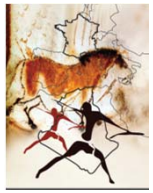
NON-VOLATILE MEMORIES: A NEW DEAL FOR OPERATING SYSTEM DESIGN?

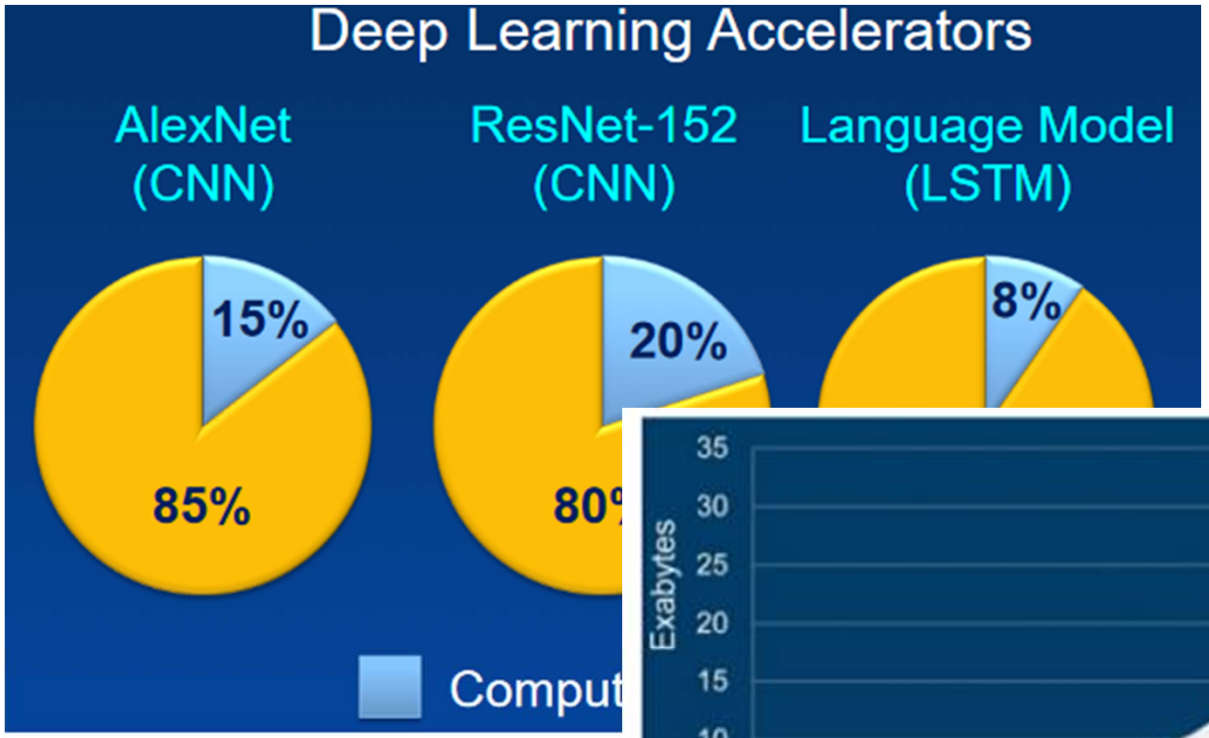
Stéphane Rubini, Jalil Boukhobza
Lab-STICC, University of Western Brittany, France

Alger, April 17th 2018



PHC Tassili

Memory Wall and Data Explosion



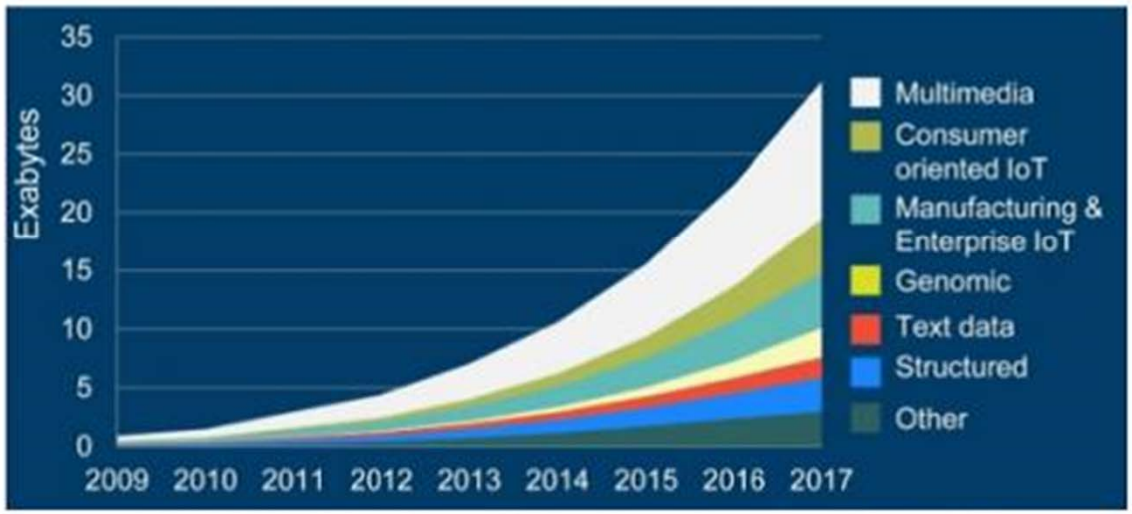
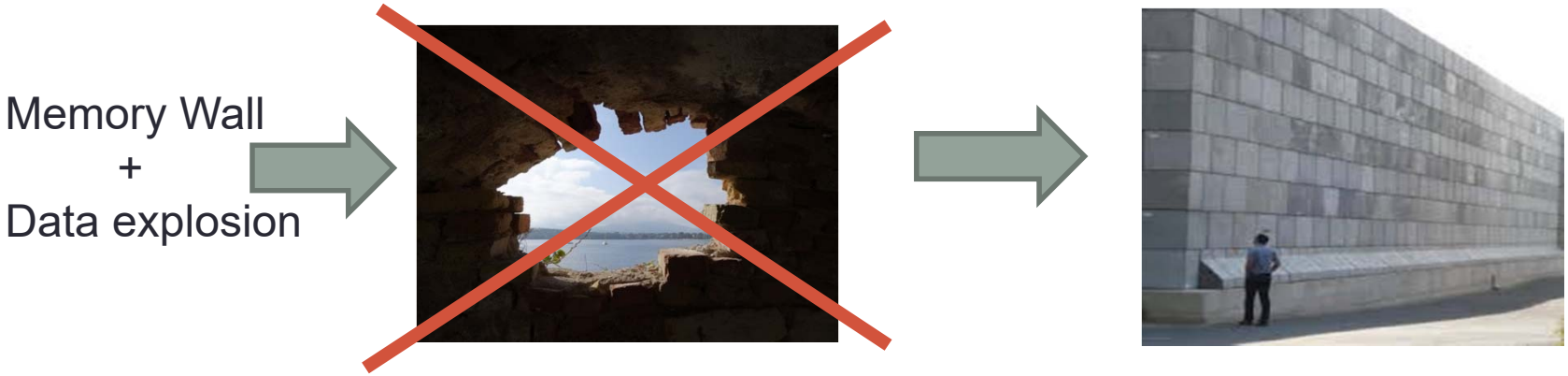


Fig. 2. Estimated data growth in various categories and overall.[1]

Memory hierarchy limits

- Memory hierarchy: tradeoff capacity/speed
  - How many levels? About 8 today…
  - Temporal locality of processor accesses?
- Radical change of computer architecture
  - Parallel access to memory array
  - In-memory computing
- Or improve the memory management in current machine
  - New memory technology
  - Improve the memory service in OS

