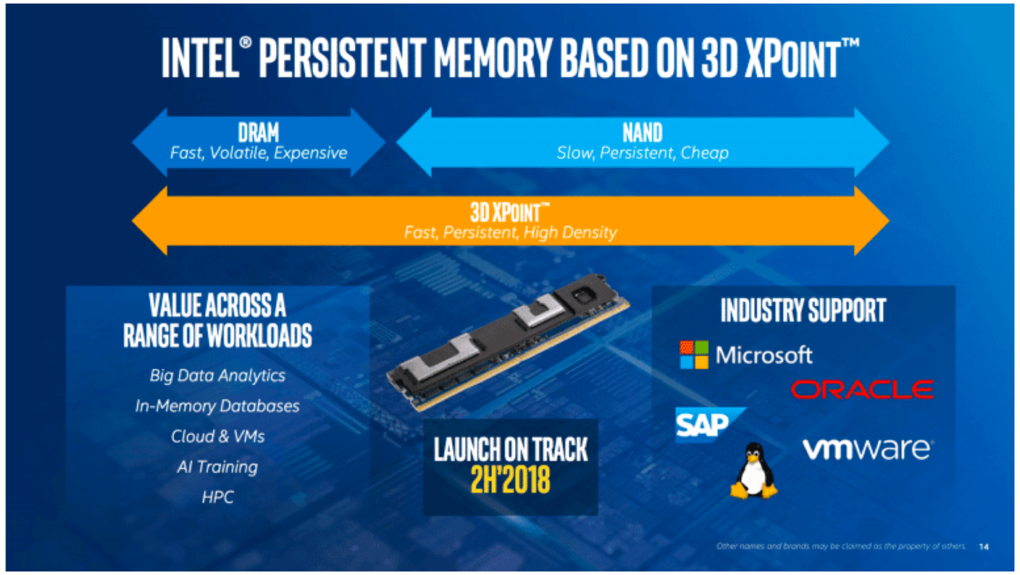
MEMSYS18

Quantifying the Performance Overheads of PMDK

William Wang, Stephan Diestelhorst

2 October 2018

NVDIMMs Shipping in 2H’2018


INTEL® PERSISTENT MEMORY BASED ON 3D XPOINT™

DRAM
Fast, Volatile, Expensive

NAND
Slow, Persistent, Cheap

3D XPOINT™
Fast, Persistent, High Density

VALUE ACROSS A RANGE OF WORKLOADS
Big Data Analytics
In-Memory Databases
Cloud & VMs
AI Training
HPC

LAUNCH ON TRACK 2H’2018

INDUSTRY SUPPORT
Microsoft
ORACLE
SAP
vmware

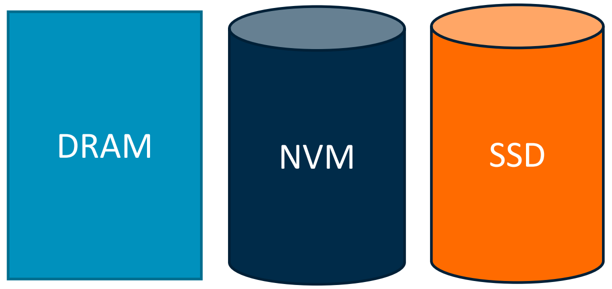
Other names and brands may be claimed as the property of others. 14

- 3DXP NVDIMM shipping 2H’2018
  - DDR4 -> $10/GB, NAND -> $1/4 per GB
  - 3DXP -> $2/GB, up to 512GB
- Arm NVDIMM-ready timeframe around 2022

