The Coming Persistence Apocalypse

Mario Wolczko
Architect
Oracle Labs
May 20, 2019

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

Overview

- Hardware Technology
- System Architecture
- The promise for software, and challenges to realization
- Some musings about Java

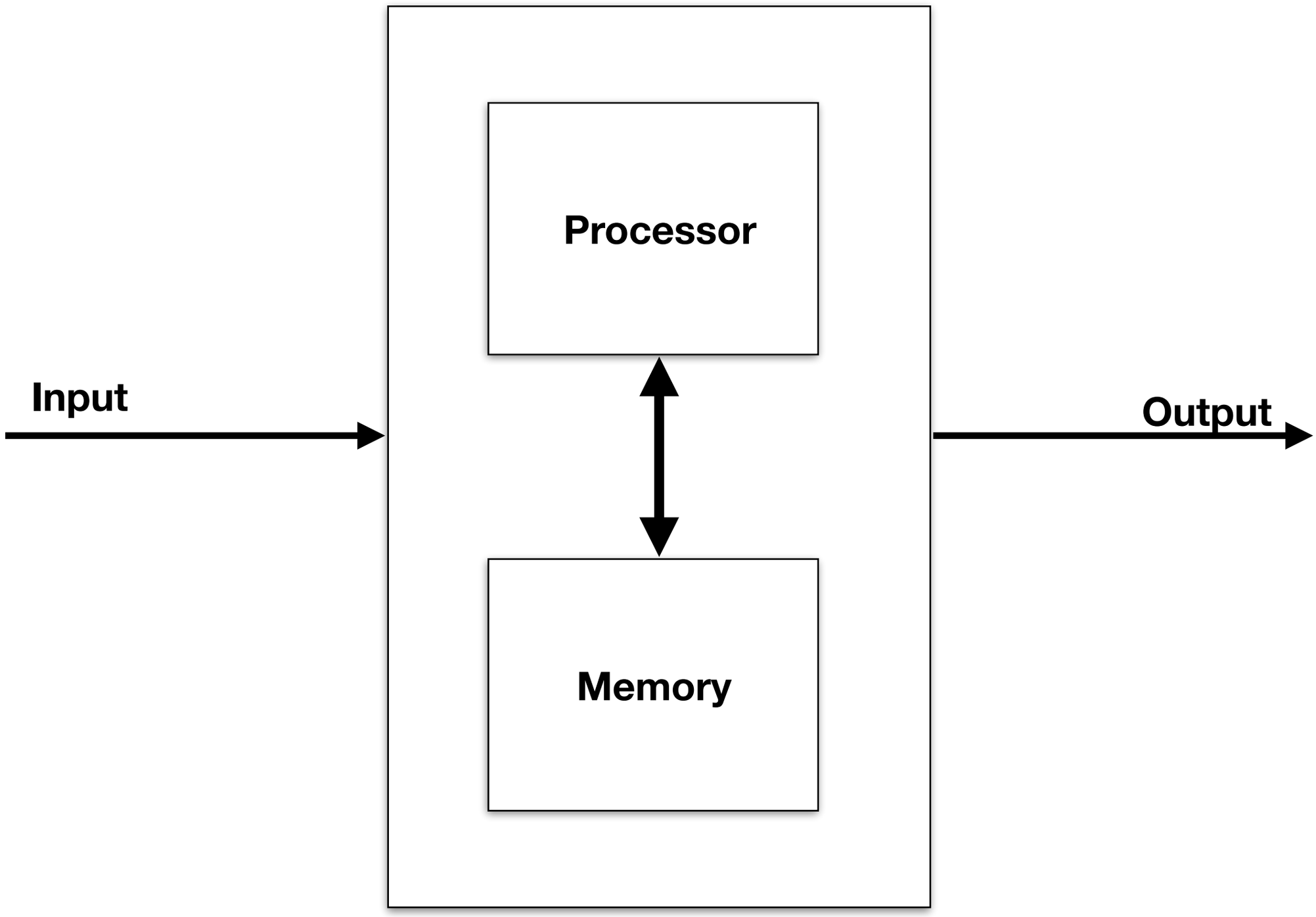
Caveats:

- Many more questions than answers
- I haven’t actually used the Intel hardware

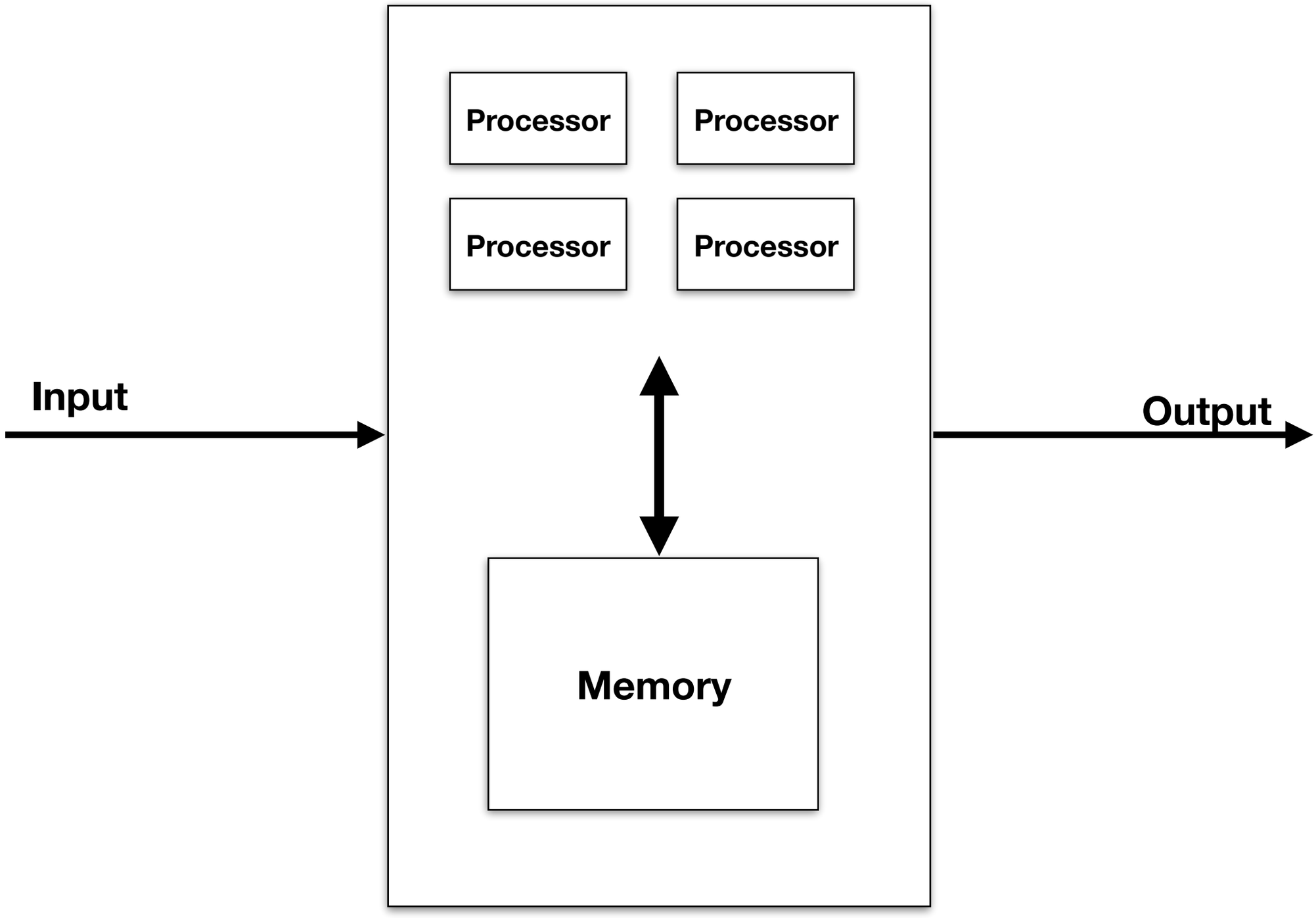
Background:

Hardware technology

Memory



Memory


Processor
Processor
Processor
Processor
Input
Output
Memory

The memory hierarchy

Cheap

Large

Fast

The memory hierarchy

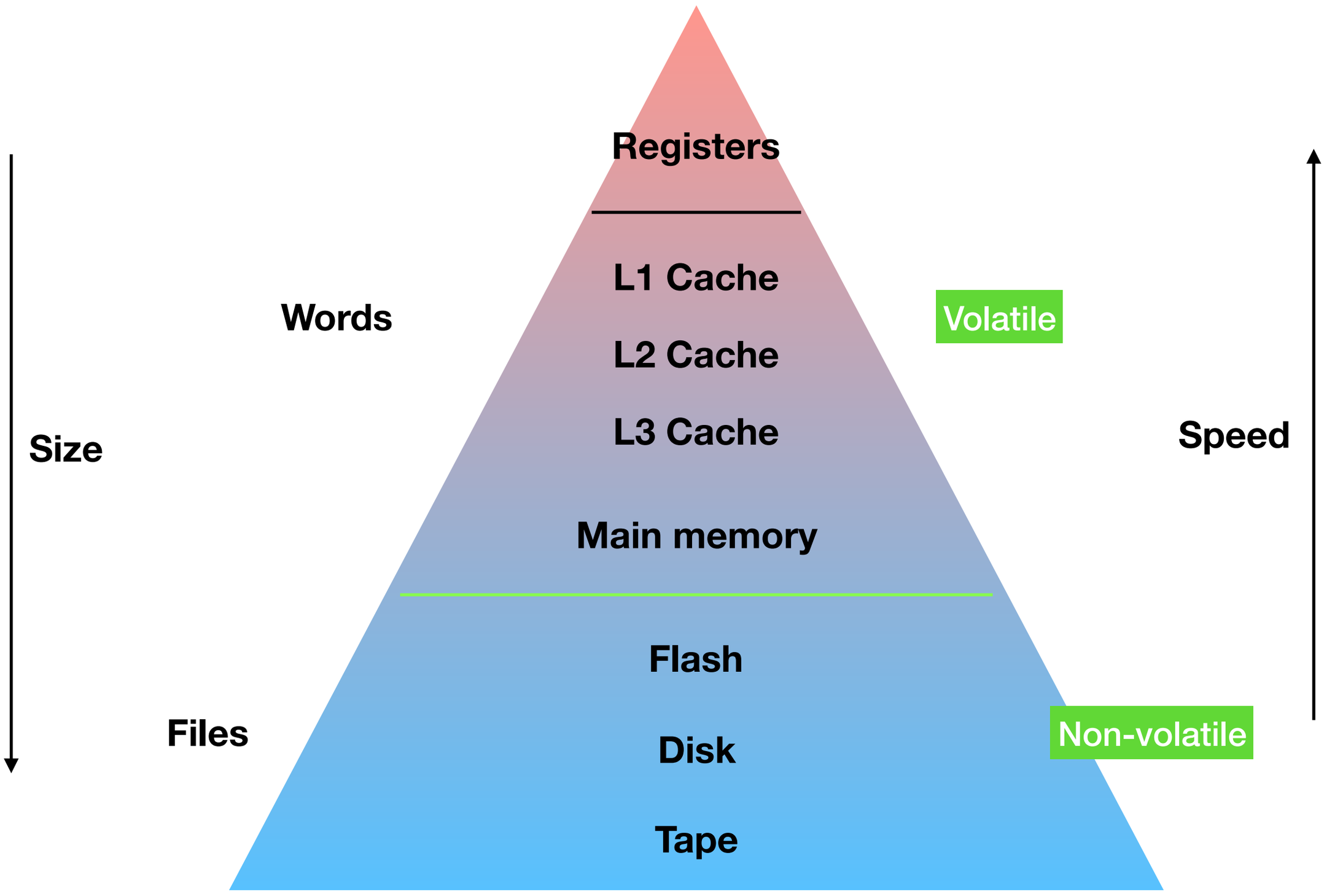
Cheap

Large

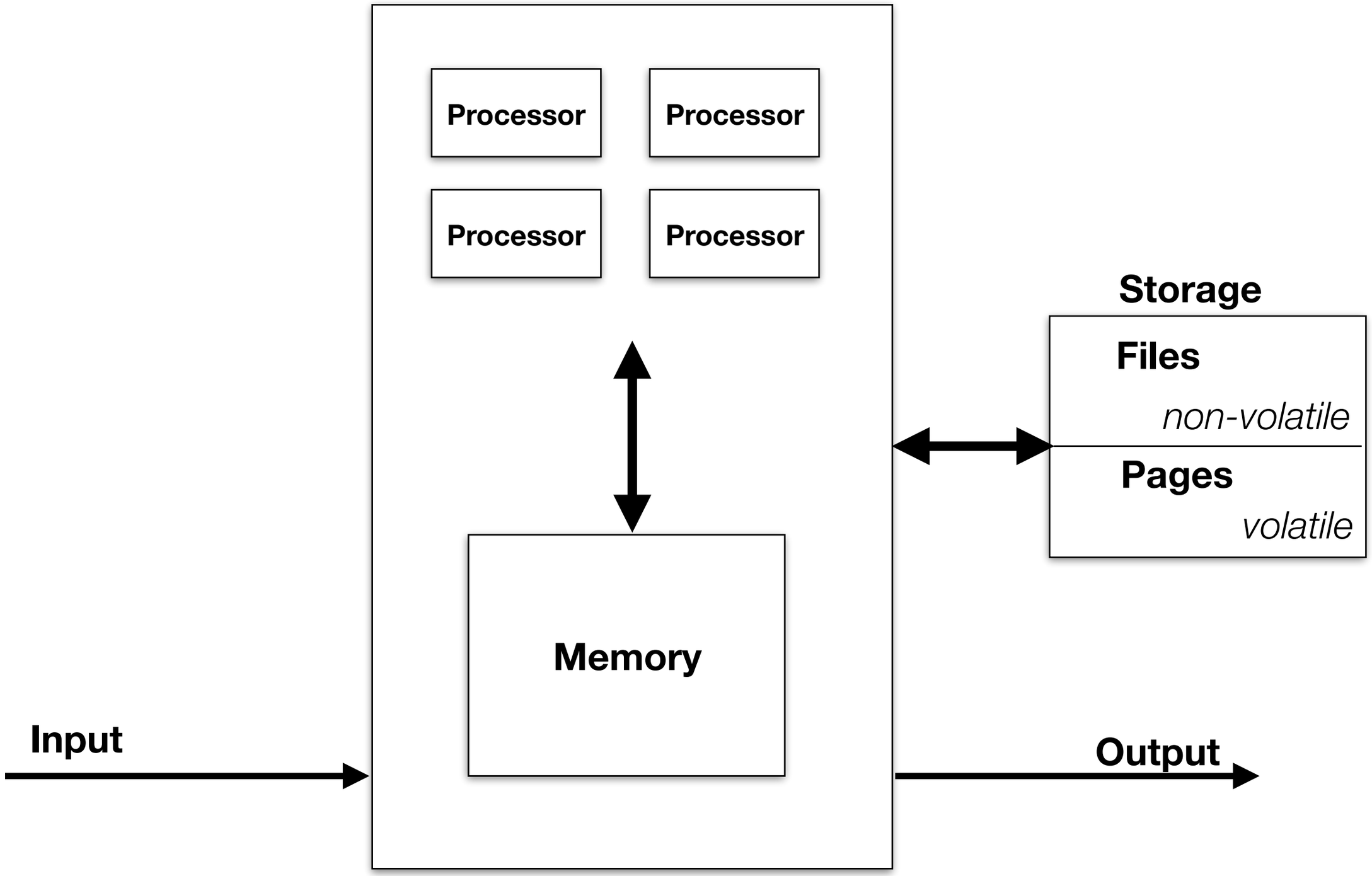
Fast

Pick any two!

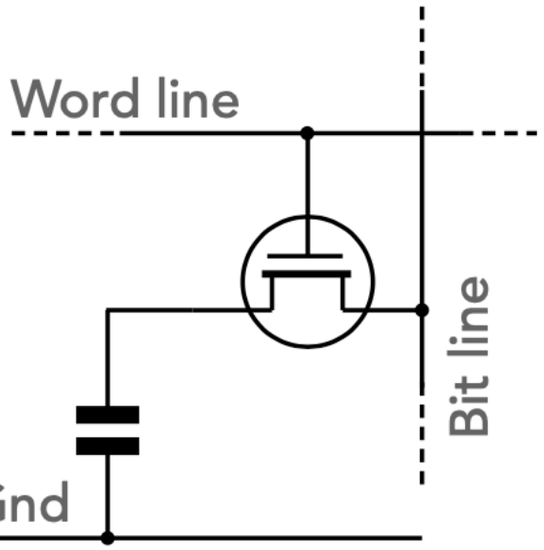
The memory hierarchy



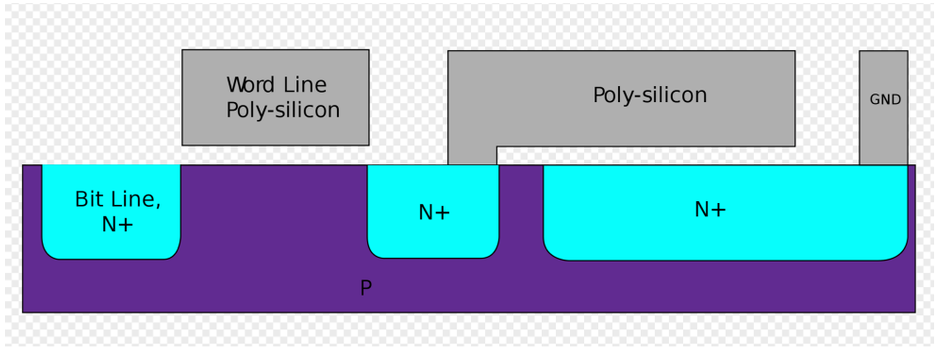
Memory


Processor
Processor
Processor
Processor
Memory
Storage
Files
non-volatile
Pages
volatile
Input
Output

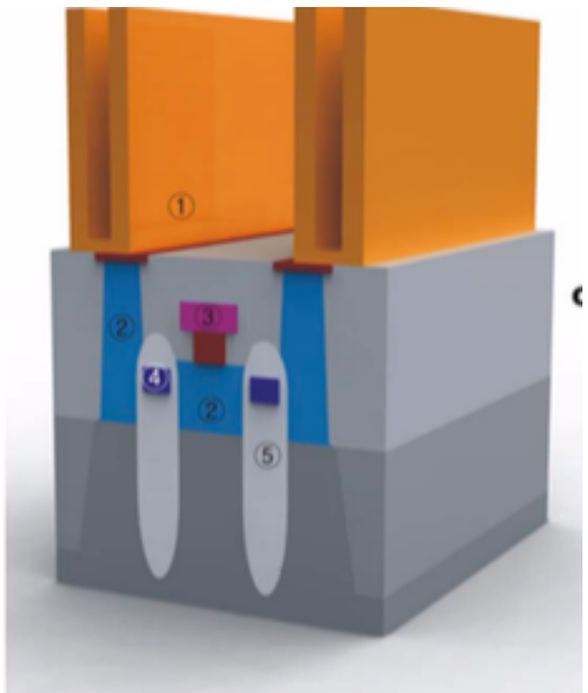
DRAM internals

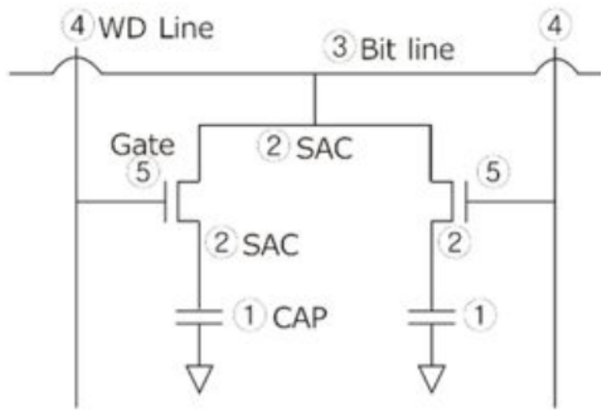


Basic dynamic RAM, DRAM memory cell



Source: Wikipedia





Source DOI: 10.1038/srep02088

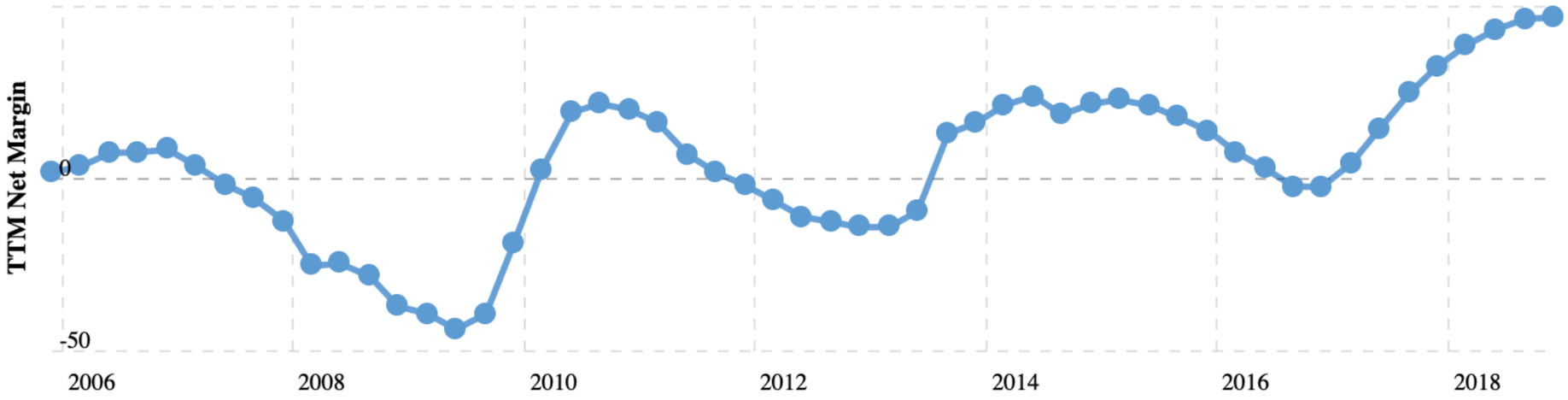
- Capacitor leakage -> refresh

Cct Diagram: https://www.electronics-notes.com/articles/electronic_components/semiconductor-ic-memory/dynamic-ram-how-does-dram-work-operation.php

