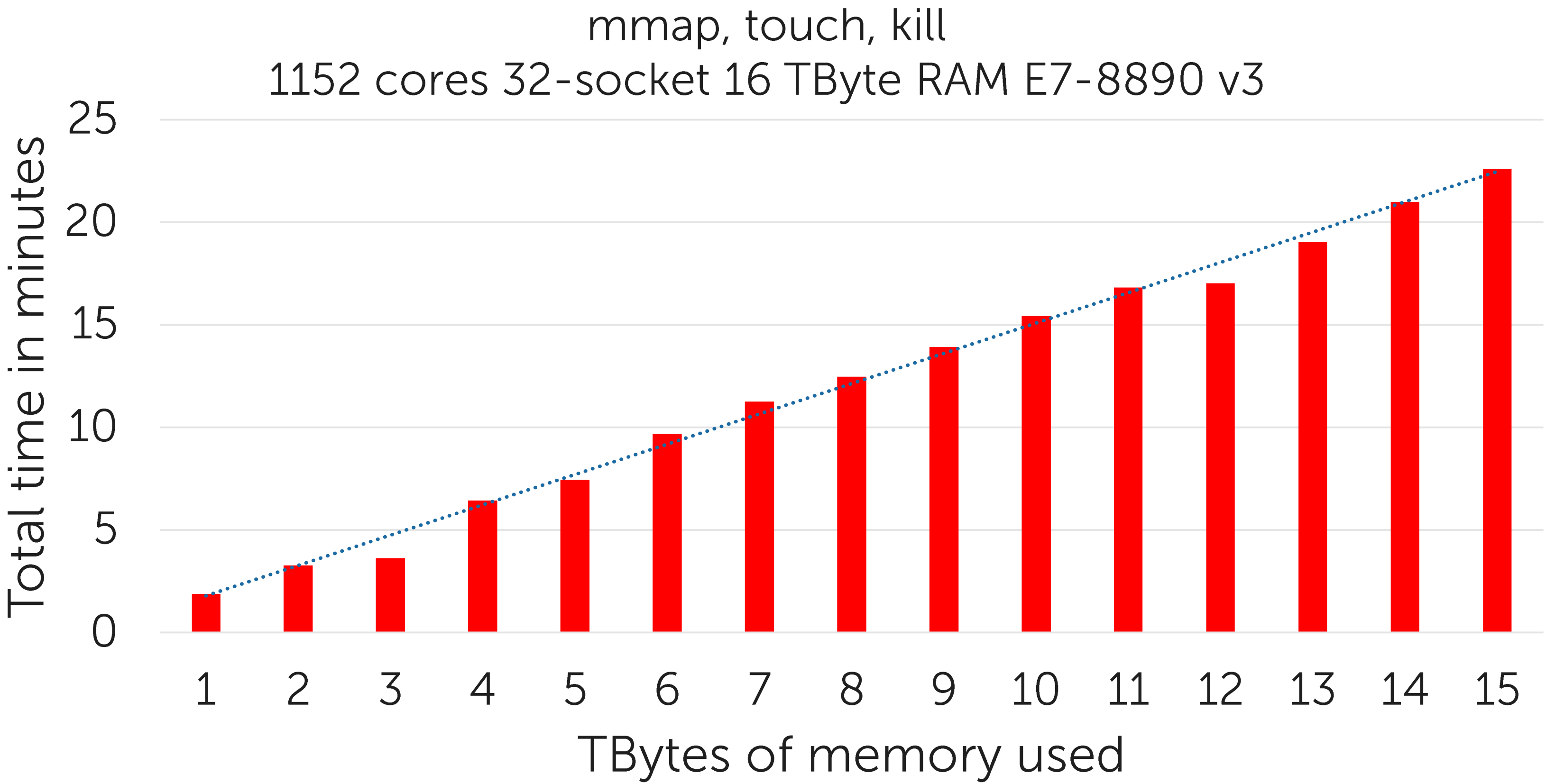
Discussion: Operating System Challenges

- Address space fragmentation
  - Only 128 Tbytes of virtual address space
  - NVRAM will push main memory capacity beyond 100 Tbytes

Newly extended to 128 Petabytes on Linux!

- Page Table (lack of) scalability
  - Memory mapping millions of files upon startup a challenge
  - Slow memory reclamation upon process termination

Duration of Process Termination



Courtesy of Robert Kettler and Daniel Booss

Tutorial Overview

Part 1: Motivation & Challenges

1. Motivation
2. NVRAM Programming Challenges
3. NVRAM Programming Models

Part 2: Data Structure Engineering for NVRAM

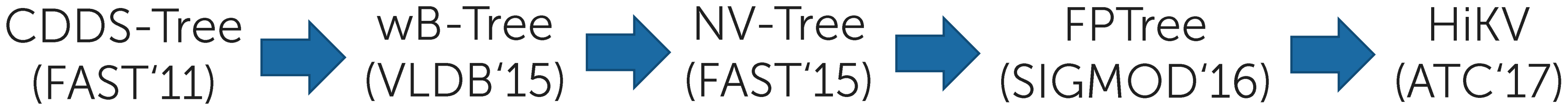
1. Persistent Memory Management
2. Data Structure Design
3. Fail-Safety Testing
4. NVRAM Performance Emulation

Data Structure Design for NVRAM

NVML includes many examples of data structure implementations
→ Linked-list, Hash table , B-Tree, KV Store

Literature focuses mostly on tree-based data structures
→ Fail-atomic updates
→ Reduce NVRAM writes

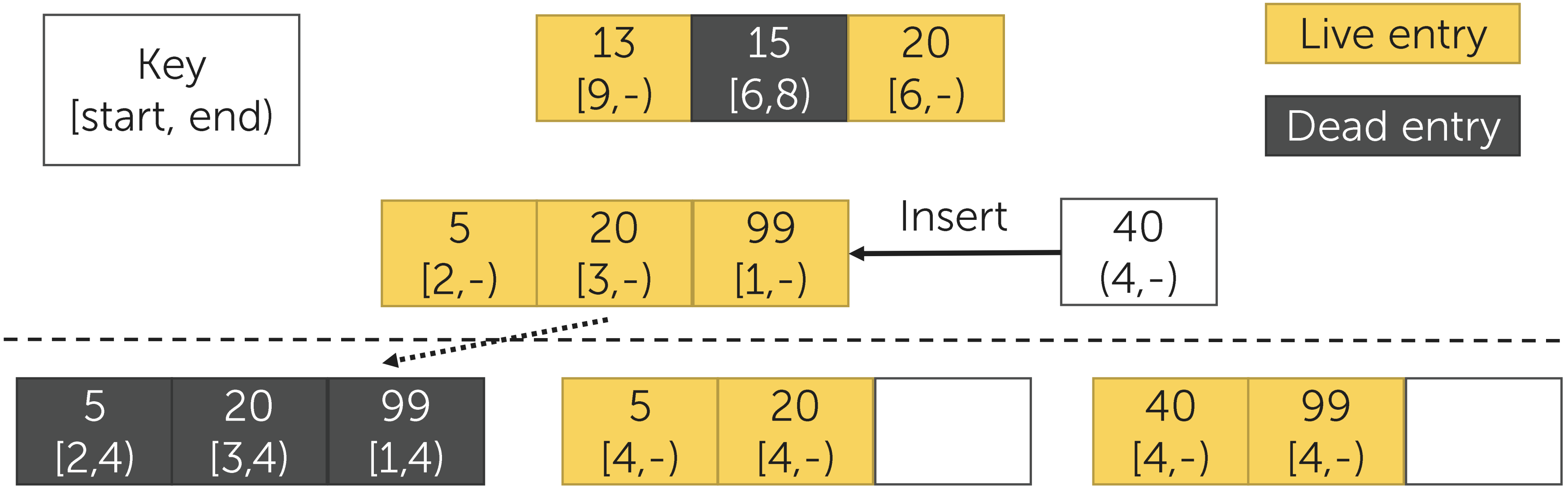
Overview



CDDS-Tree (FAST'11) → wB-Tree (VLDB'15) → NV-Tree (FAST'15) → FPTree (SIGMOD'16) → HiKV (ATC'17)

Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. FAST'11

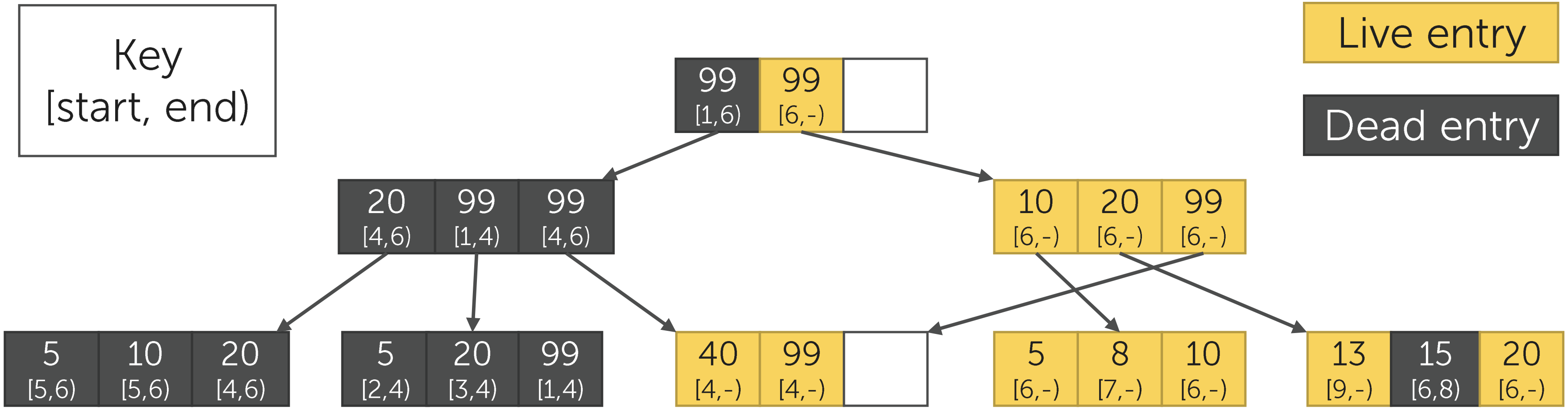
Use versioning to achieve p-atomicity



1. Set **end** timestamp of leaf entries

2. Create two new leaf nodes
3. P-atomically increment global timestamp

Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. FAST'11



Key
[start, end)

Live entry

Dead entry

99 [1,6) | 99 [6,-)

20 [4,6) | 99 [1,4) | 99 [4,6)

10 [6,-) | 20 [6,-) | 99 [6,-)

5 [5,6) | 10 [5,6) | 20 [4,6)

5 [2,4) | 20 [3,4) | 99 [1,4)

40 [4,-) | 99 [4,-)

5 [6,-) | 8 [7,-) | 10 [6,-)

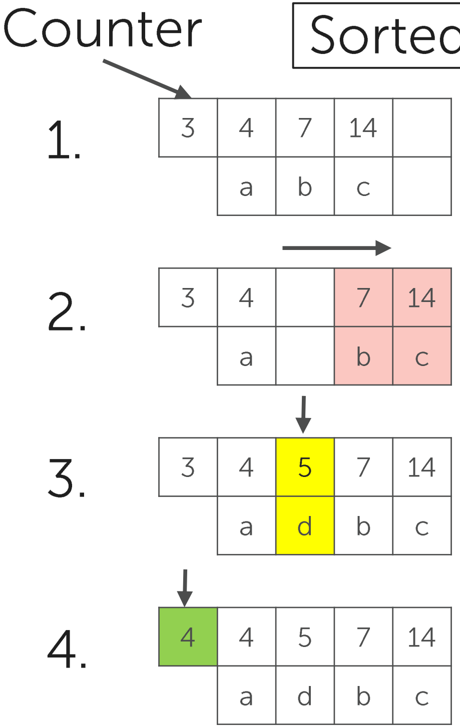
13 [9,-) | 15 [6,8) | 20 [6,-)

Recovery → undo operations based on global timestamp

Need garbage collection

Global timestamp counter is a contention point

Rethinking Database Algorithms for Phase Change Memory. CIDR'11



Sorted leaf

1.

3	4	7	14	
	a	b	c	

2.

3	4		7	14
	a		b	c

3.

3	4	5	7	14
	a	d	b	c

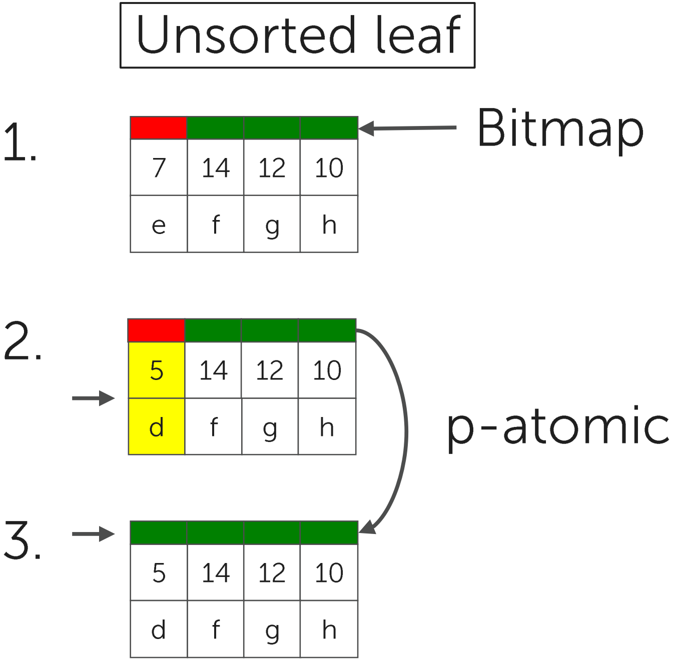
4.

4	4	5	7	14
	a	d	b	c

! Potential corruption

! Writes slower than reads

But...slower, sequential scan!



Unsorted leaf

1.

🟥	🟩	🟩	🟩
7	14	12	10
e	f	g	h

2.

🟥	🟩	🟩	🟩
5	14	12	10
d	f	g	h

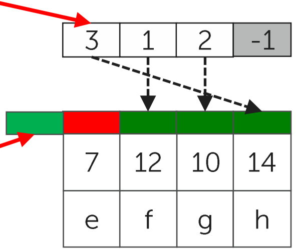
3.

🟩	🟩	🟩	🟩
5	14	12	10
d	f	g	h

p-atomicity + decreased number of writes

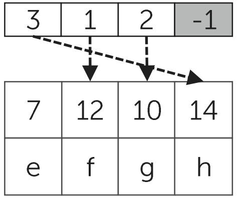
One byte per slot entry

Indirection slot array → enable binary search



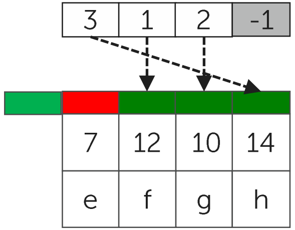
One bit reserved for slot array consistency

Slot array can be p-atomically updated up to 8 entries
→ We can do away with the bitmap



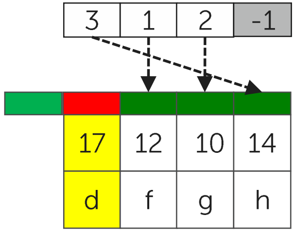
Insertion with Bitmap and Indirection Array

1.



3	1	2	-1
7	12	10	14
e	f	g	h

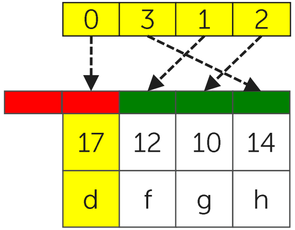
2.



3	1	2	-1
17	12	10	14
d	f	g	h

Find free slot and insert the record

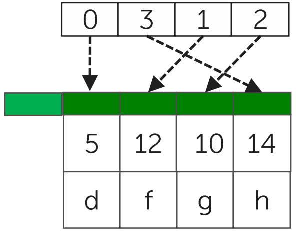
3.



0	3	1	2
17	12	10	14
d	f	g	h

Flag slot array as invalid, then update it

4.



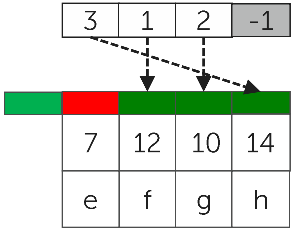
0	3	1	2
5	12	10	14
d	f	g	h

p-atomically set both new record and slot array as valid

Bitmap must be <= 8 bytes

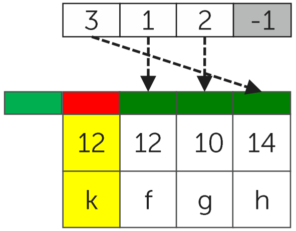
Out-of-Place Updates

1.