Multiple System Use-Cases for NVM

Faster Storage

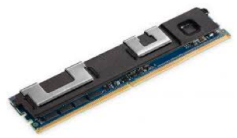
1000x faster than NAND

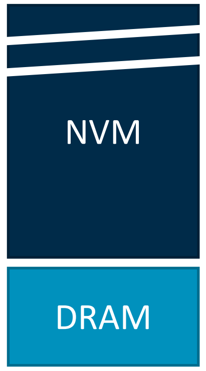


Storage
- Filesystem bottlenecks

Denser Mem

10x denser than DRAM

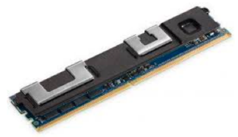


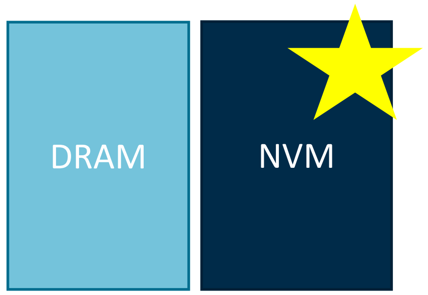


Transformative Capacity/TCO-advantage
- Endurance
- Bandwidth
- Caching

Persistent Mem

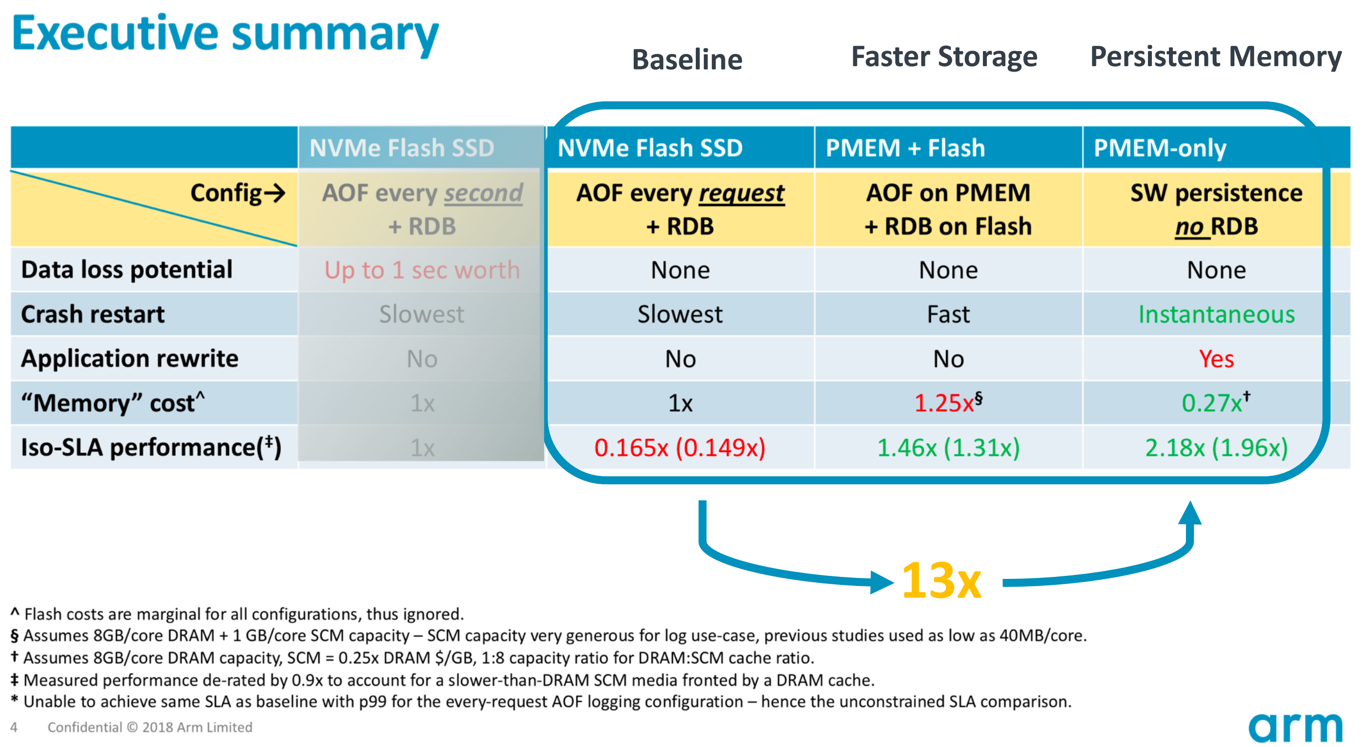
Non-Volatile





Persistency
- Failure atomicity
- Ordering
- ISA

Spectrum of Application Performance Boost with NVM



Executive summary

		Baseline	Faster Storage	Persistent Memory
	NVMe Flash SSD	NVMe Flash SSD	PMEM + Flash	PMEM-only
Config→	AOF every *second* + RDB	**AOF every *request* + RDB**	**AOF on PMEM + RDB on Flash**	**SW persistence *no* RDB**
Data loss potential	Up to 1 sec worth	None	None	None
Crash restart	Slowest	Slowest	Fast	Instantaneous
Application rewrite	No	No	No	Yes
"Memory" cost^	1x	1x	1.25x§	0.27x†
Iso-SLA performance(‡)	1x	0.165x (0.149x)	1.46x (1.31x)	2.18x (1.96x)

13x

^ Flash costs are marginal for all configurations, thus ignored.
§ Assumes 8GB/core DRAM + 1 GB/core SCM capacity – SCM capacity very generous for log use-case, previous studies used as low as 40MB/core.
† Assumes 8GB/core DRAM capacity, SCM = 0.25x DRAM $/GB, 1:8 capacity ratio for DRAM:SCM cache ratio.
‡ Measured performance de-rated by 0.9x to account for a slower-than-DRAM SCM media fronted by a DRAM cache.
* Unable to achieve same SLA as baseline with p99 for the every-request AOF logging configuration – hence the unconstrained SLA comparison.

4 Confidential © 2018 Arm Limited

arm

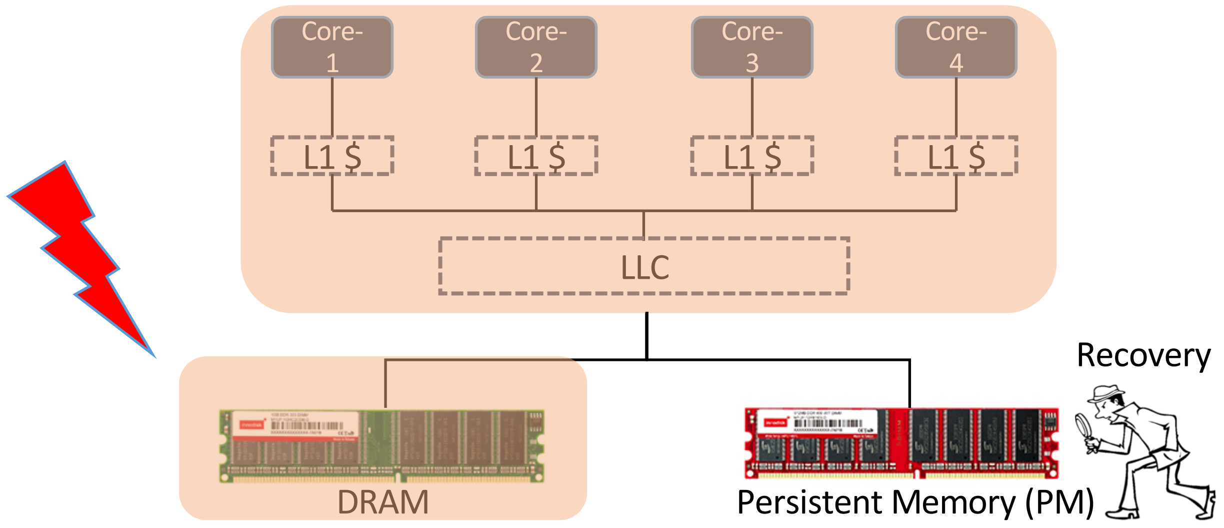
Slide Courtesy of Kshitij Sudan

redis

- ***13x** perf boost, **1/4** mem cost*
- *No data loss*
- *Instantaneous restart*
- ***Rewrite application***

