arm

Architectural Support for Persistent Memory Programming

William Wang, Arm Research

23 October 2020

Outline

- Persistent memory use cases
  - Why do we (Arm) care about persistent memory?
- Memory persistency
  - Do we have sufficient support in the Arm architecture for programming persistent memory?
- Memory consistency
  - Why should you (programmers) care about memory consistency for sequential programs?

Note: Persistent memory refers to persistent uses of non-volatile memory (NVM) here.

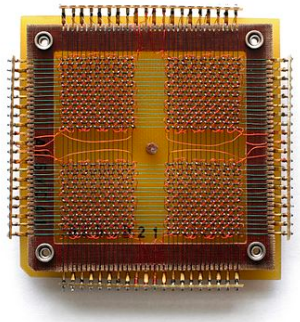
arm

Persistent Memory Use Cases

Why do we care about persistent memory?

NVM Is Real

Magnetic Core Memory



1955-1975

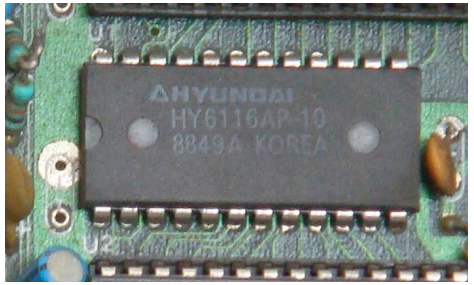
Flash Memory

1984

3DXP

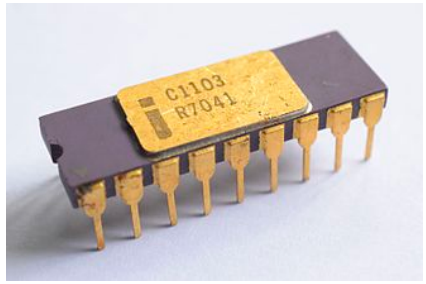
2019

1961



SRAM

1970

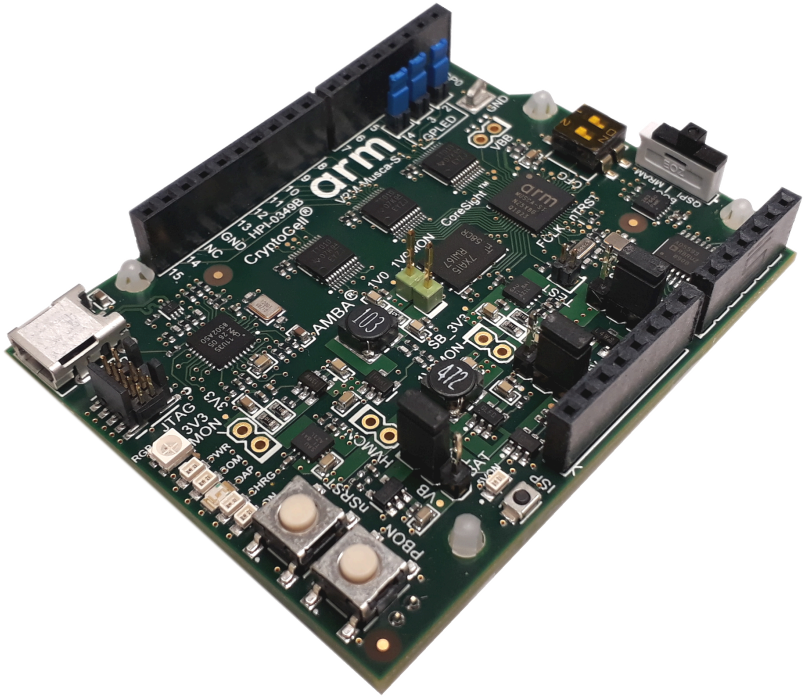


DRAM

arm

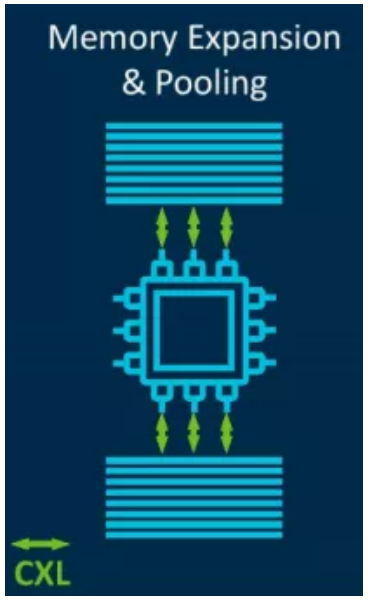
NVM Augments SRAM, DRAM, NOR, and NAND

In Embedded, Client, and Infrastructure



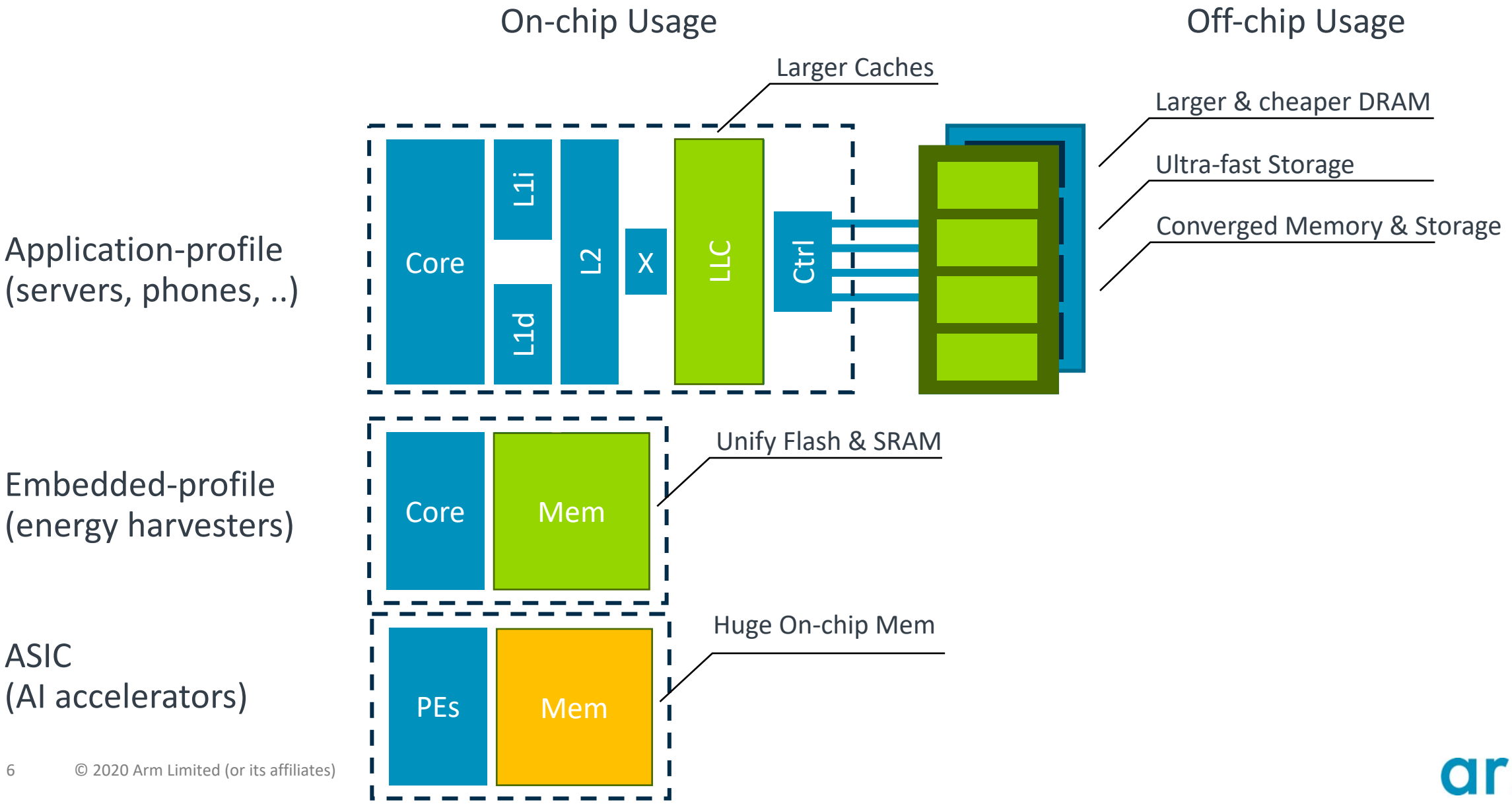
Arm MUSCA-S1 Board with MRAM at 28nm in 2019

Nokia Asha Smartphone with Micron PCM in 2012



CXL Connected Persistent Memory in Infrastructure

Non-Volatile Memory Opportunities



arm

Persistent Use Cases

Beyond 'More Memory'

- Byte addressable, denser than DRAM
- ***Today:*** new memory technologies offering density and cost improvements over DRAM
- ***Tomorrow:*** unlock performance through single memory for storage and compute

Today

More Mem

Denser than DRAM

TCO/Capacity
- Endurance
- Caching

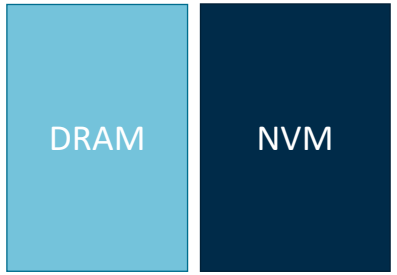
Unlock more perf out of cheaper memory

In-mem DBs, then other apps
- Data analytics
- ML training
- Financial
- HPC

Tomorrow

Persistent Mem

Non-Volatile



Persistency
- Failure atomicity
- Persist ordering
- Crash recovery
- Programming models
- ISA & uarch support

arm

Other Use Cases

Server, Networking, Edge

Databases, i.e., redo logging, in-memory index

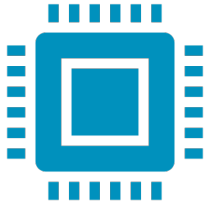
Data analytics, i.e., in-memory index

HPC and ML training, i.e., checkpointing

Animation films, i.e., append only writes

Financial pub/sub service, i.e., KDB+

Serverless restart time



Client

Restart time (apps code and shared libs in PM)

SQLite replacement (i.e., with a hashmap)

Energy efficiency (fast sleep & restore)

USB computer sticks (fast restart)

Mobile content creation (video editing)

Mobile gaming (fast loading)

IoT, Embedded, Auto

Intermittent computing, i.e., forward progress despite frequent power cycles

Industrial IoT, i.e., all data used for decision making need to be persisted

Simulator: https://github.com/UoS-EEC/fused

fast restart, fast save

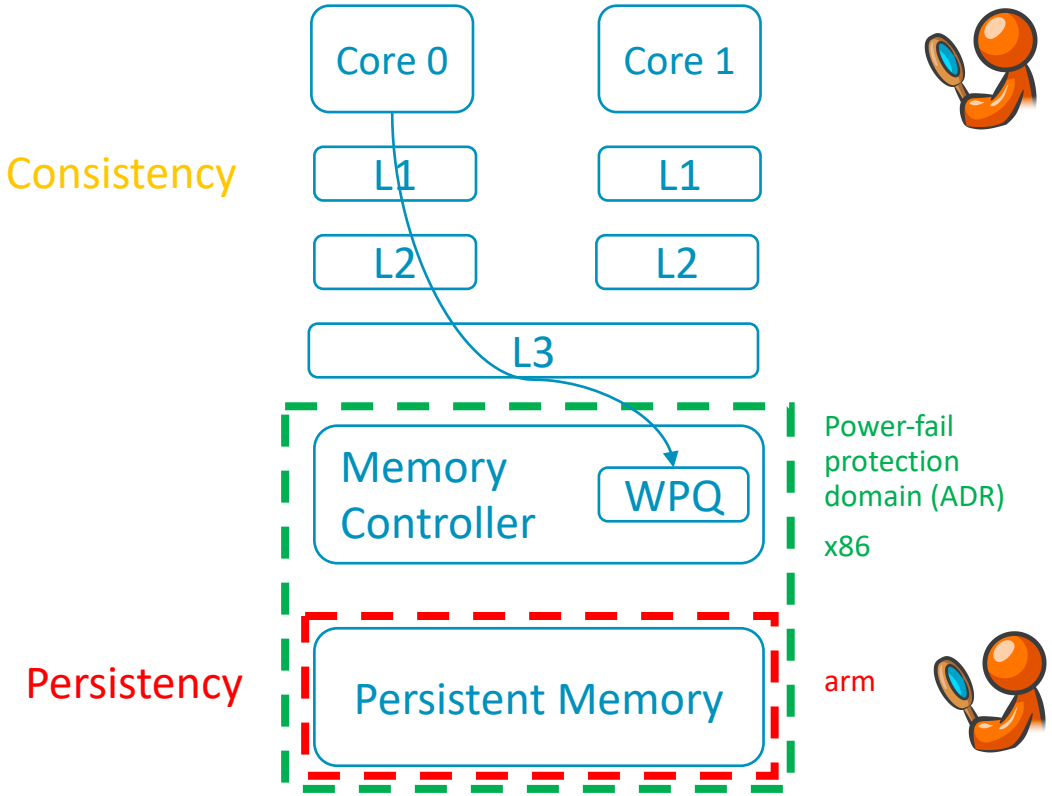
arm

arm

Memory Persistency

Do we have sufficient support in the Arm ISA for programming persistent memory?