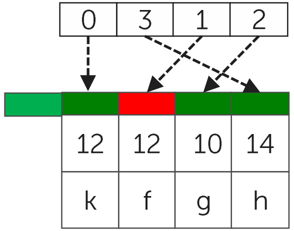
Update
(12, f) → (12, k)

2.



Find free slot and update record out-of-place

3.

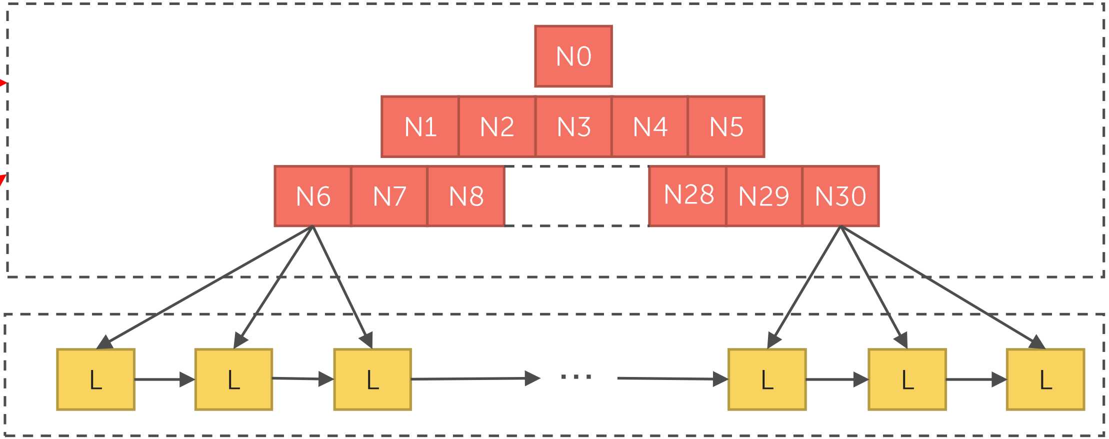


p-atomically flip validity of both old and new records

NV-Tree: Reducing Consistency Cost for NVM-based Single-Level Systems. FAST'15

Consistency of inner nodes relaxed

Selective consistency



→ Simpler algorithms

→ Less writes to NVRAM

Expensive rebuild of inner nodes when one last-level node is full

→ Cannot handle skew

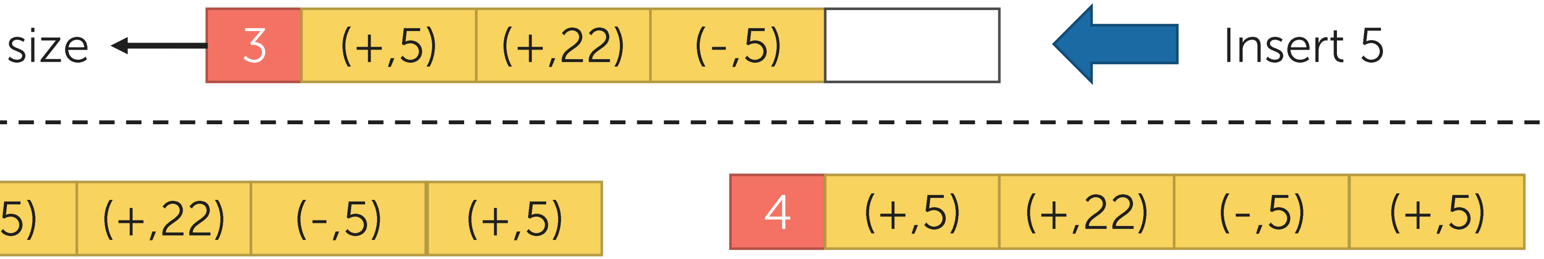
→ Large memory consumption

NV-Tree: Reducing Consistency Cost for NVM-based Single-Level Systems. FAST'15

Dresden Database Systems Group

Append-only leaf nodes

Record → `[flag(-/+), key, value]`



1. Append new record with + flag

2. p-atomically increment counter

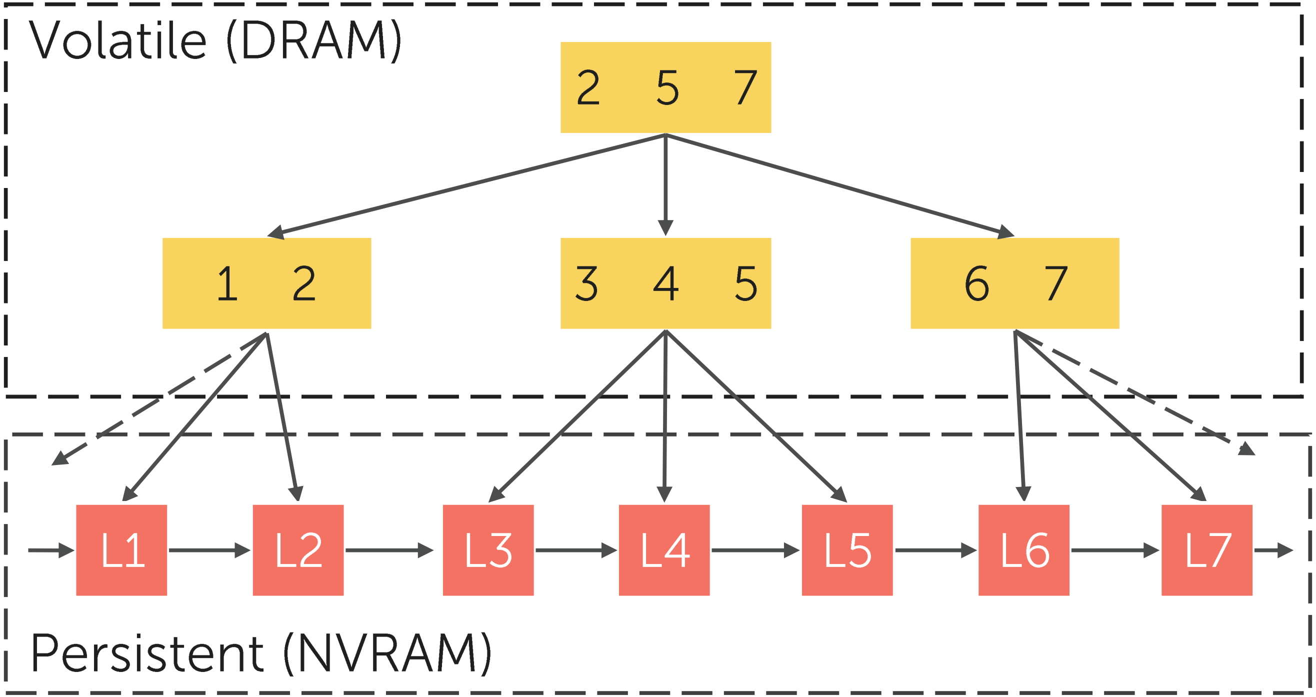
Unsorted leaf nodes → expensive linear scan

FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory.