arm

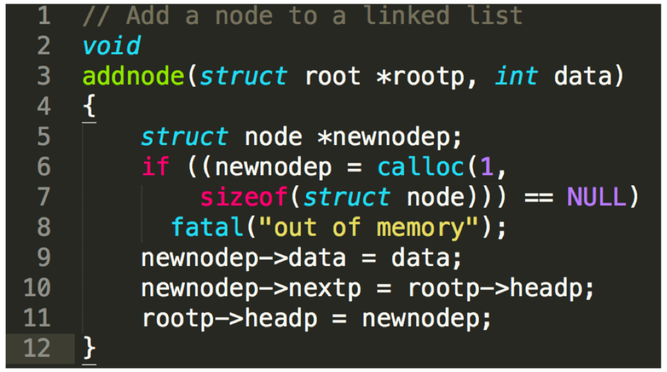
Why Application Rewrite



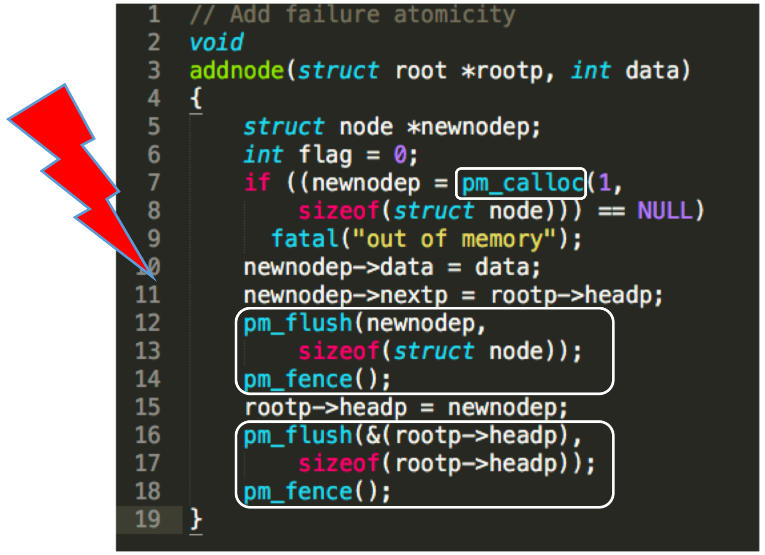
Recovery can inspect the data-structures in PM to restore system to a consistent state

- Crash consistency (failure atomicity) is needed to ensure recovery can restore system to a consistent state
  - Data move through volatile memories before they get written to PM
  - Using CPU cache flushes and fence instructions

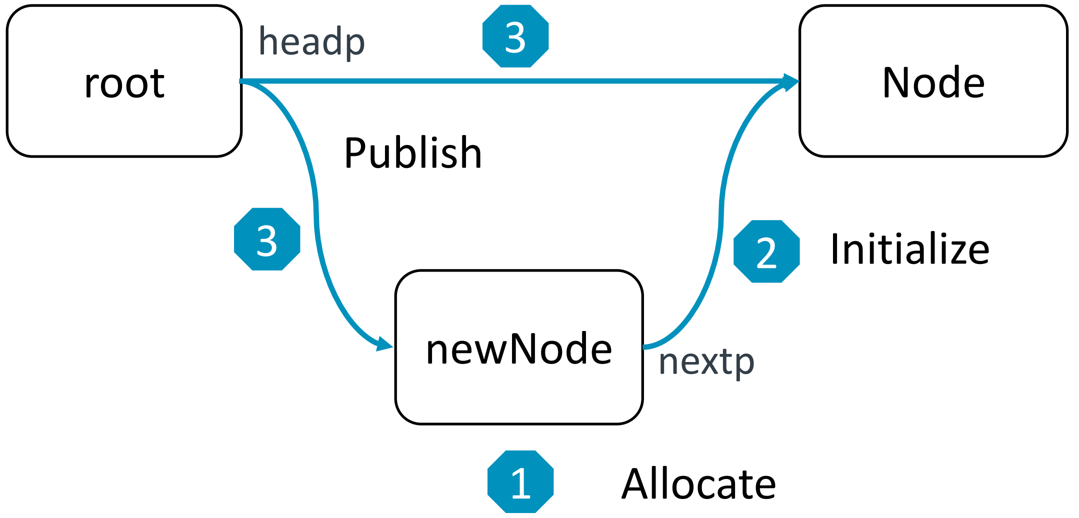
Example: Add a Node to a Linked List

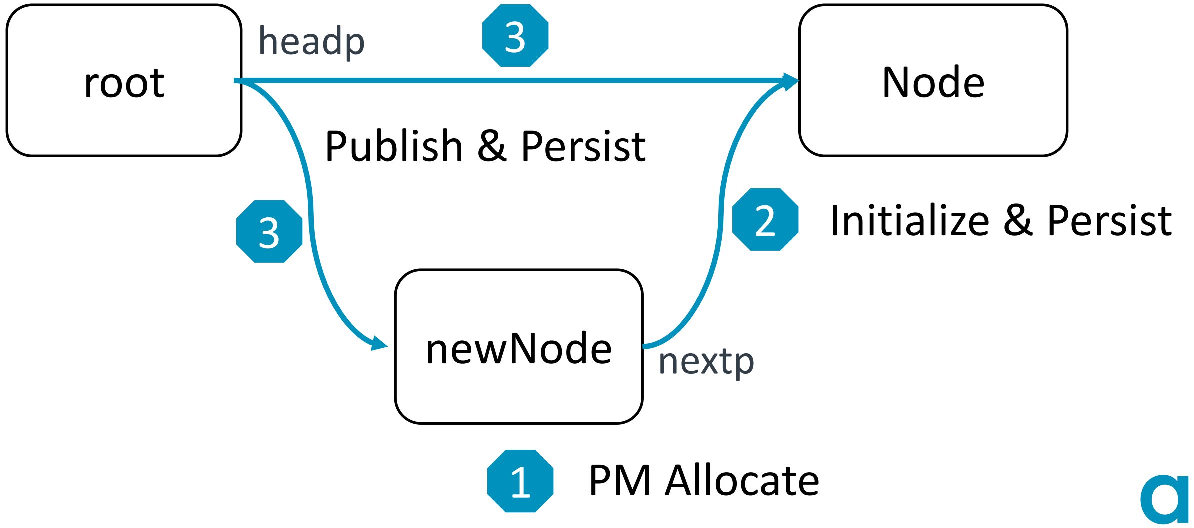


```c
// Add a node to a linked list
void
addnode(struct root *rootp, int data)
{
	struct node *newnodep;
	if ((newnodep = calloc(1,
	    sizeof(struct node))) == NULL)
	  fatal("out of memory");
	newnodep->data = data;
	newnodep->nextp = rootp->headp;
	rootp->headp = newnodep;
}
```