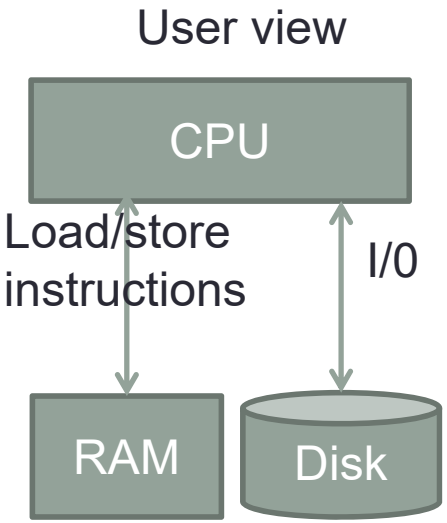


Outline

1. Introduction
2. Storage/memory convergence
   - Non-Volatile memory arrival
3. Operating System Mechanisms for NVM
   - NVM as a universal memory
   - DRAM/NVM hybrid memory architecture
     - NVM as memory
     - NVM in the middle
     - NVM as storage
4. Conclusion

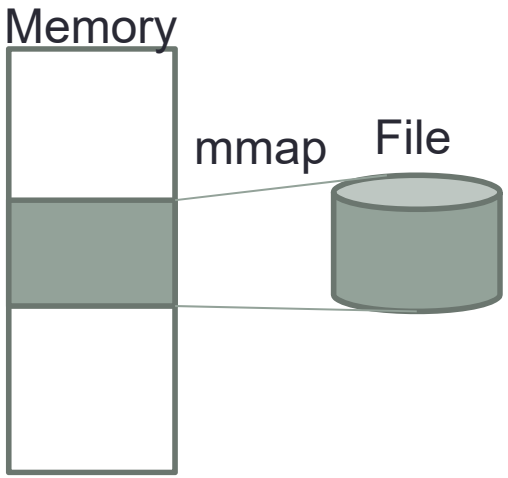
Storage and main memory

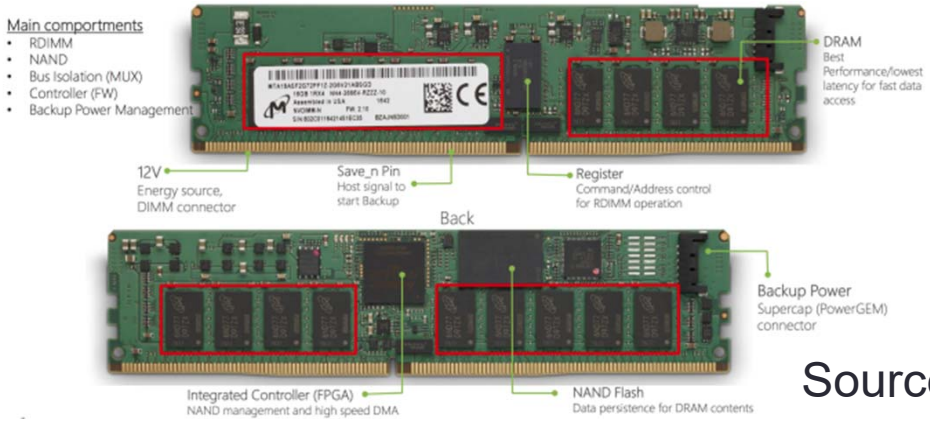
- User view: Two access models and two address spaces
  - Main Memory (primary storage)
    - Short term memory → volatile
    - Fast and direct memory CPU accesses (load/store)
  - Storage (secondary storage)
    - Long term memory → permanence
    - High capacity
    - Data identification (file name), I/O accesses



Storage/memory convergence: Fuzzy borders

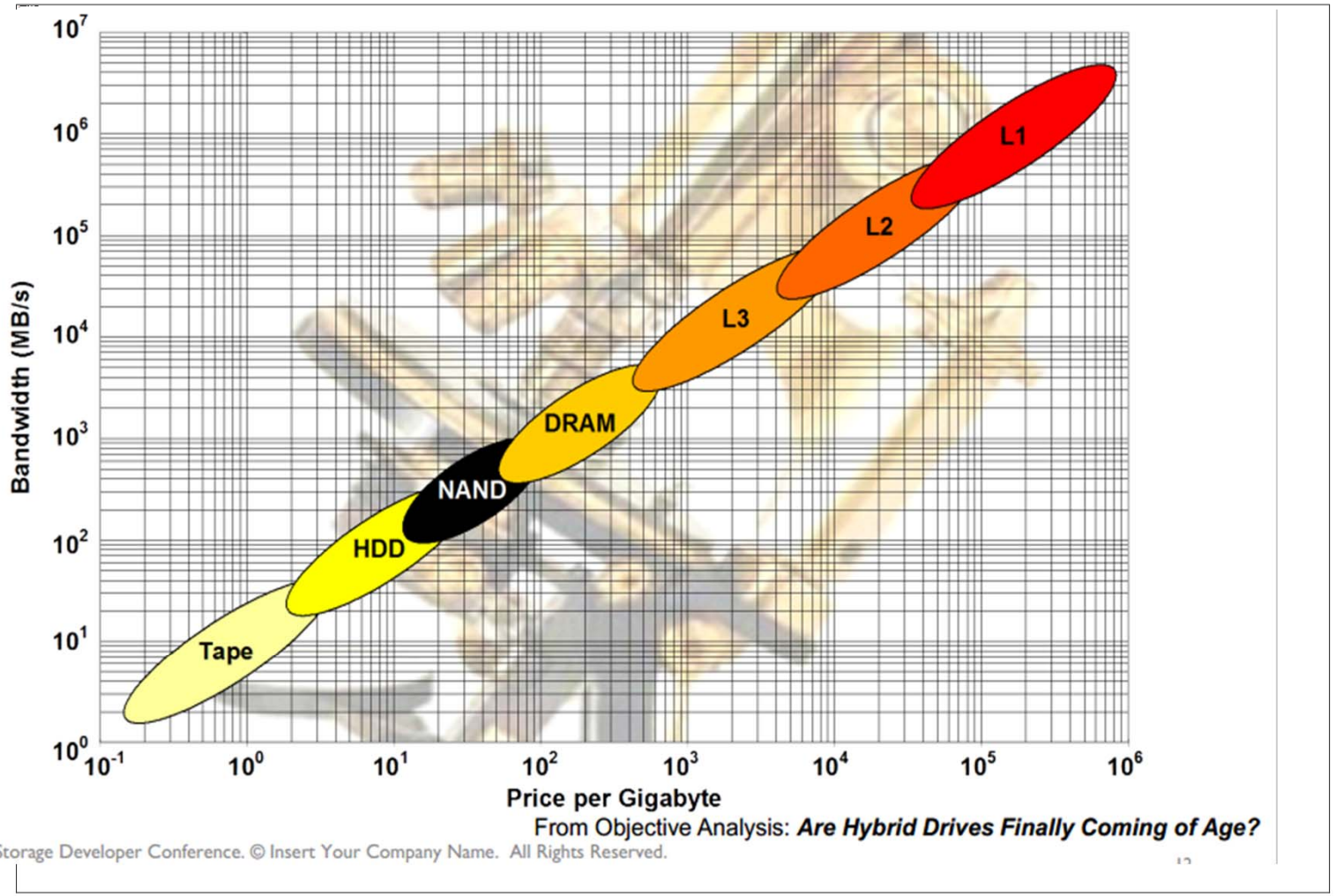
- Virtual memory
  - Memory Mapped File (*mmap()*)
  - Swap partition
- Disk-like non-volatile memory
  - Persistent RAM disk
  - Block addressable (IO API), but fast
- Memory-like non-volatile memory
  - Byte-addressable
  - But persistent
  - NVDIMM DRAM/Flash for instance