Memory vs Logic

- Why isn’t the memory on the same chip as the processor?
- How many RAM chips per CPU chip?
- Incompatible processes
- Optimize separately: fast logic vs dense, low-power memory

DRAM is a commodity



Micron Net Margin

Chart: https://www.macrotrends.net/stocks/charts/MU/micron-technology/profit-margins

Storage

- Non-volatile, large, cheap, slow
  - Disk and Flash (aka SSD)
- Transparently used to expand volatile memory: paging
- Or, via a block/file API as non-volatile memory

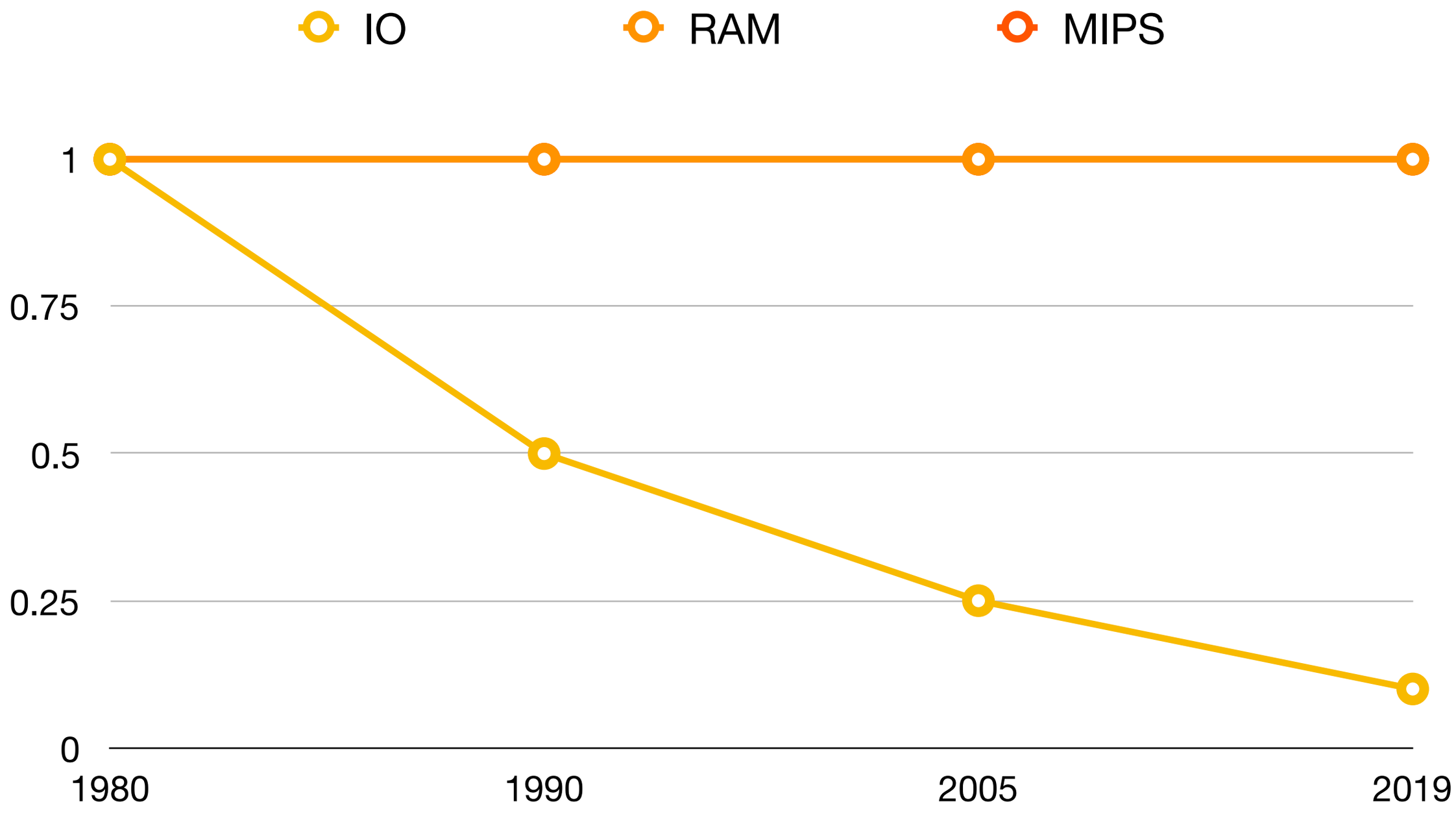
Amdahl's *Second* Law

Amdahl's *Second* Law

- For a balanced system:

For each instruction/second:
1 byte of memory, and
1 bit of I/O bandwidth

Storage has been falling behind

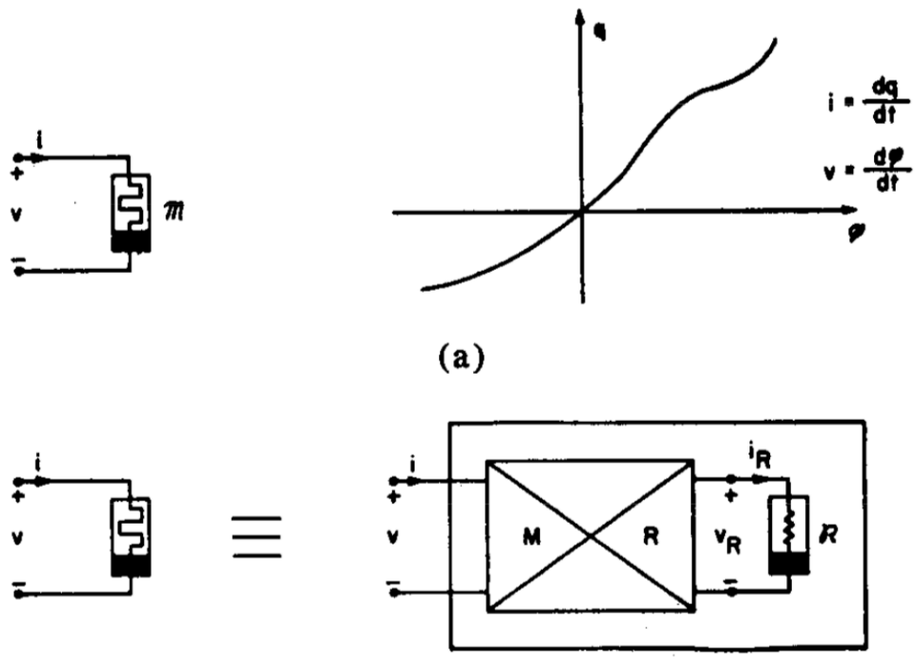


Memristor—The Missing Circuit Element

LEON O. CHUA, SENIOR MEMBER, IEEE

Abstract—A new two-terminal circuit element—called the *memristor*—characterized by a relationship between the *charge* $q(t) \equiv \int_{-\infty}^{t} i(\tau)\, d\tau$ and the *flux-linkage* $\varphi(t) \equiv \int_{-\infty}^{t} v(\tau)\, d\tau$ is introduced as the *fourth basic* circuit element. An electromagnetic field interpretation of this relationship in terms of a quasi-static expansion of Maxwell's equations is presented. Many circuit-theoretic properties of memristors are derived. It is shown that this element exhibits some peculiar behavior different from that exhibited by resistors, inductors, or capacitors. These properties lead to a number of unique applications which cannot be realized with *RLC* networks alone.

Although a physical memristor device without internal power supply has not yet been discovered, operational laboratory models have been built with the help of active circuits. Experimental results are presented to demonstrate the properties and potential applications of memristors.



I. Introduction

THIS PAPER presents the logical and scientific basis

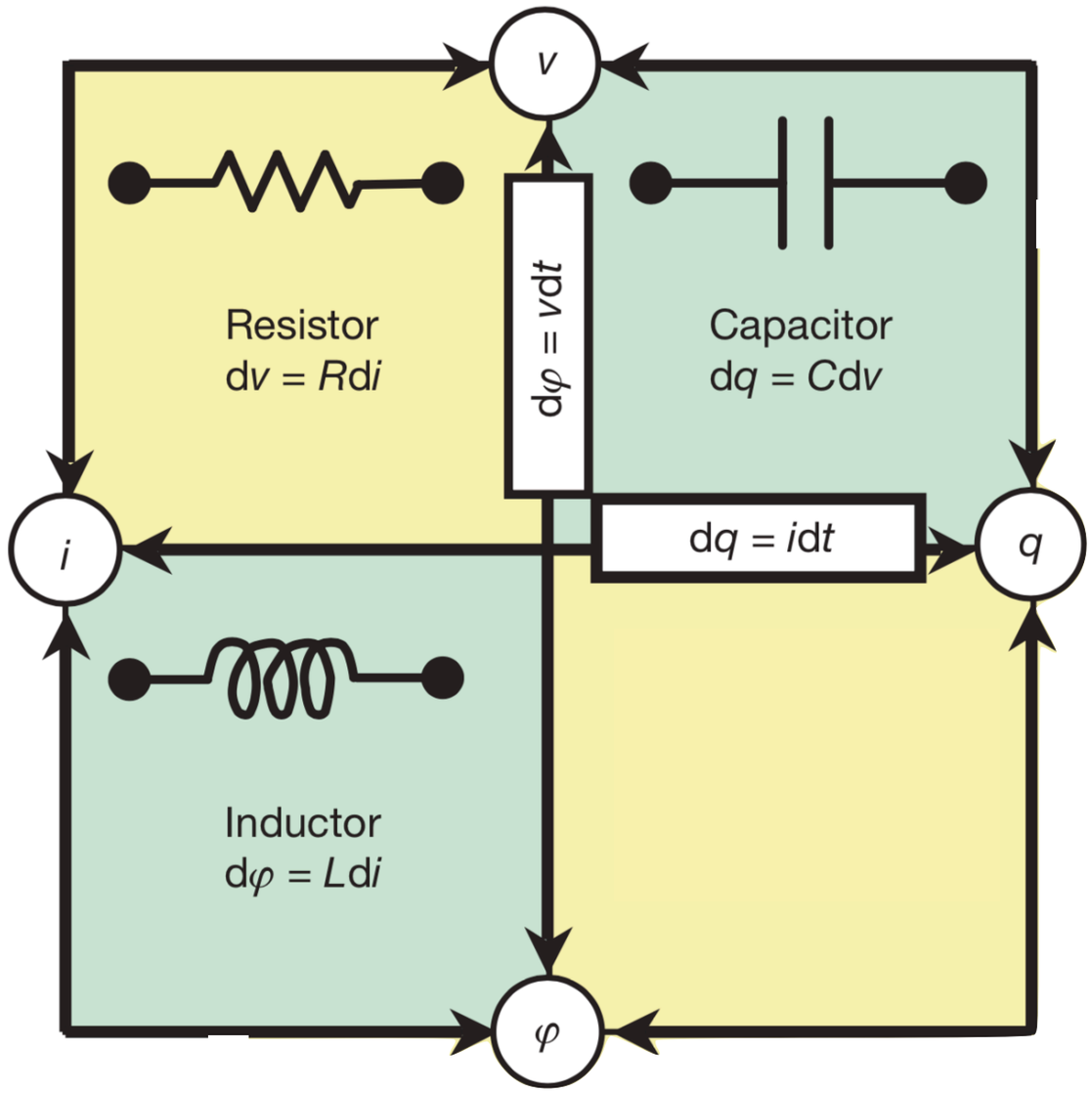


Diagram: *The missing memristor found*, Strukov et al, Nature v.453 p.80

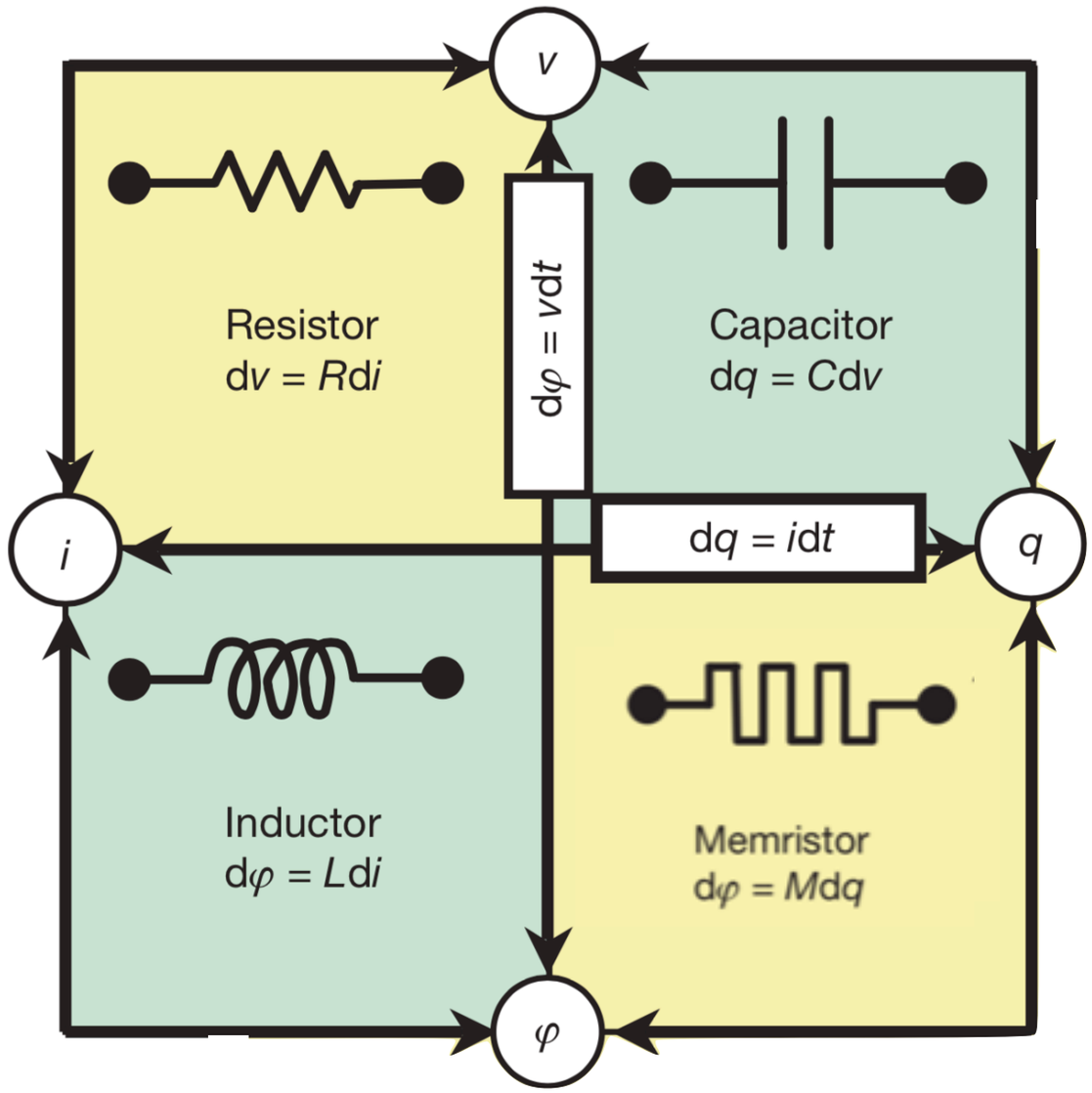
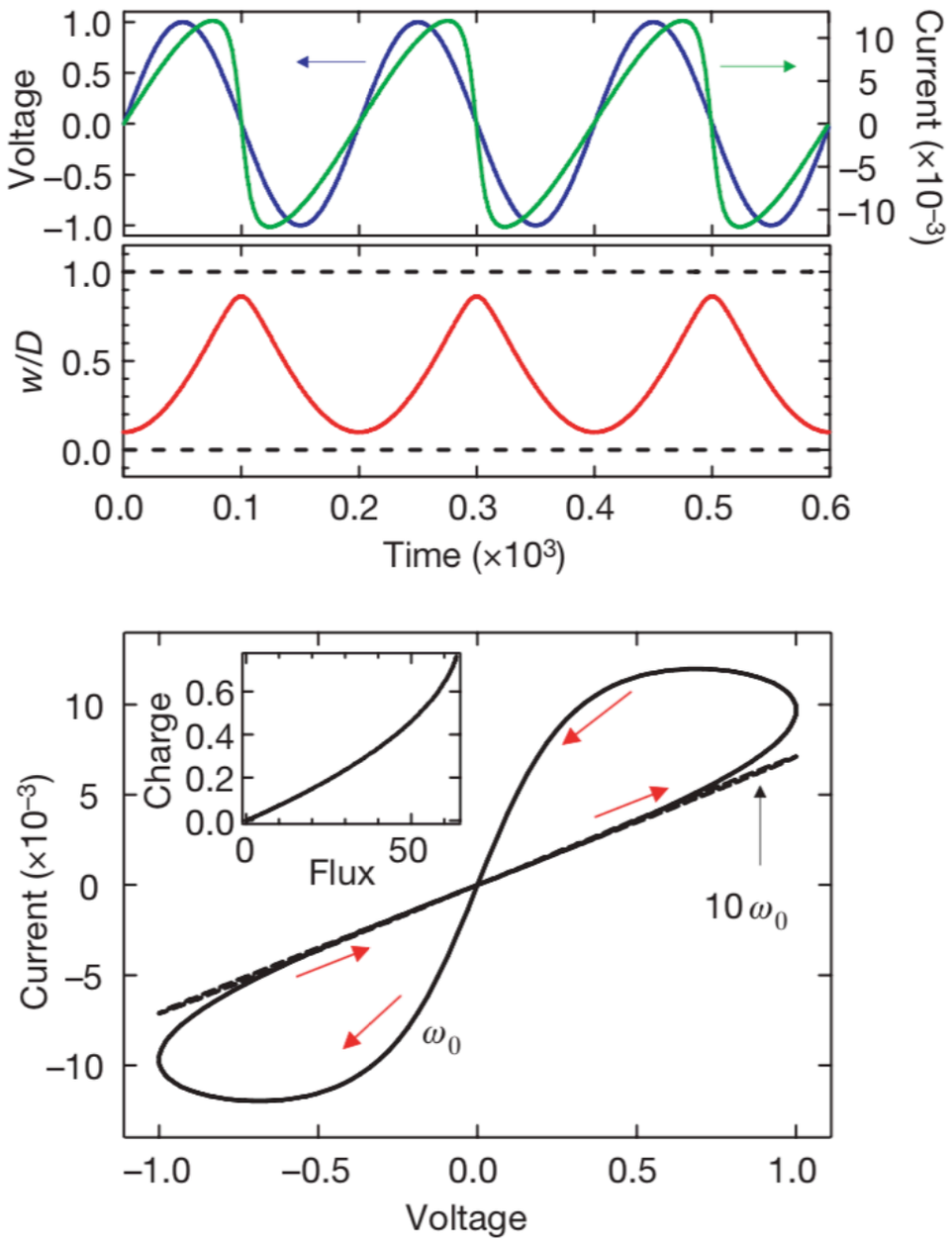


Diagram: *The missing memristor found*, Strukov et al, Nature v.453 p.80

Hysteresis — circuit “memory”



nature Vol 453|1 May 2008|doi:10.1038/nature06932

LETTERS

The missing memristor found

Dmitri B. Strukov[1], Gregory S. Snider[1], Duncan R. Stewart[1] & R. Stanley Williams[1]

Anyone who ever took an electronics laboratory class will be familiar with the fundamental passive circuit elements: the resistor, the capacitor and the inductor. However, in 1971 Leon Chua reasoned from symmetry arguments that there should be a fourth fundamental element, which he called a memristor (short for memory resistor)[1]. Although he showed that such an element has many interesting and valuable circuit properties, until now no one has presented either a useful physical model or an example of a memristor. Here we show, using a simple analytical example, that memristance arises naturally in nanoscale systems in which solid-state electronic and ionic transport are coupled under an external bias voltage. These results serve as the foundation for understanding a wide range of hysteretic current–voltage behaviour observed in many nanoscale electronic devices[2–19] that involve the motion of charged atomic or molecular species, in particular certain titanium dioxide cross-point switches[20–22].

More specifically, Chua noted that there are six different math-

propose a physical model that satisfies these simple equations. In 1976 Chua and Kang generalized the memristor concept to a much broader class of nonlinear dynamical systems they called memristive systems[23], described by the equations

$$v = \mathcal{R}(w,i)i \qquad (3)$$

$$\frac{\mathrm{d}w}{\mathrm{d}t} = f(w,i) \qquad (4)$$

where w can be a set of state variables and $\mathcal{R}$ and f can in general be explicit functions of time. Here, for simplicity, we restrict the discussion to current-controlled, time-invariant, one-port devices. Note that, unlike in a memristor, the flux in memristive systems is no longer uniquely defined by the charge. However, equation (3) does serve to distinguish a memristive system from an arbitrary dynamical device; no current flows through the memristive system when the voltage drop across it is zero. Chua and Kang showed that the i–v

nature Vol 453|1 May 2008|doi:10.1038/nature06932

LETTERS

The missing memristor found

Dmitri B. Strukov[1], Gregory S. Snider[1], Duncan R. Stewart[1] & R. Stanley Williams[1]

Anyone who ever took an electronics laboratory class will be familiar with the fundamental passive circuit elements: the resistor, the capacitor and the inductor. However, in 1971 Leon Chua reasoned from symmetry arguments that there should be a fourth fundamental element, which he called a memristor (short for memory resistor)[1]. Although he showed that such an element has many interesting and valuable circuit properties, until now no one has presented either a useful physical model or an example of a memristor. Here we show, using a simple analytical example, that memristance arises naturally in nanoscale systems in which solid-state electronic and ionic transport are coupled under an external bias voltage. These results serve as the foundation for understanding a wide range of hysteretic current–voltage behaviour observed in many nanoscale electronic devices[2–19] that involve the motion of charged atomic or molecular species, in particular certain titanium dioxide cross-point switches[20–22].