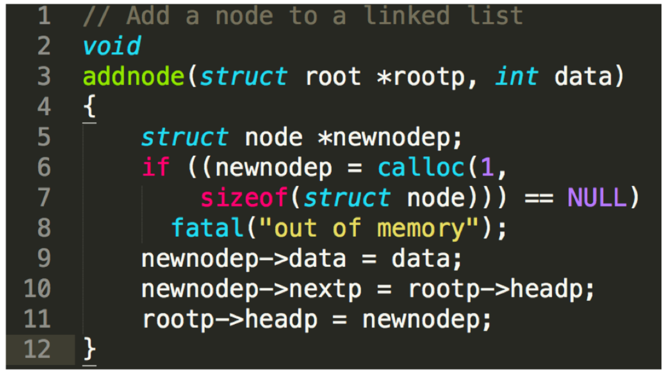


```c
// Add failure atomicity
void
addnode(struct root *rootp, int data)
{
	struct node *newnodep;
	int flag = 0;
	if ((newnodep = pm_calloc(1,
	    sizeof(struct node))) == NULL)
	  fatal("out of memory");
	newnodep->data = data;
	newnodep->nextp = rootp->headp;
	pm_flush(newnodep,
	    sizeof(struct node));
	pm_fence();
	rootp->headp = newnodep;
	pm_flush(&(rootp->headp),
	    sizeof(rootp->headp));
	pm_fence();
}
```

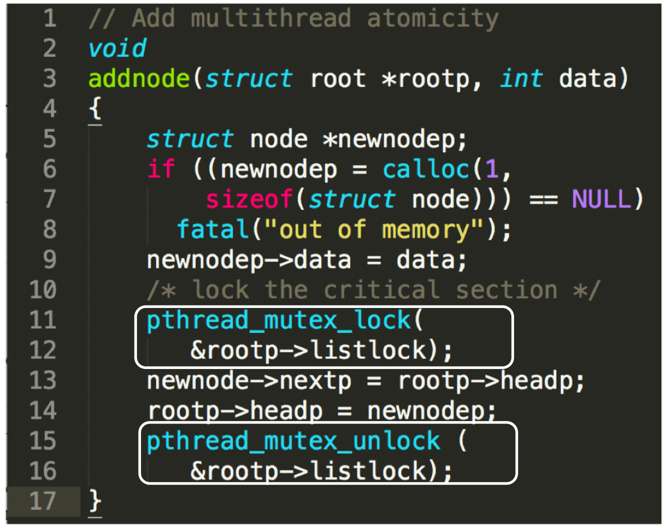




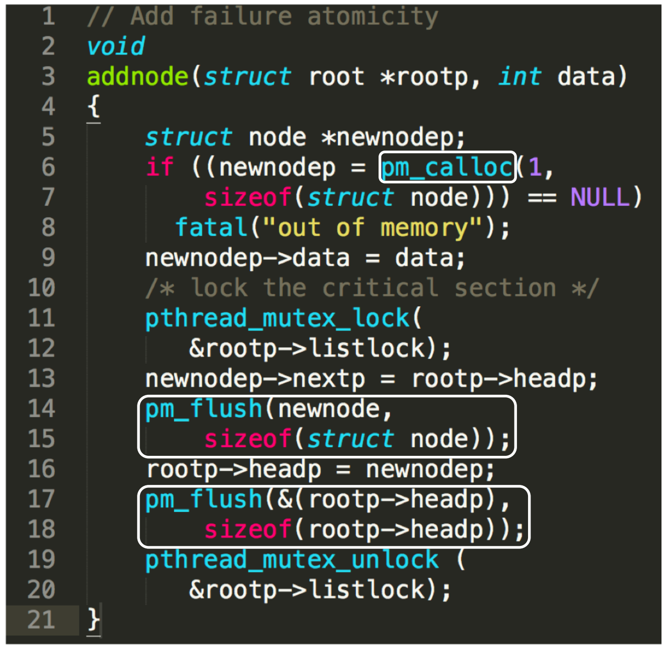
Example: Add Concurrency to the Mix



```c
// Add a node to a linked list
void
addnode(struct root *rootp, int data)
{
    struct node *newnodep;
    if ((newnodep = calloc(1,
        sizeof(struct node))) == NULL)
      fatal("out of memory");
    newnodep->data = data;
    newnodep->nextp = rootp->headp;
    rootp->headp = newnodep;
}
```



```c
// Add multithread atomicity
void
addnode(struct root *rootp, int data)
{
    struct node *newnodep;
    if ((newnodep = calloc(1,
        sizeof(struct node))) == NULL)
      fatal("out of memory");
    newnodep->data = data;
    /* lock the critical section */
    pthread_mutex_lock(
        &rootp->listlock);
    newnode->nextp = rootp->headp;
    rootp->headp = newnodep;
    pthread_mutex_unlock (
        &rootp->listlock);
}
```

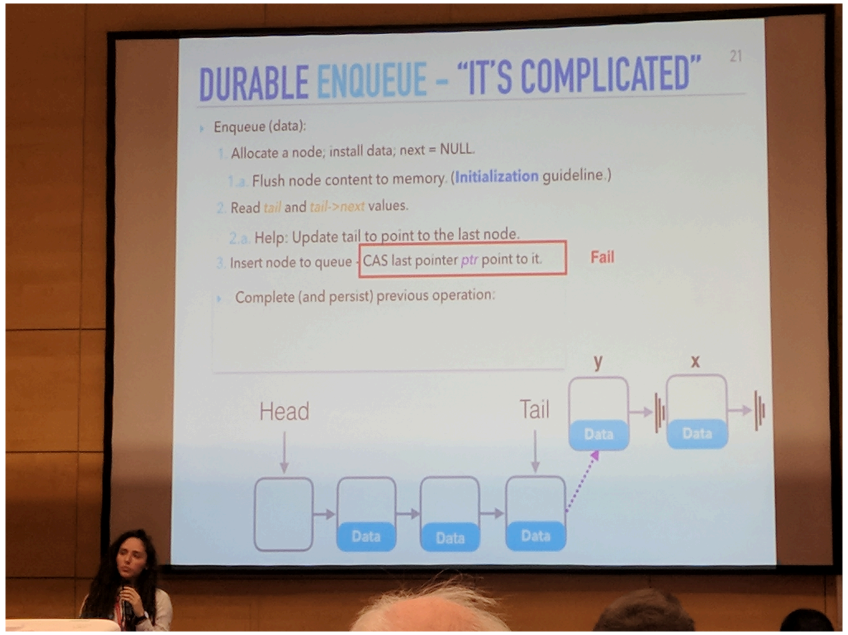


```c
// Add failure atomicity
void
addnode(struct root *rootp, int data)
{
    struct node *newnodep;
    if ((newnodep = pm_calloc(1,
        sizeof(struct node))) == NULL)
      fatal("out of memory");
    newnodep->data = data;
    /* lock the critical section */
    pthread_mutex_lock(
        &rootp->listlock);
    newnodep->nextp = rootp->headp;
    pm_flush(newnode,
        sizeof(struct node));
    rootp->headp = newnodep;
    pm_flush(&(rootp->headp),
        sizeof(rootp->headp));
    pthread_mutex_unlock (
        &rootp->listlock);
}
```