Source: micron

NAND Flash in the hierarchy



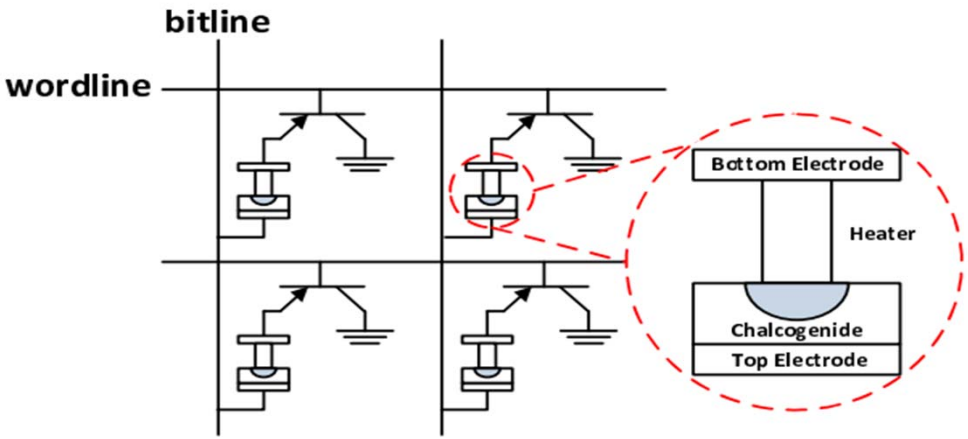
Non-Volatile Memory: characteristics

	SRAM	DRAM	HDD	NAND flash	STT-RAM	ReRAM	PCM	FeRAM
Cell size (F^2)	120-200	60-100	N/A	4-6	6-50	4-10	4-12	6-40
Write endurance	10^{16}	$>10^{15}$	$>10^{15}$ (pb: mechanical parts)	10^4-10^5	**10^{12}-10^{15}**	**10^8-10^{11}**	**10^8-10^9**	**10^{14}-10^{15}**
Read Latency	~0.2-2ns	~10ns	3-5ms	15-35 µs	**2-35ns**	**~10ns**	**20-60ns**	**20-80ns**
Write Latency	~0.2-2ns	~10ns	3-5ms	200-500µs	**3-50ns**	**~50ns**	**20-150ns**	**50-75ns**
Leakage Power	High	Medium	(mechanical parts)	Low	**Low**	**Low**	**Low**	**Low**
Dynamic Energy (R/W)	Low	Medium	(mechanical parts)	Low	**Low/High**	**Low/High**	**Medium/ High**	**Low/High**
Maturity	Mature	Mature	Mature	Mature	Test chips	Test chips	Test chips	Manufactured

Cf [5] for details

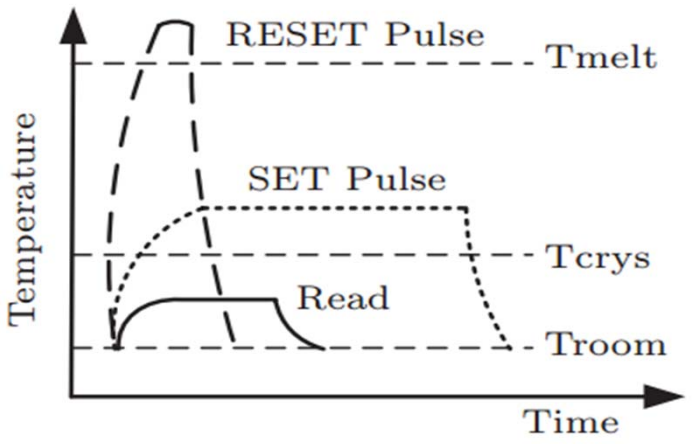
Example: Phase Change random access memory (PCM or PRAM)

- Memory cell: thin layer of chalcogenide such as Ge2Sb2Te5 (GST) + two electrodes wrapping the chalcogenide + heater
- Resistive NVM → use of resistance to represent a bit
- chalcogenide material: subject to rapid amorphous-to-crystalline phase-change process electrically initiated

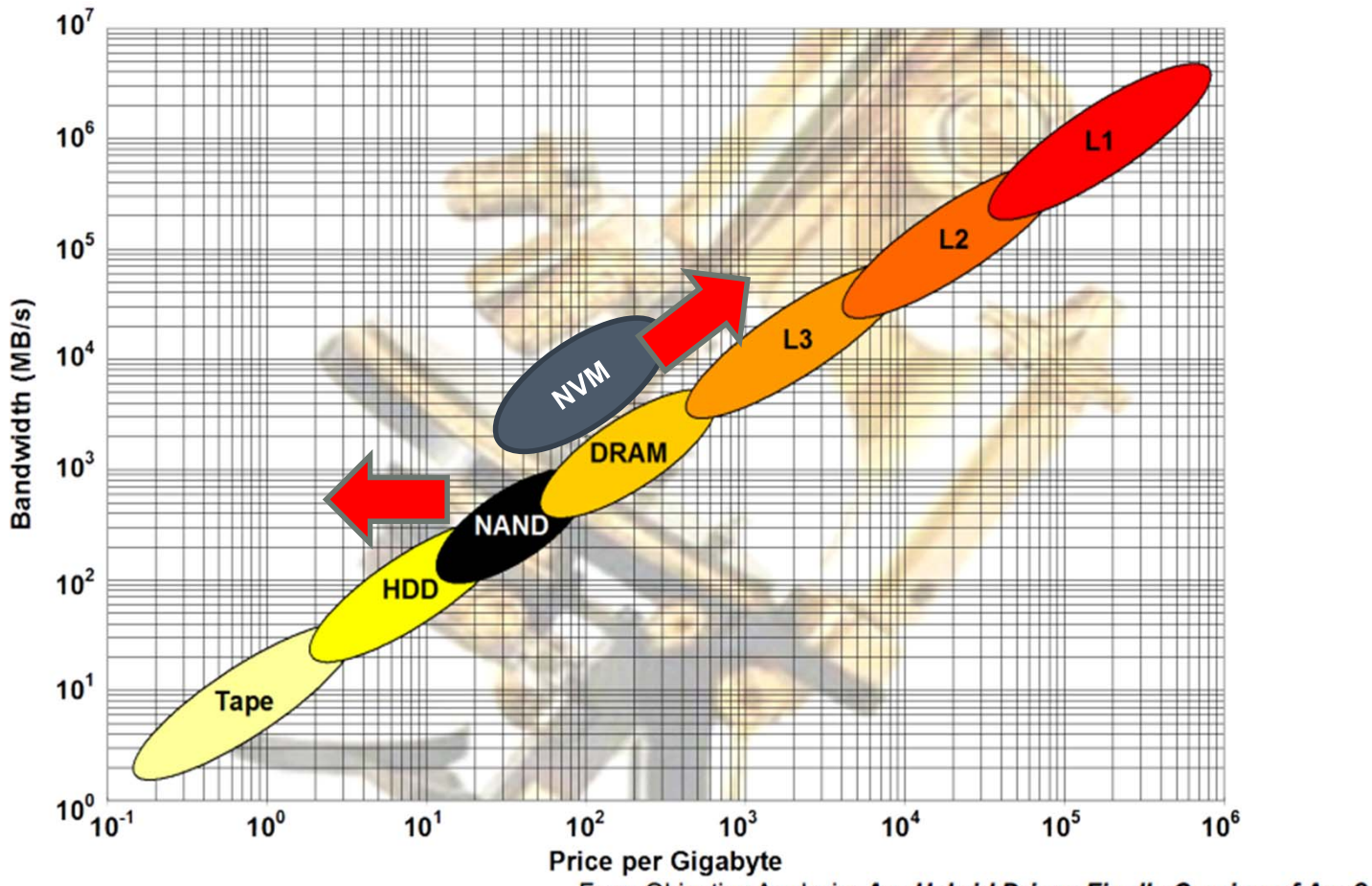


PCM (2)

- Short but high voltage pulse → GST heated above the melting temperature T_{melt}
  - Amorphous state (Reset, bit=0)
- Long pulse but low voltage → GST heated above the crystallization temperature T_{crys}
  - Crystalline state (Set, bit=1)
- Ratio between the resistance of a RESET phase [Kim08]
  - comprised between 10^2 and 10^4
  - → MLC (Multi level Cell)



NVM in the hierarchy



From Objective Analysis: ***Are Hybrid Drives Finally Coming of Age?***

12

That's new or different with NVM?