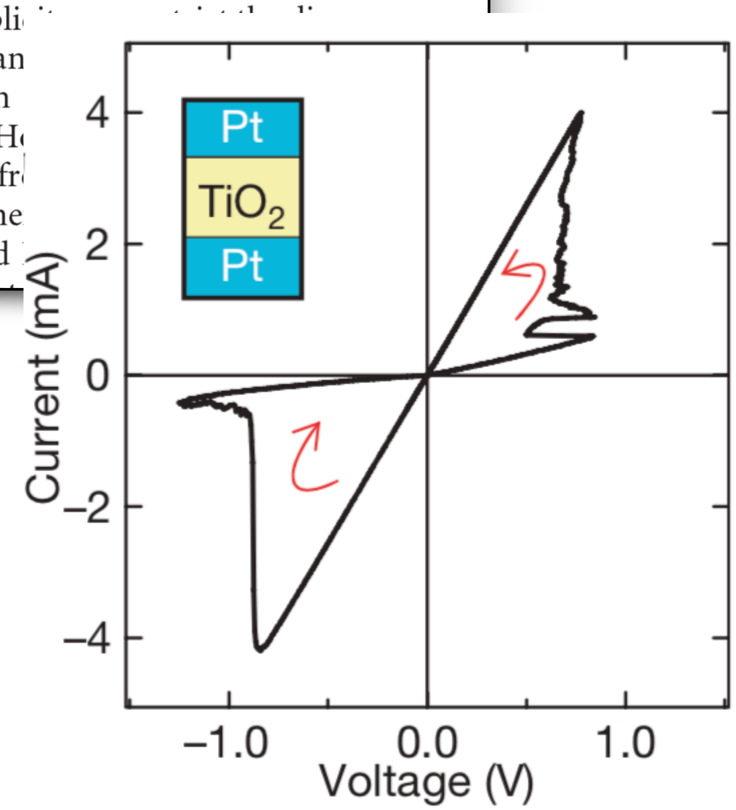
More specifically, Chua noted that there are six different math-

propose a physical model that satisfies these simple equations. In 1976 Chua and Kang generalized the memristor concept to a much broader class of nonlinear dynamical systems they called memristive systems[23], described by the equations

$$v = \mathcal{R}(w,i)\,i \quad (3)$$

$$\frac{\mathrm{d}w}{\mathrm{d}t} = f(w,i) \quad (4)$$

where w can be a set of state variables and $\mathcal{R}$ and f can in general be explicit functions of time. Here, for simpli
sion to current-controlled, time-invarian
that, unlike in a memristor, the flux in
longer uniquely defined by the charge. H
serve to distinguish a memristive system fr
device; no current flows through the me
voltage drop across it is zero. Chua and



nature Vol 453|1 May 2008|doi:10.1038/nature06932

LETTERS

The missing memristor found

Dmitri B. Strukov[1], Gregory S. Snider[1], Duncan R. Stewart[1] & R. Stanley Williams[1]

Anyone who ever took an electronics laboratory class will be familiar with the fundamental passive circuit elements: the resistor, the capacitor and the inductor. However, in 1971 Leon Chua reasoned from symmetry arguments that there should be a fourth fundamental element, which he called a memristor (short for memory resistor)[1]. Although he showed that such an element has many interesting and valuable circuit properties, until now no one has presented either a useful physical model or an example of a memristor. Here we show, using a simple analytical example, that memristance arises naturally in nanoscale systems in which solid-state electronic and ionic transport are coupled under an external bias voltage. These results serve as the foundation for understanding a wide range of hysteretic current–voltage behaviour observed in many nanoscale electronic devices[2–19] that involve the motion of charged atomic or molecular species, in particular certain titanium dioxide cross-point switches[20–22].

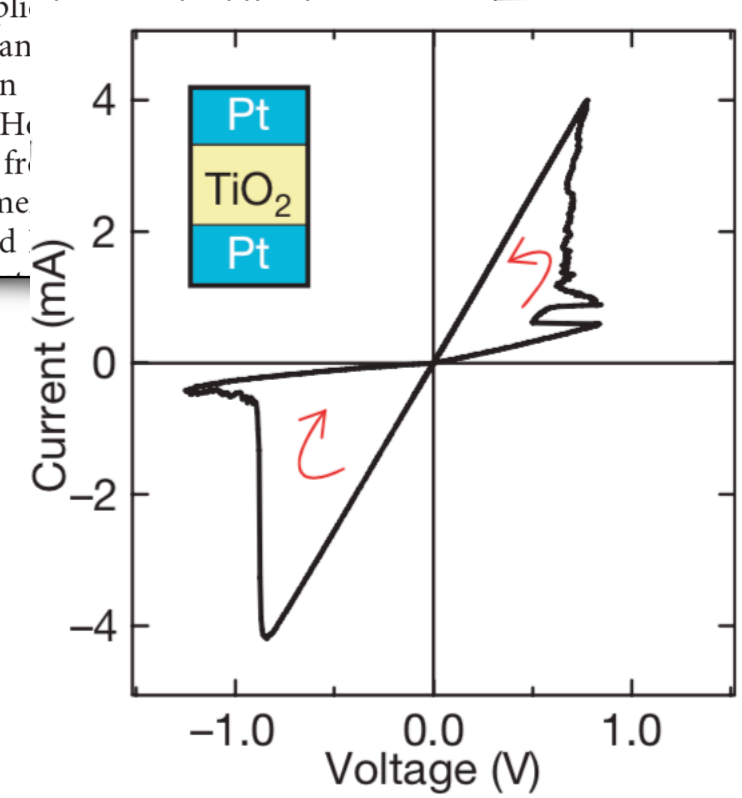
More specifically, Chua noted that there are six different math-

propose a physical model that satisfies these simple equations. In 1976 Chua and Kang generalized the memristor concept to a much broader class of nonlinear dynamical systems they called memristive systems[23], described by the equations

$$v = \mathcal{R}(w,i)i \quad (3)$$

$$\frac{dw}{dt} = f(w,i) \quad (4)$$

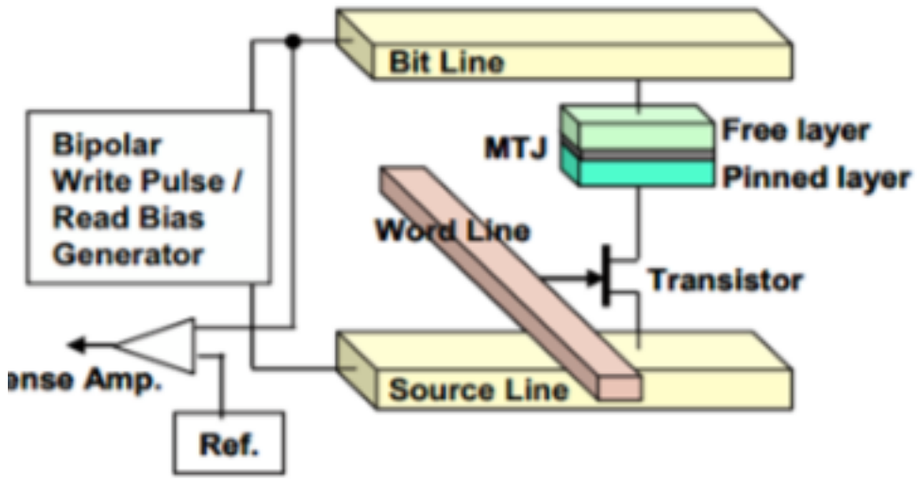
where w can be a set of state variables and $\mathcal{R}$ and f can in general be explicit functions of time. Here, for simpli
sion to current-controlled, time-invarian
that, unlike in a memristor, the flux in
longer uniquely defined by the charge. H
serve to distinguish a memristive system fr
device; no current flows through the me
voltage drop across it is zero. Chua and



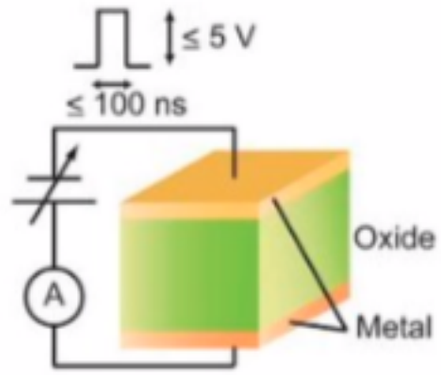
How we found the missing memristor,
R. Stanley Williams, IEEE Spectrum, Dec 2008 29—35

An explosion of technologies

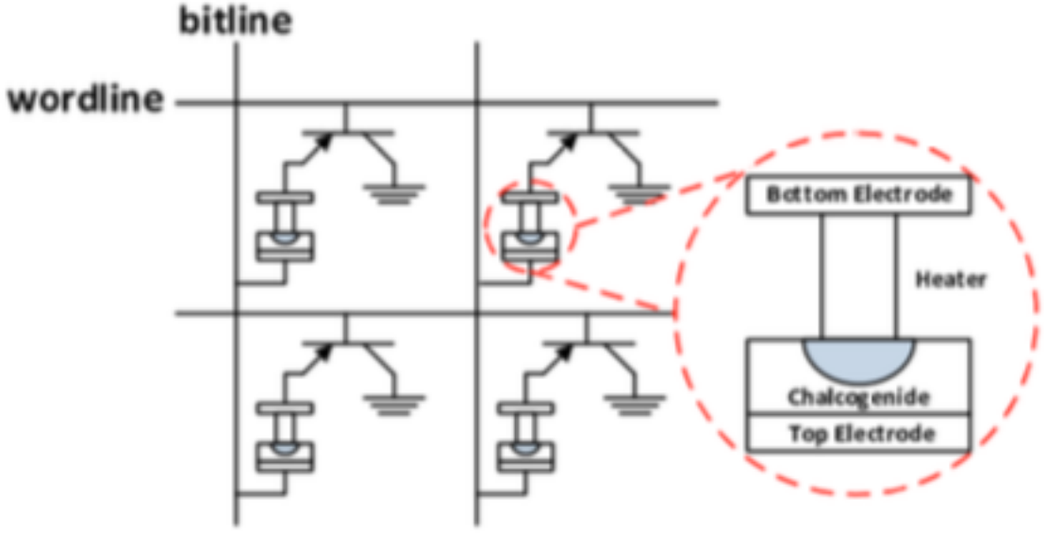
- Magnetoresistive RAM
  Spin-Torque Transfer RAM



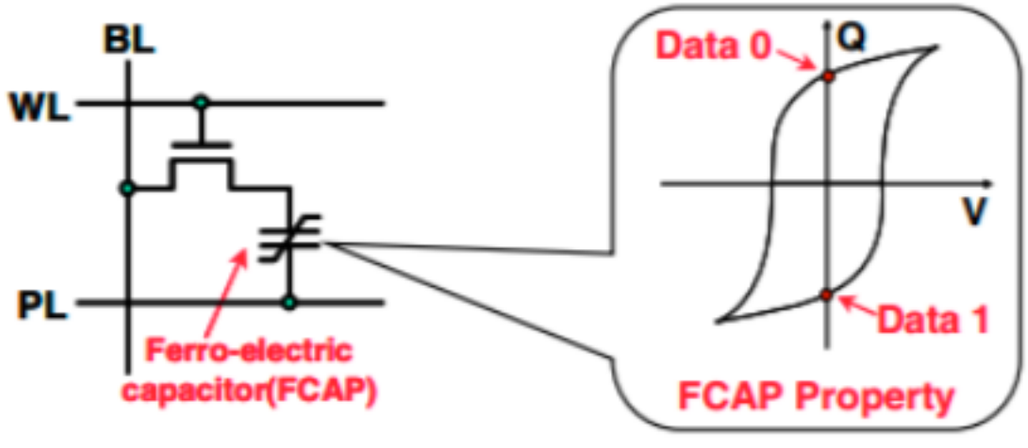
- Resistive RAM



- Phase Change Memory



- Ferroelectric RAM



Diagrams: Boukhobza et al, *Emerging NVM*, ACM Trans. Des. Aut. Elec. Sys, Jan 2018

NVRAM trends

- Order-of-magnitude density improvement over DRAM makes mass adoption highly likely

- Some companies are claiming their technology will match and surpass DRAM, e.g., Nantero, based on carbon nanotubes

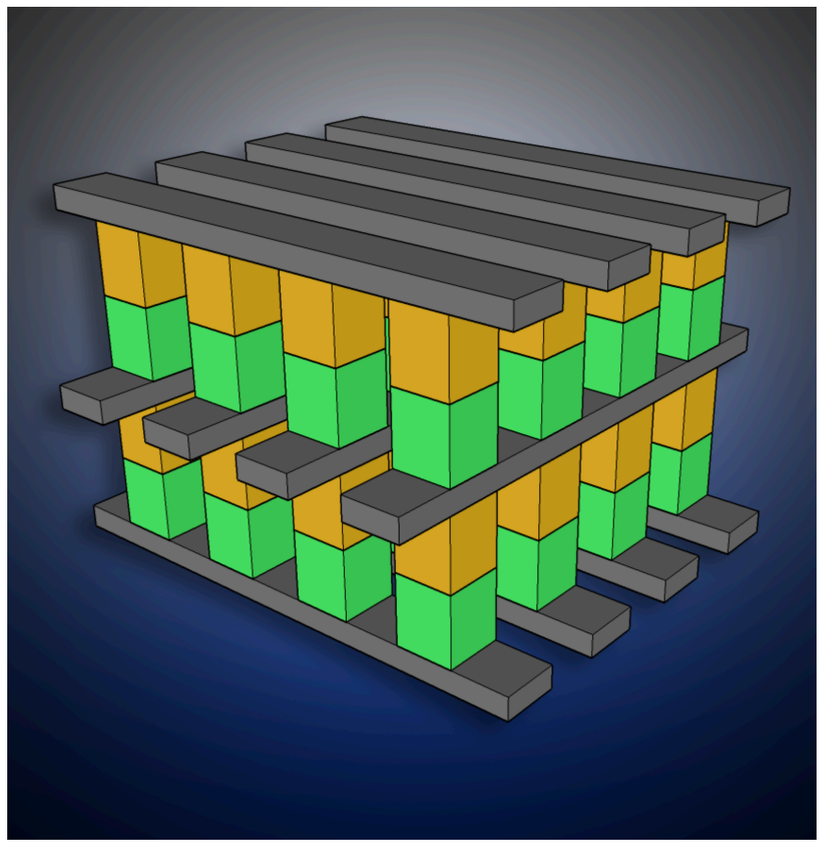


Image: https://en.wikipedia.org/wiki/3D_XPoint#/media/File:3D_XPoint.png

Software

SNIA NVM Programming Model

- Storage Networking Industry Association — a consortium of companies that develops and promotes vendor-neutral architectures and standards. snia.org

- Has been developing a NVM Programming Model since ~2012.

NPM in a single diagram

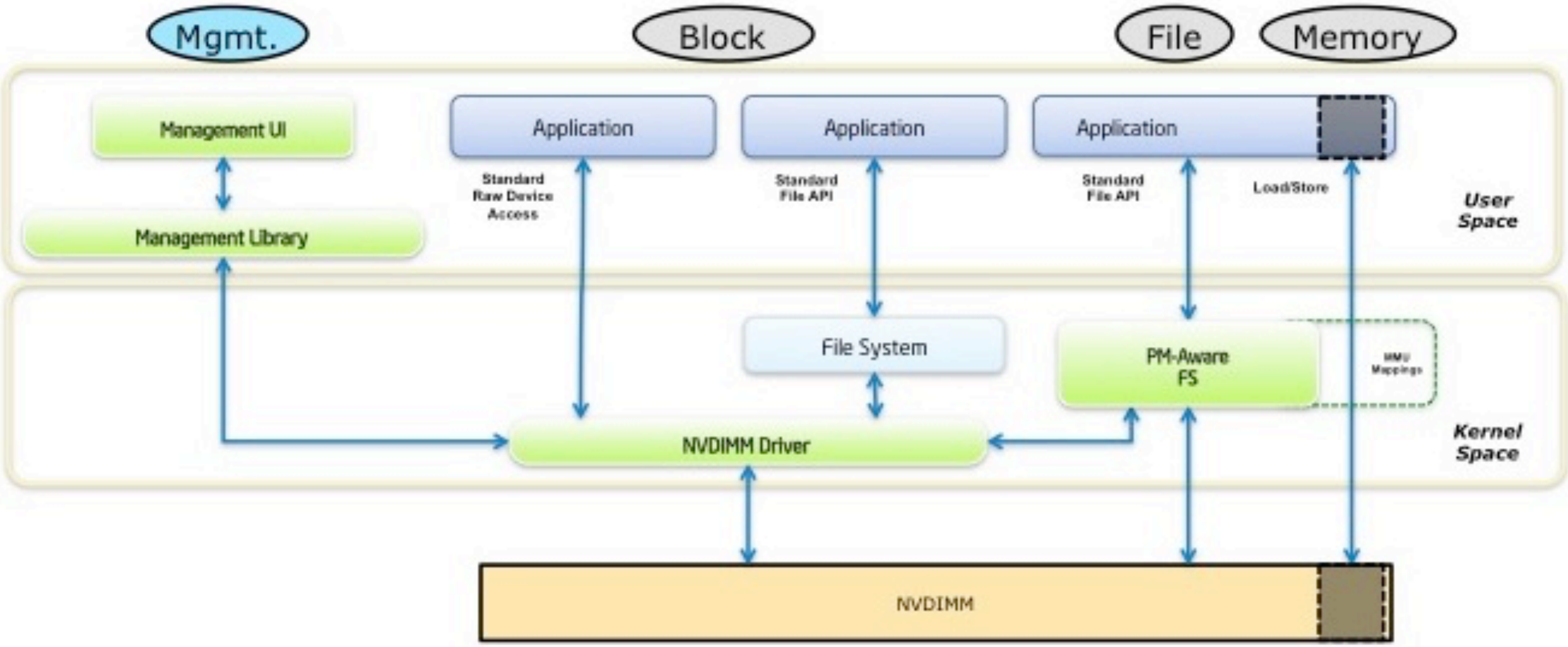
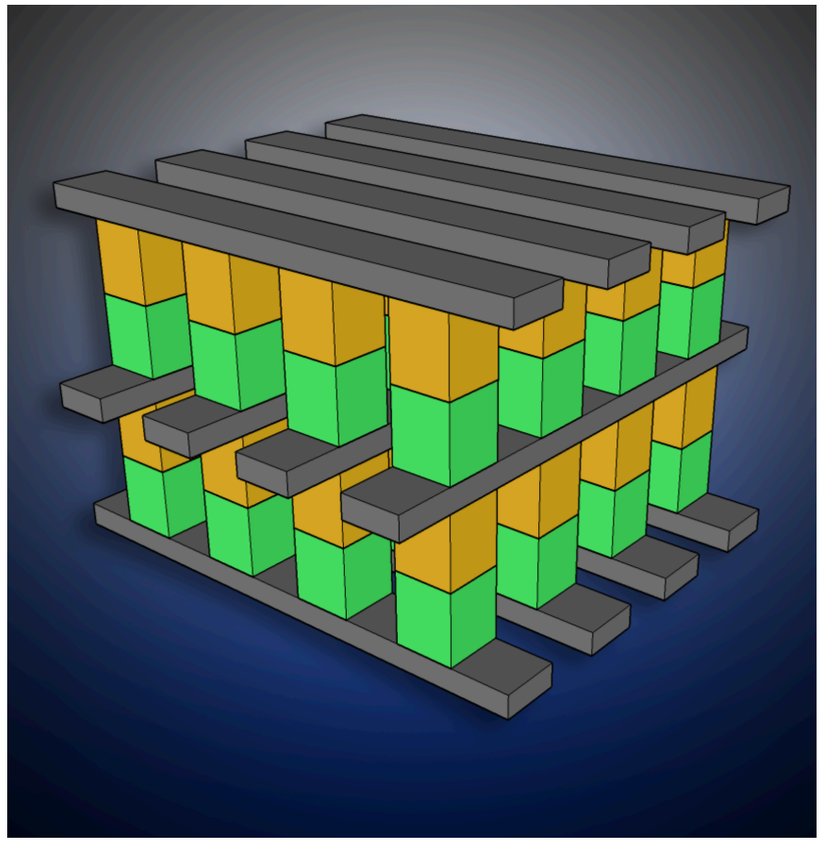


Diagram: https://pmem.io/2014/08/27/crawl-walk-run.html

Intel Optane DC Persistent Memory using 3D XPoint circuit technology

Image: https://www.storagereview.com/intel_optane_dc_persistent_memory_module_pmm



- GA 2019 April, requires Cascade Lake CPU and system
- Chips made by Micron
- ReRAM? PCM? They’re not saying.

Image: https://en.wikipedia.org/wiki/3D_XPoint#/media/File:3D_XPoint.png

Speeds and feeds

- 128Gb chips assembled into DIMMs of 128, 256 or 512GB
  (cf: biggest DRAM chip so far is 16Gb)

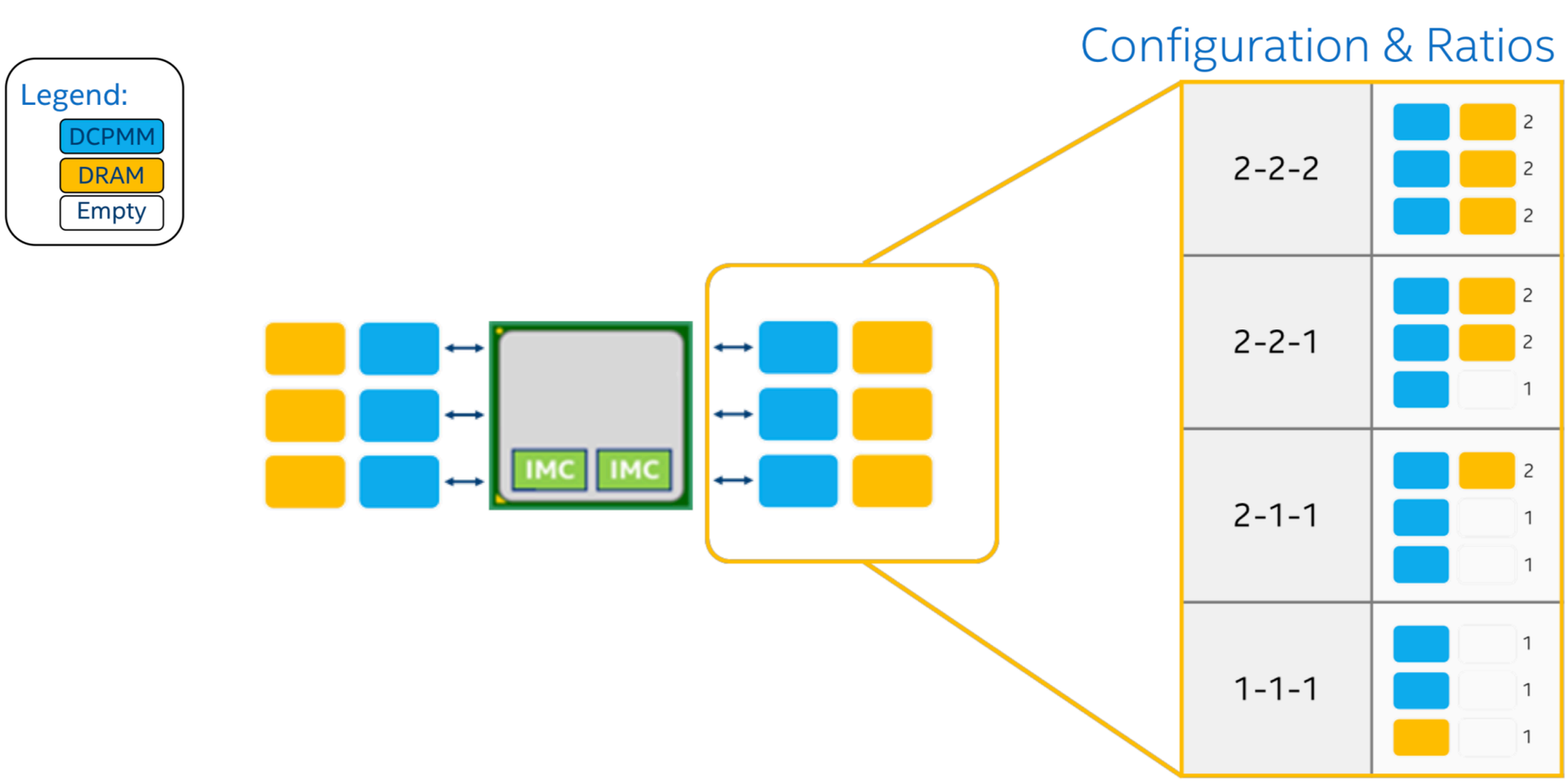
- From UCSD paper (https://arxiv.org/pdf/1903.05714.pdf):
  - Random read ~300ns (cf DRAM: 80ns; 3.75:1)
  - Read bandwidth ~⅓ DRAM, write ~⅙

- Endurance? “adequate”
  5-year warranty; probably around 10^8 writes

- Price: around 20—40% of DRAM per bit.