What can still go wrong?

Fences are missing after persists, they can get persisted out of order!!!

What about making this lock-free?



DURABLE ENQUEUE – "IT'S COMPLICATED" 21

- Enqueue (data):
  1. Allocate a node; install data; next = NULL.
     1.a Flush node content to memory. (Initialization guideline.)
  2. Read tail and tail->next values.
     2.a Help: Update tail to point to the last node.
  3. Insert node to queue - CAS last pointer ptr point to it. Fail
- Complete (and persist) previous operation.

It's Complicated

- *Oracle Labs at PPoPP'18*

Source: M. Friedman, A Persistent Lock-Free Queue for Non-Volatile Memory, PPoPP'18

Simplify lock-based persistent programming **"Persistency for Synchronization-free Regions"** in **PLDI '18**

Persistent Memory Programming Models

Native Persistence

```
pt->x = 1;
pt->y = 1;
dccvap(&pt->x)
dccvap(&pt->y)
dsb

flag=1;
dccvap(&flag)
dsb
```

Library Persistence – Atomic

```
pt->x = 1;
pt->y = 1;
pmem_persist(&pt,
sizeof(pt))

flag = 1;
pmem_persist(&flag,
sizeof(flag))
```

Library Persistence – Durable TXs

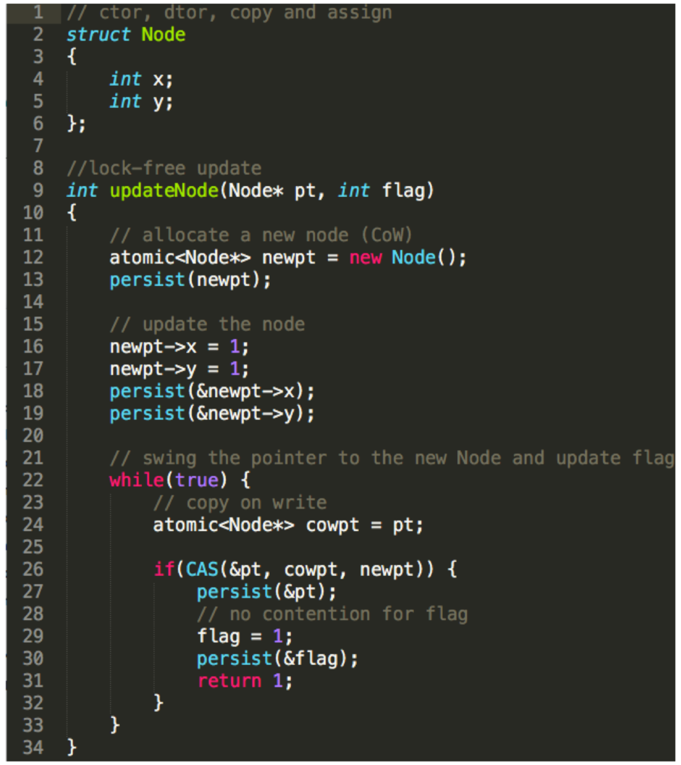
```
TX_BEGIN{
pt->x = 1;
pt->y = 1;
} TX_END
```

Programming simpler, overhead higher

arm

Add Concurrency to the Mix

Lib Persistence – Lock-free



```
// ctor, dtor, copy and assign
struct Node
{
    int x;
    int y;
};

//lock-free update
int updateNode(Node* pt, int flag)
{
    // allocate a new node (CoW)
    atomic<Node*> newpt = new Node();
    persist(newpt);

    // update the node
    newpt->x = 1;
    newpt->y = 1;
    persist(&newpt->x);
    persist(&newpt->y);

    // swing the pointer to the new Node and update flag
    while(true) {
        // copy on write
        atomic<Node*> cowpt = pt;

        if(CAS(&pt, cowpt, newpt)) {
            persist(&pt);
            // no contention for flag
            flag = 1;
            persist(&flag);
            return 1;
        }
    }
}
```

Lib Persistence – Lock-based

```
mutex.lock()
pt->x = 1;
pt->y = 1;
pmem_persist(&pt,
sizeof(pt))

flag = 1;
pmem_persist(&flag,
sizeof(flag))
mutex.unlock()
```