SIGMOD'16

Inner nodes in DRAM for better performance (~1-3% of data)

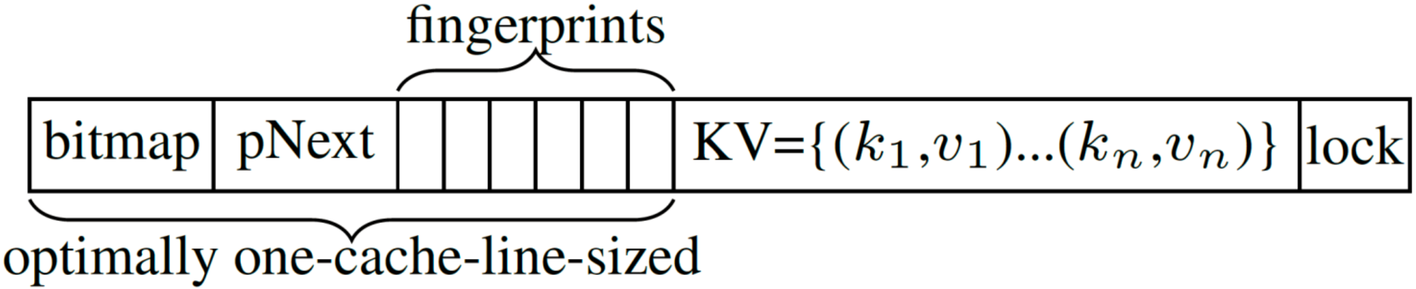
Leaves in NVRAM to ensure durability



Recovery is up to 100x faster than a full rebuild

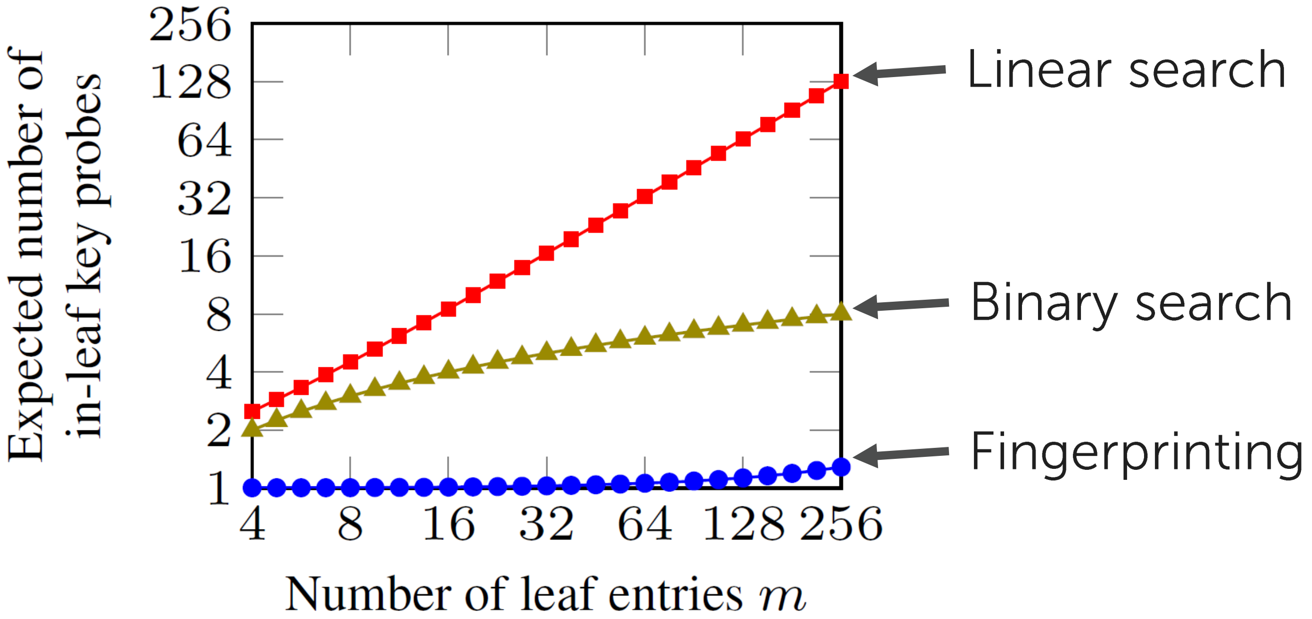
Leaf Node Layout

A fingerprint is a 1-byte hash of a key


fingerprints
bitmap
pNext
KV={(k_1,v_1)...(k_n,v_n)}
lock
optimally one-cache-line-sized

Fingerprints limit the number of key probes

Fingerprinting

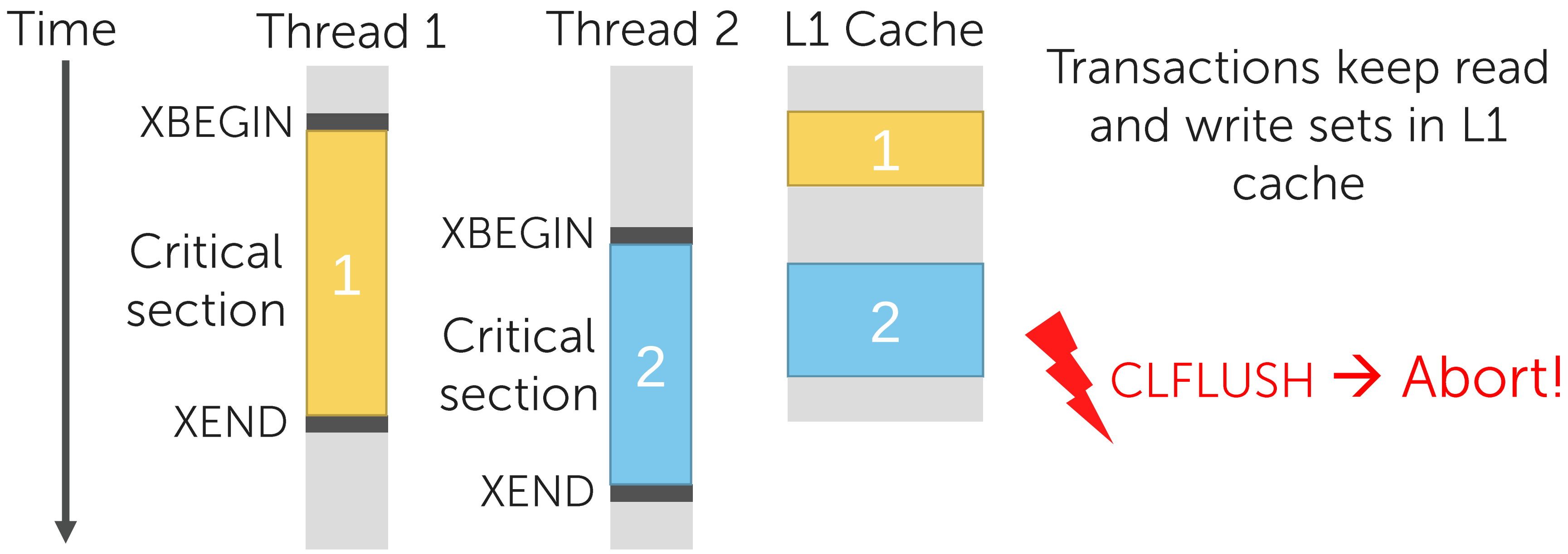


Fingerprinting limits the number of probed keys to one for leaf sizes up to 512 entries

Range scan still requires full leaf scans

Hardware Transactional Memory

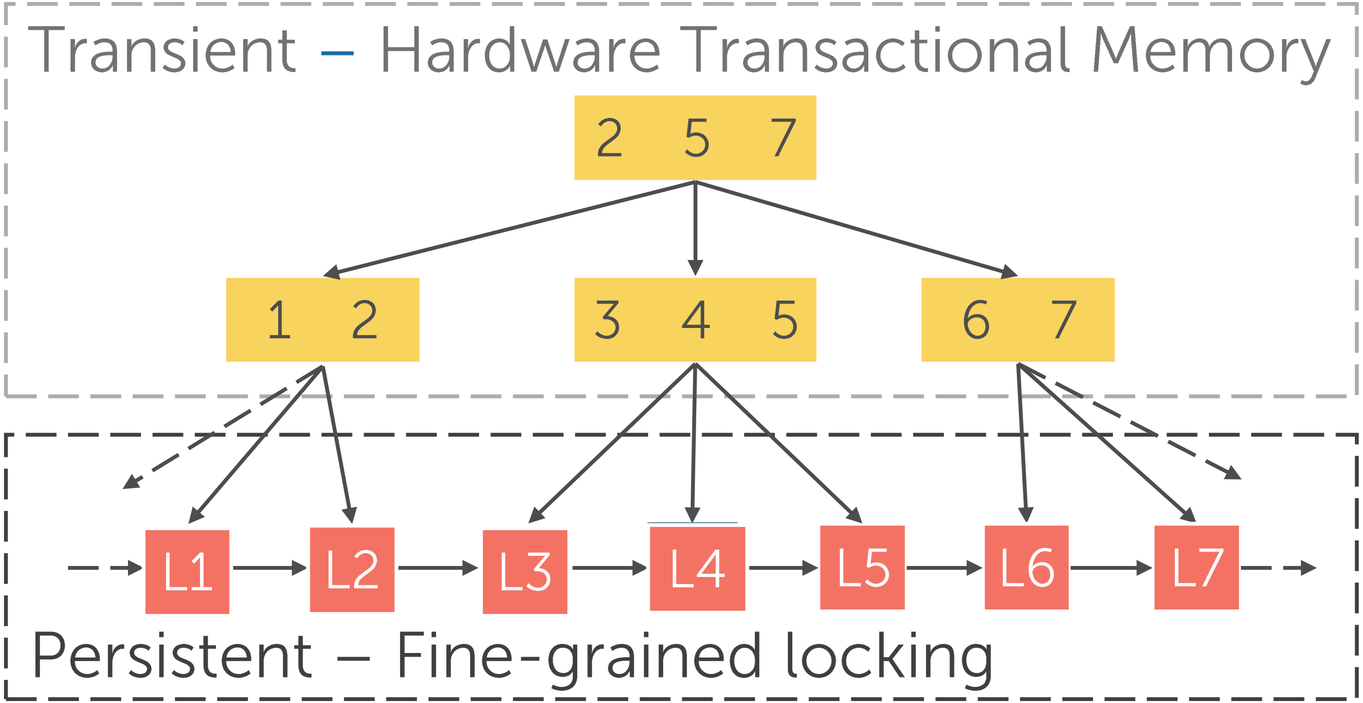
Allows optimistic execution of critical sections