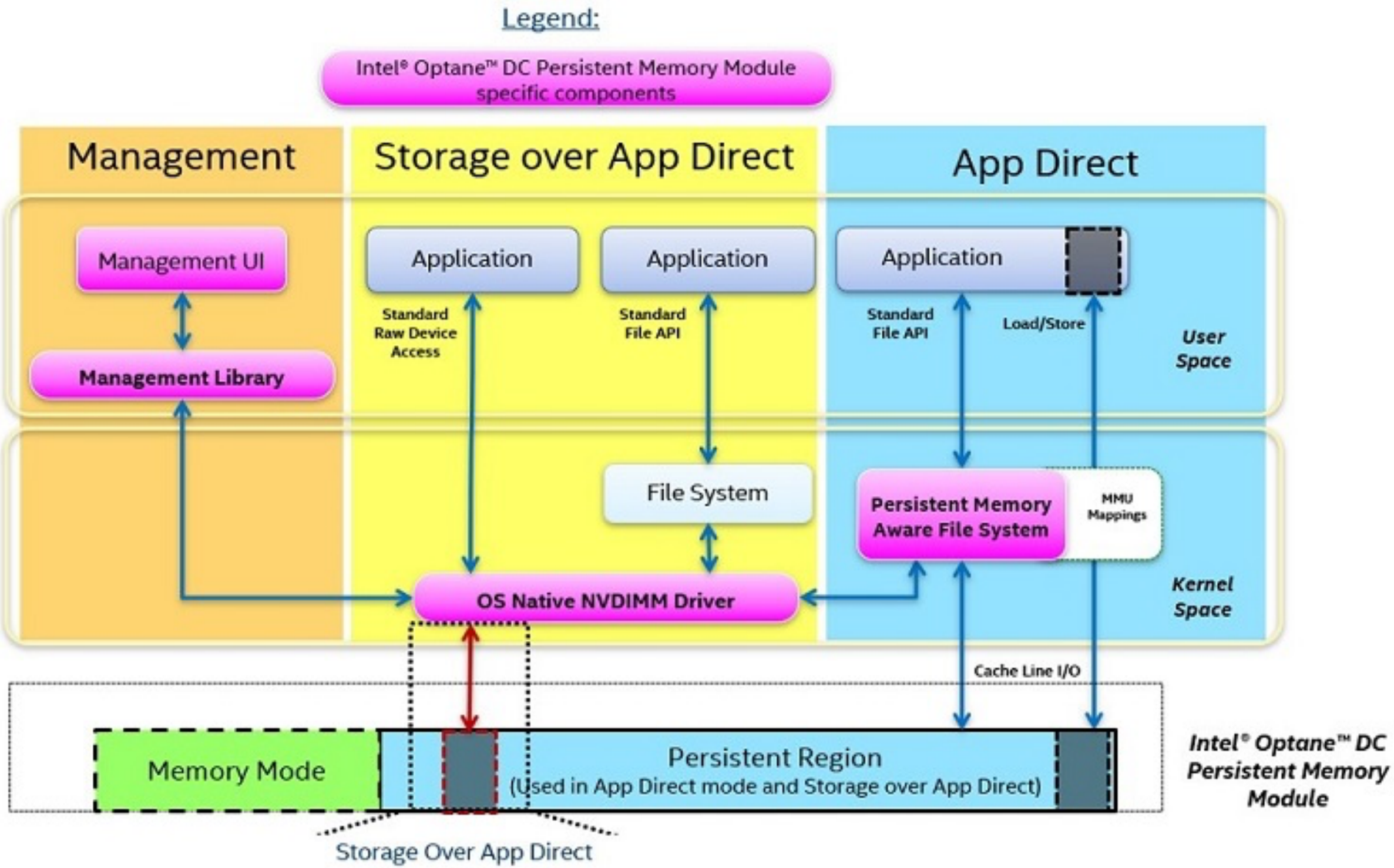
Systems

Cascade Lake system needed: up to 3TB PM/socket



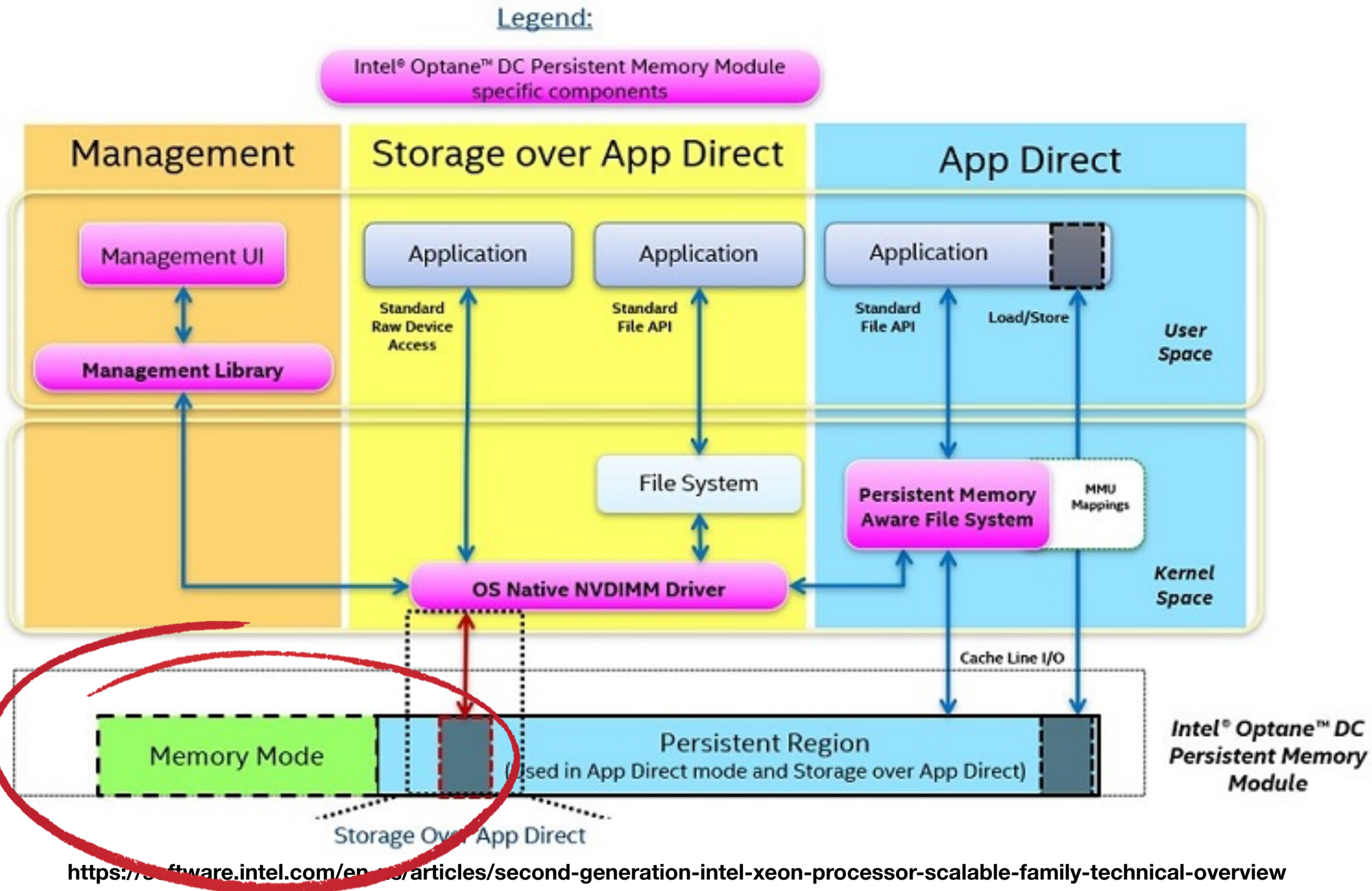
https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Population-Configuration.pdf

Operating Modes



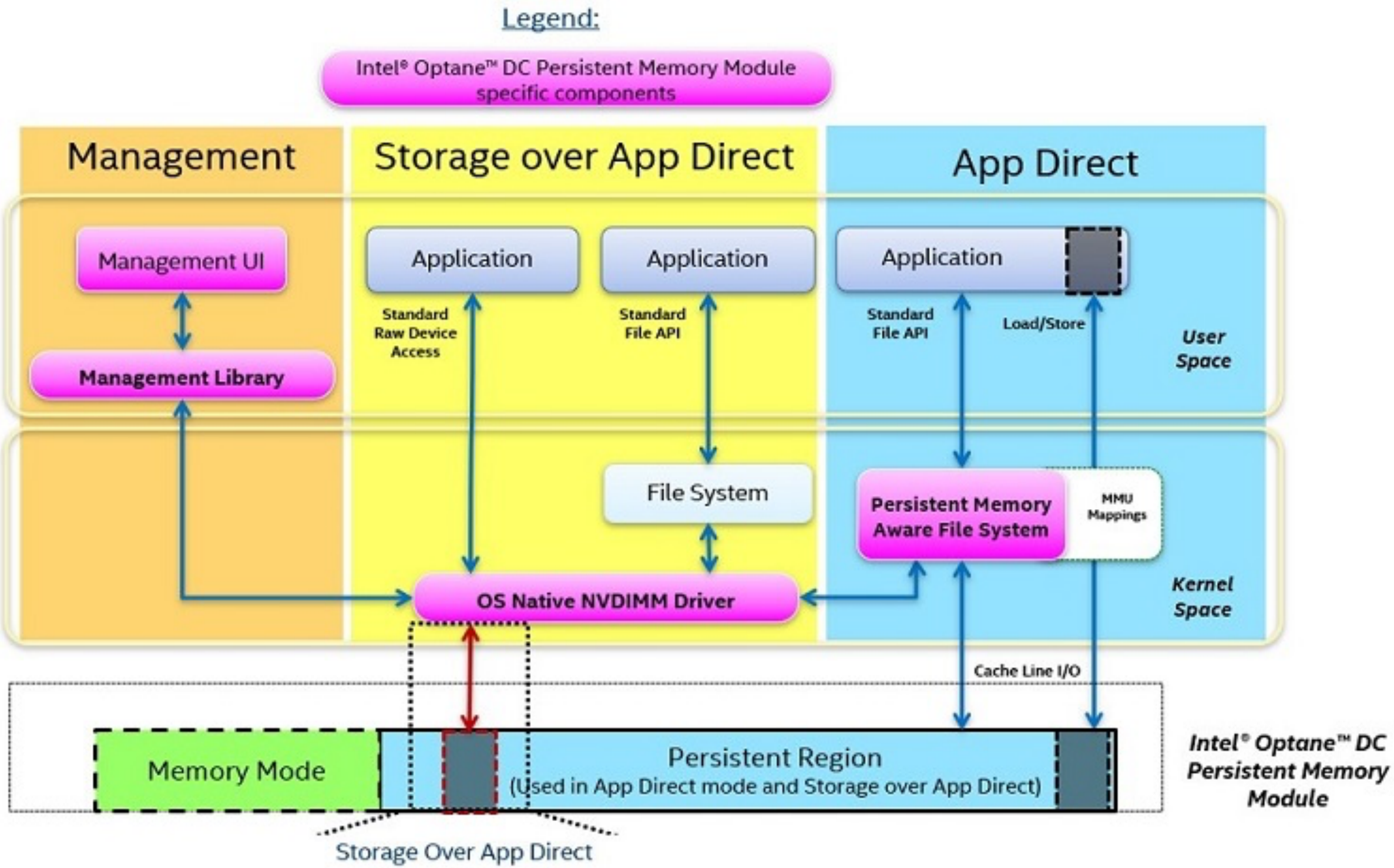
https://software.intel.com/en-us/articles/second-generation-intel-xeon-processor-scalable-family-technical-overview

Operating Modes



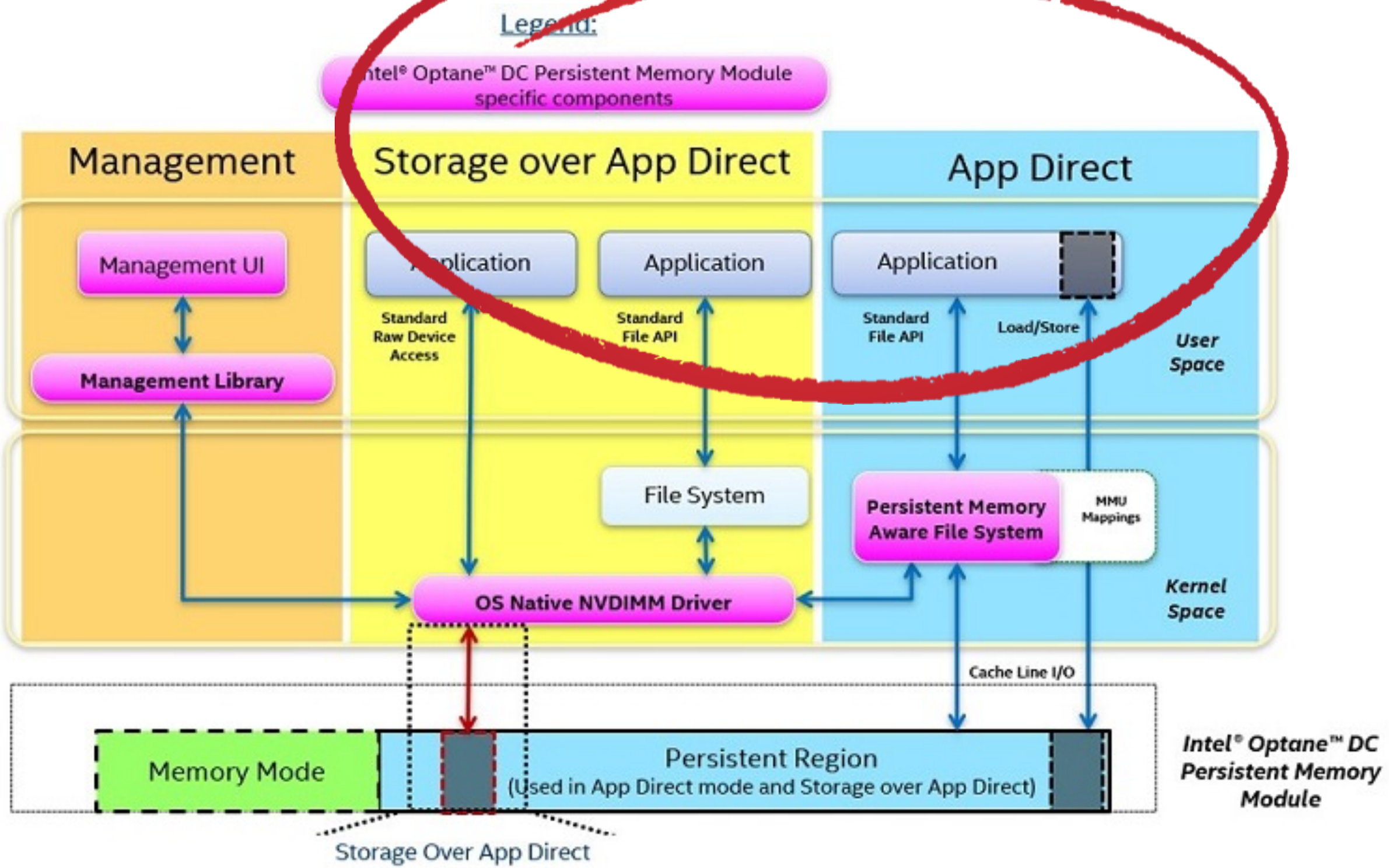
https://software.intel.com/en-us/articles/second-generation-intel-xeon-processor-scalable-family-technical-overview

Operating Modes



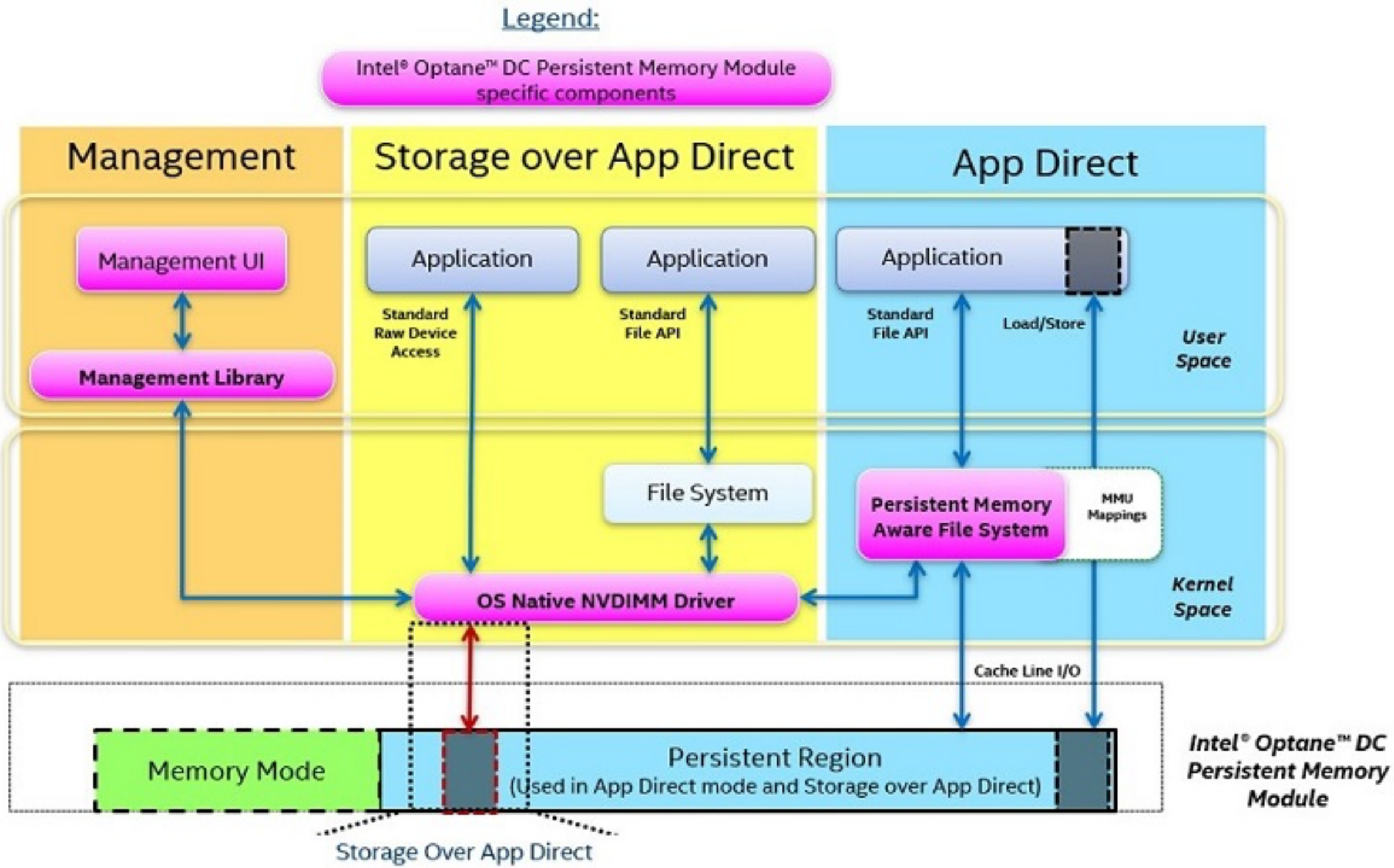
https://software.intel.com/en-us/articles/second-generation-intel-xeon-processor-scalable-family-technical-overview

Operating Modes



https://software.intel.com/en-us/articles/second-generation-intel-xeon-processor-scalable-family-technical-overview

Operating Modes



https://software.intel.com/en-us/articles/second-generation-intel-xeon-processor-scalable-family-technical-overview

Wear-leveling

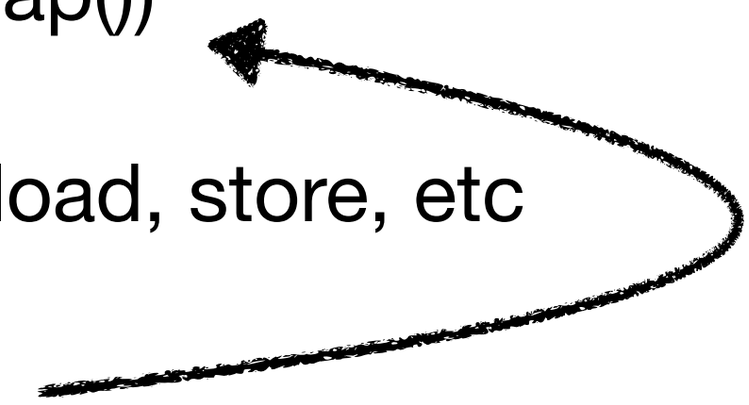
- AppDirect… “direct” access to PM?
- [according to UCSD] PM organized as 256B blocks, accessed via a mapping table.
- Writes appear to go into a buffer (fast)
- Indirection allows for bad block management and wear-leveling.
  - Spare capacity
  - Max write rate with perfect leveling yields 2M writes/block within 5y warranty