Lib Persistence – Durable TXs

```
TX_BEGIN{
pt->x = 1;
pt->y = 1;
} TX_END
```

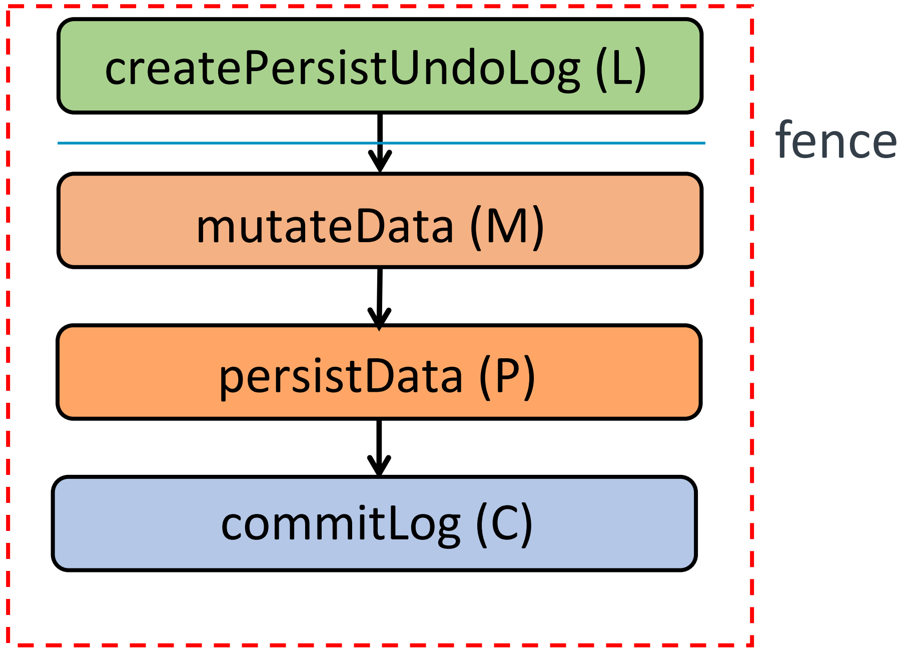
Programming simpler, overheads higher

Durable Transactions

- TXs provide clean failure semantics
- Accelerate w. Persistent HTM

```
TX_BEGIN{
pt->x = 1;
pt->y = 1;
} TX_END
```

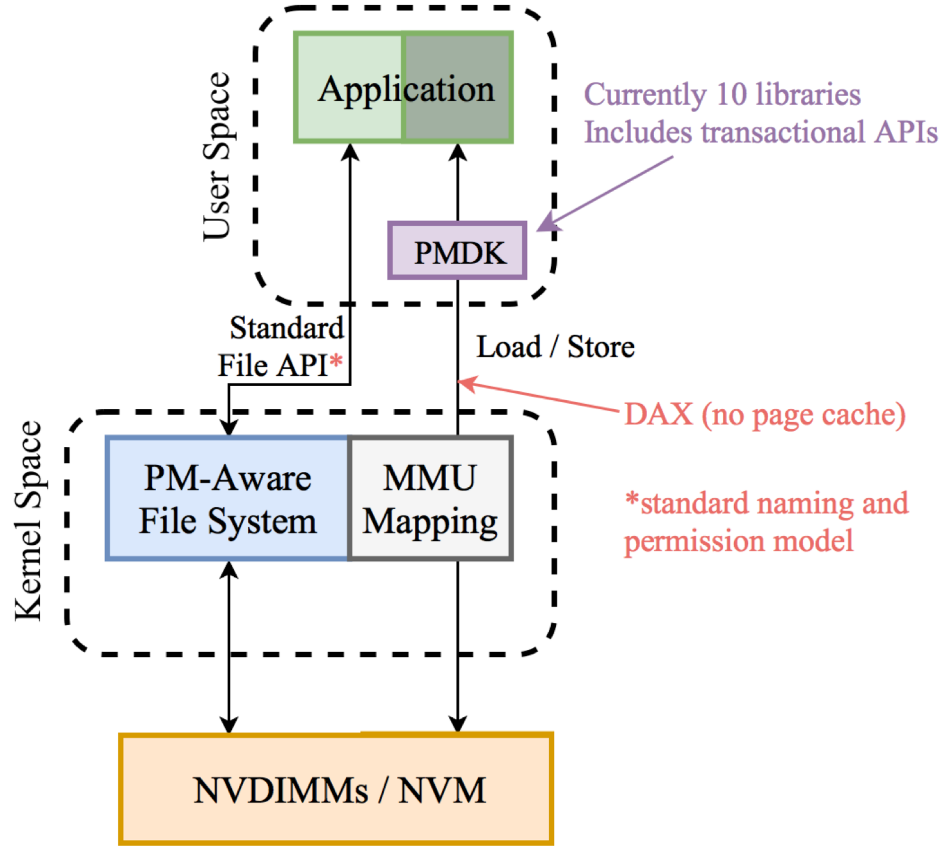
- Durable transactions guarantee Tx failure atomicity
  - Ordered persists + logging



PMDK (Persistent Memory Development Kit)

Formally NVML, 'pmem libraries'

- PMDK provides transactional APIs for persistent memory programming
  - libpmemobj transactional APIs
  - Use fine-grained logging and cache flushes
- Works on 64-bit Linux, Windows and 64-bit FreeBSD
- Supports x86 and Armv8 (experimental)



Ref: pmem.io

Analyze Cost of Durable Transactions

Logging, Persisting and Ordering

Cost of Durable Transactions

Logging, Persisting and Ordering

Persisting

- Latency too high if PoP is off-chip
- Undo log needs to be persisted in TX
- Instruction bloat due to looping through addresses with DCCVAP

Logging

- Write amplification, i.e., write twice
- Copying (costly, and cache pollution)
- Barriers with undo logging
- Memory fragmentation
- Code bloat due to logging

Ordering

- Serialize memory accesses
- Excessive # of barriers via API calls
- Persisting barriers same as ordering barriers

Performance Measurement Setup

Payload

Redis-Benchmark

TPCC

Workload

Redis

Maps (PMDK/NVML)

