

# Non-volatile main memory

Master in computer science of IP Paris

Master CHPS of Paris Saclay

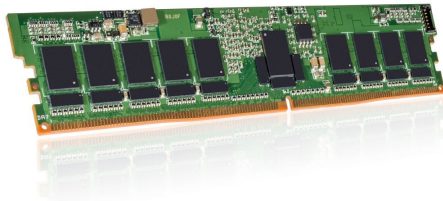
Gaël Thomas

# Non-volatile main memory

Classical volatile memory (DDR4)

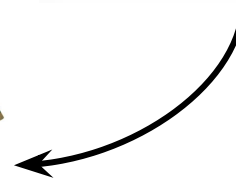
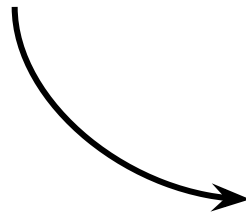
+

Persistent storage (flash)



=

Non volatile main  
memory



## Non volatile main memory

- Byte addressability, directly by the processor through the memory bus with normal load/store instructions
- Durability as a SSD
- Performance: between DDR4 and NVMe (SSD through PCIe)

# NVMM performance

## Latency

- 2x slower than a DDR4
- 1000x to 10000x faster than a NVMe

## Throughput

- 2x to 4x lower than a DDR4
- 4x to 10x higher than a NVMe

	Sequential			Random		
	Read	Write	Bandwidth	Read	Write	bandwidth
DDR4	80ns	60ns	~100Gbps	100ns	60ns	~100Gbps
Optane DC	160ns	60ns	40Gbps	300ns	120ns	14Gbps
NVMe	250 $\mu$ s	250 $\mu$ s	3Gbps	250 $\mu$ s	250 $\mu$ s	3Gbps