Using Optane PM

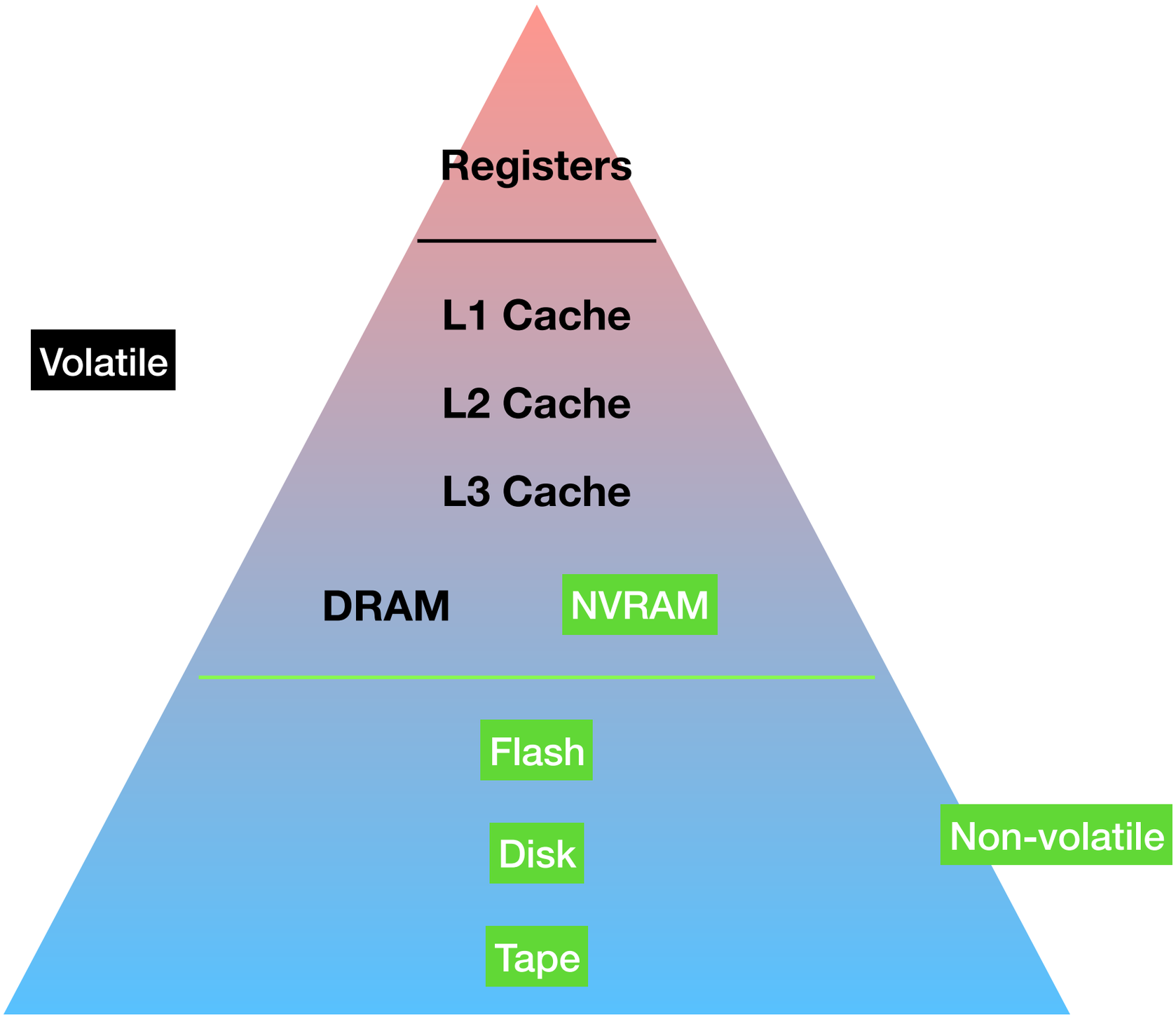
- Supported by Linux and Windows
- Treated like a disk: contains a filesystem
- mmap() can be used to map a (part of a) file into a process' address space without an intermediate buffer
  - DAX - Direct Access

Persistence lifecycle

- Create a *region* — a piece of or a whole file, intended to contain persistent data
- *Attach* the region (mmap())
- Read and write using load, store, etc
- *Detach* (close())



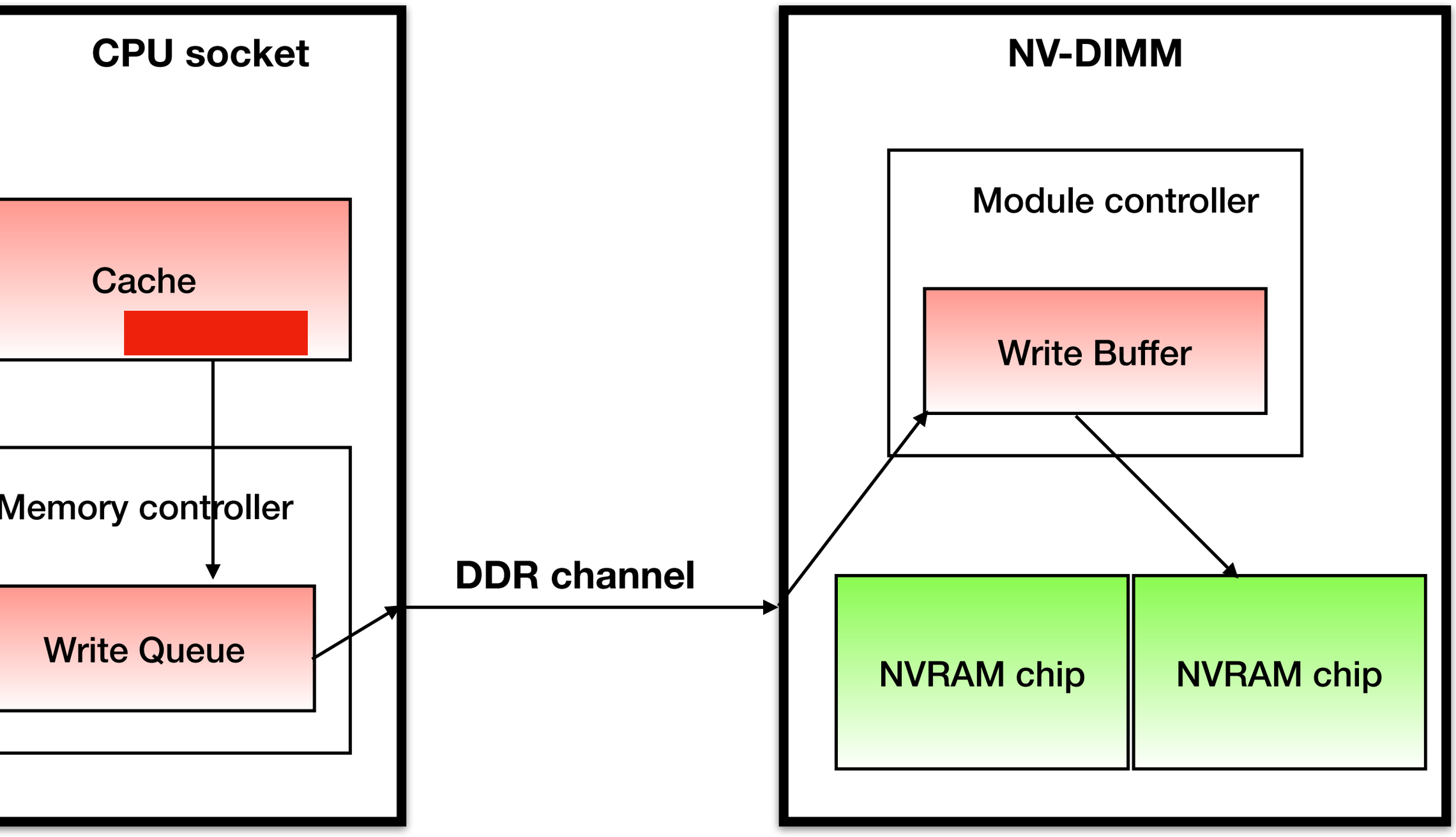
The persistence domain



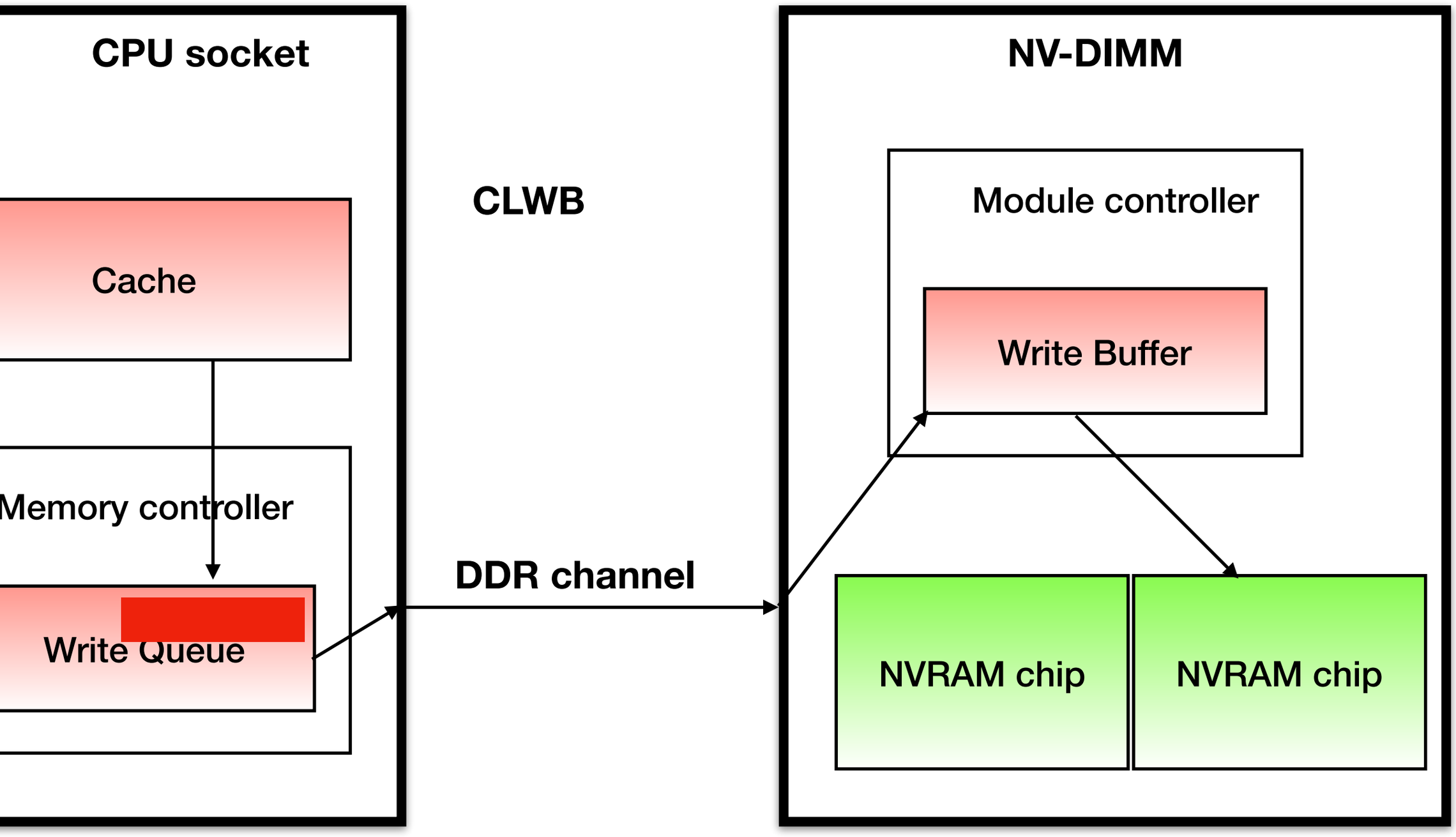
How to achieve durability

- Write to memory ⇒ update likely to be in cache
- Execute CLWB - Cache Line Write Back ⇒ asynchronously writes cache line to memory
- Execute SFENCE or MFENCE - Store/Memory Fence ⇒ waits for asynchronous write to complete