System Assumption



- Point of Persistence (PoP) at the persistent memory module or the memory controller WPQ
  - Contents in the power-fail protection domain will be saved upon power failure
- Caches and cores are still in the volatile domain
  - Contents will be lost upon power failure
- Persistency < Consistency (behind)
  - Stores need to be drained from volatile caches to PoP explicitly by software to sync persistency w. consistency

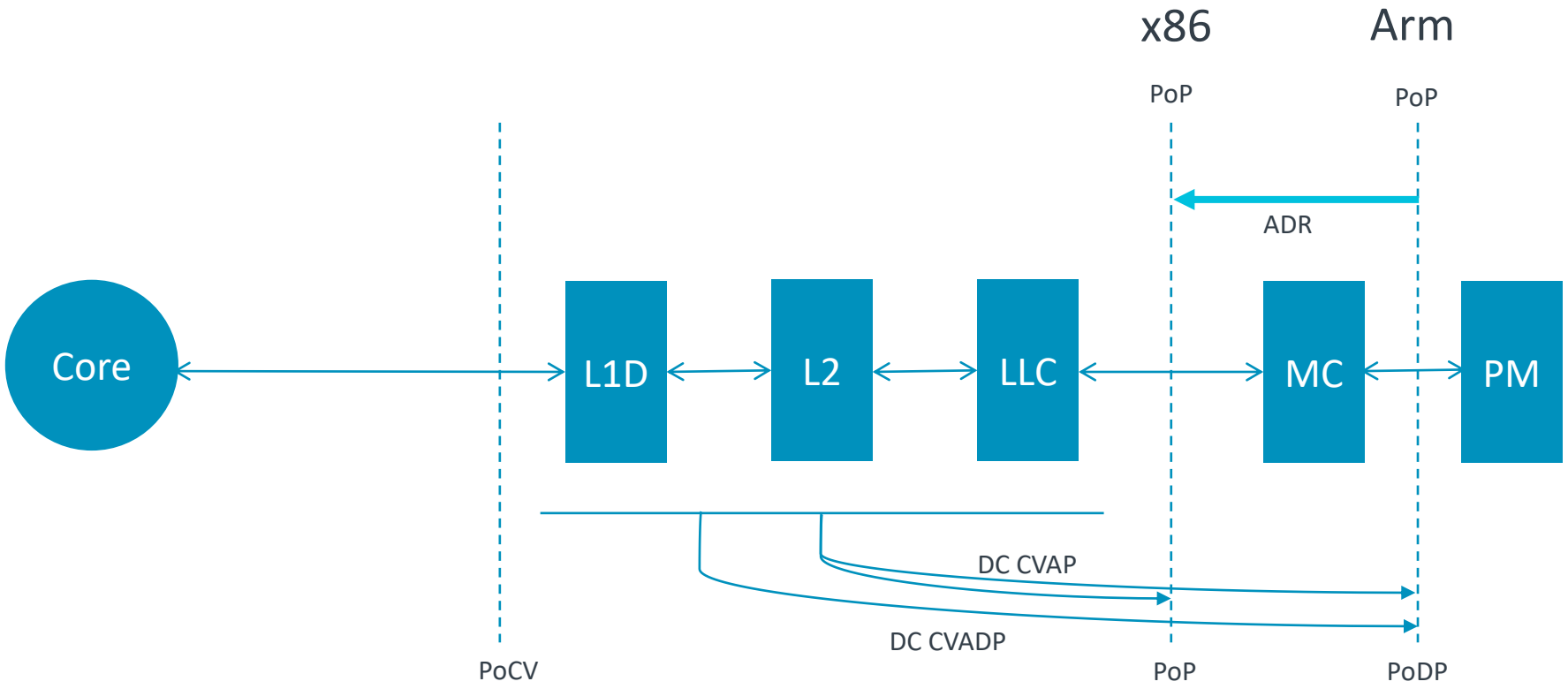
PoP: Point of Persistence

ADR : Asynchronous DRAM Refresh

WPQ: Write Pending Queue

arm

Architectural Support to Sync Visibility & Persistency



DC CVAP in Armv8.2-A and DC CVADP in Armv8.5-A

Barriers (DSB) to guarantee completion of DC CVA[D]P cache maintenance operations

PoCV: Point of Concurrent Visibility

PoP: Point of Persistence

PoDP: Point of Deep Persistence

ADR : Asynchronous DRAM Refresh

DSB: Data Synchronization Barrier

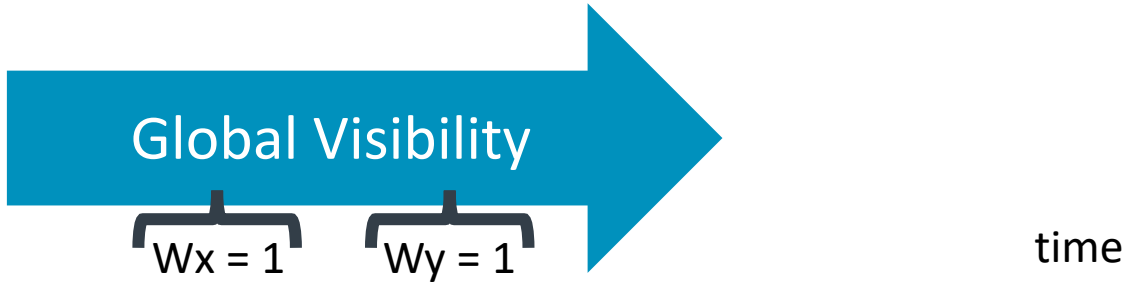
Global Visibility Order

P0
STR W0,[X1]
STR W2,[X3]

Thread 0
a: Wx = 1
po
b: Wy = 1

P0
STR W0,[X1]
DMB.ST
STR W2,[X3]

Thread 0
a: Wx = 1
dmb
b: Wy = 1



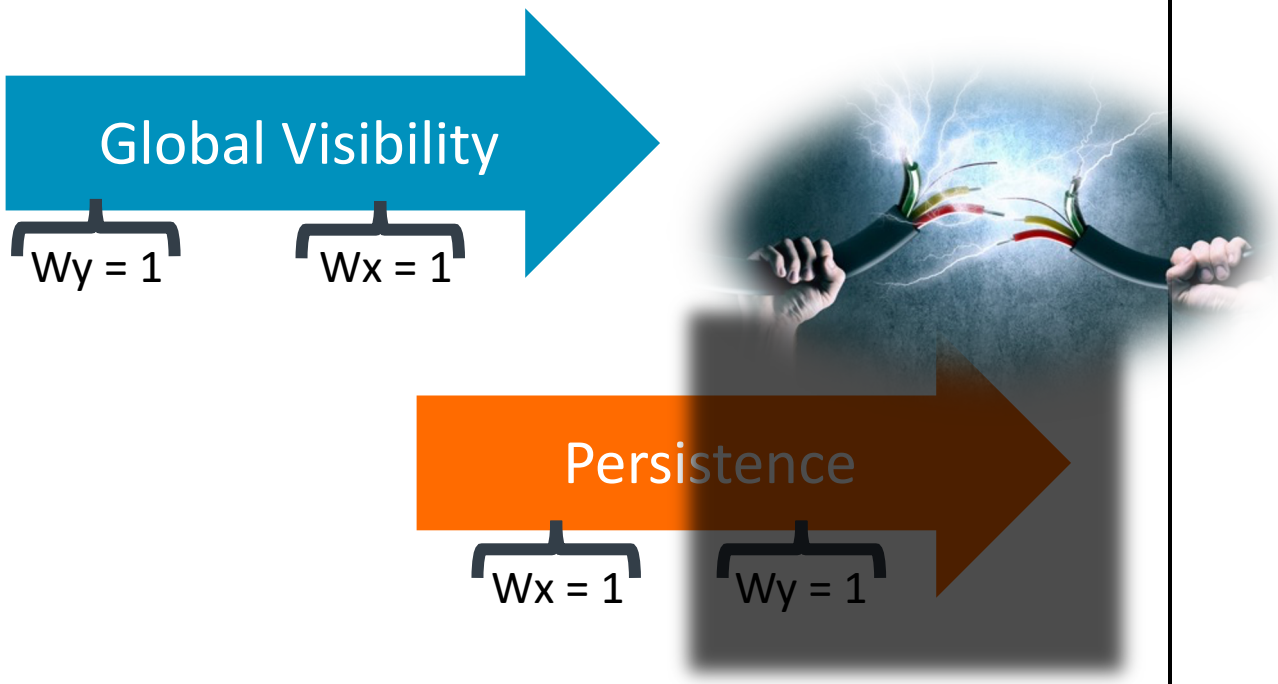
DMB: Data Memory Barrier
DMB.ST: Store barrier

arm

View of the NVM: Persist Order

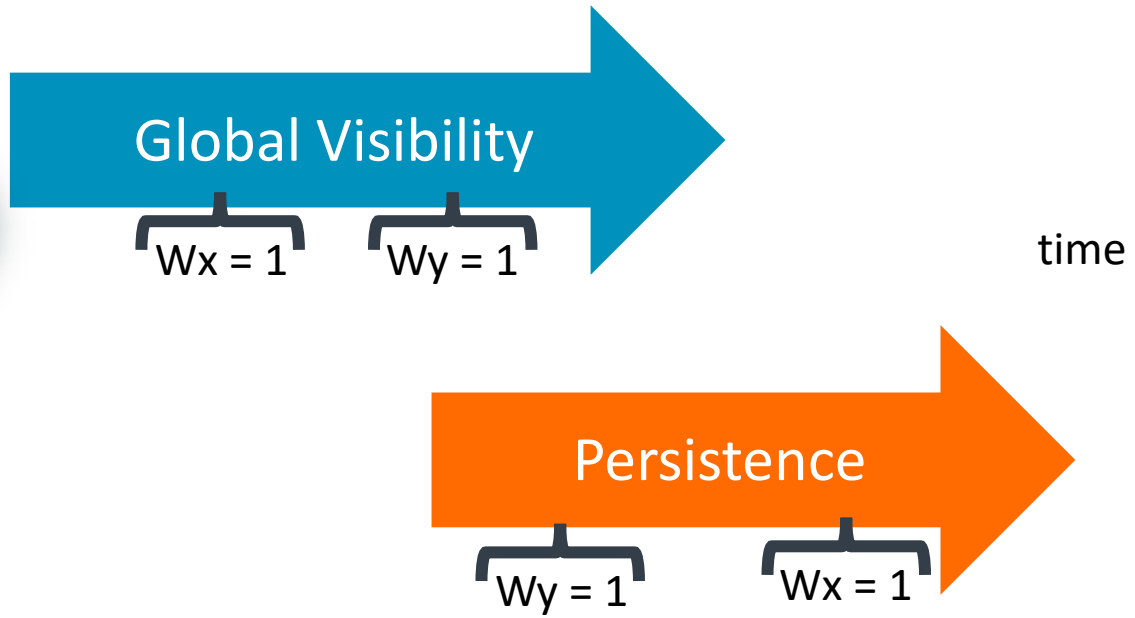
P0
STR W0,[X1]
STR W2,[X3]