- Almost all outcoming NVMs present a new mix of characteristics:
  - Read/write asymmetry (time and consumption)
  - Limited write endurance
  - Higher capacity than DRAM (quite soon)
  - Permanent (non-volatile)
  - Byte addressable

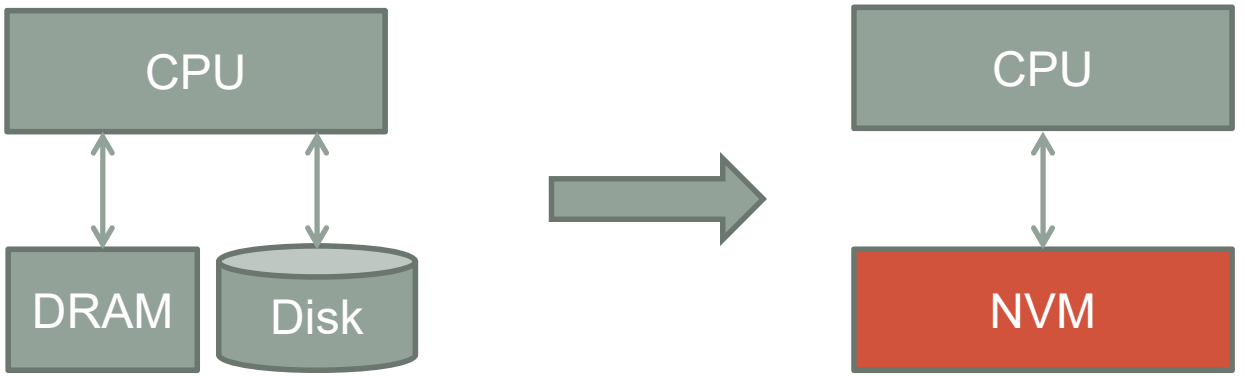
How computer OSs can take advantage of this new mix?

Outline

1. Introduction
2. Storage/memory convergence
   - Non-Volatile memory arrival
3. Operating System Mechanisms for NVM
   - NVM as a universal memory
   - DRAM/NVM hybrid memory architecture
     - NVM as memory
     - NVM in the middle
     - NVM as storage
4. Conclusion

NVM as a Universal Memory [3]

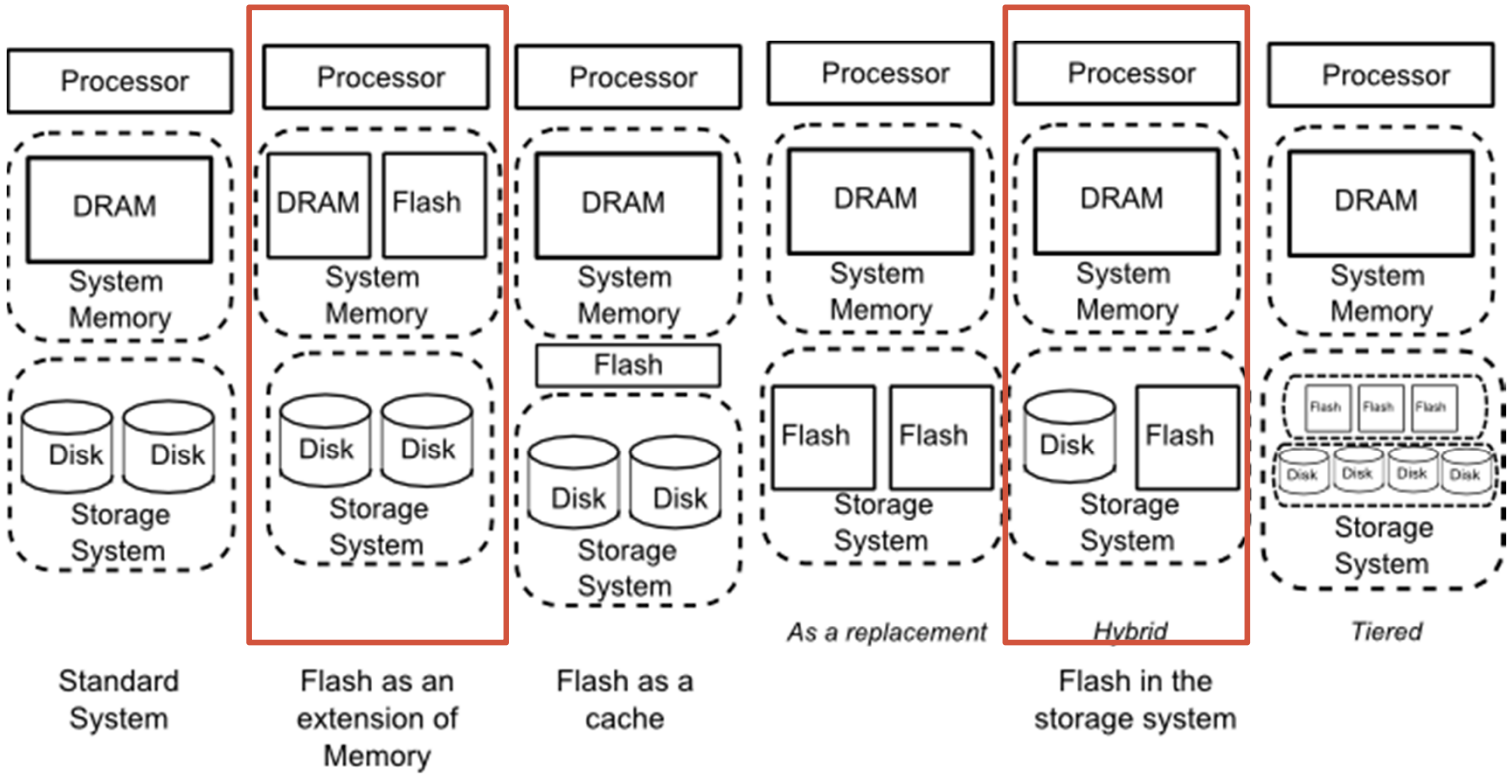
Speed + Capacity + Low consumption + Low price = universal memory



- Virtual memory :
  - DRAM is a cache for disk data (VM); Disk characteristics implies page size granularity.
- Protection different for files and memory
  - Per page and process (application)
  - Or per file and user id
- OS complexity (130000 LOC for Linux), performance cost
- No need for page concept
  - Limit the number of copies between the different level of memories
- Single address space for files and temporary data
  - Resident applications, triggered by input
  - System instant start
- Unified Protection paradigm? How to replace MMU based-mechanisms ?