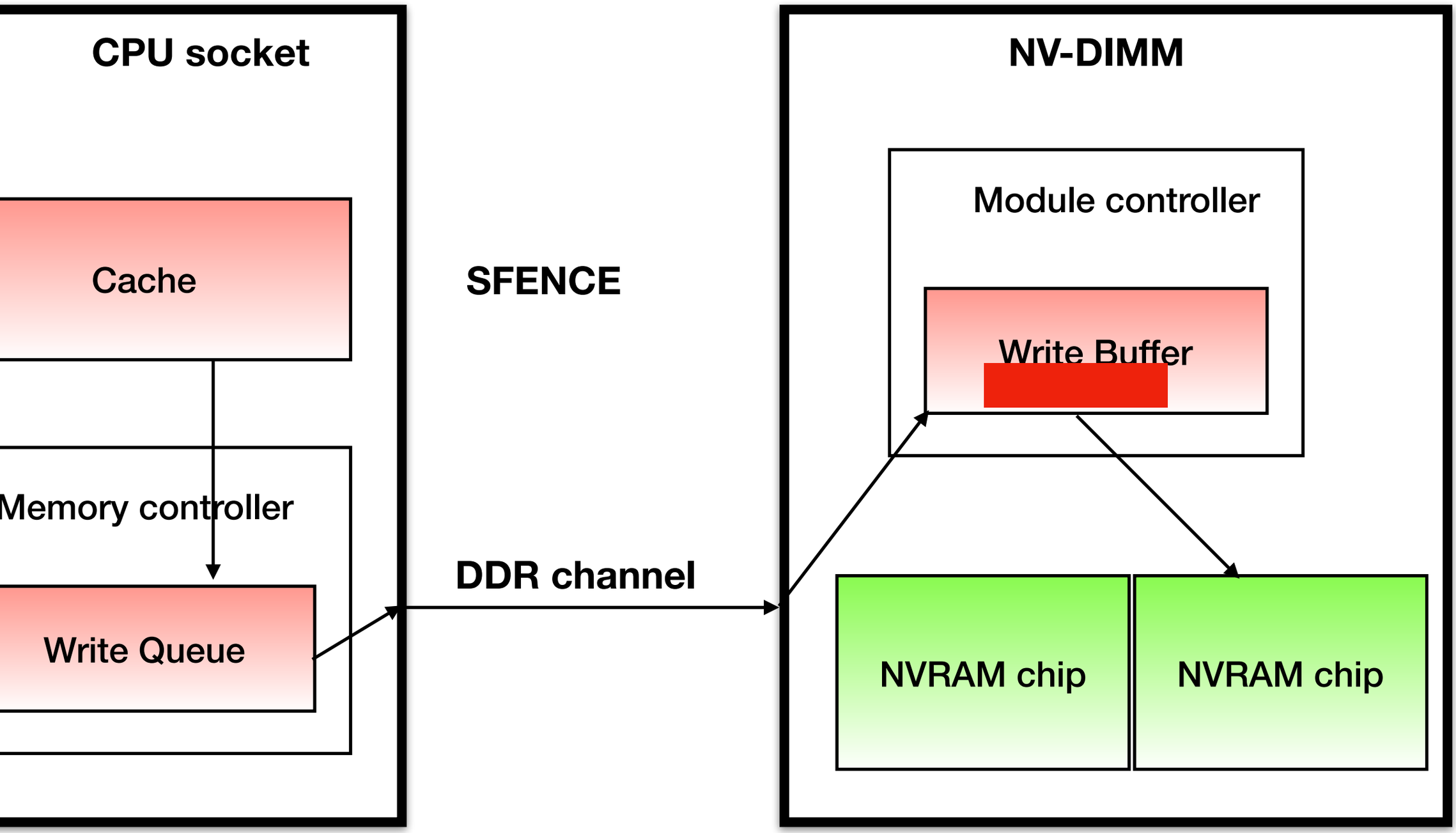
The path to NVRAM



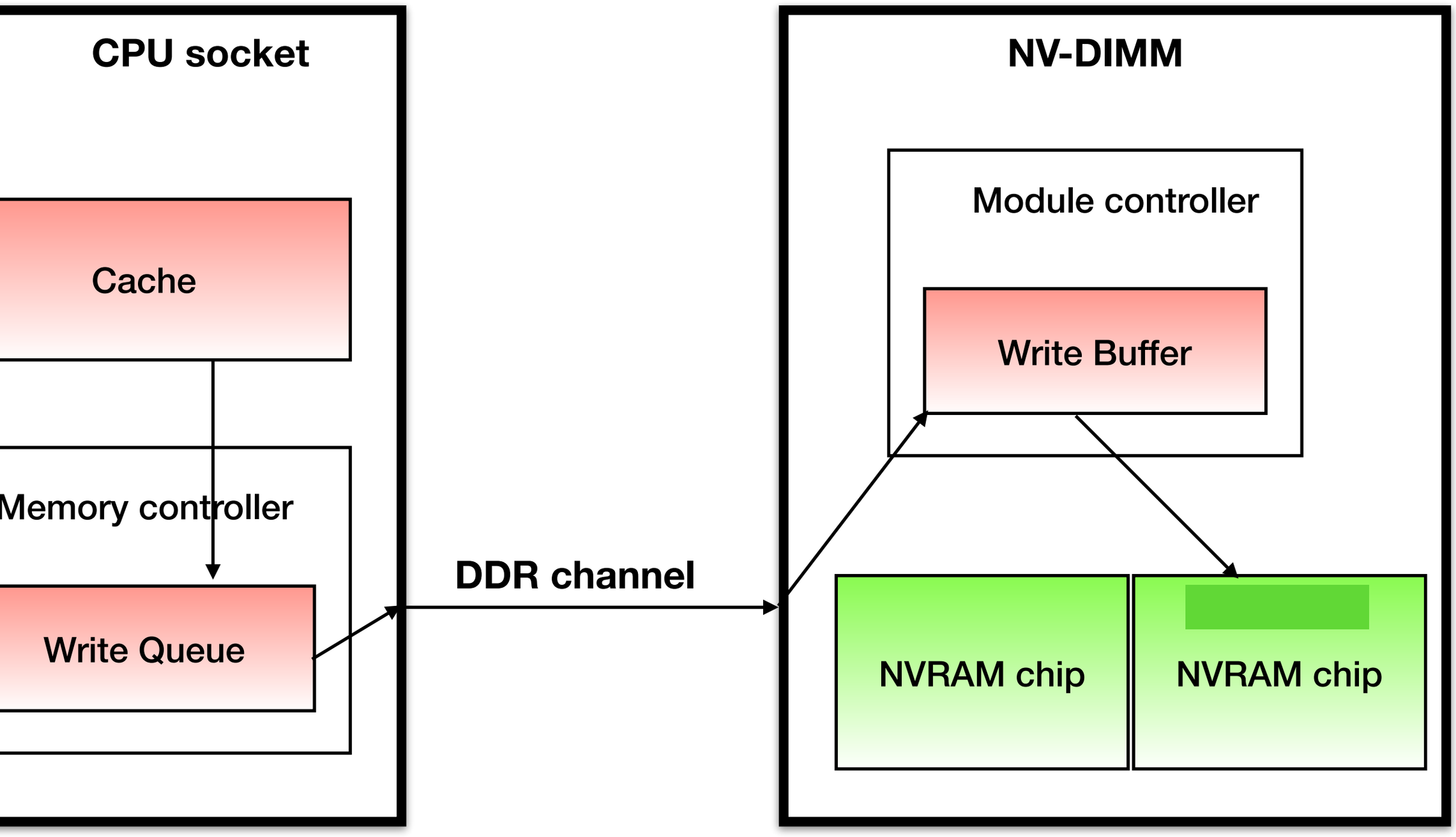
The path to NVRAM



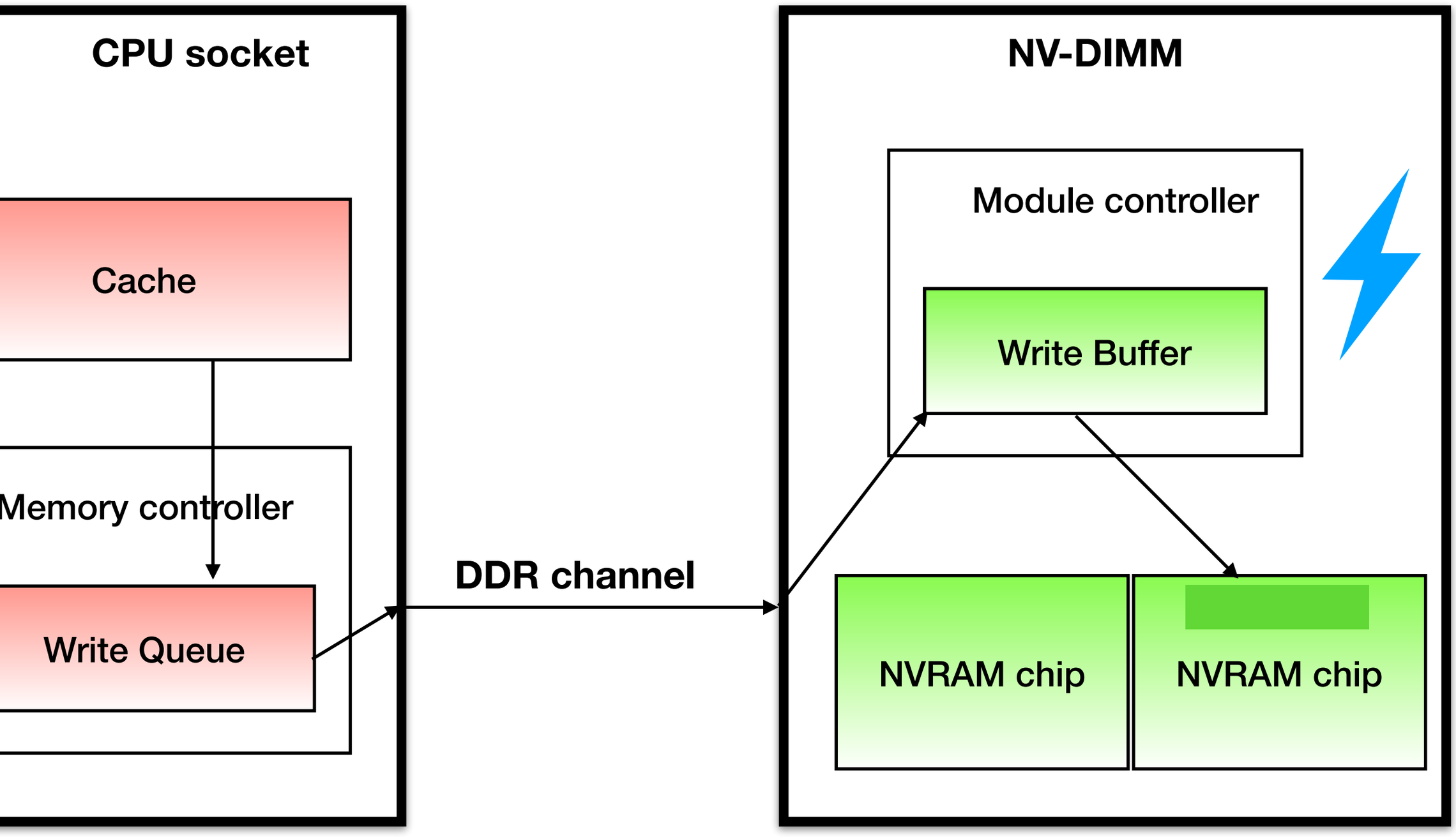
The path to NVRAM



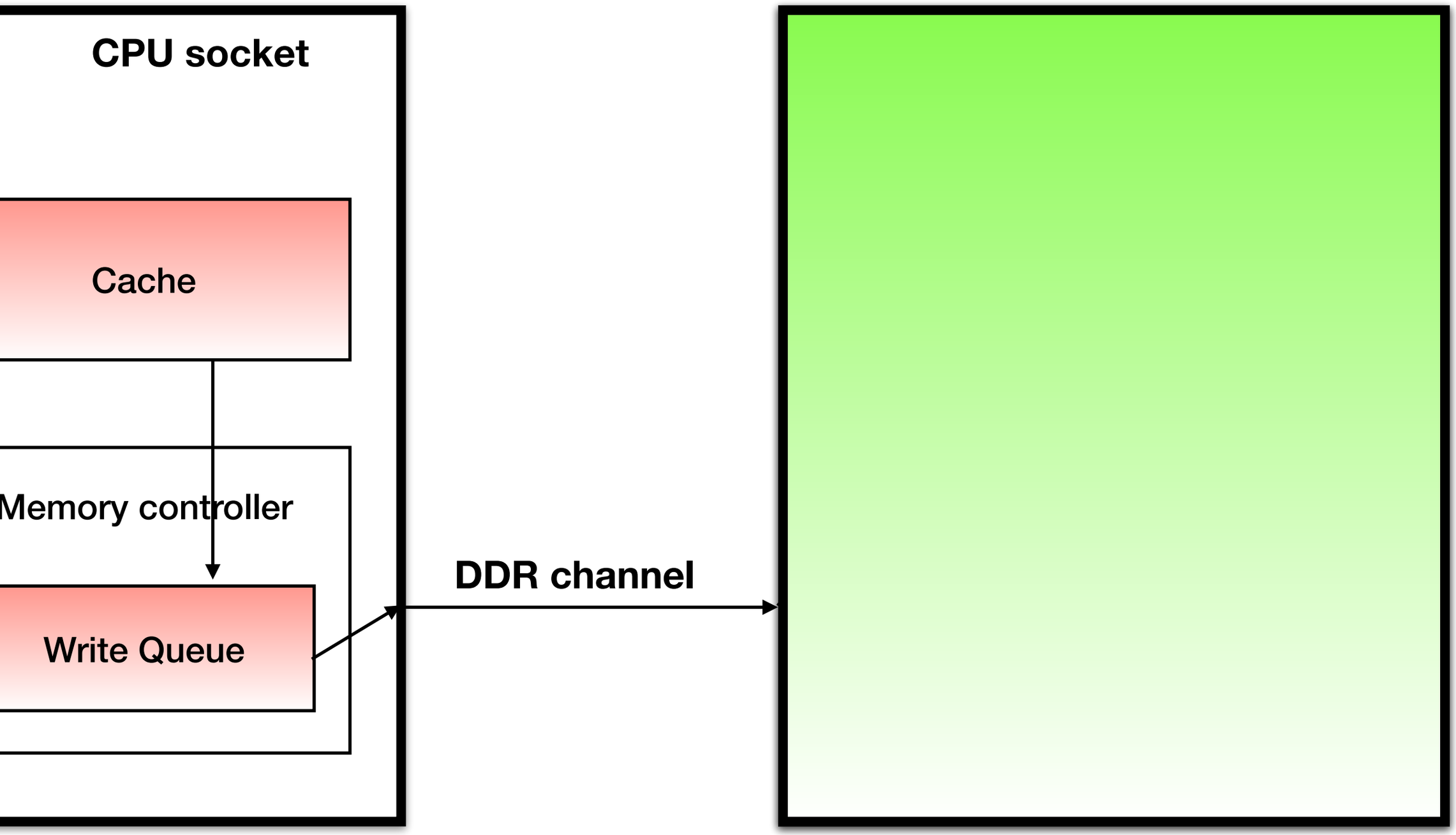
The path to NVRAM



The path to NVRAM



The path to NVRAM



Using NVRAM

Position-independent data

- Might not want to rely on mmap() mapping to the same address
- Hence, data could be either
  - position independent — e.g. self-relative, or
  - relocatable - linked after mapping.

Self-relative data

- A pointer is represented by the offset from its own address to the target.

- Just like PC-relative branch offsets.

- If pointers can point to themselves (i.e., have value 0) then the NULL pointer must have some other value (e.g., 1, which would point to a byte within the pointer).

Data Relocation

- Data will have been previously mapped at some address
- After attach, find the pointers and adjust them by the offset to the new mapping.
  - Must be able to find the pointers — tricky in some languages
- For a big file, don’t want to do this eagerly — long link pause. Could do incrementally using page protection.
- What if there’s a crash during relocation?
- Doesn’t support multiple simultaneous mappings

A new kind of dangling pointer

- After a restart and re-attach, persistent regions could have moved and volatile data existing before the restart have disappeared.

- Hence: bad idea to have references from one region to another, except from a volatile region to a persistent region. ***Lifetime correctness***

- Unless: metadata and behavior can identify such references and reconstruct or nullify before next use. Need a way to execute code at re-attach.

Durability is not directly observable

- Cache coherence means one core can read a memory location from another core's cache

- When reading a value from a NV location, written by another thread, the thread ***cannot know if that value is or was durable***

- The writer must tell the reader when it becomes durable (via a side-channel)

- Only one thread at a time executing write-writeback-fence

- One approach: maintain a volatile copy of a durable value

- See "The Observability Problem with Persistent Memory", Bill Bridge, https://medium.com/@mwolczko/non-volatile-memory-and-java-part-2-c15954c04e11

What are the advantages of persistent memory?

- Reduced startup time — no long wait while data are read from block storage into DRAM
- Reduced update latency—an update can be durable in <<1µs
- Higher capacity, density and lower cost/bit

The dream of NVRAM

The dream of NVRAM

- We compute using in-memory representations

The dream of NVRAM

- We compute using in-memory representations
- Then, we have to change representation to store
    - Requires code
    - Makes a copy
    - Takes time and energy

The dream of NVRAM

- We compute using in-memory representations
- Then, we have to change representation to store
    - Requires code
    - Makes a copy
    - Takes time and energy

Can we eliminate this second representation?

The dream of NVRAM

- We compute using in-memory representations
- Then, we have to change representation to store
    - Requires code
    - Makes a copy
    - Takes time and energy

Can we eliminate this second representation?

Lisp’s sys-out (1967), Smalltalk’s snapshots (1974)

The dream of NVRAM

- We compute using in-memory representations
- Then, we have to change representation to store
    - Requires code
    - Makes a copy
    - Takes time and energy

Can we eliminate this second representation?

Lisp's sys-out (1967), Smalltalk's snapshots (1974)

Atkinson, Bailey, Chisholm, Cockshott and Morrison,
PS-algol: a language for persistent programming (1983)

developer's The dream of NVRAM

- We compute using in-memory representations
- Then, we have to change representation to store
    - Requires code
    - Makes a copy
    - Takes time and energy

Can we eliminate this second representation?

Lisp's sys-out (1967), Smalltalk's snapshots (1974)

Atkinson, Bailey, Chisholm, Cockshott and Morrison,
PS-algol: a language for persistent programming (1983)

user's

The dream of NVRAM

The user's dream of NVRAM

- Boot and run from NVRAM—*all* software RAM-resident

The user's dream of NVRAM

- Boot and run from NVRAM—*all* software RAM-resident
- Persistent processes
  - Need to handle references to external state (e.g., network connections)

The user's dream of NVRAM

- Boot and run from NVRAM—*all* software RAM-resident
- Persistent processes
  - Need to handle references to external state (e.g., network connections)
- *All* software based on transactions and recovery

The user's dream of NVRAM

- Boot and run from NVRAM—*all* software RAM-resident
- Persistent processes
  - Need to handle references to external state (e.g., network connections)
- *All* software based on transactions and recovery
- Micro-reboot — only restart what's broken/changed

The user's dream of NVRAM

- Boot and run from NVRAM—*all* software RAM-resident
- Persistent processes
  - Need to handle references to external state (e.g., network connections)
- *All* software based on transactions and recovery
- Micro-reboot — only restart what's broken/changed
- Security is paramount

Turning the dreams into reality

- Both dreams involves the elimination of some storage (i.e., files) and its replacement by persistent memory

- For success, must understand existing requirements and satisfy them.

Why do we store?

- To persist data
    - So that the data stay around
    - To have a record of how the data used to be
- To share data
    - Multiple processes accessing the same file
    - Sending a copy

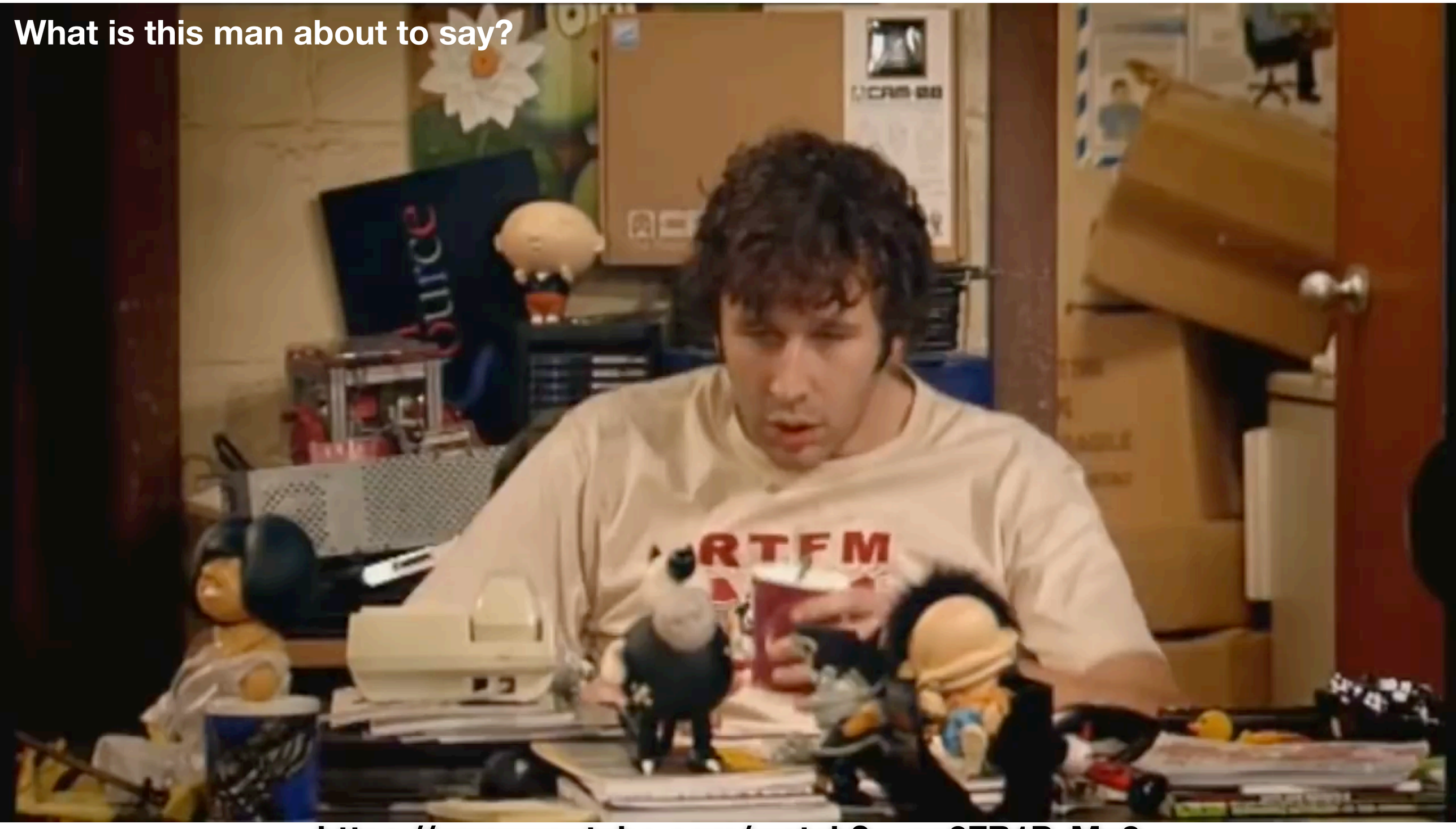
One size does *not* fit all

- Storage representations are often:
    - Simpler
    - Portable (language-, machine-, OS-independent)
    - Documented
    - Standardized

Size limitations of PM

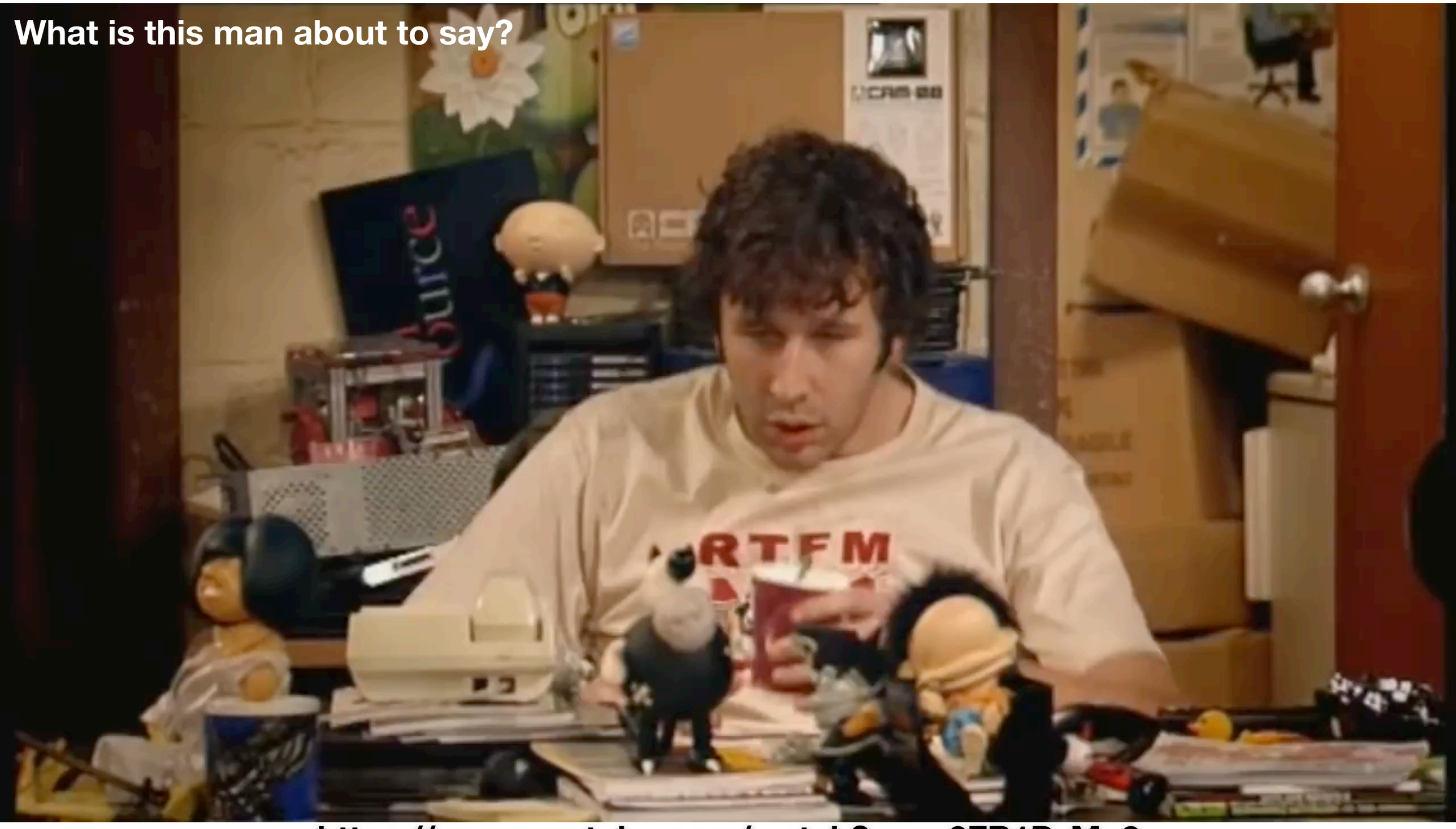
- NVRAM will have size limitations related to RAM:
  - Physical address size
  - Socket and other electrical limitations
    - Size and speed are opposed
- Virtualization can remove size limitations at the cost of speed:
  - Distributed Shared Memory (1980s)
  - Gen-Z consortium (now) — memory semantics across a network

Nightmare #1: restarts



https://www.youtube.com/watch?v=nn2FB1P_Mn8

Nightmare #1: restarts



https://www.youtube.com/watch?v=nn2FB1P_Mn8

“Have you tried turning it off and on again?”

- Poll: who has successfully restarted an application or a machine to resolve a problem in the last…
  - month?
  - week?
  - day?
- Do restarts work? Why?

Kinds of failures

Terminating

- Power loss
- Unrecoverable memory error
- Accidental termination by operator or external agent
- Fatal bug

Non-terminating

- Infinite loop / livelock
- Deadlock
- Data corruption caused by bug
- Memory leak

All handled by restart, *unless* corrupt data gets persisted

Hardware failures

- Hardware fails!
- Bit flips in DRAM are well-characterized
    - Typically a single bit, occasionally more
        - Usually due to ionizing radiation (cosmic rays, background sources)
- Consumer-grade hardware uses a parity bit to detect single-bit errors
- Enterprise-grade hardware uses multiple Error Checking and Correcting (ECC) bits for Single Error Correction, Double Error Detection (SECDED)
- Not a rare event in a data center

Errors in memory

Errors in memory

- Intel/Micron has not published failure mode or error rate data for 3D XPoint

Errors in memory

- Intel/Micron has not published failure mode or error rate data for 3D XPoint
- The modules will certainly use ECC techniques to reduce the raw error rates

Errors in memory

- Intel/Micron has not published failure mode or error rate data for 3D XPoint
- The modules will certainly use ECC techniques to reduce the raw error rates
- Existing error reporting methods (for DRAM) are used for Optane.

Errors in memory

- Intel/Micron has not published failure mode or error rate data for 3D XPoint
- The modules will certainly use ECC techniques to reduce the raw error rates
- Existing error reporting methods (for DRAM) are used for Optane.
- What happens when there is an uncorrectable error?

Errors in memory

- Intel/Micron has not published failure mode or error rate data for 3D XPoint
- The modules will certainly use ECC techniques to reduce the raw error rates
- Existing error reporting methods (for DRAM) are used for Optane.
- What happens when there is an uncorrectable error?
  - DRAM: kill the owning process and restart

Errors in memory

- Intel/Micron has not published failure mode or error rate data for 3D XPoint
- The modules will certainly use ECC techniques to reduce the raw error rates
- Existing error reporting methods (for DRAM) are used for Optane.
- What happens when there is an uncorrectable error?
  - DRAM: kill the owning process and restart
  - NVRAM: need to recover data from backup/log.

Errors in memory

- Intel/Micron has not published failure mode or error rate data for 3D XPoint

- The modules will certainly use ECC techniques to reduce the raw error rates

- Existing error reporting methods (for DRAM) are used for Optane.

- What happens when there is an uncorrectable error?

  - DRAM: kill the owning process and restart

  - NVRAM: need to recover data from backup/log.