Thread 0
a: Wx = 1
po
b: Wy = 1



P0
STR W0,[X1]
DMB.ST
STR W2,[X3]

Thread 0
a: Wx = 1
dmb
b: Wy = 1

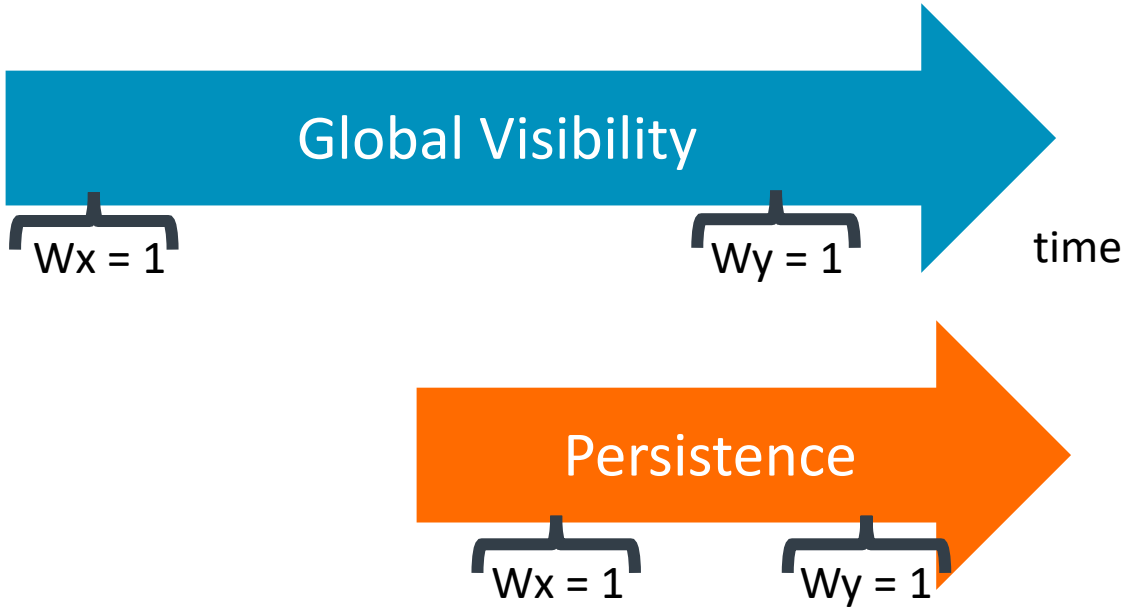


arm

Enforcing Persist Order

P0
STR W0,[X1]
DC.CVAP [X1]
DSB
STR W2,[X3]

Thread 0
a: Wx = 1
dmb
pers
b: Wy = 1

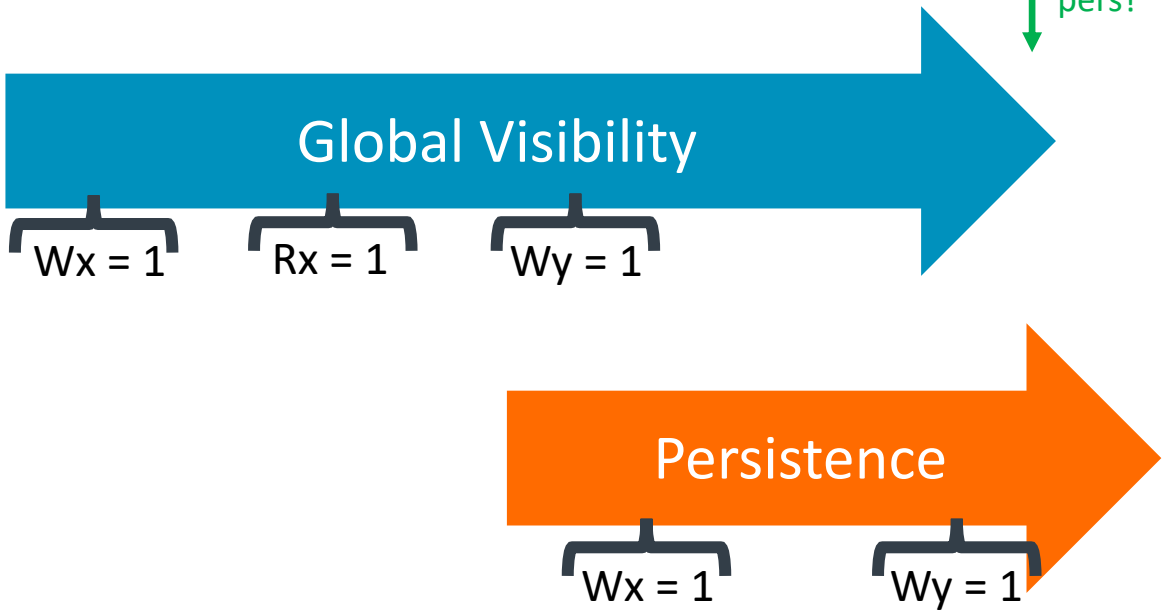


P0
STR W0,[X1]
DC.CVAP [X1]
DSB

P1
LDR W0,[X1]
DMB
STR W2, [X3]
DC.CVAP [X3]
DSB

Thread 0
a: Wx = 1
pers?

Thread 1
rfe
b: Rx = 1
dmb
c: Wy = 1
pers?

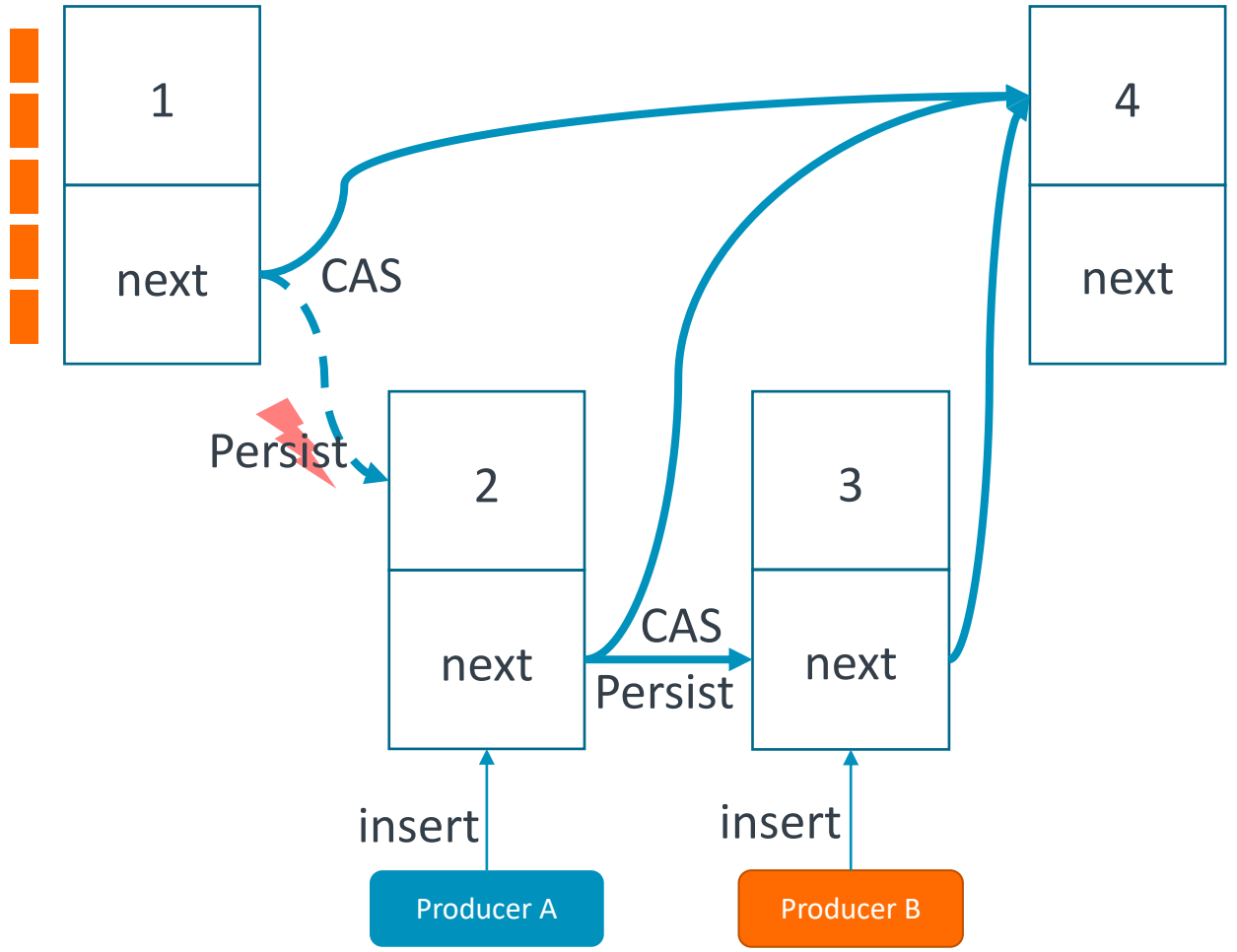


DSB: Data Synchronization Barrier

arm

Challenge: Data Loss in Concurrent Linked List

```
1. if(CAS(&last->next, next, node)) {
2.   Persist(&last->next);
3.   DSB
4. }
```



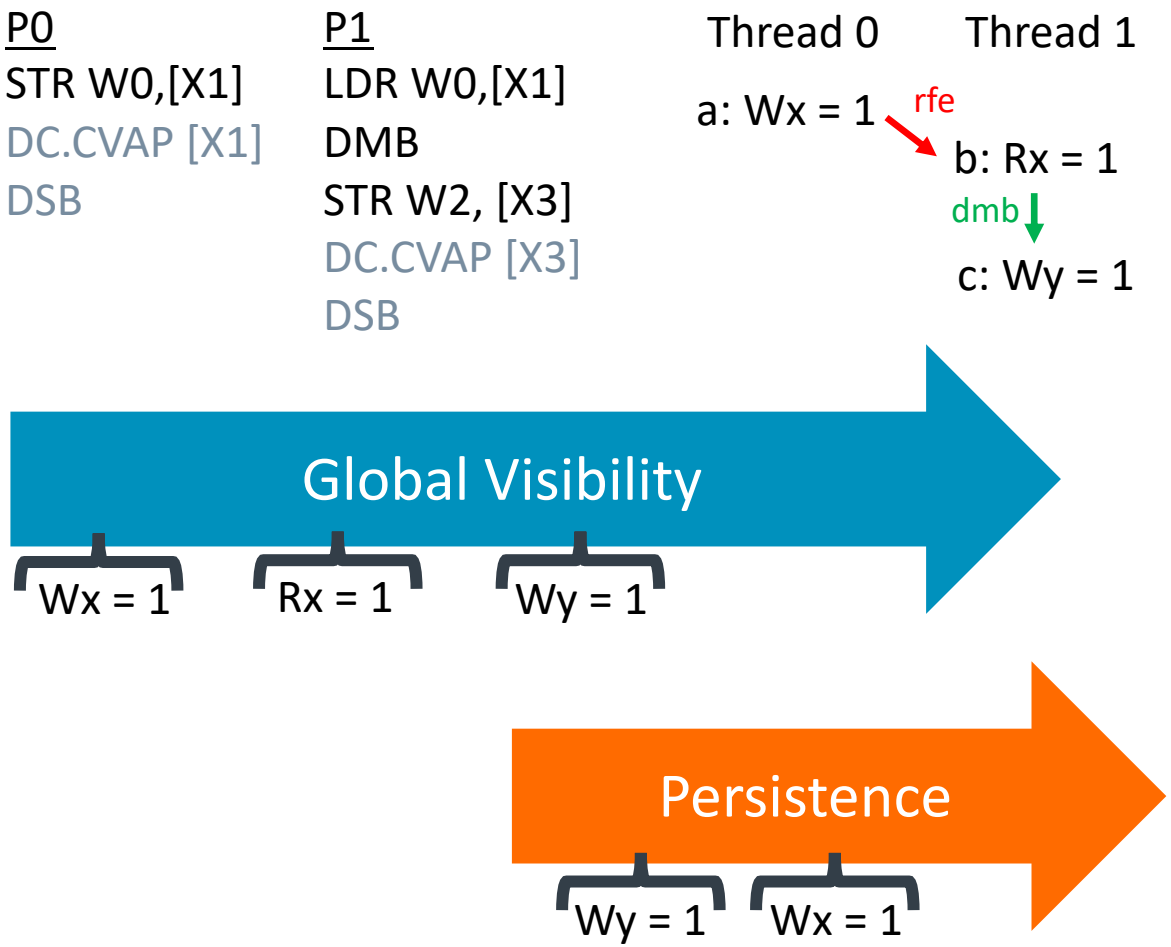
- Producer B observes A's updates, but cannot / does not enforce the persists
- The inter-thread "read of non-persistent writes" problem

arm

Solution

- Basic idea: delay consumer's persist operation until producer's persist operation is done