CPU

CLFLUSH vs. CLFLUSHOPT

Fixed frequency and ASLR off

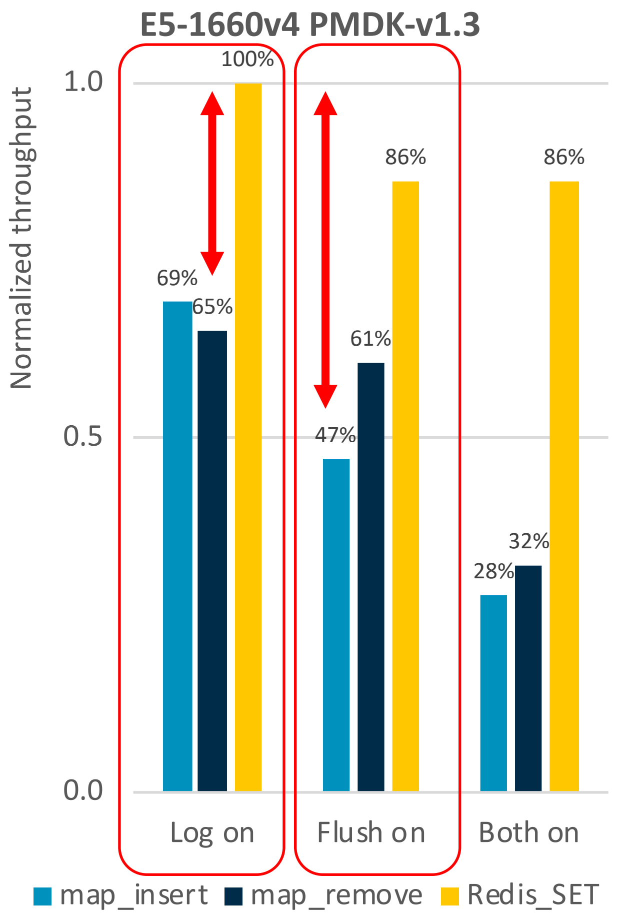
Thread and memory affinity in multi-node

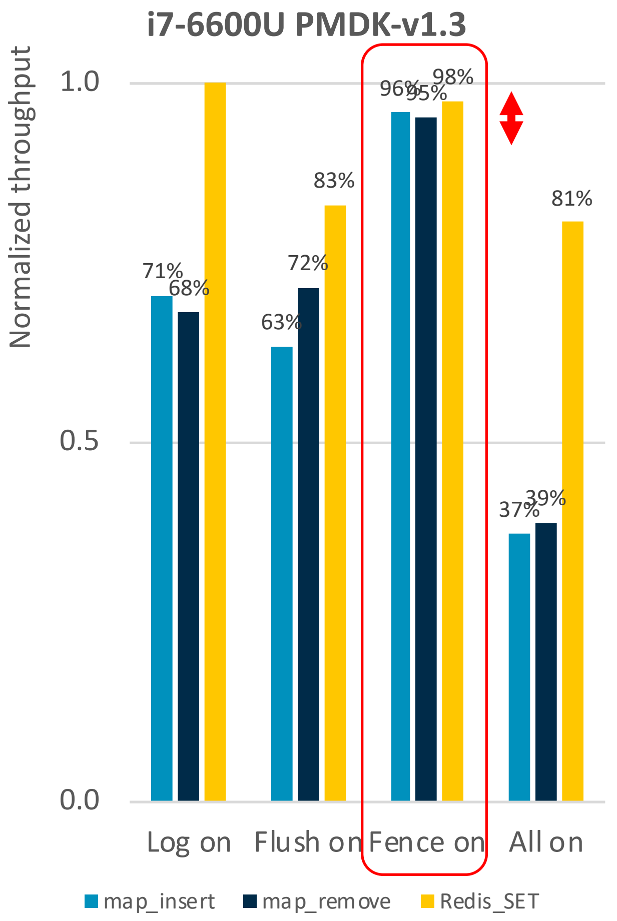
Memory

Local DRAM

Remote DRAM to emulate NVM

Flushing, Logging and Fencing Overhead



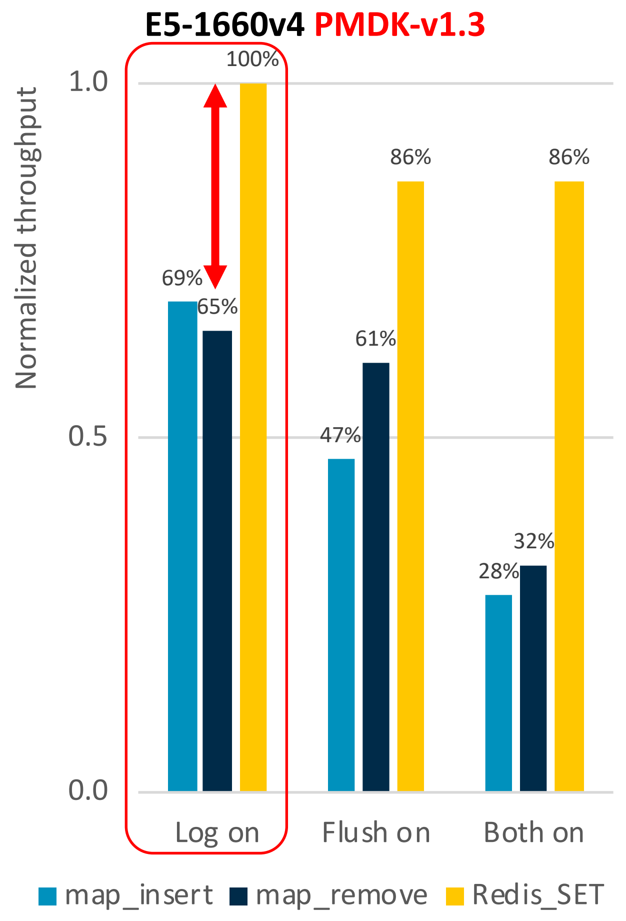


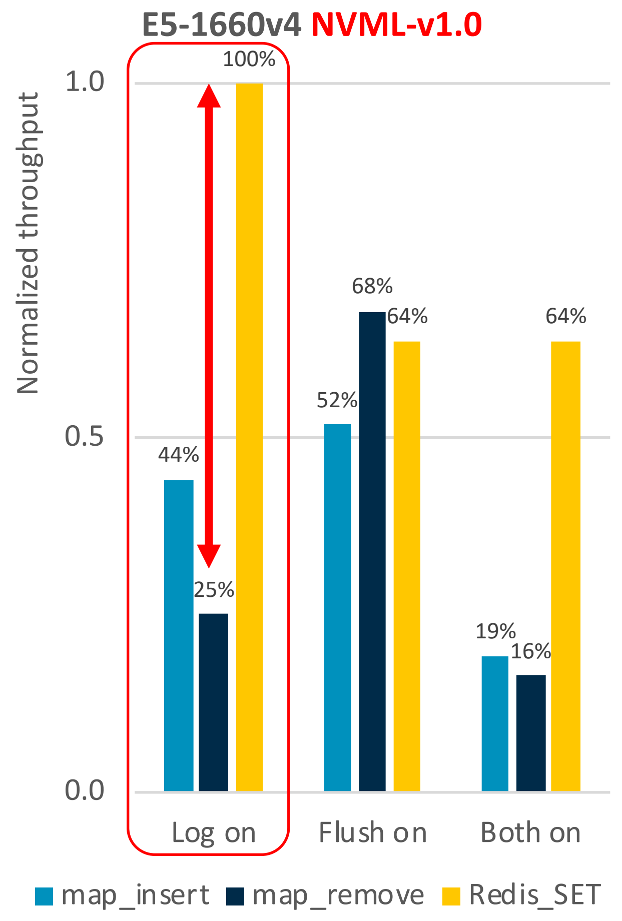
- Logging overhead 0~35%
- Flushing overhead 14~53%
- Fencing overhead 2~5%