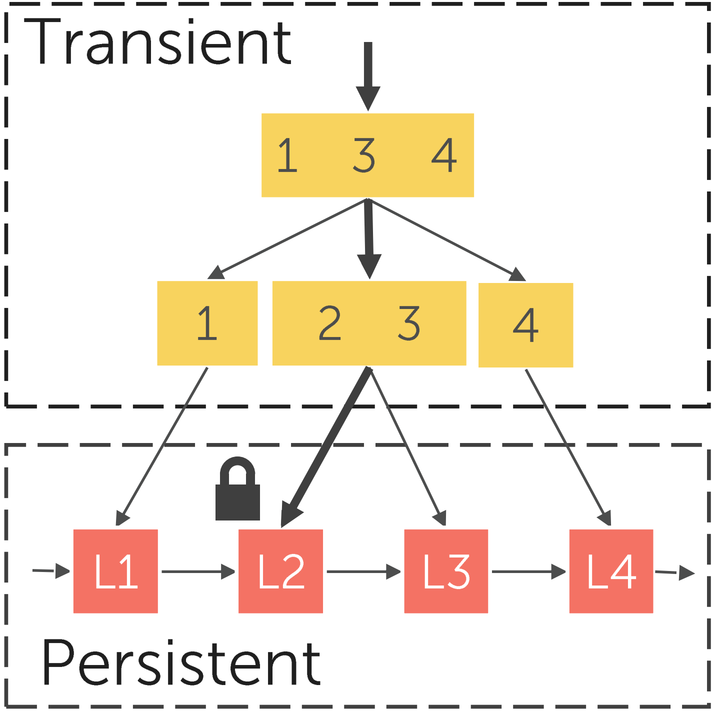
Transactions keep read and write sets in L1 cache

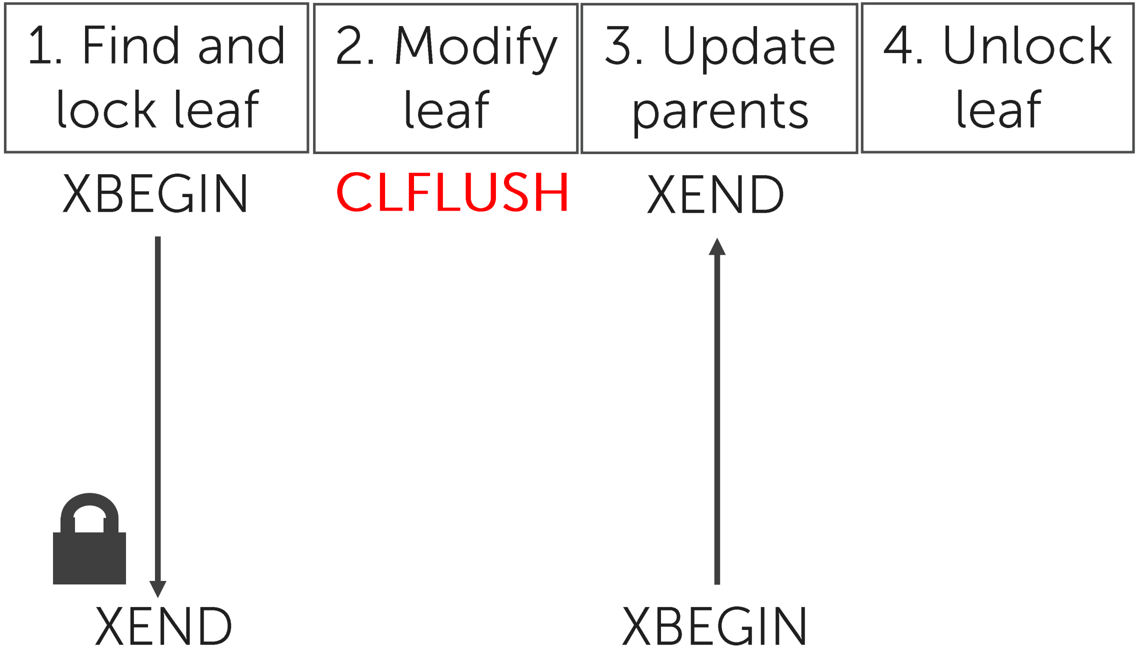
There is an apparent incompatibility between HTM and NVRAM

Selective Concurrency


Transient – Hardware Transactional Memory
2 5 7
1 2
3 4 5
6 7
L1
L2
L3
L4
L5
L6
L7
Persistent – Fine-grained locking

Selective Concurrency: Insertion





Selective Concurrency solves the incompatibility of HTM and SCM

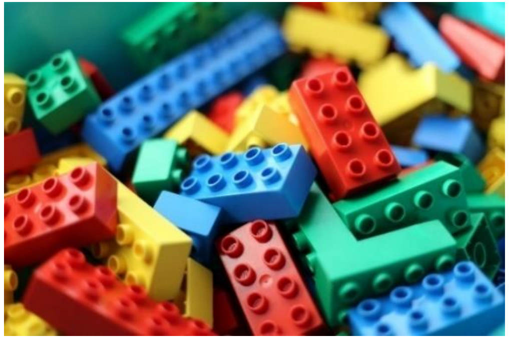
Persistent Data Structures: Summary

Achieving p-atomicity

- Versioning
- Append-only
- Out-of-place updates (e.g., using bitmaps)

Reduce NVRAM accesses

- Selective persistence
- Indirection slot array
- Fingerprints



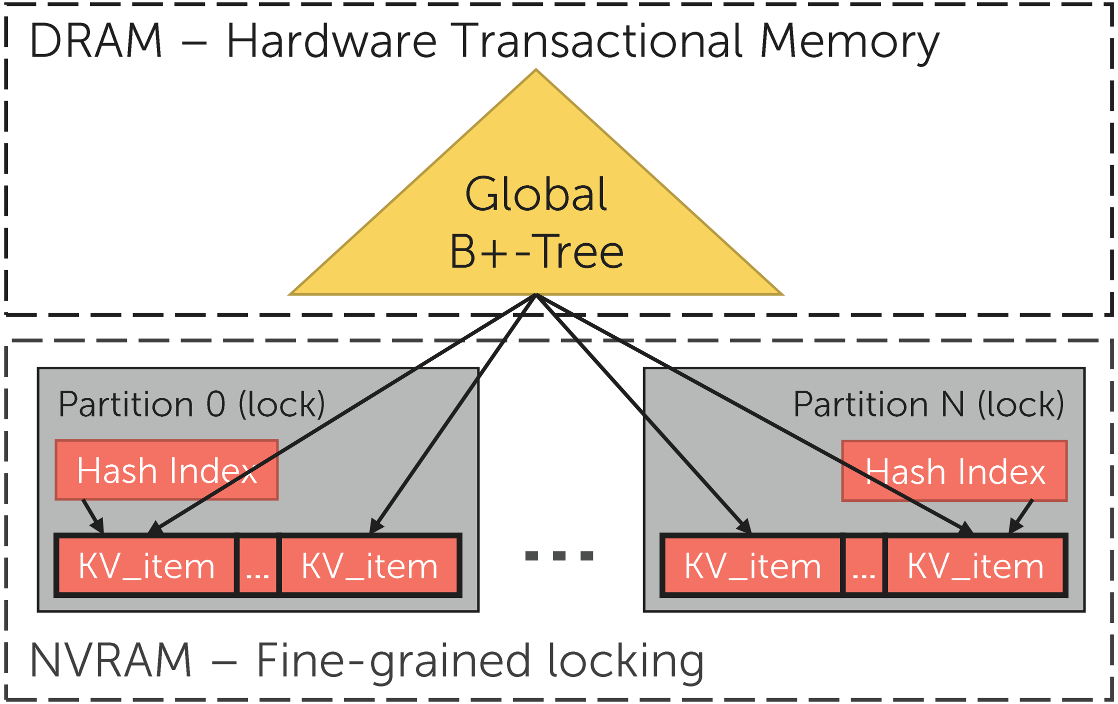
Reduce NVRAM writes

- Unsorted leaves
- Selective consistency

Concurrency scheme

- Selective concurrency

HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. USENIX ATC'17



Reused design ideas

- Selective persistence
- Selective concurrency
- Out-of-place updates

New design ideas

- Global transient B+-Tree and partitioned persistent hash index
- Asynchronous writes to the global B+-Tree

Fast point queries & Fast range queries

Discussion

All presented works propose valuable, reusable ideas!