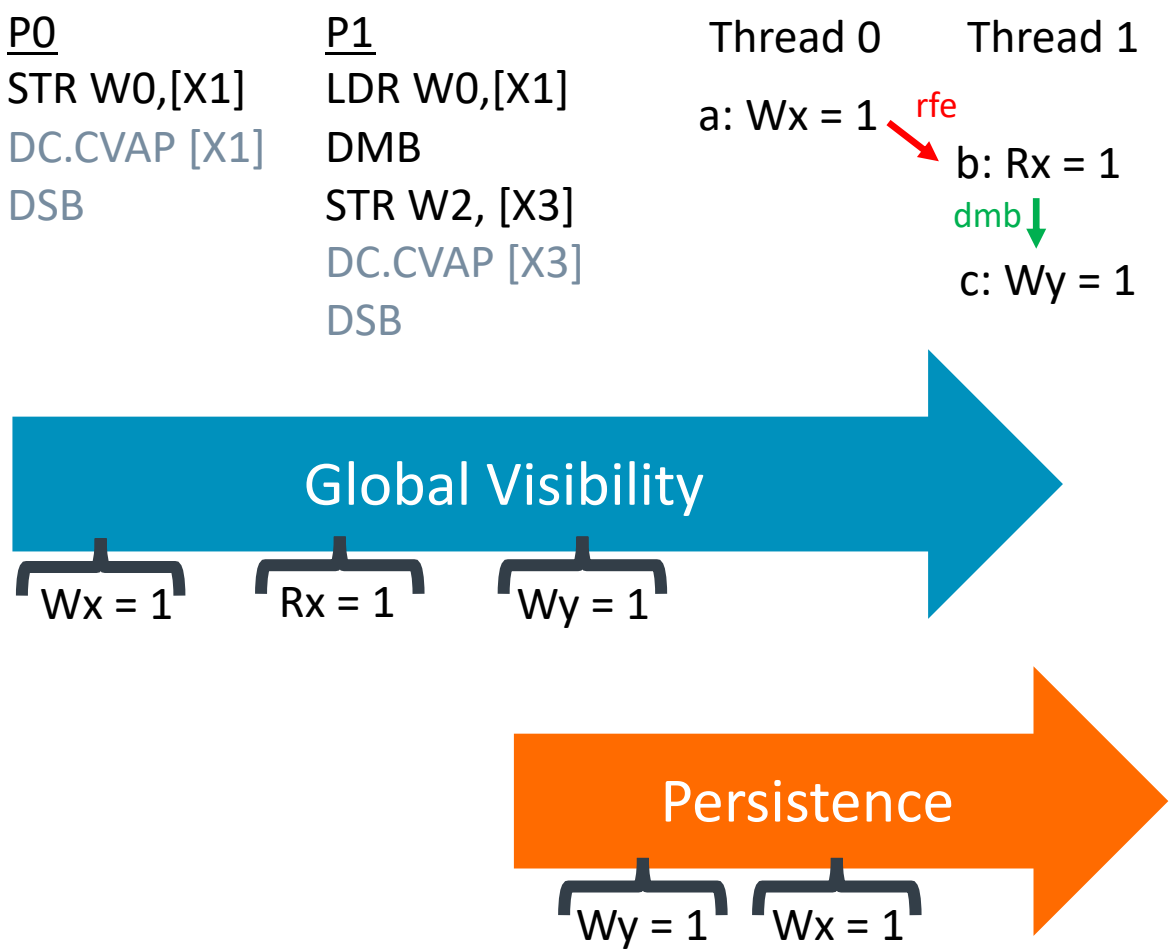
- Various arch options
  - Delay producer's visibility until persistence is done



arm

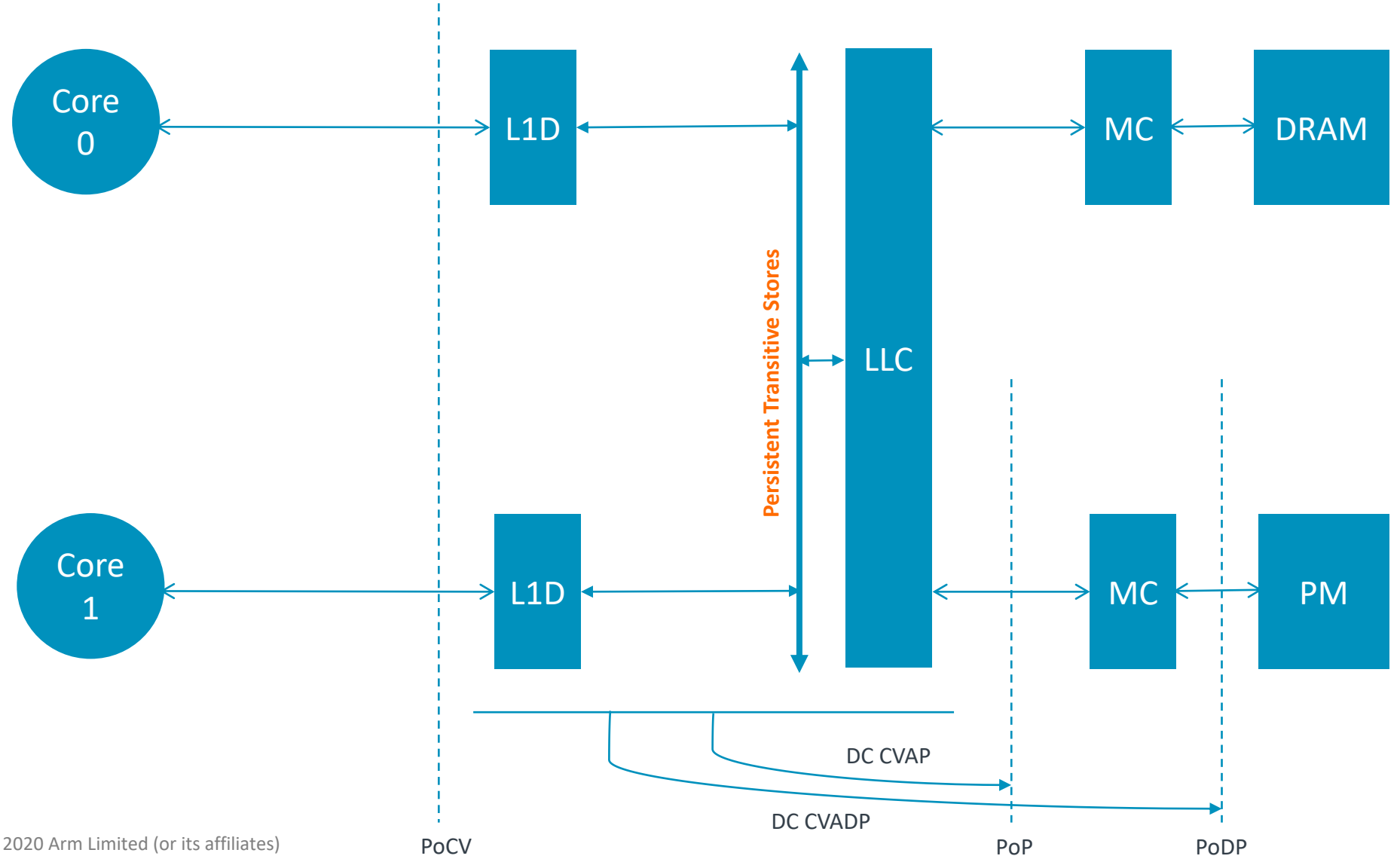
Solution

- Basic idea: delay consumer's persist operation until producer's persist operation is done
- Various arch options
  - Delay producer's visibility until persistence is done
- New instructions for combining persist and store for synchronizing stores
  - A variant is to detect stores to persistent regions at address translation and automatically persist



arm

Persistent Transitive Stores to Synchronize Visibility & Persistency



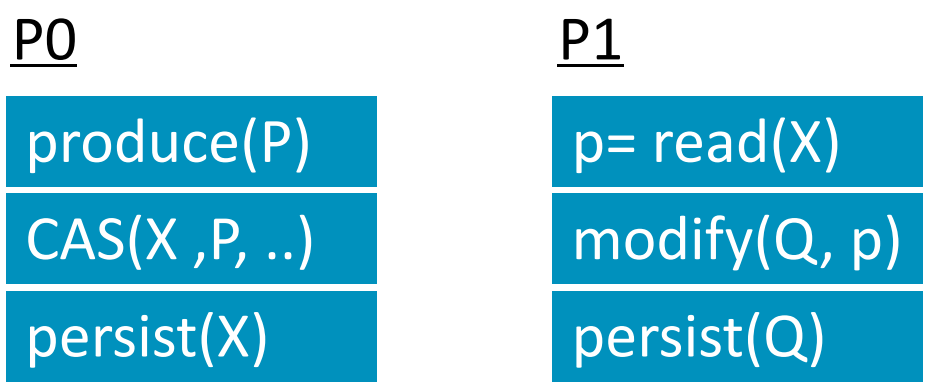
PoCV: Point of Concurrent Visibility

PoP: Point of Persistence

PoDP: Point of Deep Persistence

In Software

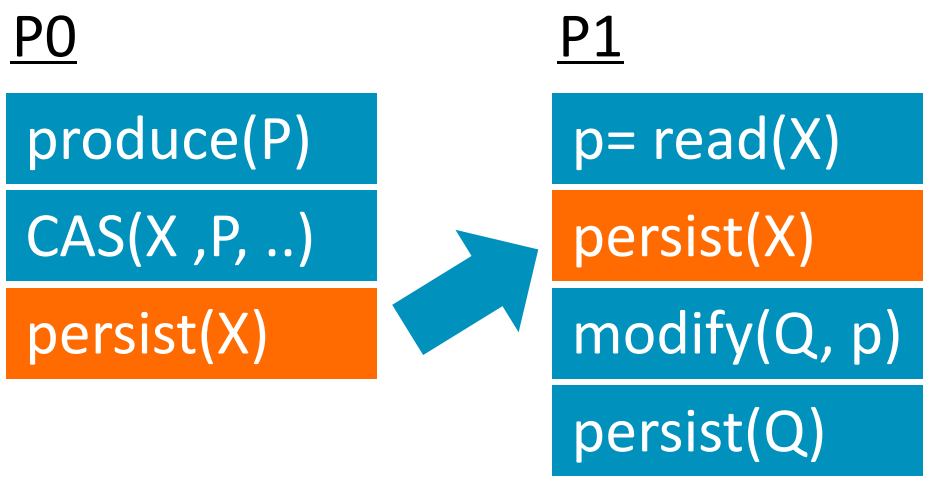
- Readers persist all locations read -> bloat, slow
- Tell reader to persist / to wait
- Single out-of-band location -> scalability??
- Multiple out-of-band locations -> hello mini-STM ?!?
- Borrow payload -> steals payload bits



arm

In Software

- Readers persist all locations read -> bloat, slow
- Tell reader to persist / to wait
- Single out-of-band location -> scalability??
- Multiple out-of-band locations -> hello mini-STM ?!?
- Borrow payload -> steals payload bits



arm

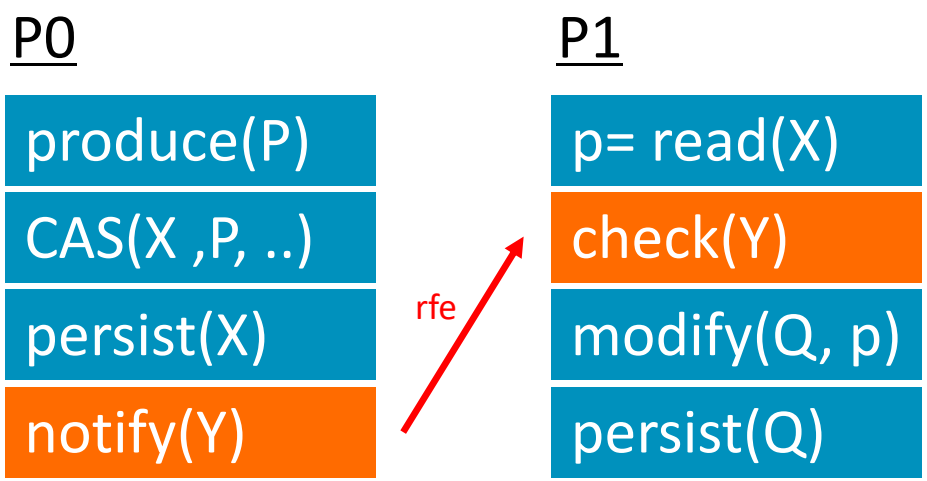
In Software

- Readers persist all locations read -> bloat, slow
- Tell reader to persist / to wait
- Single out-of-band location -> scalability??
- Multiple out-of-band locations -> hello mini-STM ?!?
- Borrow payload -> steals payload bits

"There is always a software way around the problem if you are aware of it, but that is not a reliable solution. The best solution is to design processors so that a load from a persistent memory location will only see data that is persistent."