# How to use a NVMM

## ■ At the hardware level

- Recorded by the BIOS as any memory bank
- Marked as non volatile

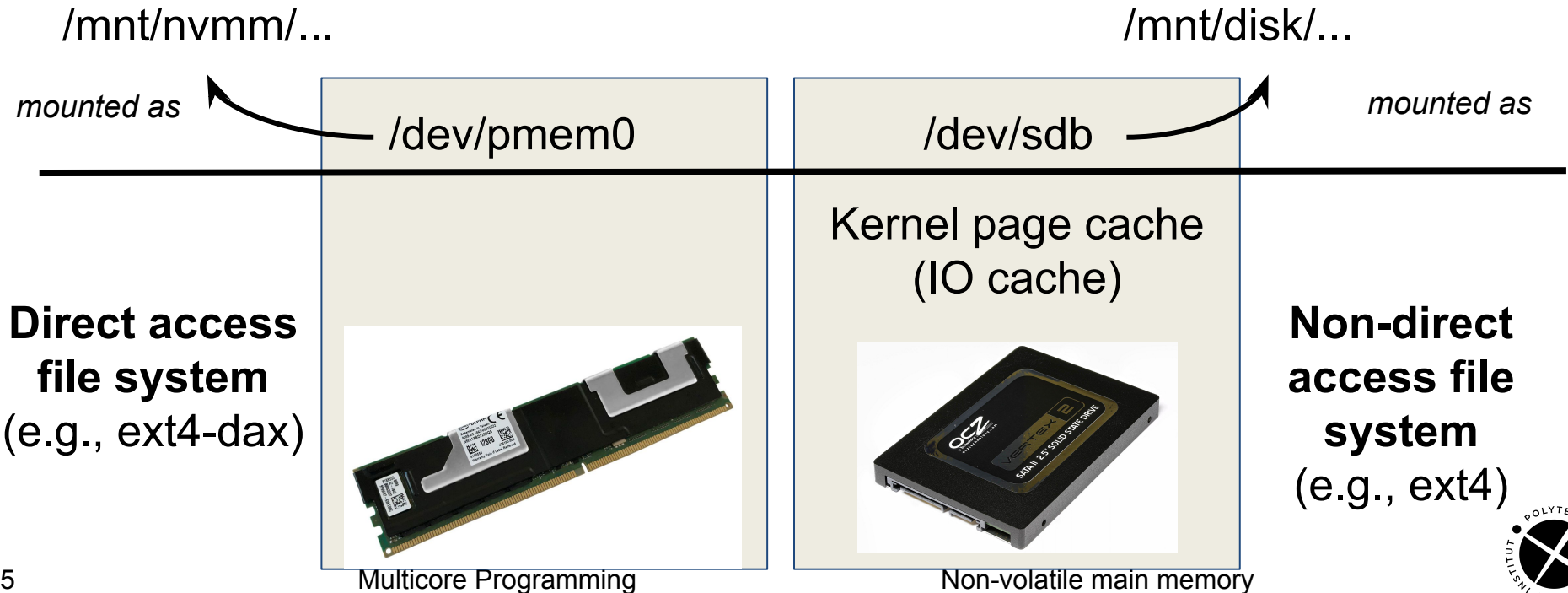
## ■ At the system level with Linux

- Exposed as a device
- Formatted with a direct access file system (DAX)
  - Bypass the IO cache => direct load/store to the NVMM
- Accessible with classical IO functions (read, write and mmap)
  - In case of mmap, direct access

# How to use a NVMM with Linux

NVMM exposed as `/dev/pmem0`  
Formatted with `ext4-dax`  
=> bypass the kernel page cache  
Mounted in `/mnt/nvmm`

SSD exposed as `/dev/sdb`  
Formatted with `ext4`  
=> use the kernel page cache  
Mounted in `/mnt/disk`

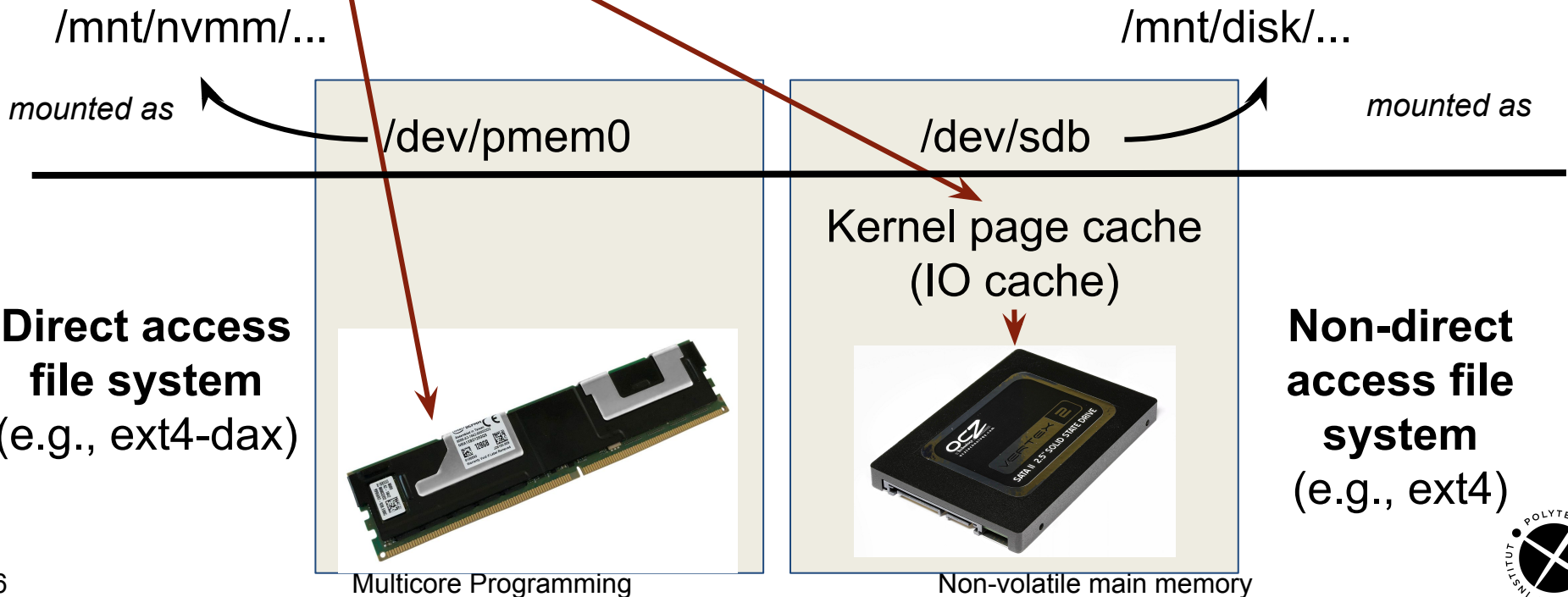


# How to use a NVMM with Linux

```
int fd = open("/mnt/[nvmm,disk]/myfile", O_RDWR | O_CREAT);  
ftruncate(fd, 1024*1024);  
char* addr = mmap(NULL, 1024*1024, PROT_READ | PROT_WRITE,  
MAP_SHARED_VALIDATE | MAP_SYNC, fd, 0);  
addr[0] = 'a';
```