But…some are…

- Oblivious to failure-induced memory leaks
- Do not use a recoverable addressing scheme
- Mix concurrency atomicity with p-atomicity

Using a sound programming model is a must to move to building more complex systems

Tutorial Overview

Part 1: Motivation & Challenges

1. Motivation
2. NVRAM Programming Challenges
3. NVRAM Programming Models

Part 2: Data Structure Engineering for NVRAM

1. Persistent Memory Management
2. Data Structure Design
3. Fail-Safety Testing
4. NVRAM Performance Emulation

Bug Example

Persist = sfence + clwb + sfence

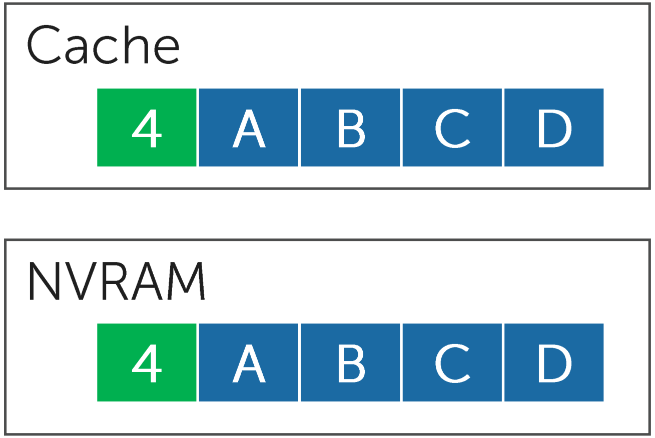
Simplified array append operation:

```
array[size] = val;
persist(&array[size]);
size++;
persist(&size);
```

Correct code

```
array[size] = val;
size++;
persist(&size);
```

Missing persist



Cache

4	A	B	C	D

NVRAM

4	A	B	C	D

Valgrind Persistent Memory Extension

https://github.com/pmem/valgrind

Experimental effort to catch errors related to persistent memory.

Program must tell Valgrind about persistence primitives
→ persistent memory mappings, flushes, fences, etc.

Currently indicates when:

- Writes are not guaranteed to be durable (e.g., missing flush)
- Multiple writes are made to the same location without flushing the first one
- Flushes made to non-dirty cache lines

Yat: A Validation Framework for Persistent Memory Software. USENIX ATC'14

Dresden Database Systems Group

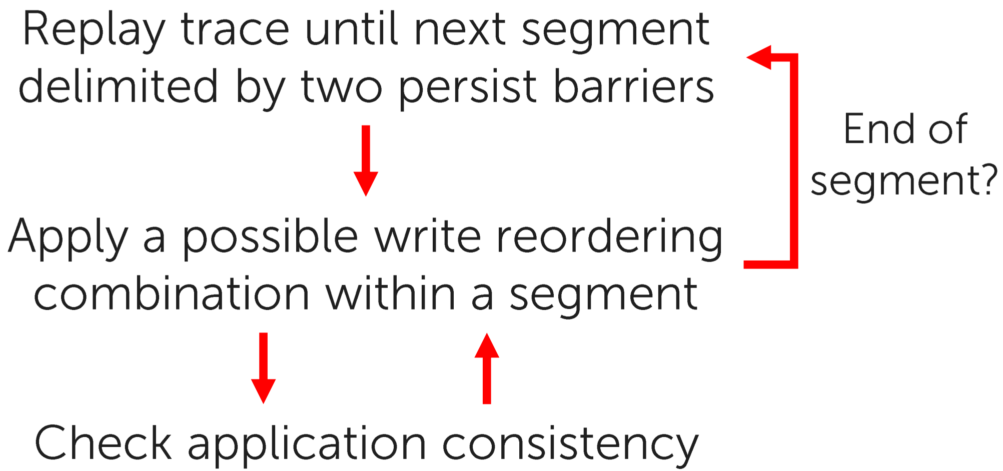
Record-and-replay approach

1. Record

Collect write instructions within the address range of NVRAM

Use virtualization to trace NVRAM primitives as VMM exits

2. Replay



Replay trace until next segment delimited by two persist barriers

Apply a possible write reordering combination within a segment

Check application consistency

End of segment?

Yat: A Validation Framework for Persistent Memory Software. USENIX ATC'14

Dresden Database Systems Group

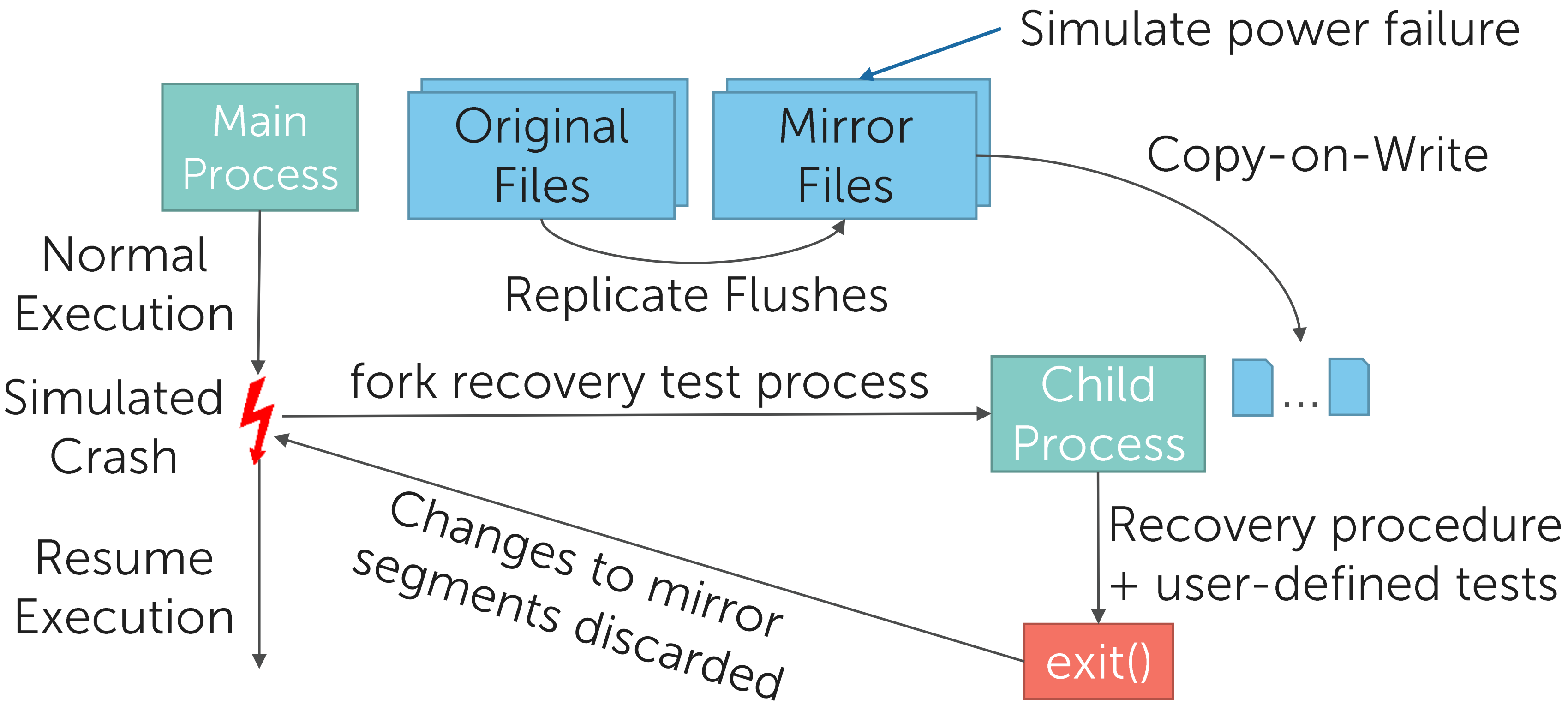
Evaluation: Testing PMFS, an NVRAM-optimized filesystem

Test	write	clflush	pm barrier	Segments		Combinations		Time	
				Total	Thresh.	Total	Thresh.	Total	Thresh.
T1	506	372	131	131	12	15K	4K	44m	15m
T2	54K	14K	6K	6K	4K	789M	1M	5.2y	3d
T3	158K	53K	15K	14K	6K	•	2M	•	5d

+ extensive coverage – slow

On Testing Persistent-Memory-Based Software.