More realistic approaches…

Here, NVM flavor is Flash memory…

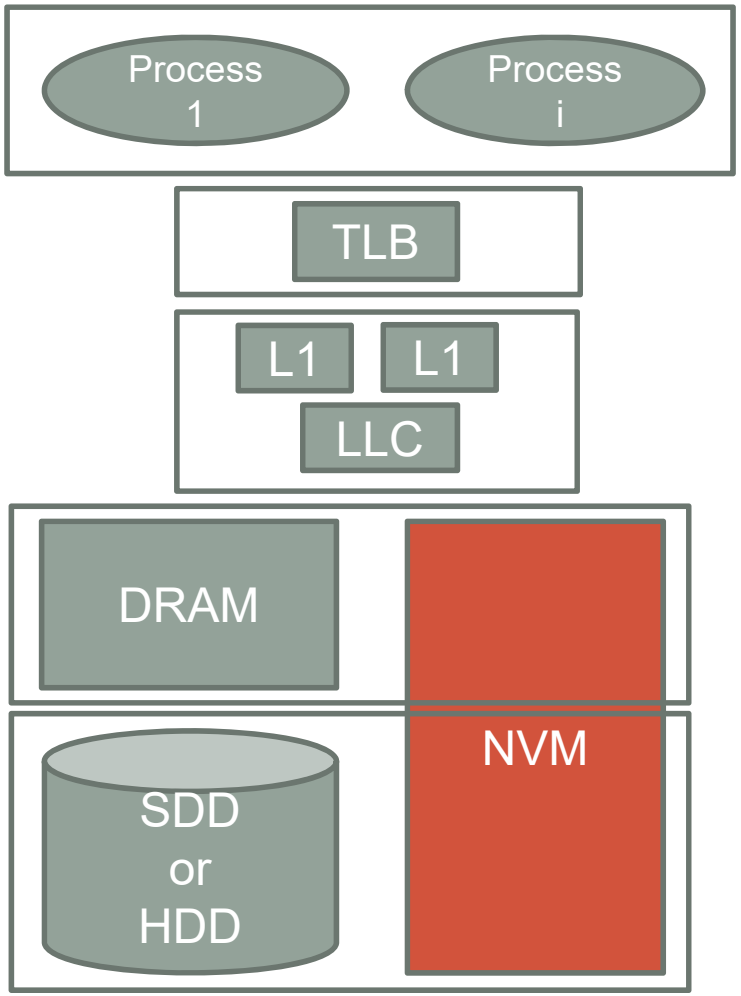


Outline

1. Introduction
2. Storage/memory convergence
   - Non-Volatile memory arrival
3. Operating System Mechanisms for NVM
   - NVM as a universal memory
   - DRAM/NVM hybrid memory architecture
     - NVM as memory
     - NVM in the middle
     - NVM as storage
4. Conclusion

A case study
NVM in a Hybrid memory Architecture

- Horizontal integration



Processor

Virtual memory management

Cache memories

Main memory

Storage memory

Software integration options

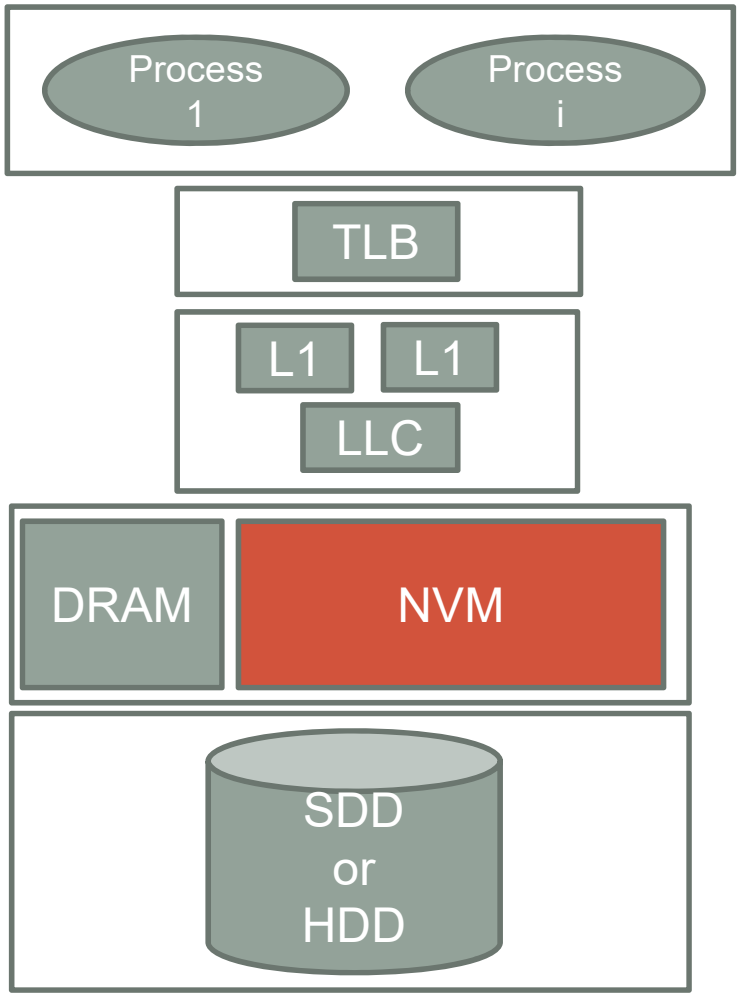
- Use NVM as a “classical” main memory
  - Benefit: capacity, static consumption
  - Challenges: write (latency, power, endurance)
    - → migrant store
- Use NVM as a permanent memory
  - Benefit: speed, per word access
  - Challenge: consistency
    - → nvm_malloc
- Use NVM as a “classical secondary memory” through a file system
  - Benefit: (speed), legacy code
  - Challenge: capacity
    - → PMFS

Software integration options

- Use NVM as a “classical” main memory
  - Benefit: capacity, static consumption
  - Challenges: write (latency, power, endurance)
    - → migrant store
- Use NVM as a permanent memory
  - Benefit: speed, per word access
  - Challenge: consistency
    - → nvm_malloc
- Use NVM as a “classical secondary memory” through a file system
  - Benefit: (speed), legacy code
  - Challenge: capacity
    - → PMFS

NVM as a “classical” main memory

- Enabled by per word access capability
- NVM benefits
  - Capacity, better density scaling than DRAM
  - Static consumption
  - Predictability of access times
- NVM problems
  - Write (latency, power, endurance)



NVM as a “classical” main memory

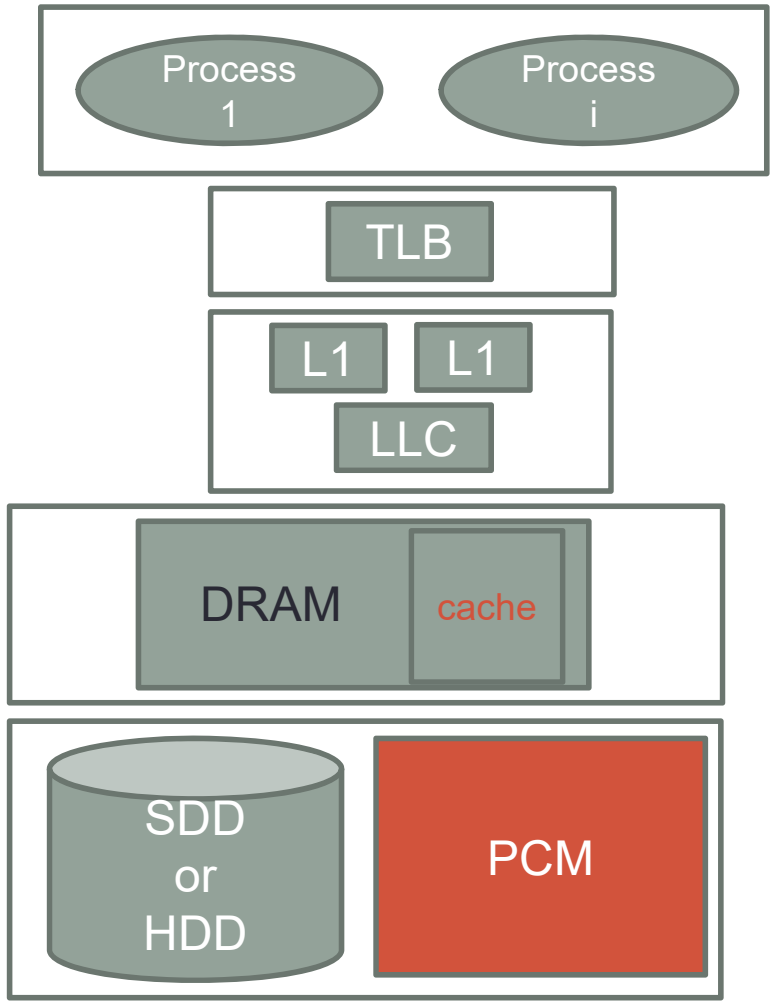
- Segmentation
  - Read-intensive segments: text (code), input data
  - Write-intensive segments: stack, heap, uninitialized global data
- Capacity and Power consumption
  - Decrease the memory static power if a part of DRAM is replaced by NVM.
  - NVM slower: How to choose the punished process?
    - Allocation and/or migration rules
      - Copy-on-write, write-intensive detection
      - Choice of the applications, API?
    - DVFS governor adaptation f(cpu time, #read, #write)