Baseline: PMDK without flushing/fencing and logging on

- Workloads: Map insert/remove, Redis Set. Implemented with NVML v.10 and v1.3 libpmemobj transactions
- Platforms: Intel E5-1660v4 with CLFLUSH and Intel i7-6600U with CLFLUSHOPT, single node with local DRAM

Software Opt Can Reduce Logging Overhead





- Software opt can reduce logging overhead
  - NVML v1.3 (left) reduces logging overhead over NVML v1.0 (right)
  - Redis-SET performance not affected by logging overhead

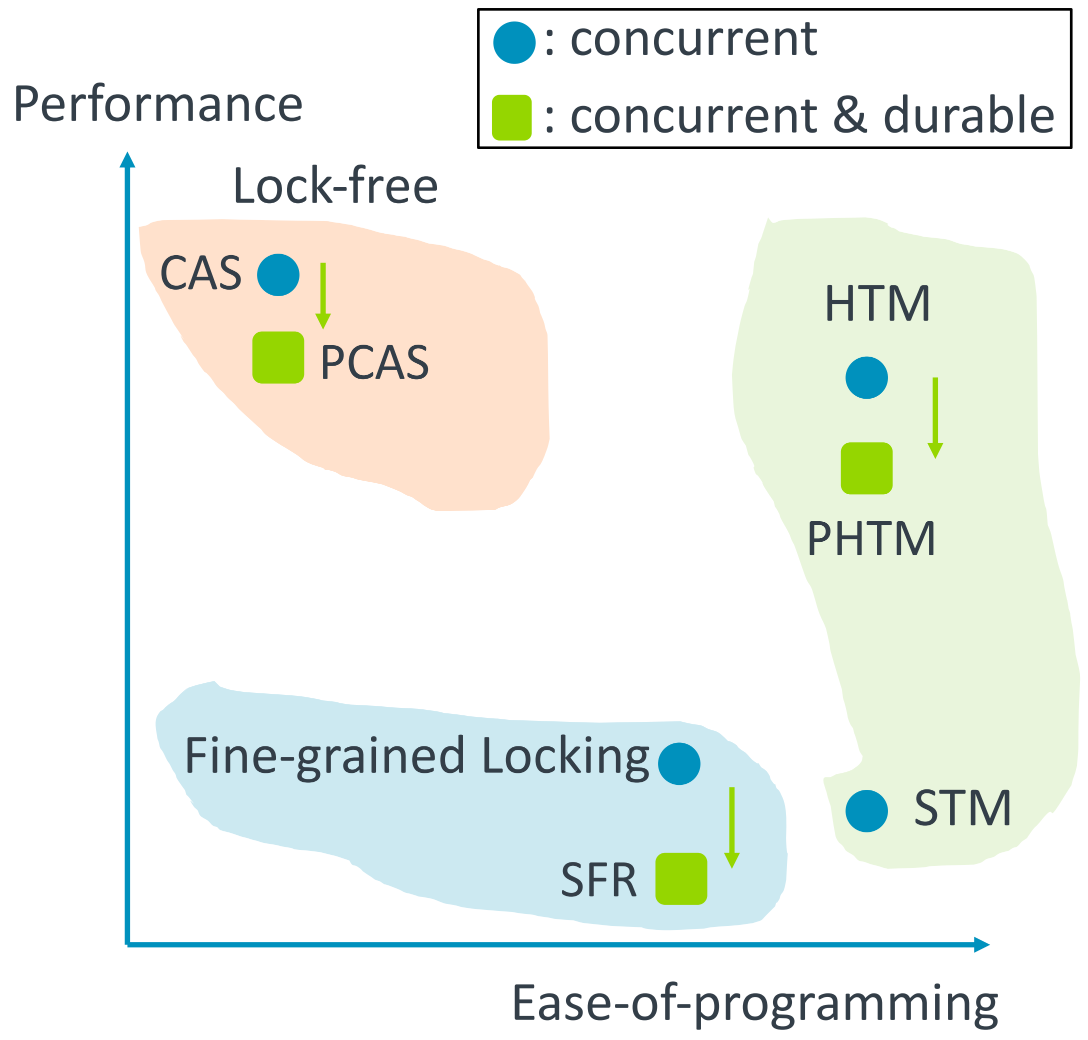
Baseline: PMDK without flushing/fencing and logging on

- Workloads: Map insert/remove, Redis Set. Implemented with NVML v.10 and v1.3 libpmemobj transactions
- Platforms: Intel E5-1660v4 with CLFLUSH and Intel i7-6600U with CLFLUSHOPT, single node with local DRAM

Summary of Performance Overheads Analysis

- Flushing overhead up to 53%.
- Logging overhead up to 35%.
- Fencing overhead up to 5%
- These overheads can be reduced by
  - Software optimizations
  - Hardware accelerations

Ease-of-Programming vs Performance



Programming Model	Ordering	Logging	Persisting	Translation
Atomics	✓	✗	✓	✓
TM	✓	✓	✓	✓
Locking	✓	✓ (log elision)	✓	✓

Ease-of-Programming

Persistent CAS as drop-in replacement for CAS

Persistent HTM as drop-in replacement for HTM

Compiler instrumented failure atomicity for locking

Performance

Relaxed memory persistency model

Hardware accelerations

Software optimizations

Thank You!

Danke!

Merci!

谢谢!

ありがとう!

Gracias!

Kiitos!

감사합니다

धन्यवाद