- Mario Wolczko and Bill Bridge (Oracle)

Source: https://medium.com/@mwolczko/non-volatile-memory-and-java-part-2-c15954c04e11



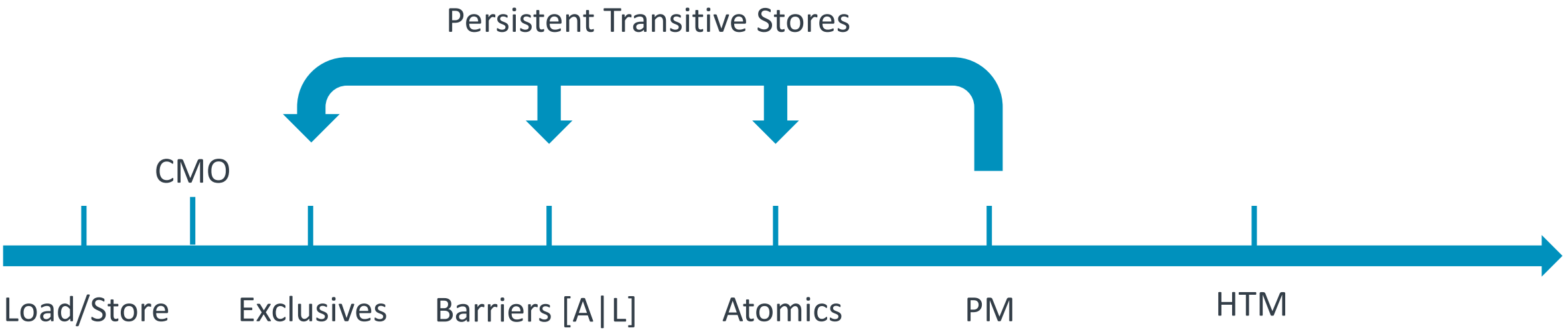
arm

Summary: Persistent Transitive Stores

- Persistent memory introduces a new level of reasoning
- Arm ISA extensions for flushing to *point of (deep) persistence:* DC CVA[D]P
  - Arm v8.2 DC CVAP, Arm v8.5 DC CVADP

- Simple persist operations do not allow transitive ordering of persists
- Tricky case closing store of lock-free section
- Extending the ISA (and uarch) to synchronize *visibility and persist* orders

Architectural Support for Memory



CMO: Cache Maintenance Operation

PM: Persistent Memory

HTM: Hardware Transactional Memory

arm

Use Cases for Persistent Atomics

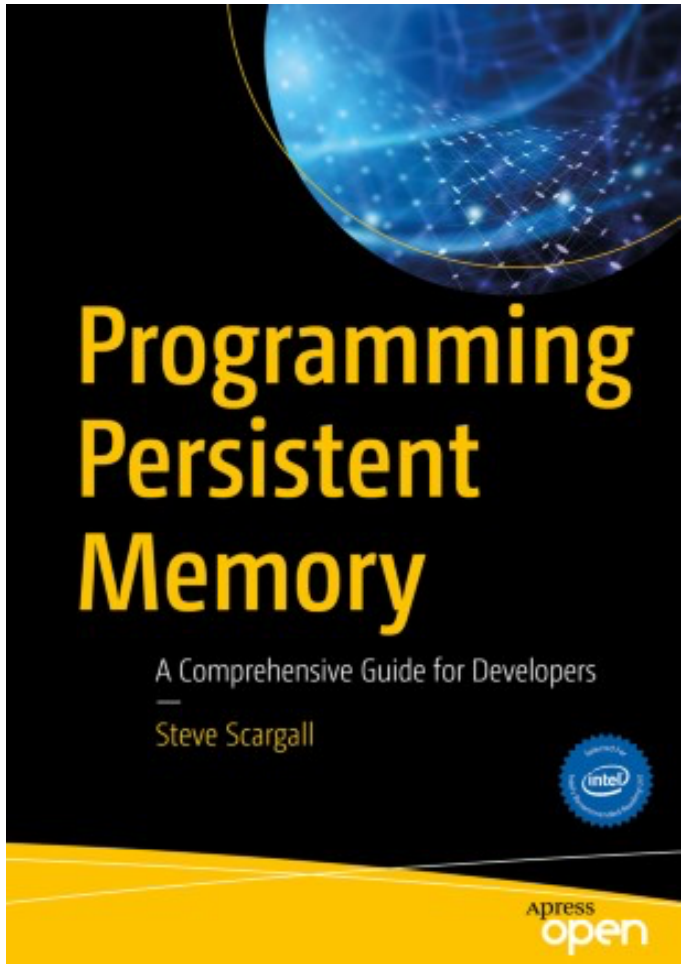
- **Lock-free data structures** for filesystems, databases, k-v stores, and caching tiers

Data Structures	Example Implementations	Applications
B+ Trees	BZTree, and Crab-tree, Masstree, noveLSM, FAST-FAIR B+-Tree, WORT, FPTree, NV-tree, WB+-Tree, B+-Tree, CDDS B-Tree	**Filesystems and databases:** Microsoft Hekaton, HANA, Timesten, SQLite, LevelDB/RocksDB/Cassandra (LSM Tree), NOVA, ext4-DAX
Hashmaps	NVC-hashmap, CCEH, LevelHashing, Dali, PFHT	**Key-value stores:** Redis, Memcached, Pelikan
Queues	LogQueue	**Persistent log queues:** Oracle DB, SQL server
Skiplists	NV-skiplist	**Databases and KV stores:** MemSQL

- **Synchronization primitives** in languages, libraries, runtimes and compilers for PM

Software Stacks	Synchronization Primitives	Examples
Applications	Locks, lock-free atomics, STM	MySQL, Tomcat, Nginx (sync intensive)
Runtimes	• Interpret language functions to runtime builtin implementations • Concurrent GC in runtime implementations	• *Synchronized* in Java to intrinsic lock or monitor lock • v8, OpenJDK, go-runtime
Kernels	spinlock, ticket spinlock, mcs queued spinlock, clh queued spinlock mutex, semaphore, reader-writer lock, read-copy-update	Linux kernel
Languages	Locks and atomics: Java, C11/C++, C#, Golang, JS, NodeJS, WASM; TM: C/C++	*Synchronized* in Java/C++, *lock* in C#
Libraries	mutexes, semaphores	pthreads, Windows threads
Compilers	atomics in languages get mapped to compiler builtin implementations	GCC __atomic_ Builtins, LLVM __atomic_
ISA	PCAS[A|L], PSWP[A|L], P[LD|ST]ADD[A|L]	Persistent atomics

Concurrency on Persistency Memory : It's Complicated



" We also explain that atomic operations cannot be used inside a [PMDK] transaction while building lock-free algorithms without transactions. ***This is a very complicated task if your platform does not support eADR.***"

Source: https://link.springer.com/chapter/10.1007/978-1-4842-4932-1_14

arm

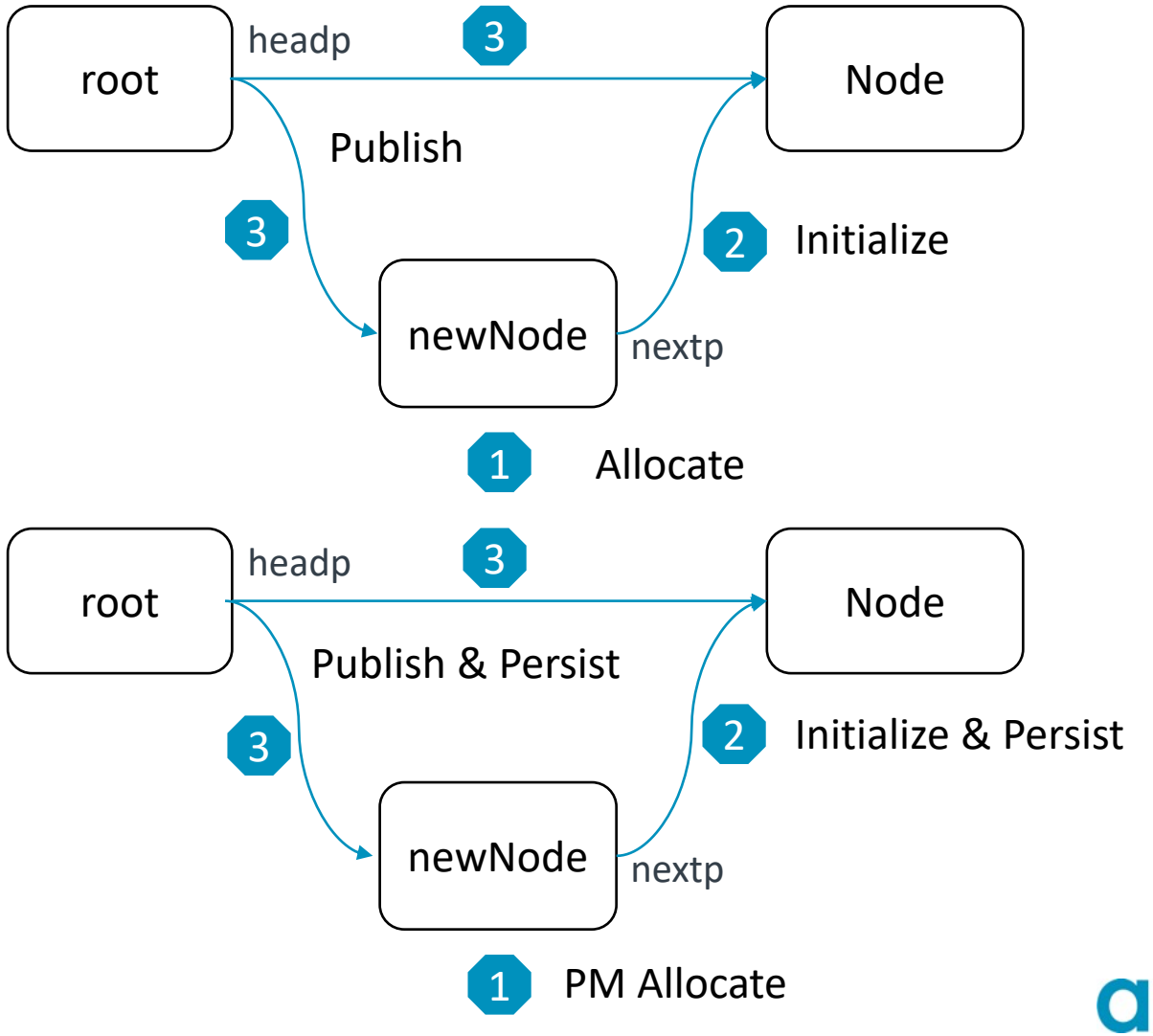
Memory Consistency

Why should you care about memory consistency for sequential programs?