*direct write  
to NVMM*

*write in the volatile page cache, which is  
eventually propagated to disk*



# Crash management

- A crash may happen at any time
  - At recovery, the NVMM state is still there
  - The state is not necessarily consistent
  - An application has to cleanup this state

## ■ Example

```
struct id {                                struct id* id = mmap(...);
    char name[256];
};                                          strcpy(id->name, "Pikachu");
```

In case of crash inside `strcpy`, `id->name` may contain inconsistent values (neither "" nor "Pikachu")

# Crash management

## ■ Solution: use transactions!

- Manually manage the transaction (see below)
- Or use a high-level library such as the `pmdk`

## ■ Principle of solution

```
struct id {  
    char name[256];  
    bool committed;  
};
```

```
struct id* id = mmap(...);  
  
if(!id->isCommitted) { /* recover */  
    strcpy(id->name, "Pikachu");  
    id->committed = true;  
}
```



# Out-of-order execution

■ Modern processors execute instructions out of order

- `id->committed` may be executed before `strcpy`
- The NVMM state at recovery is thus unknown with our code

■ Principle of solution: enforce the ordering

■ Problem: a memory fence enforces the ordering in the processor cache, but the cache lines can be flushed in any order....

```
if(!id->isCommitted) { /* recover */  
    strcpy(id->name, "Pikachu");  
    memory_fence();  
    id->committed = true;  
}
```

# Fighting out-of-order execution

- Introduces two new instructions

- `pwb(char* addr)`: adds the cache line that contains `addr` in a flush queue that ensures a FIFO order

- `pfence()`: ensures that the stores and `pwb`s that precede are executed before the stores and `pwb`s that succeed

# Fighting out-of-order execution

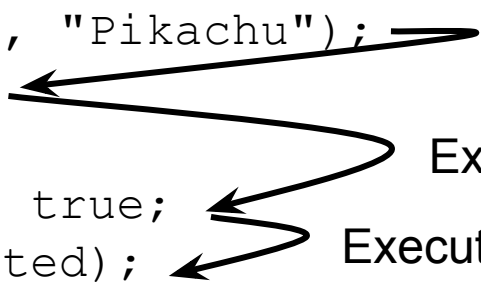
```
struct id* id = mmap(...);

if(!id->isCommitted) { /* recover */
    strcpy(id->name, "Pikachu");
    pwb(id->name);
    pfence();
    id->committed = true;
    pwb(&id->committed);
}
```

Executed before (same variable)

Executed before (pfence)

Executed before (same variable)



=> the cache line that contains `id->committed` is flushed  
after the cache line that contains `id->name`

=> `id->committed` propagated to NVMM after `id->name`

=> at recovery

`(id->committed == true => id->name == "Pikachu")`

# Stronger guarantees

■ pwb/pfence only ensures the propagation order of the cache lines

- Does not ensure that a cache line is actually flushed

■ Sometime, we need stronger guarantees

- For example, only execute a code if a data is durable
- Impossible with only pwb/pfence

■ Example: durable linearizability, which essentially ensures that a write becomes visible to other threads if the write is durable

# Stronger guarantees

■ A third instruction: `psync()`

- Acts as a `pfence()`
- And ensures that the cache line is actually propagated to NVMM

```
struct id {                                // NVMM
    char name[256];
    bool committed;
};
_Atomic bool visible; // volatile memory
```

## Thread 1

```
strcpy(id->name, "Pikachu");
pwb(id->name);
pfence();
id->committed = true;
pwb(&id->committed);
psync();
atomic_store(&visible, true);
```

## Thread 2

```
while(!atomic_load(&visible)) {
}

printf("%s is durable\n",
       id->name);
```

# Implementation with a pentium

■ pwb implemented with clwb (cache line write back)

■ pfence implemented with sfence (store fence)

- Pentium is a TSO: already ensures that a store after store is not reordonanced
- The sfence additionally ensures that a clwb is not reordonanced because a clwb is a store-like instruction
- => pfence ensures that the stores and pwb are not reordonanced

■ psync implemented with sfence (store fence)

- In case of crash, enough residual energy in a pentium to flush the pending cache lines to the NVMM

# To take away

- NVMM: durability for the cost of volatile memory access
  - Same order of magnitude, but slower than DDR4
- Exposed in Linux through a direct access file system
  - For example ext4-dax
  - Direct access with open/mmap
- Three new instructions to enforce ordering
  - pwb: add a cache line to the flush queue
  - pfence: store fence + prevent reordering of pwb
  - psync: pfence + ensures cache lines written to NVMM