DaMoN 2016

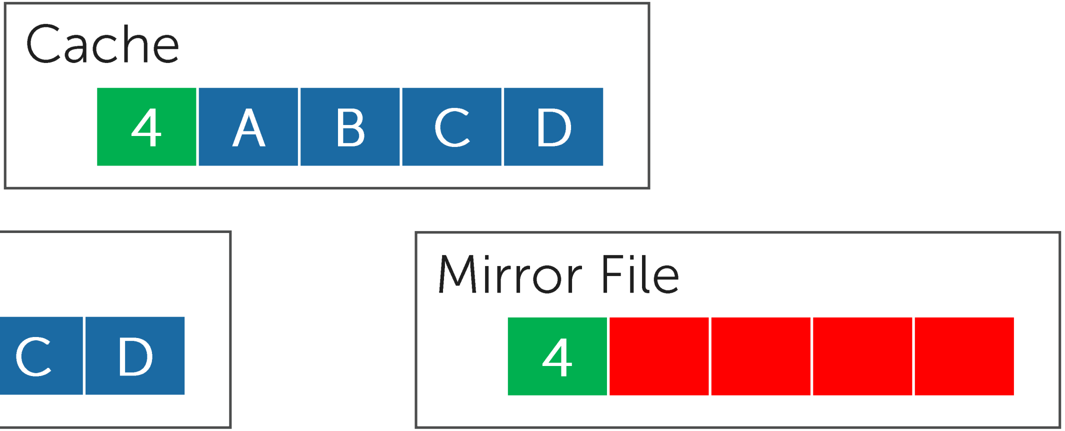


+ Fast and automated – Not exhaustive

Bug Example Revisited

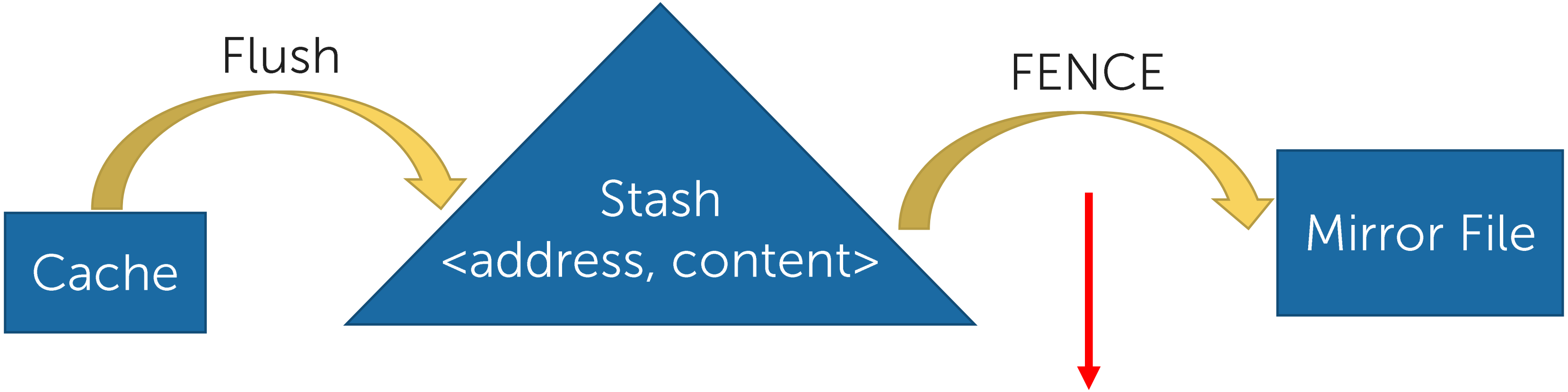
```
array[size] = val;
size++;
persist(&size);
```

Missing persist



Mirror files allow to catch missing persist primitives

Memory Reordering



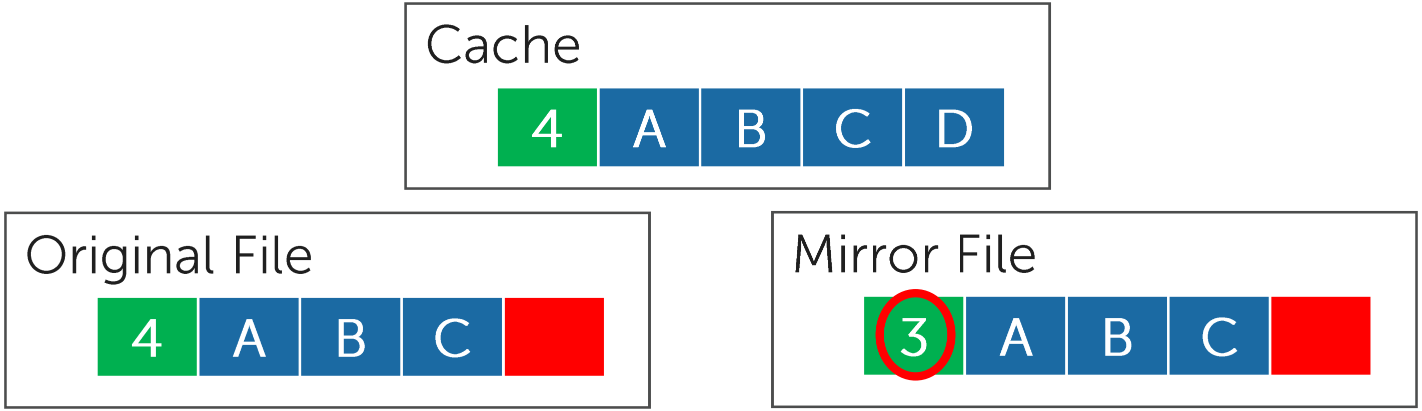
Apply a reordering combination

Catch errors resulting from wrongfully unordered flushes

Limitations

```
array[size] = val;
size++;
persist(&array[size]);
persist(&size);
```

New **size** might be made durable before new **value**



Cache

4 A B C D

Original File

4 A B C 🟥

Mirror File

3 A B C 🟥

Durability reordering of writes cannot be detected

Testing of Multi-Threaded Programs

```
mutex m1, m2;
if(m1.try_lock()){
    ctr1++;
    persist(&ctr1);
    m1.unlock();
~~}else if(m2.try_lock()){~~
    ~~ctr2++;~~
    ~~persist(&ctr1);~~
    ~~m2.unlock();~~
}
```

Single-threaded execution

Argument should be **&ctr2**

Single-threaded fail-safety + concurrency correctness

≠

Multi-threaded fail-safety

Summary

- ➢ No “free lunch”: aggravated corruption risks
- ➢ Exhaustive testing practically infeasible
  → Strong theoretical guarantees are a prerequisite
- ➢ Simple testing techniques that cover a wide range of bugs