Example: Adding a Node to a Linked List

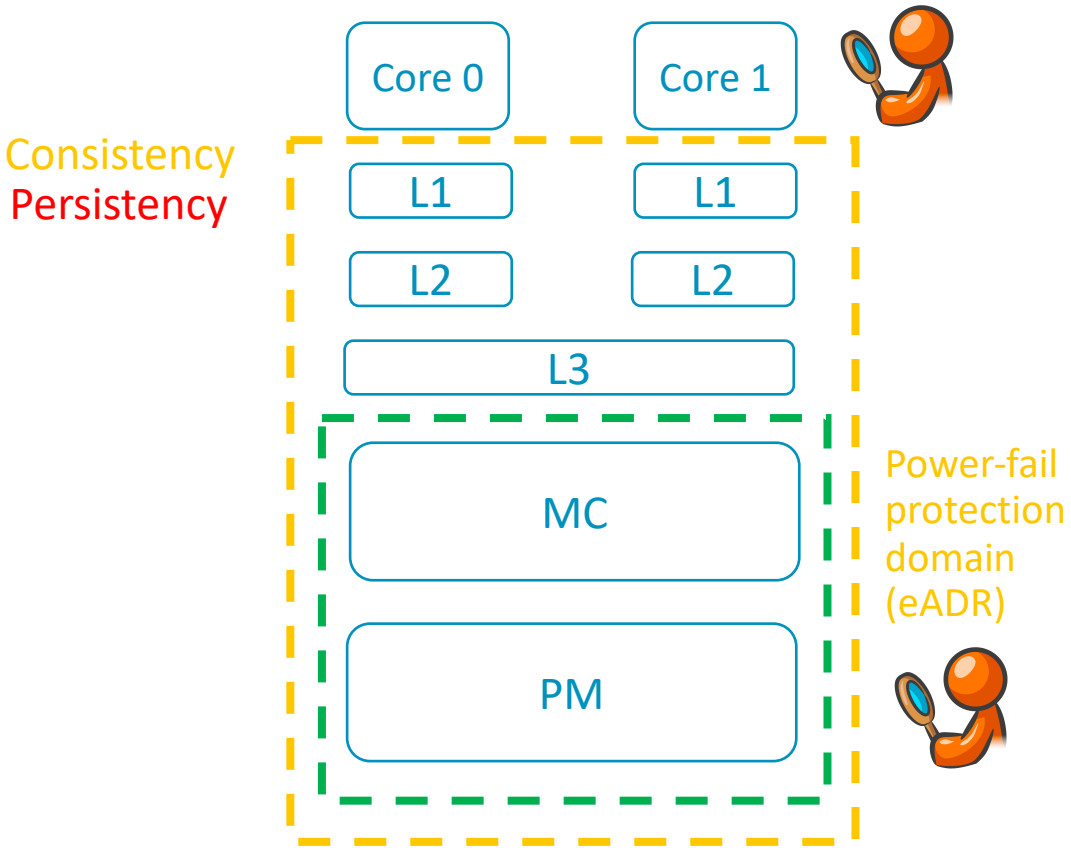
```
1  // Add a node to a linked list
2  void
3  addnode(struct root *rootp, int data)
4  {
5      struct node *newnodep;
6      if ((newnodep = calloc(1,
7          sizeof(struct node))) == NULL)
8        fatal("out of memory");
9      newnodep->data = data;
10     newnodep->nextp = rootp->headp;
11     rootp->headp = newnodep;
12 }
```

```
2  void
3  addNode(struct root *rootp, int data)
4  {
5      struct node *newnodep;
6      if((newnodep = pm_calloc(1,
7          sizeof(struct node))) == NULL)
8        fatal("out of memory");
9      newnodep->data = data;
10     newnodep->nextp = rootp->headp;
11     pm_flush(newnodep,
12         sizeof(struct node));
13     pm_fence();
14     rootp->headp=newnodep;
15     pm_flush(newnodep,
16         sizeof(struct node));
17     pm_fence();
18 }
```



arm

eADR Simplifies Persistent Programming, but Not Sufficient



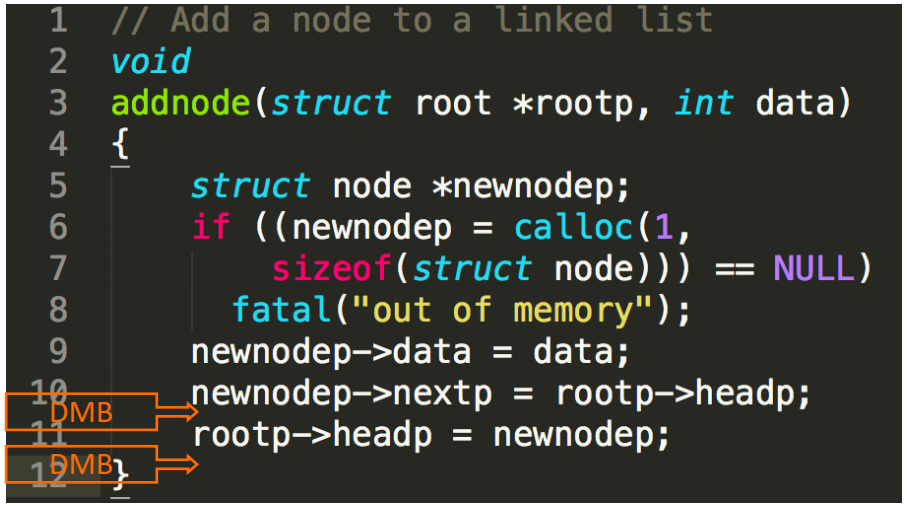
- CPU cache hierarchy in the power-fail protection domain (PoP)
  - Contents will be saved upon power failure
- Persistency == Consistency
  - Concurrent programs ✓
  - Is that sufficient for sequential programs?
- Globally visible stores in the cache hierarchy will be persistent too
  - No need to DC CVAP
  - No need to use barriers?
    - No, simple sequential programs need to reason about memory consistency

Arm's Weak Memory Model: W->W Reordering Allowed

P0	P1
str A=1	while(flag==1){};
str flag=0	print A

P1 can read a stale copy of A, as **str flag=0** can get executed before **str A=1**.

Use **DMB** (or stlr) between the two stores on P0 to serialize the two stores.



Even though caches are in the PoP, no need to **PERSIST**, but **DMB** is needed for Arm.

Non-TSO needs DMB for sequential programs correctness with persistent memory. [First DMB]

While not an issue on TSO, barriers are still needed for visibility due to store buffering. [Second DMB]

arm

Solutions – Applications, Compilers, Languages, ISA, or uArch?

- All such sequential programs get patched (w. DMB) to run on systems with PM
- Compilers implement a stricter memory model（such as TSO）if the target architecture has a weaker memory model.
  - By disallowing certain reorderings such as W->W in compiler passes, and by inserting memory barriers (including store releases) in the right places
- Languages provide an option to specify stricter memory models (such as C++), but legacy code will need to be ported to leverage the feature.

SW

HW

- Tighten memory models to TSO, is that an option?
  - Architecture supports TSO extension
  - Stricter microarchitectural implementations that disallow reordering stores
- Extend power-fail protection to store buffers
  - As instructions are committed in order, despite being executed OoO

Barriers Can Be Expensive

OpenJDK inserted barrier

```
@Benchmark
    public void testLargeConstArray(Blackhole bh)
throws Exception {
        int localArrlen = ARR_LEN;
        for (int i = 0; i < LENGTH; i++) {
            Object[] tmp = new Object[localArrlen];
            bh.consume(tmp);
        }
    }
```

DMB.st

DMB ishst

```
0x0000ffff9426dee0: str xzr, [x10]
0x0000ffff9426dee4: prfm pstl1keep, x12, #320
0x0000ffff9426dee8: str w20, [sp]
0x0000ffff9426deec: stp x19, x13, sp, #8
0x0000ffff9426def0: str x14, sp, #24
0x0000ffff9426def4: dmb ishst
0x0000ffff9426def8: ldr x1, sp, #8
0x0000ffff9426defc: bl 0x0000ffff8c826c00
```

DMB can be expensive

Performance on m6g [N1] got improved 30% after removing the store barriers for objects initialization.

CPU	Normally	Remove store barriers
m6g.16xlarge	22805.389 ± 334.201 ns/op [Average]	**15942.065 ± 63.785 ns/op [Average]**

arm

No Barriers Get Inserted by Compilers

On Arm's Weak Memory Models

Source Code

```c
#include <stdio.h>
#include <stdlib.h>

struct Node{
    int data;
    struct Node* next;
};

struct Head{
    struct Node * next;
};

int main() {
    // create a singly linked list
    struct Head * head = malloc(sizeof(struct Head));

    // insert a node
    // allocate
    struct Node* node = malloc(sizeof(struct Node));
    // initialise
    node->data = 1;
    node->next = NULL;
    // publish
    head->next = node;
    // check the value
    printf("%d\n", head->next->data);
}
```

gcc v7.4.0 on Centriq2400

```
00000000004005bc <main>:
  4005bc:       a9be7bfd        stp     x29, x30, [sp,#-32]!
  4005c0:       910003fd        mov     x29, sp
  4005c4:       d2800100        mov     x0, #0x8                        // #8
  4005c8:       97ffffaa        bl      400470 <malloc@plt>
  4005cc:       f9000ba0        str     x0, [x29,#16]
  4005d0:       d2800200        mov     x0, #0x10                       // #16
  4005d4:       97ffffa7        bl      400470 <malloc@plt>
  4005d8:       f9000fa0        str     x0, [x29,#24]
  4005dc:       f9400fa0        ldr     x0, [x29,#24]
  4005e0:       52800021        mov     w1, #0x1                        // #1
  4005e4:       b9000001        str     w1, [x0]
  4005e8:       f9400fa0        ldr     x0, [x29,#24]
  4005ec:       f900041f        str     xzr, [x0,#8]
  4005f0:       f9400ba0        ldr     x0, [x29,#16]
  4005f4:       f9400fa1        ldr     x1, [x29,#24]
  4005f8:       f9000001        str     x1, [x0]
  4005fc:       f9400ba0        ldr     x0, [x29,#16]
  400600:       f9400000        ldr     x0, [x0]
  400604:       b9400001        ldr     w1, [x0]
  400608:       90000000        adrp    x0, 400000 <_init-0x430>
  40060c:       911b2000        add     x0, x0, #0x6c8
  400610:       97ffffa8        bl      4004b0 <printf@plt>
  400614:       52800000        mov     w0, #0x0                        // #0
  400618:       a8c27bfd        ldp     x29, x30, [sp],#32
  40061c:       d65f03c0        ret
```

gcc v5.4.0 on ThunderX2

```
0000000000400610 <main>:
  400610:       a9be7bfd        stp     x29, x30, [sp,#-32]!
  400614:       910003fd        mov     x29, sp
  400618:       d2800100        mov     x0, #0x8                        // #8
  40061c:       97ffff95        bl      400470 <malloc@plt>
  400620:       f9000ba0        str     x0, [x29,#16]
  400624:       d2800200        mov     x0, #0x10                       // #16
  400628:       97ffff92        bl      400470 <malloc@plt>
  40062c:       f9000fa0        str     x0, [x29,#24]
  400630:       f9400fa0        ldr     x0, [x29,#24]
  400634:       52800021        mov     w1, #0x1                        // #1
  400638:       b9000001        str     w1, [x0]
  40063c:       f9400fa0        ldr     x0, [x29,#24]
  400640:       f900041f        str     xzr, [x0,#8]
  400644:       f9400ba0        ldr     x0, [x29,#16]
  400648:       f9400fa1        ldr     x1, [x29,#24]
  40064c:       f9000001        str     x1, [x0]
  400650:       f9400ba0        ldr     x0, [x29,#16]
  400654:       f9400000        ldr     x0, [x0]
  400658:       b9400001        ldr     w1, [x0]
  40065c:       90000000        adrp    x0, 400000 <_init-0x430>
  400660:       911c4000        add     x0, x0, #0x710
  400664:       97ffff93        bl      4004b0 <printf@plt>
  400668:       52800000        mov     w0, #0x0                        // #0
  40066c:       a8c27bfd        ldp     x29, x30, [sp],#32
  400670:       d65f03c0        ret
  400674:       00000000        .inst   0x00000000 ; undefined
```

"I have thought about this in the past and for (A) if they are allowed to re-order at all then this is a problem regardless of whether they do or not (B) **can we reliably detect the type of data structure being used so that we can insert barriers automatically, I would say in some cases yes but in others no. The uncertainty will be the problem**."