MigrantStore [4]

- Cache of NVM page in DRAM
  - Endurance, and speed of DRAM
  - Capacity of PCM memory
- Management through VM mechanisms
  - A kind of “swap”
- Performance issue
  - Small cache / PCM size

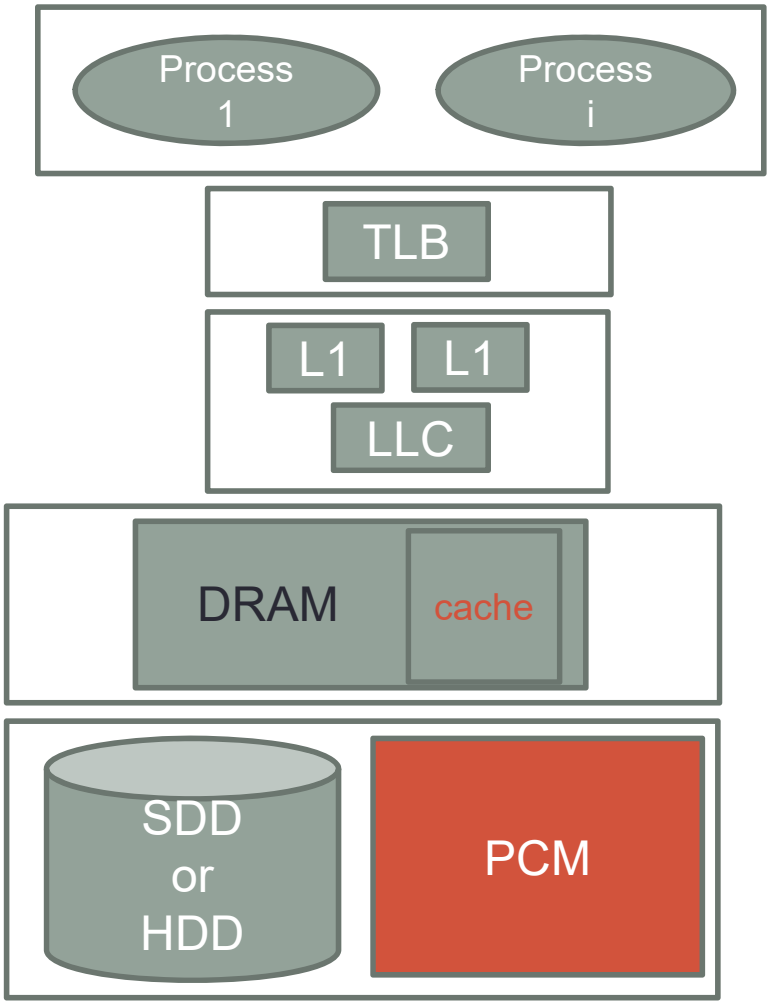
    → a lot of migration

    → limited re-use rate due to hardware cache absorption



MigrantStore: performance issue

- Isolate the heavily accessed pages:
  - →Placement into *MigrantStore* after some number of accesses (cache miss) has occurred on a page
- Replacement strategy
  - LRU: information build from the scan of referenced bits of page tables entries (PTE)
  - Too expensive for frequent migration
    - →proposition of a new hardware buffer, that stores the addressed pages between 2 migrations



Software integration options

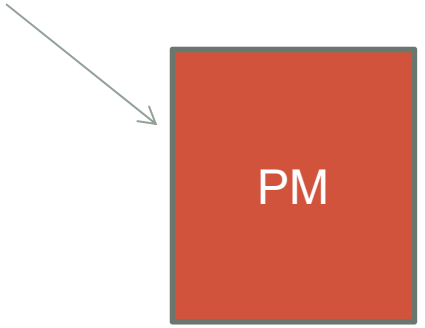
- Use NVM as a “classical” main memory
  - Benefit: capacity, static consumption
  - Challenges: write (latency, power, endurance)
    - → migrant store
- Use NVM as a permanent memory
  - Benefit: speed, per word access
  - Challenge: consistency
    - → nvm_malloc
- Use NVM as a “classical secondary memory” through a file system
  - Benefit: (speed), legacy code
  - Challenge: capacity
    - → PMFS

Pointer and persistence

Revisiting a common student error!

1. Save linked list nodes in PM, while mapped at VM address VA1

```
struct node * { int data; struct node * next; };
struct node *n1=malloc(sizeof(*n1));
struct node *n2=malloc(sizeof(*n2));
n1->next=n2; …
```



2. And next, reuse them later at VM address VA2…

How to retrieve n1 value?

→ need for a permanent data name space

→ need for permanent allocator data structure

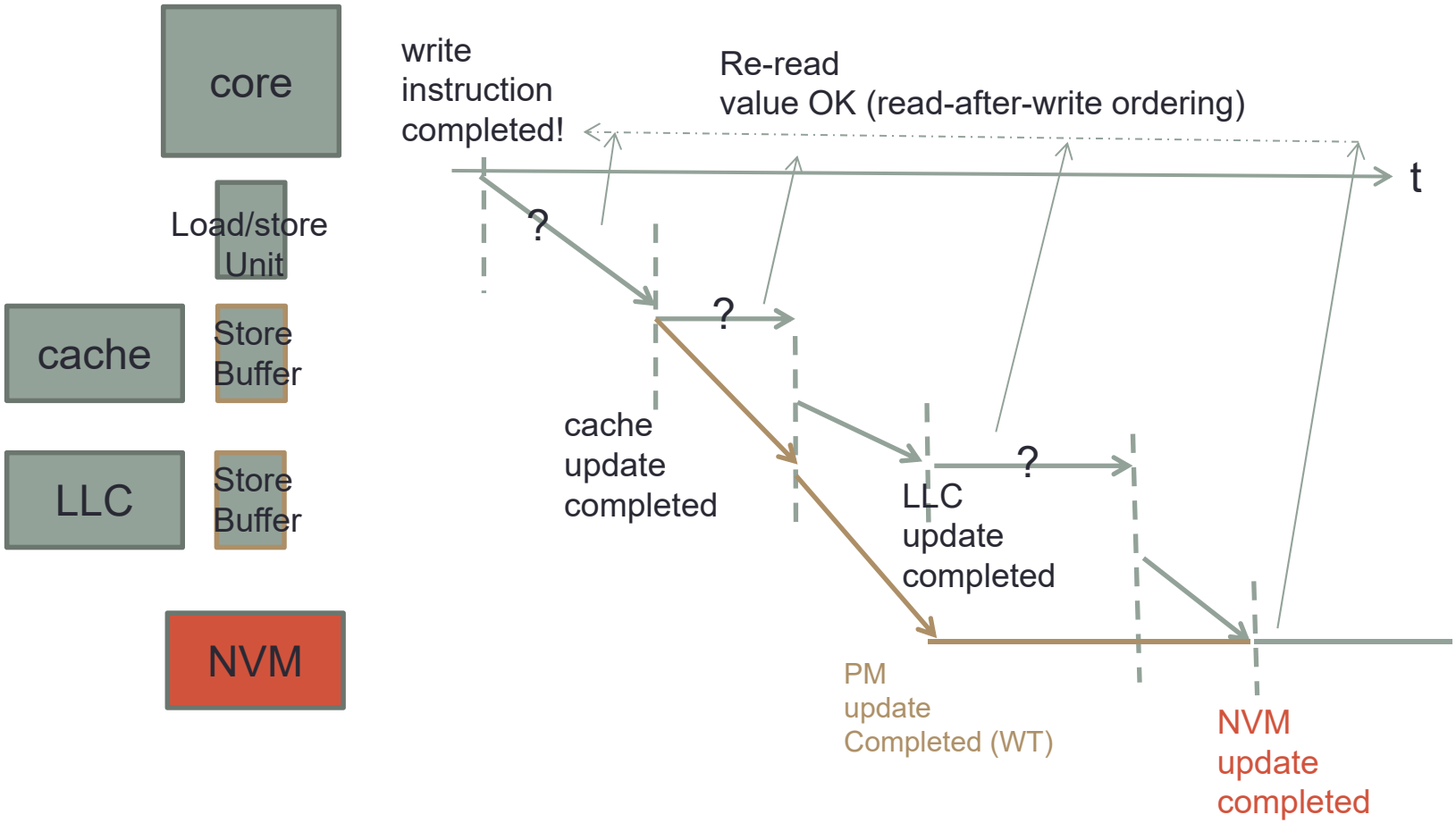
Where are we going to by following *n1->next* field?