Some programming required

Security

- Malware often exploits access to memory
  - Buffer overrun, return-oriented programming, etc.
- NVM increases the attack surface: more stuff in memory, things which were in the filesystem now just a load or store away
- Optane encrypts “data at rest”

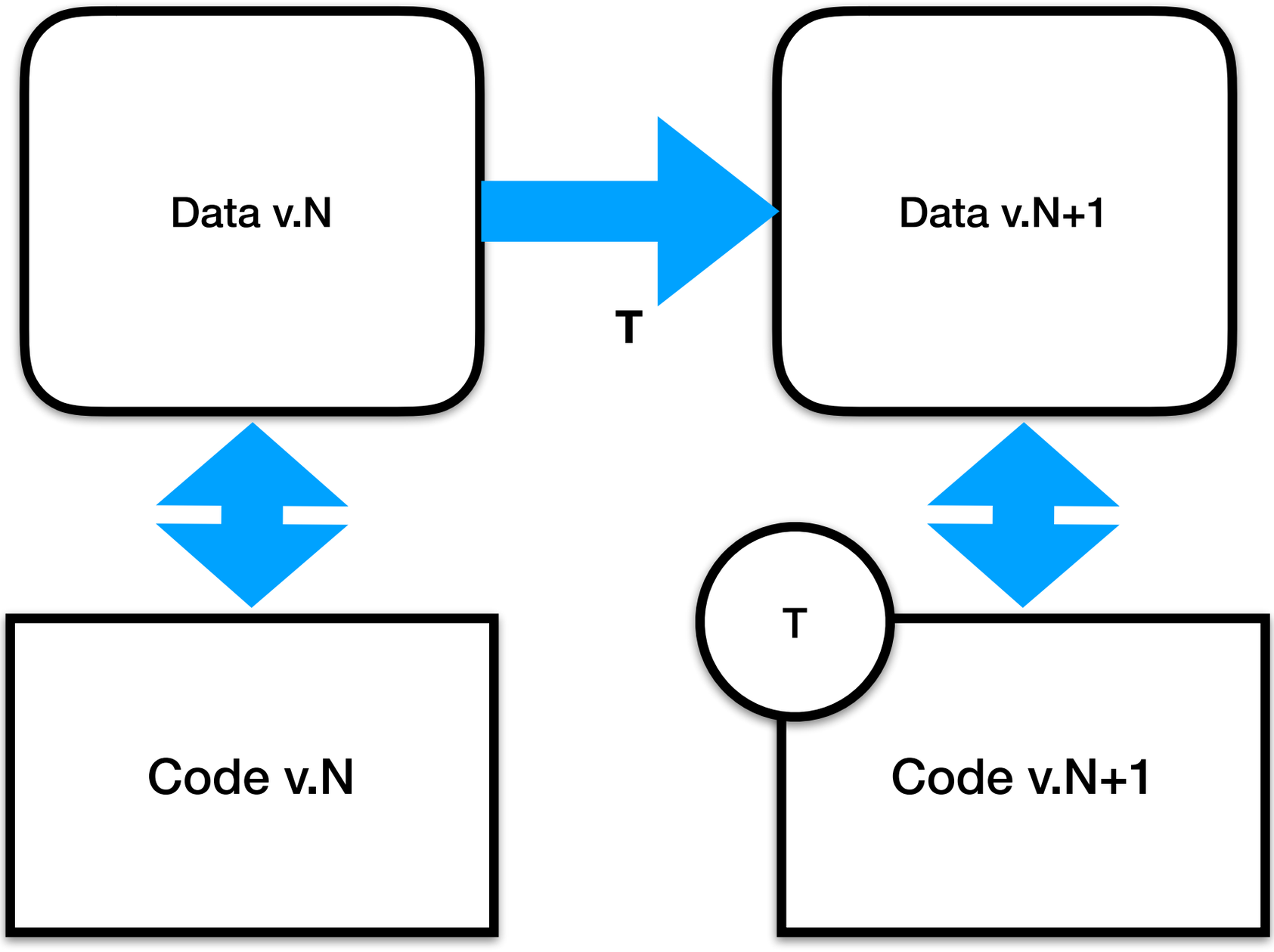
The Coming Persistence Apocalypse Part 2

Mario Wolczko
Architect
Oracle Labs
May 21, 2019

Nightmare #2: Software changes…

- …and with it, the structure of data
- If the data persist, we must be able to restructure as the software changes.
- Databases call this *schema evolution*.
- Most languages have no conceptual framework for this.
- Interactive systems typically address this, e.g., Smalltalk
- Can also be tackled at the binary level (e.g., ksplice)
- Generally described as *Dynamic Software Update*.

Transforming data as code evolves



Challenges

- Preserving relationships: identity, equality, sharing, etc. Example: if two variables refer to the same object, and the object is updated, then they should still refer to the same object after, even if they are on the stack, in objects, globals, etc.

- What are the types and values available in T? Need to be able to refer to both old and new types.

- Order of update?

- Updating derived values

- The typical approach is to provide default values for new variables, and let ad-hoc code perform re-initialization

- Primitive operations are needed of the kind
  - “Find all values of this type”
  - “Find all references to this object”

    Tricky, given that ad-hoc code is being run (and new objects and values are being created)
  - “Replace all references to A by references to B” (A and B are of different types)

Language issues

Consistency

- Software must indicate consistent states
- Runtime must be able to recover to a consistent state
- Typically involves taking some kind of snapshot from time-to-time, and logging important events between snapshots

Restarting after abnormal termination
1. Externally caused

- Having restarted after abnormal termination, must be able to recover persistent data to a consistent state

- If termination was caused by an **external** event, then the state just before termination was consistent — but we have lost all volatile data and the external environment has changed.

- Must recover to the state after the last completed change*, and discard updates made from incomplete changes.

* Or run forward to a new consistent state

Restarting after abnormal termination
1. Externally caused

- Having restarted after abnormal termination, must be able to recover persistent data to a consistent state

- If termination was caused by an **external** event, then the state just before termination was consistent — but we have lost all volatile data and the external environment has changed.

mostly

- Must recover to the state after the last completed change*, and discard updates made from incomplete changes.

* Or run forward to a new consistent state

Restarting after abnormal termination
2. Bugs

- If termination was caused by a **bug**, then we need to determine an earlier, correct state, recover to that, and somehow deal with all the inputs received since that time.

- This is tricky.

- Maybe this is why the current separation into memory and storage works: storage operations are usually correct, and most bugs are confined to RAM

A fallback strategy

- Memory is a scratchpad and cache
- Make this explicit using PM: separate the core data from the cached structures and the scratchpad
- The scratchpad can be volatile
- The caches can persist, but are discarded on failures of behavior
- Correctness of the core data must be preserved at all costs

Consistency

- Software must indicate consistent states
- Runtime must be able to recover to a consistent state
- Typically involves taking some kind of snapshot from time-to-time, and logging important events between snapshots

Consistency — of what?

- From a single data structure — transactions on objects
- … to the entire of the program state — snapshot the heap

Database transactions

- Snapshot - update - commit (or abort and roll back)
- If executed serially, commit can be global. Early DBs were like this.
- To improve performance, transaction are executed concurrently.
- Optimistic concurrency — abort if transactions conflict
- Overall control is in the client program — outside the DBMS

ACID

ACID

- Atomic - each transaction executes “all or nothing”

ACID

- Atomic - each transaction executes “all or nothing” ✓

ACID

- Atomic - each transaction executes “all or nothing” ✓
- Consistent

ACID

- Atomic - each transaction executes “all or nothing” ✓
- Consistent ✓

ACID

- Atomic - each transaction executes “all or nothing” ✓
- Consistent ✓
- Isolated - transactions execute as if serialized

ACID

- Atomic - each transaction executes “all or nothing” ✓
- Consistent ✓
- Isolated - transactions execute as if serialized ?

ACID

- Atomic - each transaction executes “all or nothing” ✓
- Consistent ✓
- Isolated - transactions execute as if serialized ?
- Durable

ACID

- Atomic - each transaction executes “all or nothing” ✓
- Consistent ✓
- Isolated - transactions execute as if serialized ?
- Durable ✓

Is recovery implicit or explicit?

- Implicit:
  - Failure terminates program
  - At restart, state is rolled back
- Explicit:
  - A transaction can be aborted but execution continues
  - Can’t rollback everything

Specifying persistence

- By type
- Per object
    - At allocation
    - Dynamically — move objects
- By reachability

Type

- How to avoid code duplication?
    - Some code applies to both volatile and persistent values
- Can legacy code be used without change? On persistent objects?

Static types

- Static type checking assumes a consistent type regime in the software before any data are created

- Instead, we will need to compose systems from type-checked software and an independently checked heap (or heap region)

Compiling persistent code

- Compiler should emit write-backs and syncs for persistent updates
- In addition to type soundness, must enforce lifetime correctness
    - Shouldn’t reference volatile data from persistent objects
- Automatically maintaining recovery metadata

Persistence by reachability

- Designate some objects/values as *persistent roots*

- Anything reachable from the roots is automatically persistent; everything else is volatile.

Persistence by reachability

- Reference assignment can result in immediate relocation to persistent memory
    - Of a subgraph
    - Old referenced value can become volatile - can be moved lazily, or GCed
    - How to avoid entangling the core state with other state?
    - More on this shortly…

Programming language and runtime issues

- GC: must be resilient to failure, scale to enormous heaps
- Unmanaged code — trashing the persistent managed heap
  - Unmanaged persistent data?
- Heap layout stability and implementation-independence
- Monolithic persistent heap? Or partitioned, modular?
- Objects which defy persistence

Objects which defy persistence

- If an object refers to external state it may fail when the external state changes and the object does not reflect the change. Examples:
  - Files and file descriptors, network connections
  - Process IDs
  - Environment and locale
  - Stack frames for external code
  - ...

Resumption issues

- But what if the external state changes while the object is in hibernation?
- Do we check all the objects when we wake? At next use?
- How do we even know to check? File descriptor is an int, PID is an int, filename is a string, …
- **Code must be explicit about capturing external state, and provide a way to check and update.**

Case study: NVM-Direct

- An extension of C for persistence, by Bill Bridge
  https://github.com/oracle/nvm-direct
- Explicit separation of volatile memory from persistent regions
- Adds nested transactions, associated locks and compensation code
- A transaction is associated with a region
- Pointers to NVM, assignments to NVM: use new syntax
  - Compiler generates flushes and undo records
- Structs allocated in NVM must be declared persistent
  - Compiler and library use self-relative pointers

An example

```
typedef persistent struct mystruct mystruct;
persistent struct mystruct {
    nvm_mutex mutex;
    int count;
}
nvm_desc desc; // region descriptor
int increment(mystruct ^my){
    int ret;
    @ desc {
        nvm_xlock(%my=>mutex);
        my=>count@++;
        ret = my=>count;
    }
    return ret;
}
```

From https://github.com/oracle/nvm-direct/blob/master/NVM_Direct_Presentation.pdf

Software evolution in NVM-Direct

- In NVM-Direct, the programmer is responsible for deciding when a new version has been introduced.
  - Allows for bug fixes and other small changes
    - No need for complex algorithms to infer when a change is significant.
- Every version of each type has an associated unique ID
- The program and the data contain the ID and they must match
- The program can supply code to update data to a more recent version.

Java and NVM

- Millions of programmers
- Billions of lines of code
- My goal: large-scale near-term adoption
- Must limit disruption to existing code and practices

Existing approaches

- Intel Persistent Collections for Java
  - Off-heap persistence, Persistent types
- Espresso: persistent `new`
- AutoPersist: persistence by reachability
- PJama (c.1999) and related

AutoPersist: persistence by reachability

- Shull, Huang and Torrellas (3 papers so far)
  PLDI: https://doi.org/10.1145/3314221.3314608

- Split the heap into persistent and volatile parts, transparently to the application

- Label some statics with @durable_root annotation

- When an object becomes reachable from a persistent root, move it to the persistent heap

  - Forwarder left behind, cleared at next GC

- Add per-thread failure-atomic regions

  - Write-ahead undo logging

- Heuristics to determine when to allocate in persistent heap

Persistence by reachability: is it a good thing?

- Reachability is a non-local property
- A single assignment can cause arbitrary data to become or cease to be reachable
- Newly reachable data has to be durable by the end of the enclosing transaction: could result in a long pause
- Unreachable live data should eventually be moved to DRAM
- Some objects should not/cannot be persisted
- Could result in persistent memory leaks

Partitioned heaps
(following NVM-Direct)

- Partition the heap into volatile and persistent regions, visible to the programmer.

- Persistent regions are mapped from DAX files.

- Each object is allocated in a specific region.

- Disallow cross-region references, except from volatile regions.

- When `main` begins, there is a single volatile region.