NOTES: Both the compiler and the CPU can reorder stores. Barriers prevent CPU from reordering. However, barriers are not designed to prevent compilers from reordering, and should not be used for this purpose due to runtime artifact.

For the sequential program discussed, compiler reordering should not break the sequential execution mental model for developers. Therefore no compiler reordering should be allowed, otherwise the compiler optimization would break an important principle. Compiler barriers have been introduced as a result, such as *_barrier()* in the kernel, or *asm volatile ("":::"memory"),* that prevents compiler from reordering, and with no runtime performance impact.

However, OoO CPU store buffers can still reorder stores and should be prevented with CPU barriers.

arm

uArch Implementations vs. Arch Specifications

- DMBST/STLR must be used in the producer thread

- Stores are rarely reordered on TX and HI, a bit more frequent on QC
  - 10 (TX/HI) in 100M, 0.5M(QC) in 100M

P0	P1
str A=1	while(flag==1){};
str flag=0	print A

Testname	Hi1616	QC2400	TX2
LB000	0	0	0
3.LB000	0	0	0
4.LB000	0	0	0
W+RW+RW+RW000	0	0	0
WWC000	0	0	0
IRWIW000	0	0	0
IRIW000	0	0	0
W+RR+WW+RR000	0	0	0
WW+RR+WW+RR000	0	0	0
LB001	0	1	0
3.LB001	0	0	0
4.LB001	0	0	0
W+RW+RW+RW001	0	1	0
WWC001	0	1	0
IRWIW001	0	1	0
IRIW001	1	1	0
W+RR+WW+RR001	0	1	0
WW+RR+WW+RR001	1	1	0
LB002	0	0	0
3.LB002	0	0	0
4.LB002	0	0	0
W+RW+RW+RW002	0	0	0
WWC002	0	0	0
IRWIW002	0	0	0
IRIW002	0	0	0
W+RR+WW+RR002	0	0	0
WW+RR+WW+RR002	0	0	0
2+2W000	1	1	1
2+2W001	1	1	1
2+2W002	1	1	1
2+2W003	1	1	1
3.2W000	1	1	1
SB000	1	1	1
3.SB000	1	1	1
2+2W004	1	1	0
3.2W001	1	1	0
SB001	1	1	0
3.SB001	1	1	0
2+2W005	1	1	0
3.2W002	1	1	0
SB002	1	1	0
3.SB002	1	1	0
2+2W006	0	0	0
3.2W003	0	0	0
SB003	1	1	0
3.SB003	1	1	0
2+2W007	0	0	0
3.2W004	0	0	0
SB004	0	0	0
3.SB004	0	0	0

Notes:
Hi = Hi1616, 64C, A72
QC = QC2400, 48C
TX = TX2-9800, 2S*32C*4T

arm

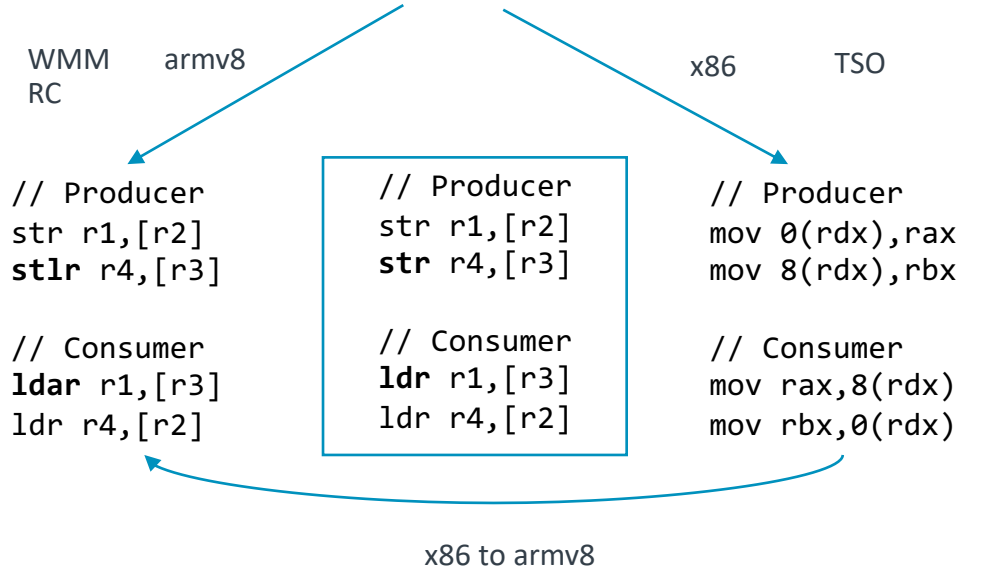
Software Porting from TSO to WMM

Barriers Are Hard to Get Right

- DBT (x86->Arm)
  - Need to add fences (STLR/LDAR, DMB)
    - Hard problem to identify all cases, if not overusing
- Applications porting from TSO -> WMM
  - Recompile, if w. language-level consistency model
  - Add fences (STLR/LDAR, DMB), if not
    - Tedious, easy to overuse or underuse barriers
- Silicon can support x86-TSO and WMM
  - Set an MSR to get x86-TSO dynamically
  - So the code in the middle would run okay

```
// Producer
*data = 1;
atomic_store_explicit(&flag, 1, memory_order_release)

// Consumer
if (atomic_load_explicit(&flag, 1, memory_order_acquire))
  assert(*data != 0);
```

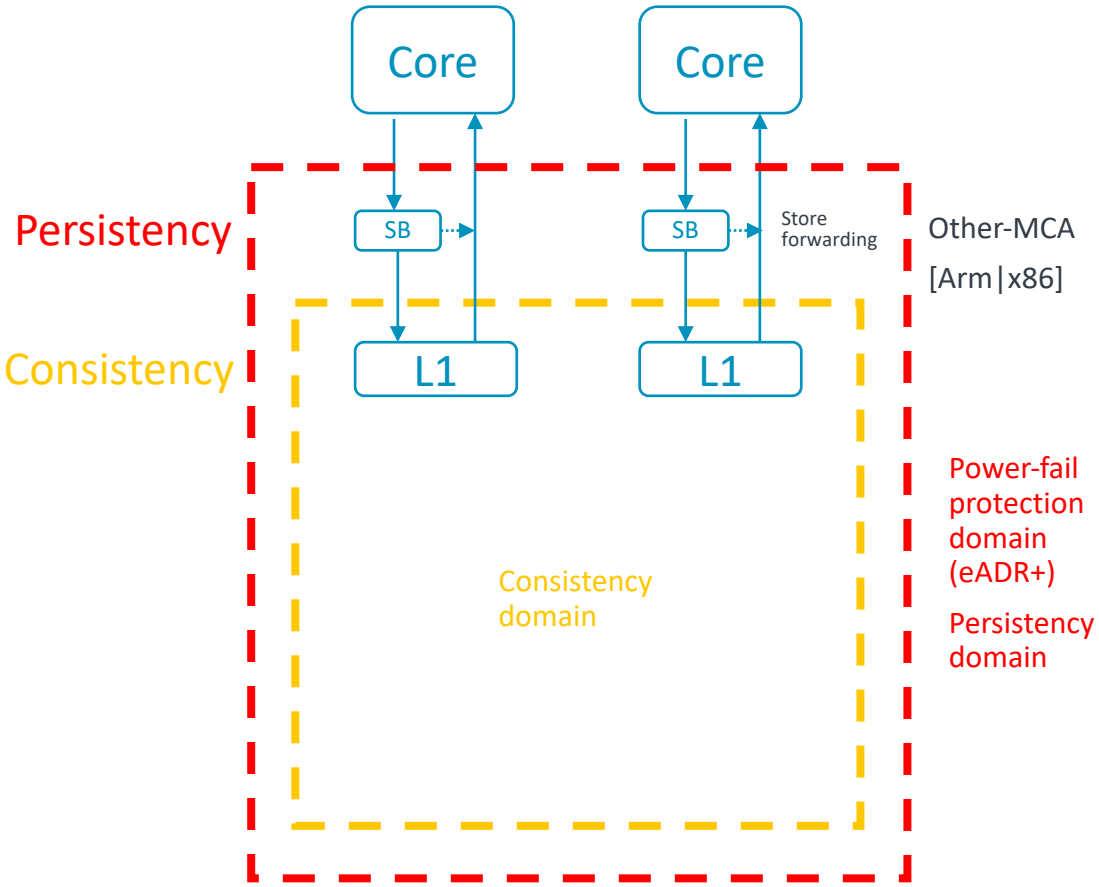


x86 to armv8

A compiler targeting either architecture directly would produce correct code. However, binary translation that does not account for differences in consistency models would lead to the invalid outcome becoming observable!

DBT needs to insert fences, otherwise tricky bugs get introduced. Or, processors support TSO as well.

Extend Power-fail Protection to Store Buffers

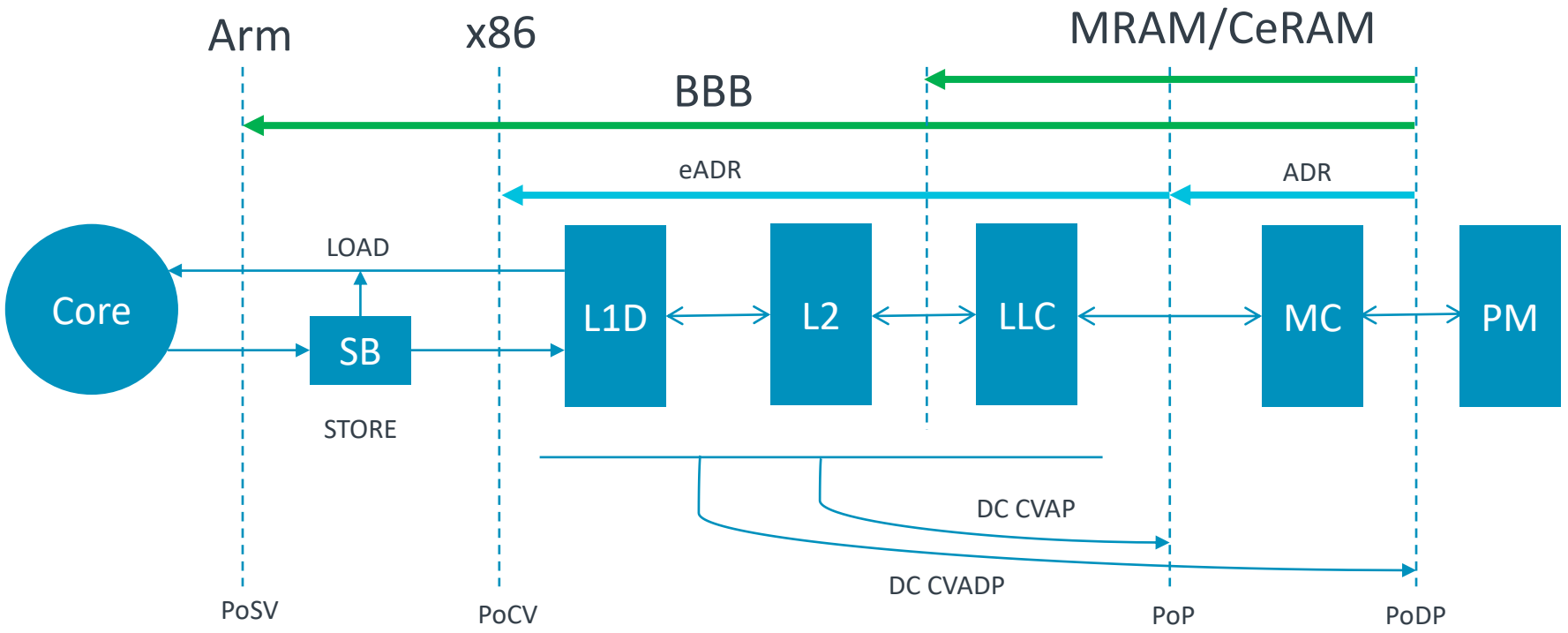


- CPU store buffers in the power-fail protection domain (PoP) too
  - Contents will be saved to PoP
- Stores are executed OoO but committed in order
  - No need to order w. barriers explicitly
- Consistency == Persistency
  - Concurrent programs ✓
- Persistency > Consistency (ahead)
  - Persistency at SB
    - WMM stores get persisted in order, despite can be made visible OoO, barriers would already have been needed for concurrency.
  - Sequential programs continue to execute correctly without barriers
    - Language support may be needed to prevent compiler reordering

arm

Microarchitectural Support to Sync Visibility & Persistency

<= Microarchitectural Support



Architectural Support =>

PoSV: Point of Sequential Visibility

PoCV: Point of Concurrent Visibility

PoP: Point of Persistence

PoDP: Point of Deep Persistence

BBB: Battery-Backed Buffers

arm

Summary

Summary

- Problems
  - Persist ordering across threads
  - Persist ordering within a thread
- Solutions [*]
  - Persistent transitive stores
  - Battery-backed buffers
- Feedback
  - Please send to `william.wang@arm.com`