→ volatile allocators don't care about the position of the reservation, the permanent ones must do.

Use PM as a permanent memory: ACID properties

- Assumption:
  - use a transactional model is too expensive at the main memory level
- Atomicity, Consistency, Isolation, Durability
- Durability:

Processor memory writing: the long way

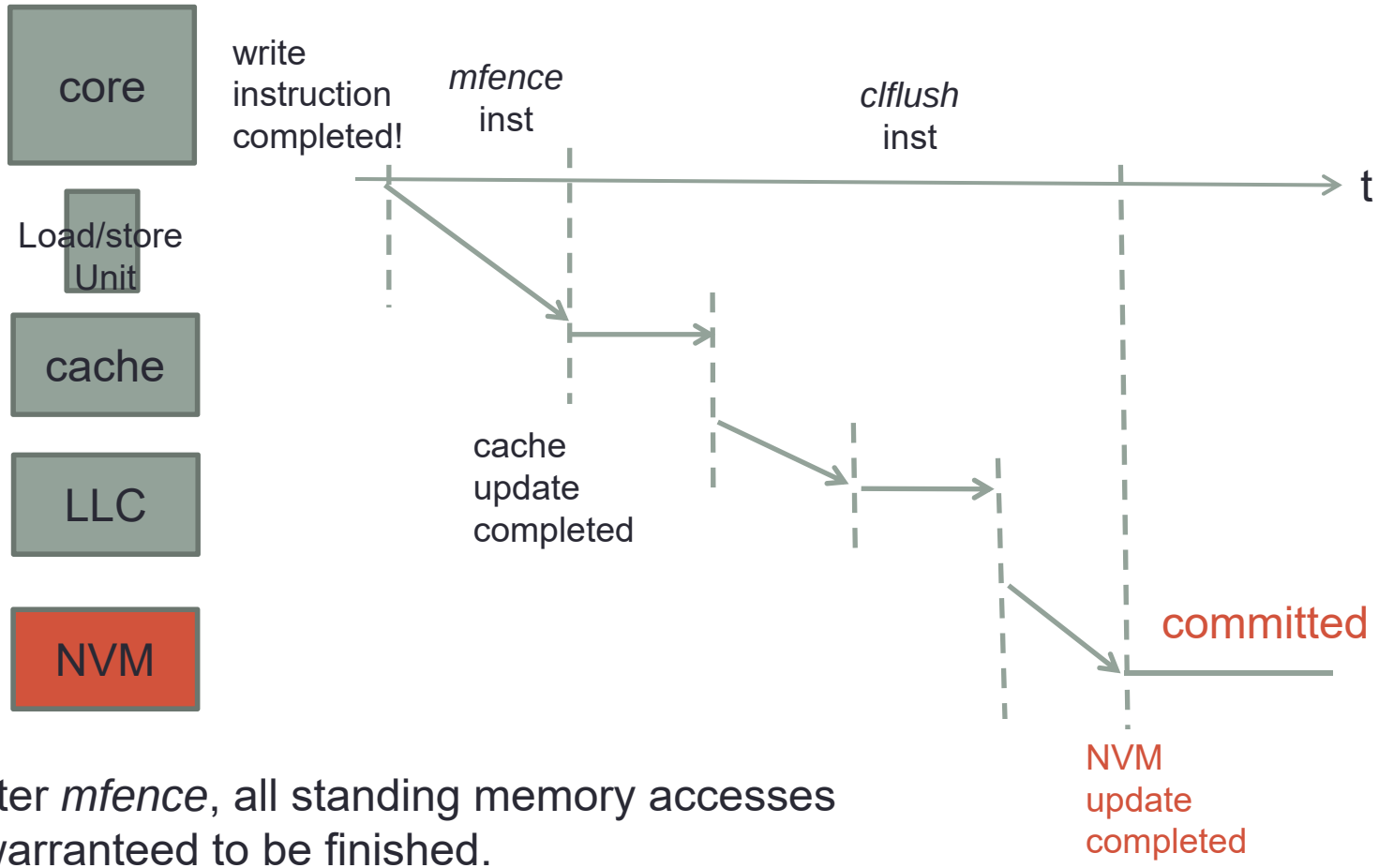


Durability

- Long write path increases the risk of data lost
- “Un-cached” accesses?
  - Increase the number of writes into NVM memory (cache absorption)
  - Increase access latencies, power consumption and memory wear.

→”software” triggered synchronization of the memory hierarchy

Processor memory writing: commit

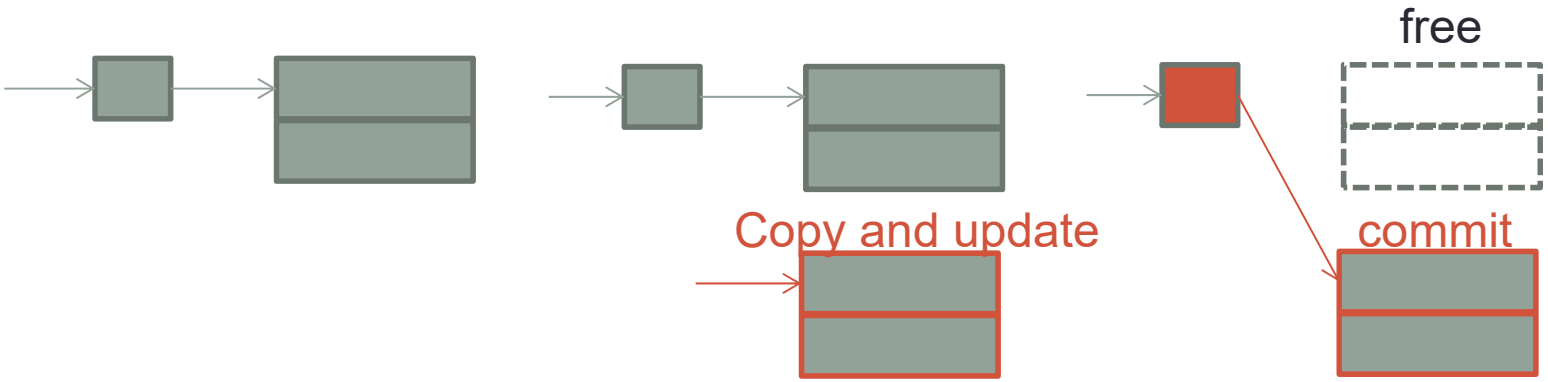


- After *mfence*, all standing memory accesses are warranteed to be finished.
- After *clflush*, all cache lines containing the address content are flushed.