Tutorial Overview

Part 1: Motivation & Challenges

1. Motivation
2. NVRAM Programming Challenges
3. NVRAM Programming Models

Part 2: Data Structure Engineering for NVRAM

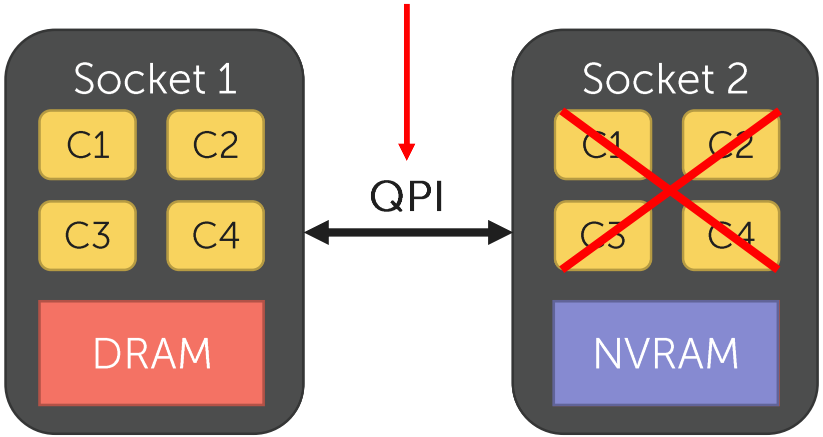
1. Persistent Memory Management
2. Data Structure Design
3. Fail-Safety Testing
4. NVRAM Performance Emulation

NUMA-Based NVRAM Emulation

Higher latency, lower bandwidth

Bind application to socket 1

- `numactl`
- `libnuma`



Memory of socket 2 as emulated NVRAM

How?

Can deactivate QPI links to further increase latency on larger systems

+ Micro-architectural behavior not affected

- Limited latency settings, symmetric latency

Using DRAM as Emulated NVRAM

Two alternatives

Mount a `tmpfs` filesystem and bind memory to a specific processor

```
mount -t tmpfs -o size=1G tmpfs /mnt/pmem
mount -o remount,mpol=bind:1 /mnt/pmem
```

Reserve a DRAM region at boot time and mount a DAX filesystem on it

`memmap=32G!64G` kernel parameter → reserve 32G of RAM starting from 64G

```
mkfs.ext4 /dev/pmem0
mount -o dax /dev/pmem0 /mnt/pmem
```

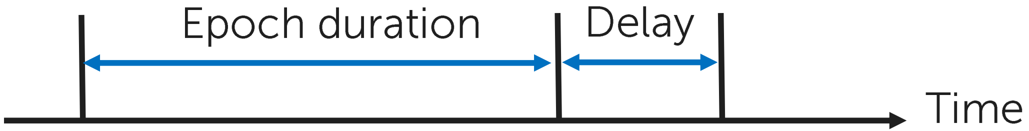
Further details: http://pmem.io/2016/02/22/pm-emulation.html

Quartz: A Lightweight Performance Emulator for Persistent Memory Software. Middleware'2015

https://github.com/HewlettPackard/quartz

Emulates bandwidth by utilizing the DRAM thermal control

Models average application perceived latency
→ Inject delays at boundaries of **epochs**

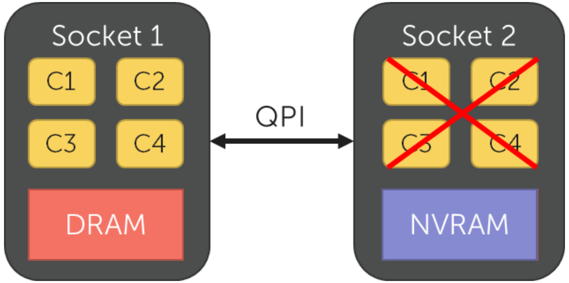


Delay = (Stalled cycles / Average latency) X (NRAM latency – DRAM latency)

Quartz: A Lightweight Performance Emulator for Persistent Memory Software. Middleware'2015

https://github.com/HewlettPackard/quartz

Can emulate two memory regions: DRAM + NVRAM
→ Delays based on remote memory access stalls



How to use Quartz?

Preload user-mode library
→ Registers threads → manages **epochs** and injects delays

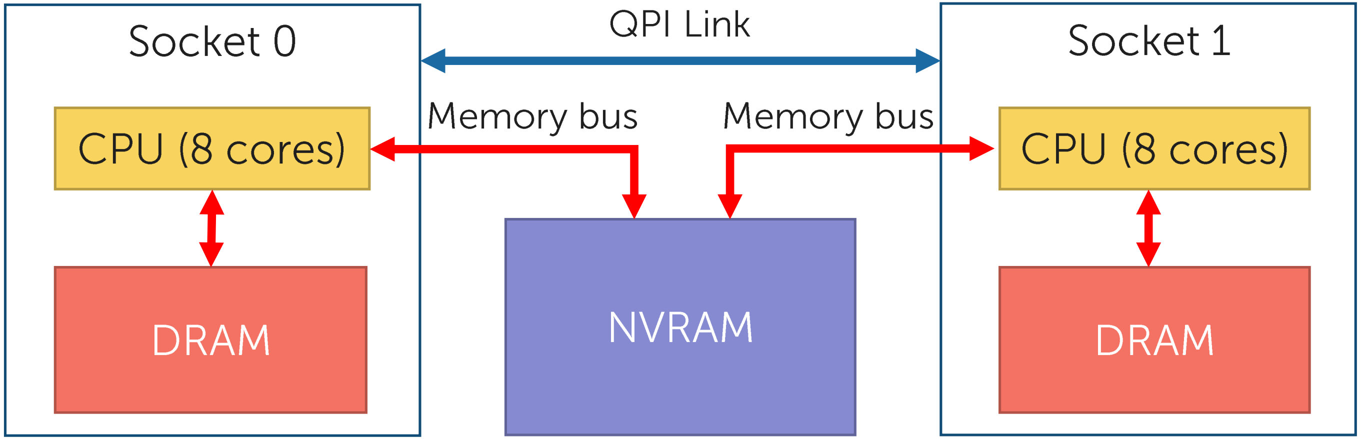
+ Wide range of latency/bandwidth settings

– Less reliable than NUMA-based emulation, symmetric latency

Intel NVRAM Emulation Platform. Subramanya R. Dulloor. Systems and Applications for Persistent Memory. PhD Thesis, 2015.

Emulates bandwidth by utilizing the DRAM thermal control

Increases latency using microcode patch



+ Accurate, microcode-based, uses memory bus

- Not widely available, symmetric latency

Access through Intel virtual Lab → Requires sponsor from Intel

Summary

- Available, easy-to-use NVRAM latency and bandwidth emulation techniques
  - NUMA-based emulation
  - Quartz
  - Intel‘s NVMEP

- Reliable performance emulation

- Limitation
  - Symmetric read/write latencies

Tutorial Wrap Up

Part 1: Motivation & Challenges

1. Motivation
2. NVRAM Programming Challenges
3. NVRAM Programming Models

Part 2: Data Structure Engineering for NVRAM

1. Persistent Memory Management
2. Data Structure Design
3. Fail-Safety Testing
4. NVRAM Performance Emulation

Hands-on session

Room: Buckingham

Tuesday, 4-6 p.m.

Distribute bootable USB drives with Ubuntu 16.04.2 (sponsored by SAP)

Walk through code examples using Intel's NVM Library

Join us and write your first NVRAM data structure!

References: NVRAM Programming Models

Approach	Reference
Mnemosyne	Mnemosyne: Lightweight Persistent Memory. Volos et al. In ASPLOS'11
NV-Heaps	NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. Coburn et al. In ASPLOS'11
Intel NVML	Intel NVM Library. http://pmem.io/
Atlas	Atlas: Leveraging Locks for Non-volatile Memory Consistency. Chakrabarti et al. In ACM SIGPLAN Notices '14.
REWIND	REWIND: Recovery Write-Ahead System for In-Memory NonVolatile Data-Structures. Chatzistergiou et al. In VLDB'15.
PAllocator	Memory Management Techniques for SCM-Based Database Systems. Oukid et al. VLDB 2017. To appear.

References: Persistent Memory Allocators

Allocator	Reference
nvm_malloc	nvm malloc: Memory Allocation for NVRAM. Schwalb et al. In ADMS@VLDB'15.
NVML	Intel NVM Library. http://pmem.io/
Makalu	Makalu: Fast Recoverable Allocation of Non-volatile Memory. Bhandari et al. In OOPSLA'16.
NVMalloc	Consistent, durable, and safe memory management for byte-addressable non volatile main memory. Moraru et al. In TRIOS'13.
WAlloc	WAlloc: An Efficient Wear-Aware Allocator for Non-Volatile Main Memory. Yu et al. In IEEE IPCCC'15.