	Persistent transitive stores	Battery-backed buffers
Performance		
Improvement	Small	Big
Programmability		
Concurrency	Yes	Yes
Failure atomicity	No	No
Persist ordering	Yes	Yes
Crash recovery	No	No
Persistent memory management	No	No
Portability	High	Low
Implementation		
ISA architecture (instructions)	Yes	No
System architecture (registers)	No	Yes
Microarchitecture	Yes	Yes
Interconnect	Yes	No

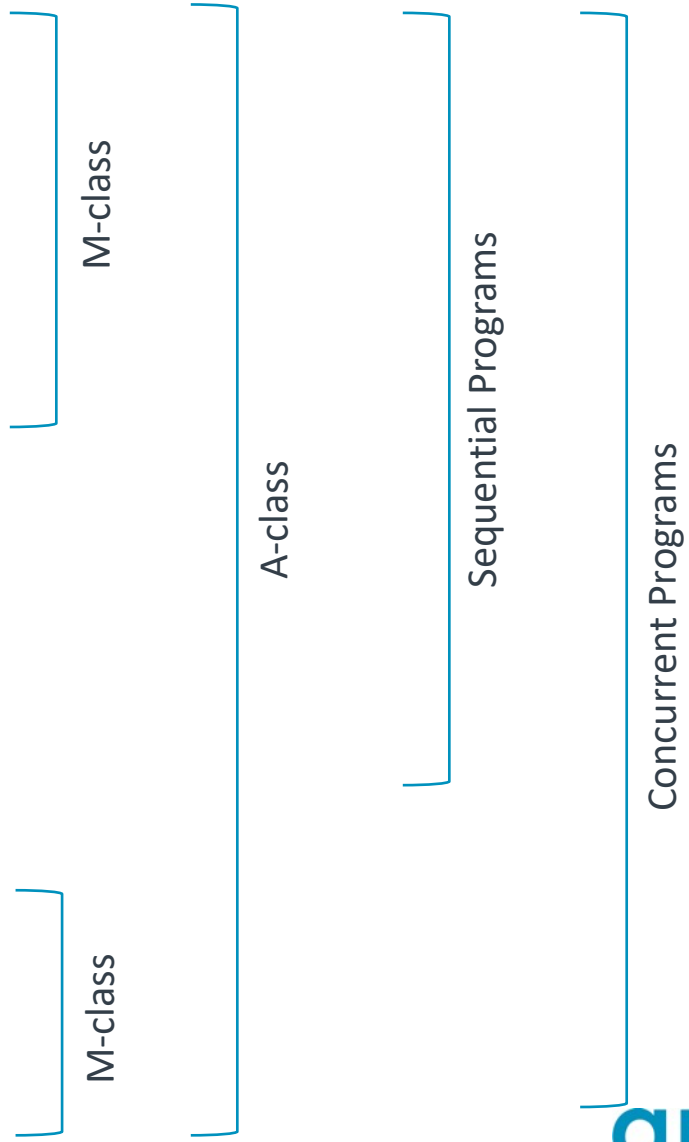
[*] This is research and not (yet) part of a committed architecture/product.

arm

Other Persistent Memory Programming Challenges

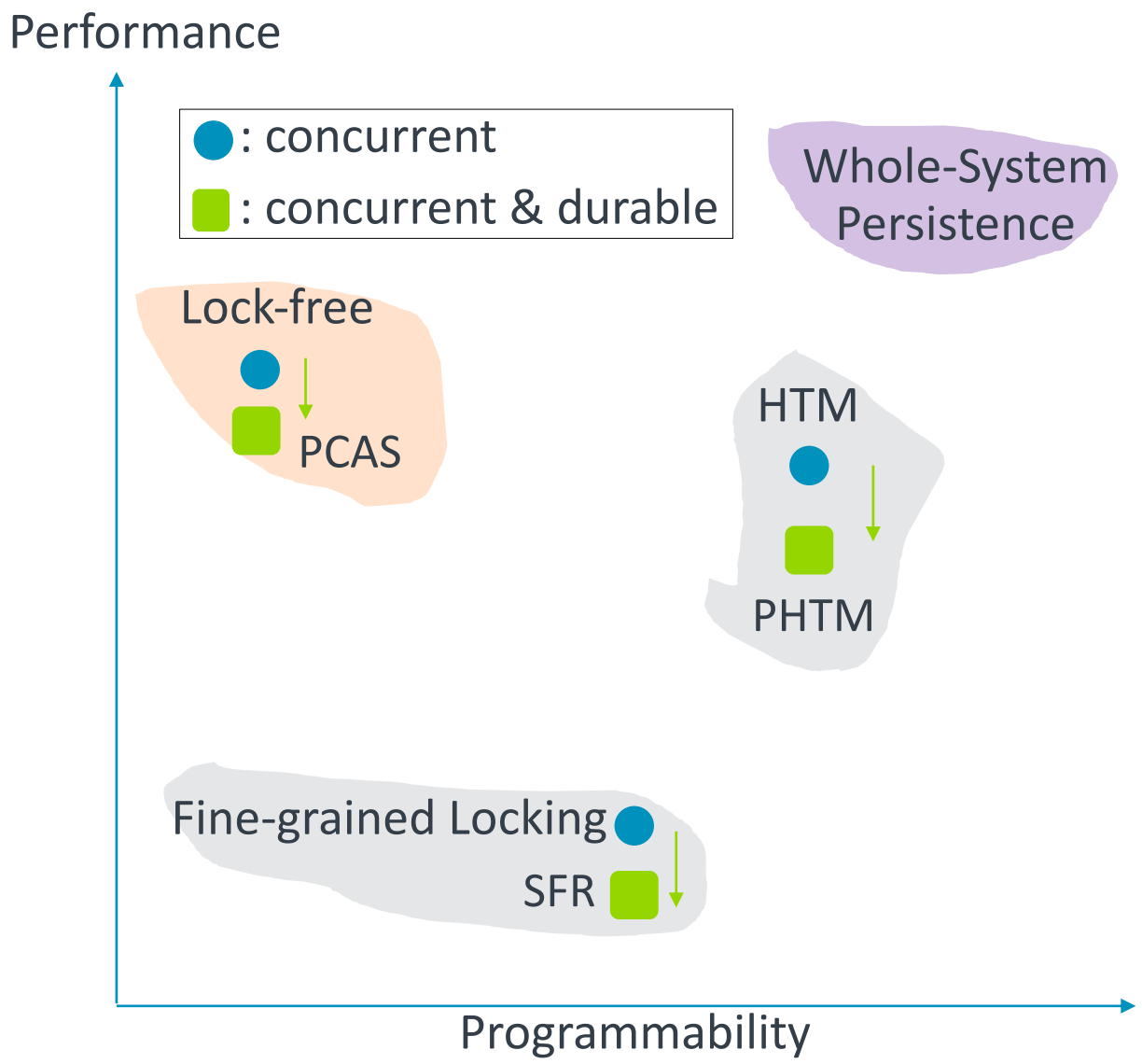
Persistent Memory Programming Challenges

- Persist ordering
  - Relaxed memory persistency models [arch & uarch]
  - Weak memory consistency models [sw & arch] – with eADR
- Failure atomicity
  - PSTM [sw]
  - HW logging [uarch & arch]
- Addressing/crash recovery
  - Persistent pointers [sw & arch]
  - Pointer swizzling at crash recovery [sw]
- Persistent memory management
  - Metadata crash consistency, GC [sw]
- Concurrency
  - Persistent transitive stores [arch]
  - PHTM/PSTM [uarch/sw]
  - Locking [sw]



arm

Addressing the Persistent Programming Challenges



Programmability

Persistent CAS as drop-in replacement for CAS [SPAA'19]

WSP for IoT apps [DAC'19, ISPASS'20]

Compiler instrumented failure atomicity for SFR [PLDI'18]

Persistent HTM as drop-in replacement for HTM [US Patent 10,445,238 B1]

Performance

Relaxed memory persistency models [ISCA'20]

Hardware accelerations – BBB[HPCA'21], HW logging

Software optimizations

arm

Languages Support for Persistent Memory

Language	Applications	Extensions	Failure atomicity	Addressing	Memory Management	Concurrency
C	Oracle DB, SAP HANA, MS Hekaton, Redis	NVM-Direct (Oracle), PMDK (Intel)	STM	Fat pointers	metadata crash consistent	Locking
C++	SQL Server	PMDK (Intel), STL containers, ATLAS(HPE)	STM CS-based FASE	Fat pointers (P0773R0)	metadata crash consistent	Locking
Go	Kubernetes, Docker, Redis	Go-pmem (VMWare)	STM	Pointer swizzling at recovery	pnew, pmake, GC, heap metadata crash consistent	Locking
Java	Cassandra, ActiveMQ	OpenJDK JEP-352, mashona (RedHat), PCJ (Intel)				

More info: William Wang et.al. , Language Support for Memory Persistency, IEEE MICRO Top-picks 2019

OS Support for PM - Persistent Memory Objects

Addressing Protection, Translation and Persistence Holistically, along w. PM & Capabilities

Virtual Memory

YES

- Same VA to apps, ease of prog.
  - No overlays to move data from storage to memory
- Illusion of large PA to all apps
- VA >> PA w. demand paging

NO

- High translation overheads w. TB NVMM
- Page tables won't survive reboots
- Sharing across processes
- Page granularity protection, mapping and migration

Single-level Store

YES

- Objects survive reboots
- No MM <-> FS serialization/deserialization
- No fixed page granularity translations
- Fine-grained protection
- Inter-process sharing (PGAS)
- No context switch TLB flushes

NO?

- Distributed systems (naming)
- Performance?

Example SAS OS

Single Address Space OS (SAS OS)

- Multics (1965)
- IBM i [OS/400] (1988)
- Opal (1995)
- Twizzler (2020)

Arm Capabilities the Morello Board: https://developer.arm.com/architectures/cpu-architecture/a-profile/morello

arm

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
ধন্যবাদ
תודה

Thanks Stephan Diestelhorst, Richard Grisenthwaite, Nigel Stephens, Robert Dimond, Stuart Biles, David Weaver, Matt Horsnell, Thomas Grocutt, Wendy Elsasser, Nikos Nikoleris, Andreas Sandberg, Joseph Yiu, Rod Crawford, Andrew Sloss, Mitch Ishihara, Dave Rodgman, Gustavo Petri, Jade Alglave, Will Deacon, Alex Waugh, Ola Liljedahl, Magnus Bruce, Bobby Batacharia, Travis Walton, David Bull, Shidhartha Das, Shiyou Huang, Sivert Sliper, Mohammad Alshboul, Mike Filippo, Gagan Gupta, Jay Lorch, Bret Toll, Ben Chaffin, Nagi Aboulenein, Dai Zong, Jonathan Halliday, Hans-J. Boehm, Pedro Ramalhete, Virendra Marathe, and Mario Wolczko for their valuable feedbacks and insightful discussions.