Use NVM as a permanent memory: ACID properties

- Atomicity, Consistency, Isolation, Durability
- Durability: flush instructions, Write Through
  - →performance issue
- Isolation: mutual exclusion between concurrent threads
- Consistency: integrity constrains? Upper software layer…
- Atomicity (all or nothing)
  - Memory controllers ensure atomic write of words

  → Careful update order



Allocator nvm_malloc [1]

- NVM-aware memory allocator
  - 3 steps process: reservation, initialization and activation
    1. *void * nvm_reserve(int size);*
       reserve an area of size byte in NVM memory,
       update allocator “shadow meta-data” in DRAM
    2. Initialize data in memory area
    3. *Void nvm_activate(const void *p, void * link, const void* target …);*
       mark area as permanent, and atomically link it with another object
- Recovery status

  Reserved state: memory will be recovered as “free“

  Activated state: memory will be recovered as “in use“

Allocator nvm_malloc (2)

- Need for a name to locate formerly created data, and use it again.
  - String ID associated to allocated memory area
  - Allow to locate the address of a previously allocated area through a hashmap structure

*void * nvm_reserve(const char *id, int size);*

*void nvm_activate(const char * id);*

*void * nvm_get_id(const char *id);*

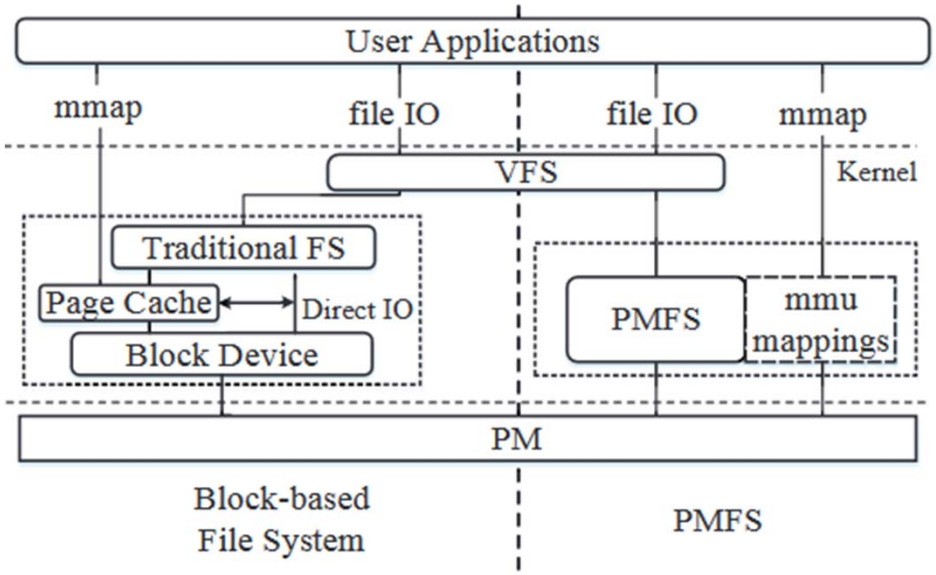
- In practical, only the “root” of the data structure needs for an ID.

Software integration options

- Use NVM as a “classical” main memory
  - Benefit: capacity, static consumption
  - Challenges: write (latency, power, endurance)
    - → migrant store
- Use NVM as a permanent memory
  - Benefit: speed, per word access
  - Challenge: consistency
    - → nvm_malloc
- Use NVM as a “classical secondary memory” through a file system
  - Benefit: (speed), legacy code
  - Challenge: capacity
    - → PMFS

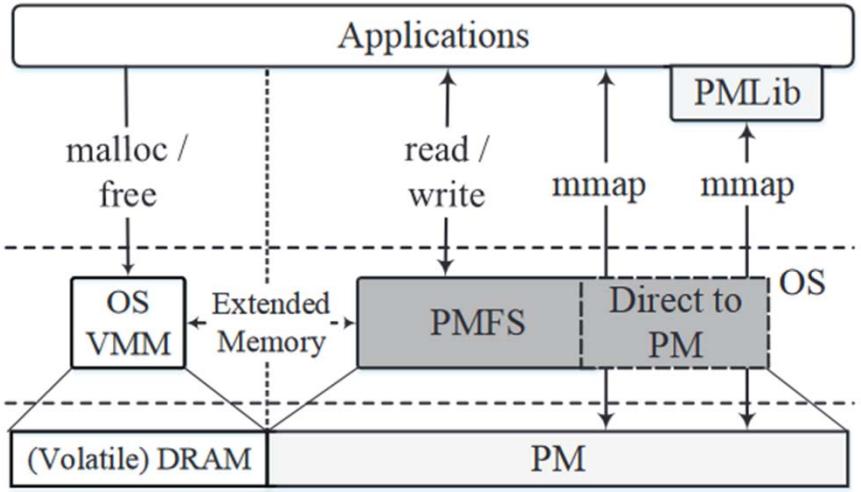
NVM through a File System

- Rest on a block device interface
  - "NVMTL": Generalization of the Flash Translation Layer concept
  - Allow for using legacy File System (*ext4* for instance)
  - But, all NVM aren't so complex to manage than NAND Flash
  - Performance issues
    - Multiple page/buffer copies, read-modify-write per block
- Rest on a byte-addressable memory interface
  - PM mapped in kernel VM space (controlled by OS)
  - NVM aware file system
  - Performance: direct copy between NVM and application buffers
  - Protection against OS pointer corruption?



PMFS (Intel [2])

- PM-aware file system
- Posix API
- FS implementation:
  - Allocation by page, extend
  - B-Tree Indexation
  - Protection in kernel mode
    - CPU dependent
      - Intel x86: read only pages + override only when CR0.WR bit set
      - PMFS set CR0.WR bit before writing
    - Consistency
      - “Undo” journaling (fine-grained (64 bytes) logging) for meta-data
      - Copy on Write
      - Atomic in-place update



From [2]

Outline

1. Introduction
2. Storage/memory convergence
   - Non-Volatile memory arrival
3. Operating System Mechanisms for NVM
   - NVM as a universal memory
   - DRAM/NVM hybrid memory architecture
     - NVM as memory
     - NVM in the middle
     - NVM as storage
4. Conclusion

Conclusion

- Some Operating system updates already exist for using NVM
  - XIP (eXecute In Place) + permanent state: consequences?
- Missing hardware mechanisms for heterogeneous memory ("NUMA+" extension)
  - Application memory profile → Accounting read or write memory references per page (hardware support)
  - Protection model (MMU based versus File System based)
- Temporary, configuration, session, result data

  →API to guide memory placement
- Embedded real-time system
  - predictability, power consumption?
- NVM "killer application"?
  - Look at the main application classes and find the good strategy (Assia deals with Big Data!)

References

1. *nvm_malloc: Memory Allocation for NVRAM*, D. Schwalb et al., International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Hawaii, USA, 2015.
2. *System software for persistent memory*, S. R Dulloor, Sanjay Kumar, A.Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, J. Jackson, Proceedings of the 9th European Conference on Computer Systems, 2014
3. *Operating System Implications of Fast, Cheap, Non-Volatile Memory. K.* Bailey, L. Ceze, S. Q. Gribble, Henry M. Levy, Usenix *HotOS*. 2011.
4. *MigrantStore: Leveraging Virtual Memory in DRAM-PCM Memory Architecture.* H. B. Sohail, B. Vamanan, and T. N. Vijaykumar. Technical report, Purdue University, Department of Electrical and Computer Engineering, 2012.
5. *Emerging NVM: A Survey on Architectural Integration and Research Challenges.* J. Boukhobza, S. Rubini, R. Chen, Z. Shao, *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, *23*(2), 2017