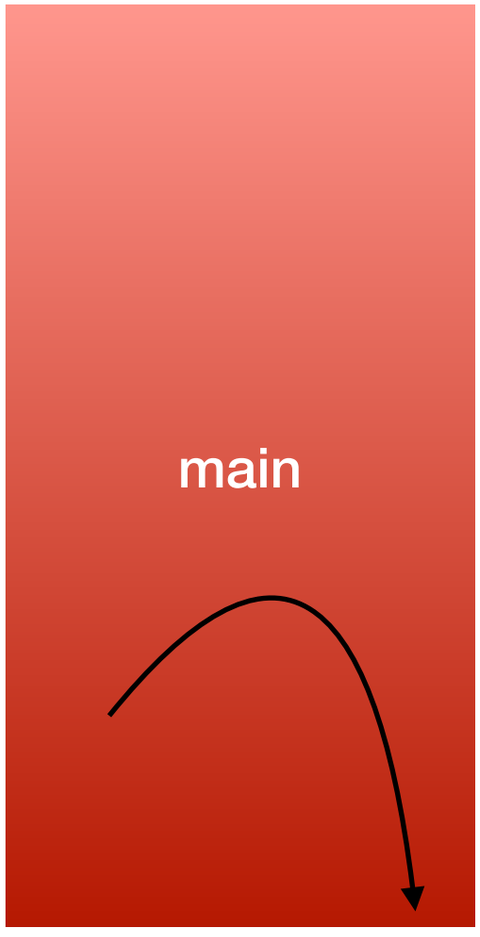
P1

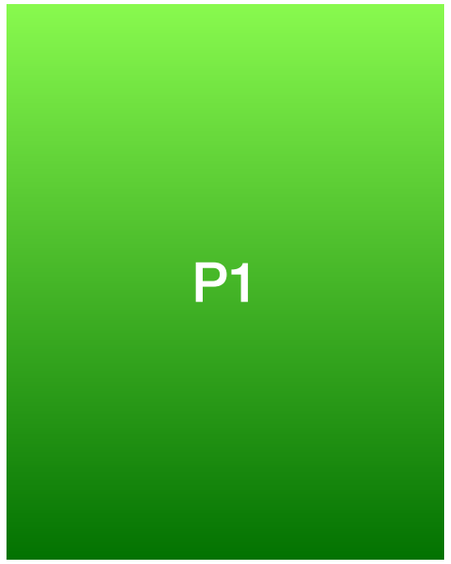
main

P2

volatile

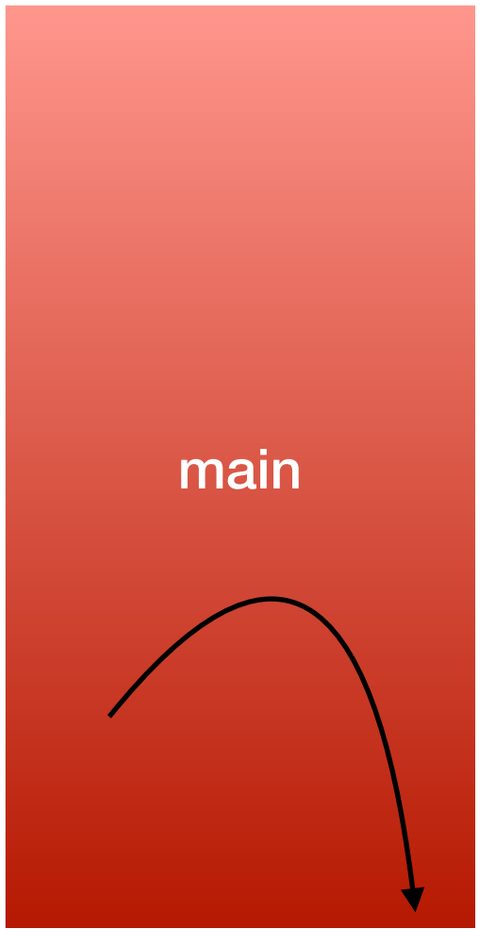
Persistent

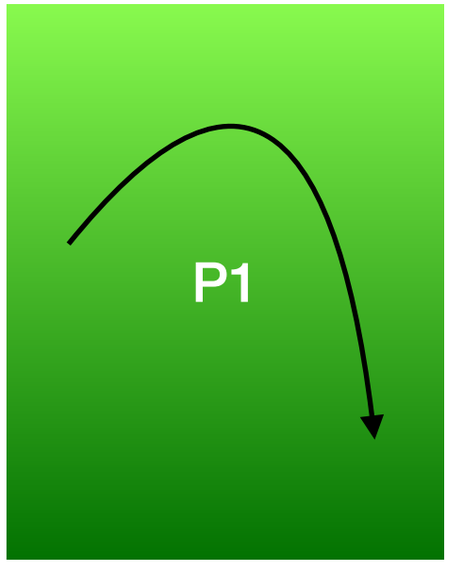




volatile

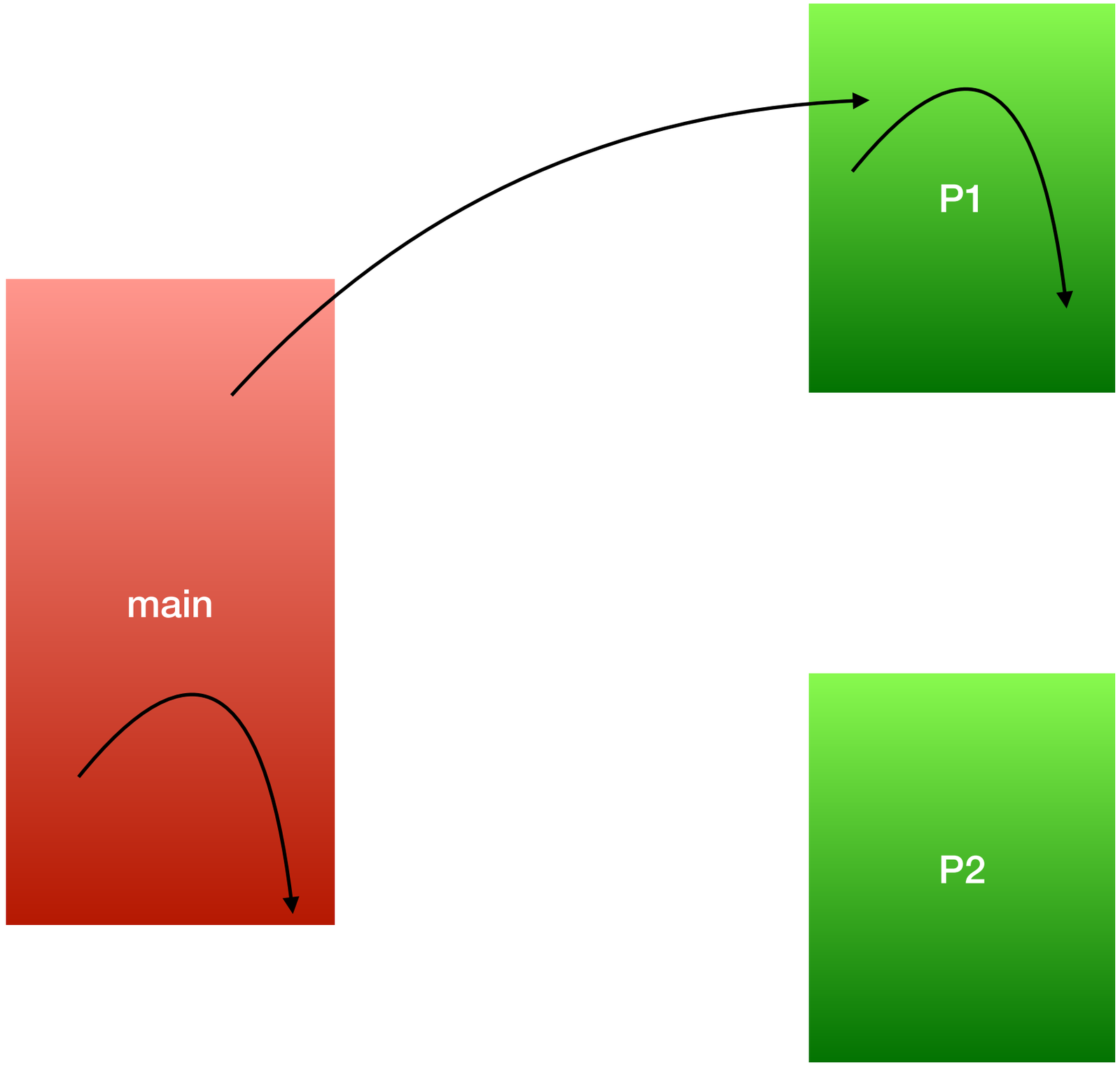
Persistent





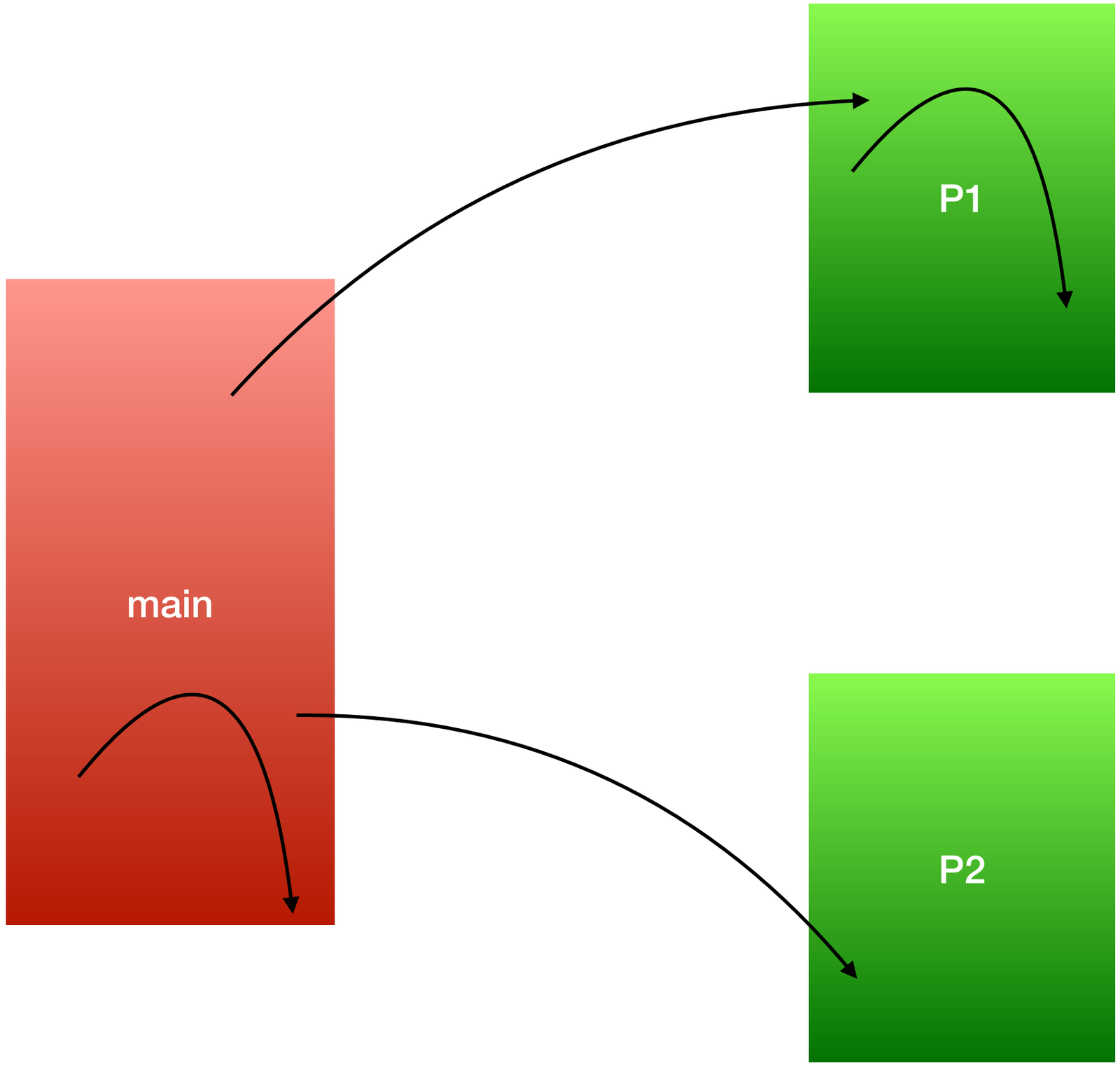
volatile

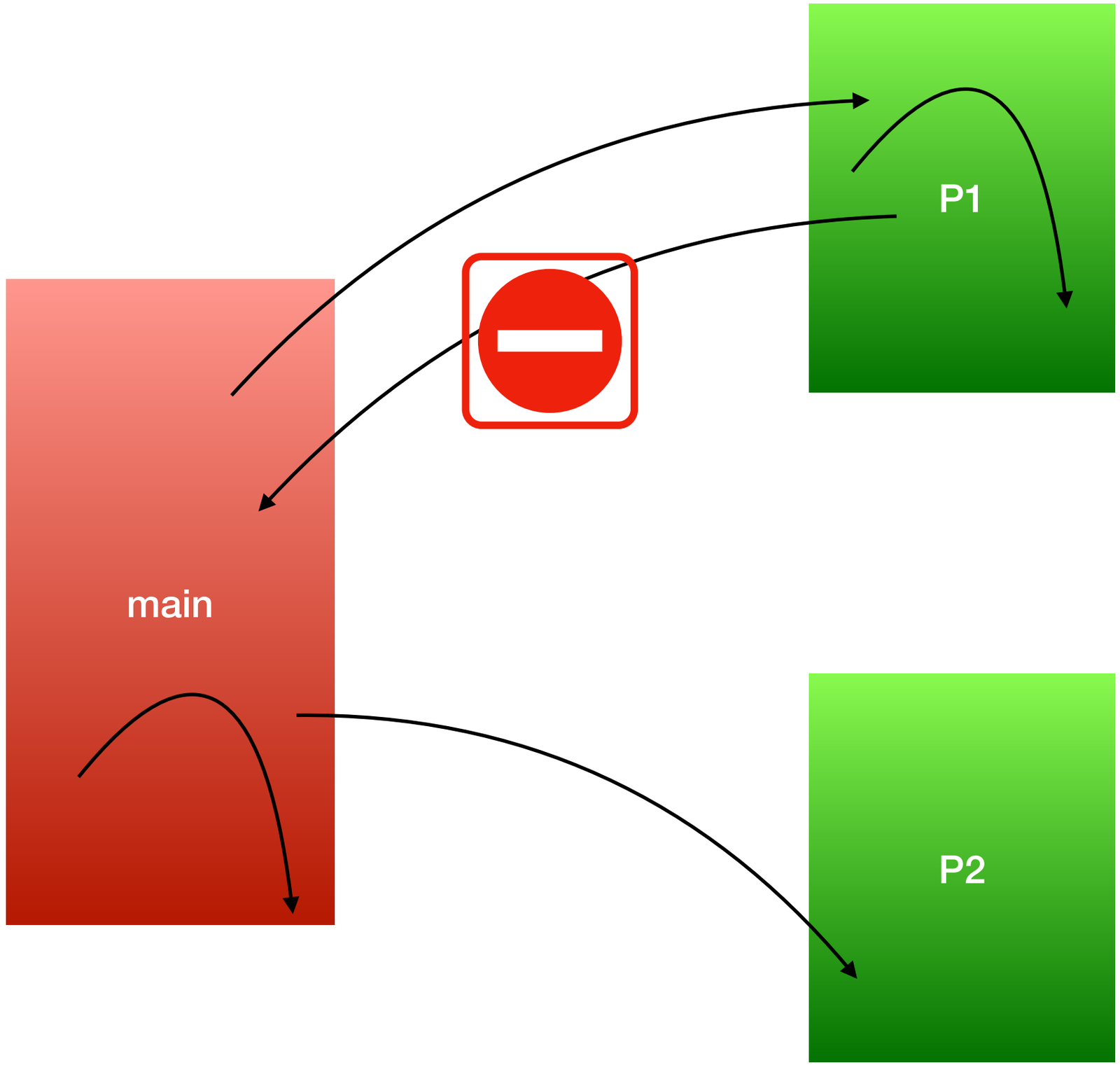
Persistent

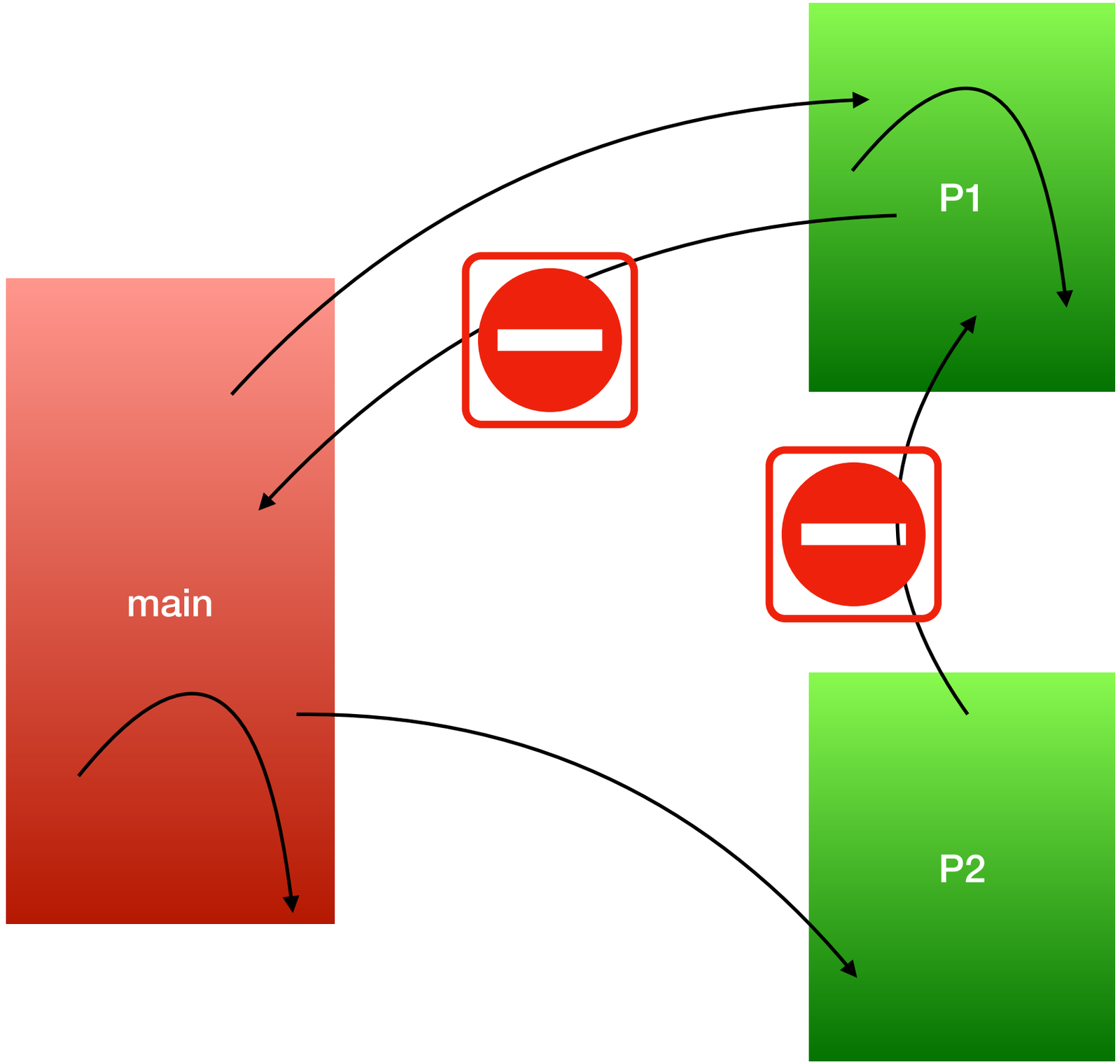


volatile

Persistent


main
P1
P2
volatile
Persistent


P1
main
P2
volatile
Persistent


P1
main
P2
volatile
Persistent

The current region

- Each thread has a notion of *the current region* — all allocations by the thread are in the current region.

- The current region can be changed. Code can be wrapped to use a specific region, e.g.:

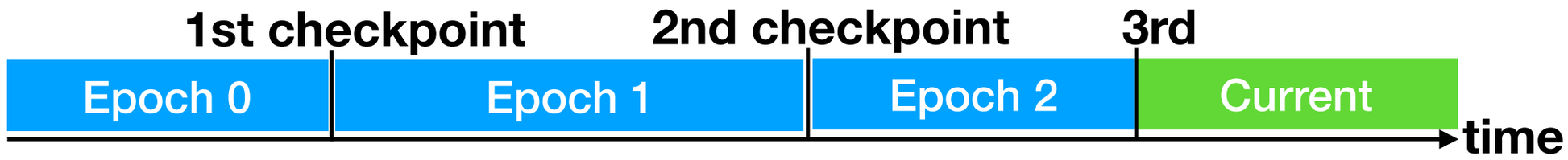
```
myRegion.run(() -> {h = new HashMap();});
```

Checkpoints

- The program can invoke a checkpoint primitive which snapshots the current state of one or more regions
- At re-attach, a region is recovered to the last checkpoint
- Regions can be checkpointed independently or together. Example: checkpoint a log after every addition, the DB less frequently

Epochs

- Each region goes through a series of epochs. An epoch is the period between two checkpoints (or before the first).



- A checkpoint freezes the state of the region within that epoch. The data within an old epoch are immutable.

- When an object is modified for the first time within the current epoch, a copy is made in NVM. All subsequent modifications within the current epoch are to this copy.

- The running program can only observe the most recent version of an object.

Recovery

- When a warm start occurs, and the latest epoch is uncommitted, recovery takes place.

- Recovery discards the current, uncommitted epoch and reverts to the state at the end of the previous epoch.

- Locks held on objects in the region are re-initialized.

Managing checkpoints

- Each checkpoint persists until explicitly discarded.
- If the oldest checkpoint is deleted, objects in that checkpoint without modified copies in the next epoch are absorbed into the next epoch. If an object is superseded by a copy in the next epoch, it can be discarded.
- An intermediate-age checkpoint can also be deleted, merging the two epochs around it.
- We can also recover to any checkpoint, discarding all the later epochs.

The programmer's burden

To convert an existing application to use NVM, the programmer:

- Partitions the heap into DRAM and NVM regions
- Adds the region creation and loading logic,
- Wraps NVM object creations in `Region.run(Runnable)`.
- Adds calls to `checkpoint()` at places where the region is consistent.

Example: simple phone directory

(following an example by Eliot Moss)

```
public class PhoneDirectory {
  static HashMap<String, String> dir;

  public static void main(String[] args) {
    …
  }

}
```

```
static HashMap<String, String> dir;
```

…

Inside main():

```
Region<HashMap<String, String>> region;

try {
  region = Region.attachOrCreate(“phonedir.region",
                () -> { return new HashMap<String, String>(); });
} catch (InvalidRegionFileException |
         ClassCastException | IOException ex) {
  System.err.println(“Region error");
  System.exit(1);
}

dir = region.root();
```

Further along in main():

```
// command received to add an entry for name and number
…
addEntry(name, number);
```

```
private void addEntry(String name, String number)
  throws PDException
{
  if (dir.containsKey(name)) {
    throw new PDException("name already in directory");
  }
  dir.put(name, number);
}
```

Without persistence

```
private void addEntry(String name, String number)
  throws PDException
{
  if (dir.containsKey(name)) {
    throw new PDException("name already in directory");
  }
  region.run(() -> dir.put(name, number));
  region.checkpoint();
}
```

Region management added - but there’s a bug

```
private void addEntry(String name, String number)
  throws PDException
{
  if (dir.containsKey(name)) {
    throw new PDException("name already in directory");
  }
  region.run(() -> dir.put(new String(name),
                           new String(number)));
  region.checkpoint();
}
```

```
private void addEntry(String name, String number)
  throws PDException
{
  if (dir.containsKey(name)) {
    throw new PDException("name already in directory");
  }
  region.run(() -> dir.put(new String(name),
                           new String(number)));
  region.checkpoint();
}
```

- The String parameters must be copied to the region.

```
private void addEntry(String name, String number)
  throws PDException
{
  if (dir.containsKey(name)) {
    throw new PDException("name already in directory");
  }
  region.run(() -> dir.put(new String(name),
                           new String(number)));
  region.checkpoint();
}
```

- The String parameters must be copied to the region.
- For immutable data, the JVM could do this automatically.

Using partitioned heaps

- Inherently volatile objects reside in a volatile partition
    - Their classes are annotated, so that they are never created in persistent partitions (e.g., threads)
    - Or, add behavior to be invoked at attach to recreate state (e.g., files)
- Inter-region references can be handled like this too.

Implicit checkpoints

- main() is the only transaction
- return 0 is commit (checkpoint), non-zero is abort
- Attached regions specified by config info, not code
- Viable iff startup is very fast
- Simple!

Implementation sketch

Assumptions, requirements and desiderata

- A region can be loaded anywhere (although it may be possible to have it commonly re-loaded at the same address). Hence it must be position-independent, or cheaply (and preferably incrementally) relocatable.

- We'd like checkpoints to be fast, so they can be used relatively often

- We'd like checkpoints of small regions to be even faster

- Steady-state performance of long-running programs is paramount

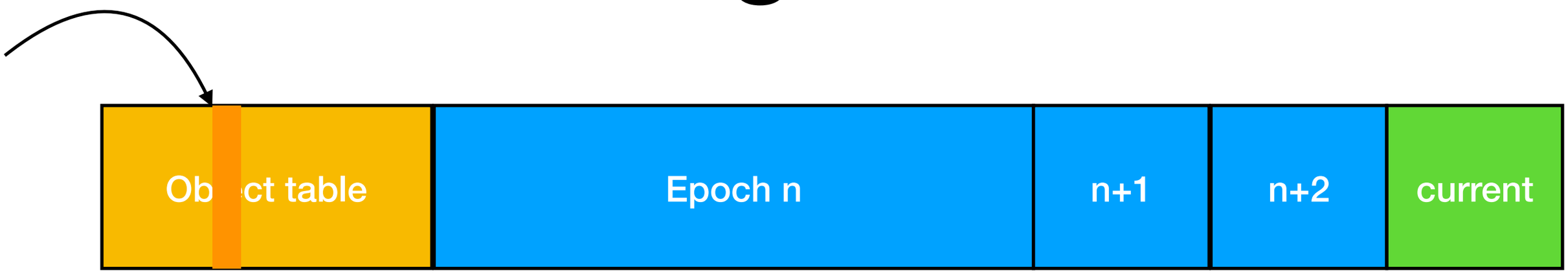
- Crashes and recovery are infrequent

- NB: NVM is much cheaper than DRAM, so we can trade space for time —but caches won't get bigger or cheaper because of NVM.

The heap structure of a region

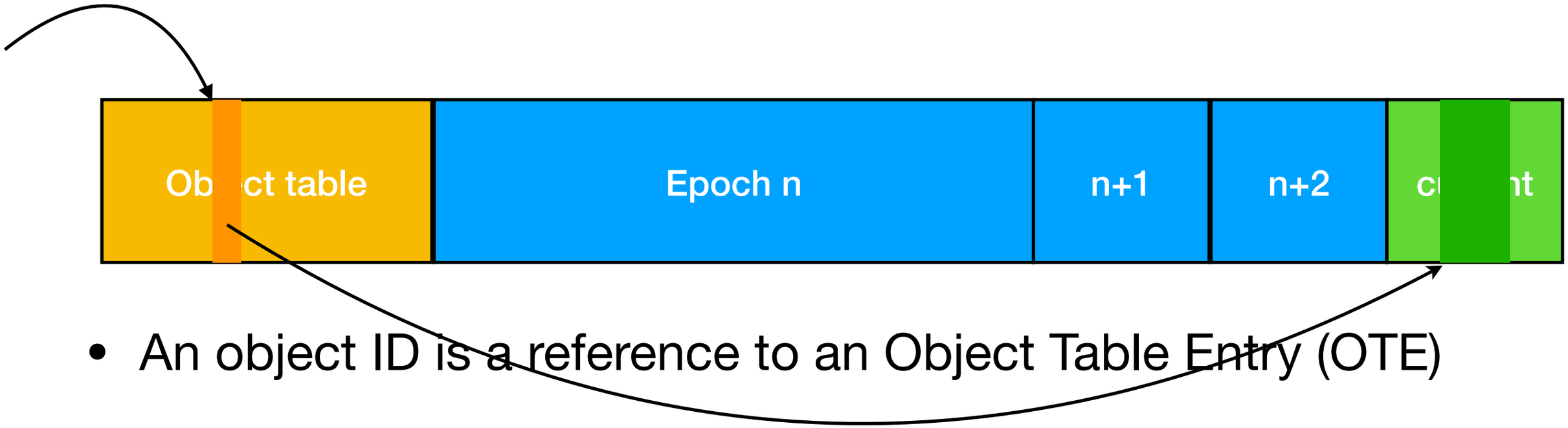
- An object ID is a reference to an Object Table Entry (OTE)
  - Self-relative when in the heap
- The OT references the most recent version of an object.
- Each version references the next oldest from its header (reserved header space).

The heap structure of a region



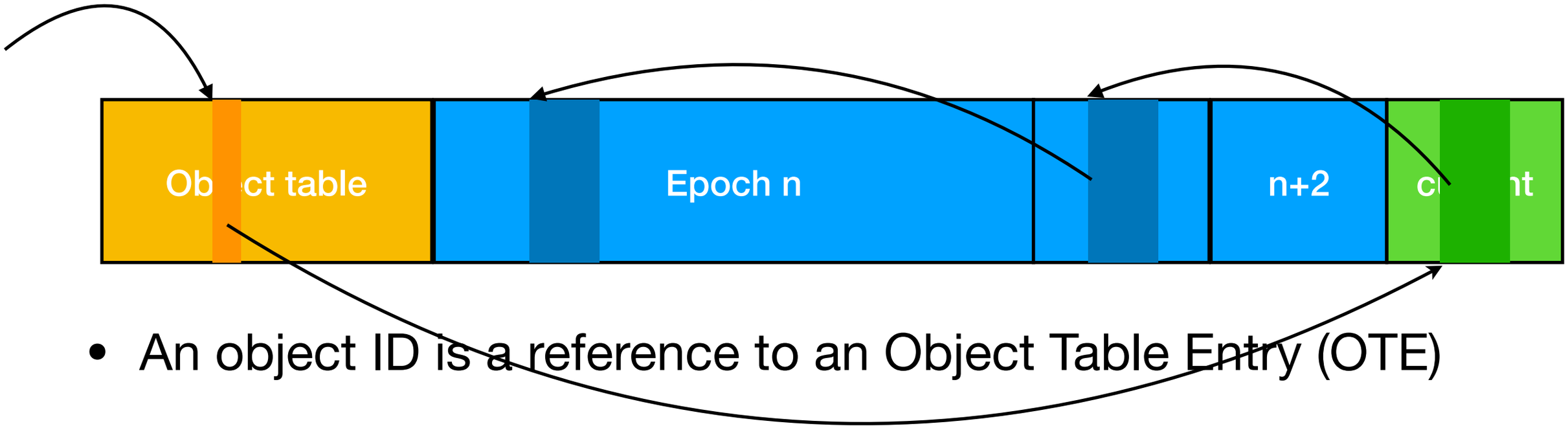
- An object ID is a reference to an Object Table Entry (OTE)
  - Self-relative when in the heap
- The OT references the most recent version of an object.
- Each version references the next oldest from its header (reserved header space).

The heap structure of a region



- An object ID is a reference to an Object Table Entry (OTE)
  - Self-relative when in the heap
- The OT references the most recent version of an object.
- Each version references the next oldest from its header (reserved header space).

The heap structure of a region



- An object ID is a reference to an Object Table Entry (OTE)
  - Self-relative when in the heap
- The OT references the most recent version of an object.
- Each version references the next oldest from its header (reserved header space).

Some details

- Write-protect old epochs to catch updates; trap makes a copy in the current epoch.
- Need a store barrier to check for cross-region stores.
    - A region is actually chunked; each chunk obeys alignment restrictions. Regions have a unique ID in the chunk header. DRAM regions have ID=0.

Research questions

1. Is this a useful model? When is it not usable? What kinds of mistakes are made, and what is the mitigation?

2. Which libraries and apps need to be changed? How do we find them?

3. Make it work; make it fast. How fast?

    - Peak performance, checkpoint latency

Some other research topics

The time horizon

- Short-term:
  - Must address existing languages, software, practices
- Long-term:
  - What is the best way to construct persistent software?

VM topics

- GC at Terabyte scale and beyond
    - Largest CascadeLake machine has 24TB of PM, 6TB of DRAM. Does GC scale to this?
- Caching compiled code — what? how? When to recompile?
- How to make transactions efficient in your favorite language/ runtime
- Optane puts wear leveling in the DIMM. Could you do better having it managed by the OS/VM?

VM implementation-independent, modular object and heap layout

- For heaps to be portable across VMs for the same language

- For a VM to be able to support multiple formats simultaneously

How to backup a live heap

- Can it be done independently of the application?

- Piggy-backed on GC?

What are the implications of self-relative data?

- Should there be hardware support? (addressing modes, instructions, …)

- Language support in an unmanaged language? (NVM-Direct)

Dynamic software update and schema evolution

- Most languages haven’t been addressed
- What if you don’t have source code? (ie binary patch)
- Multiple versions co-resident?
  - UpgradeJ

Security

- How to minimize the attack surface of persistent memory?

- How to detect tampering?