A Support for **Persistent Memory** in **Java**

Anatole Lefort

Advisors: Pierre Sutra, Gaël Thomas

Télécom SudParis
Institut Polytechnique de Paris

Ph.D. Defense – Mar. 24th, 2023

A Support for Persistent Memory in **Java**

Anatole Lefort

Advisors: Pierre Sutra, Gaël Thomas

Télécom SudParis
Institut Polytechnique de Paris

Ph.D. Defense – Mar. 24th, 2023

Data Persistence

*Computer programs deal with **two kinds** of data.*

- **Transient:**
  - **Limited Lifetime:** renewed at every program execution.
    - do not survive crashes.
  - hosted in Main Memory

- **Persistent:**
  - **Extended Lifetime:** recalled and reused in subsequent executions.
    - remain consistent even in the wake of a failure.
  - hosted on Storage Devices

Data Persistence

“Persistence is the ability of data to outlive the instance of a program”

“Data remain persistent for the extent of time during which they may be recalled and used by a program”

Common media: **HDD, Flash disks**

- **Durable**: resist reboots, power loss
- **Large capacity:** at least TBs
- **Slow access latency:** 200μs-15ms

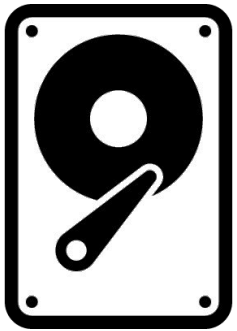
Data Persistence

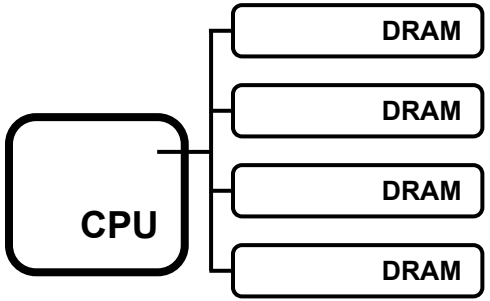
“Persistence is the ability of data to outlive the instance of a program”

“Data remain persistent for the extent of time during which they may be recalled and used by a program”

Common media: **HDD, Flash disks**

- **Durable**: resist reboots, power loss
- **Large capacity:** at least TBs
- **Slow access latency:** 200μs-15ms





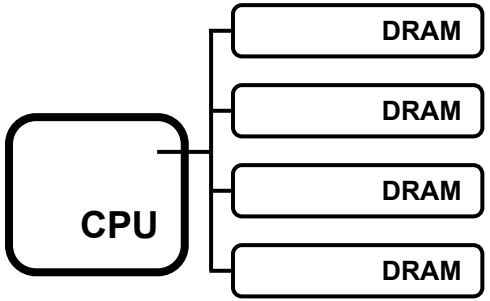
Data Persistence

“Persistence is the ability of data to outlive the instance of a program”

“Data remain persistent for the extent of time during which they may be recalled and used by a program”

Common media: **HDD, Flash disks**

- **Durable**: resist reboots, power loss
- **Large capacity:** at least TBs
- **Slow access latency:** 200µs-15ms



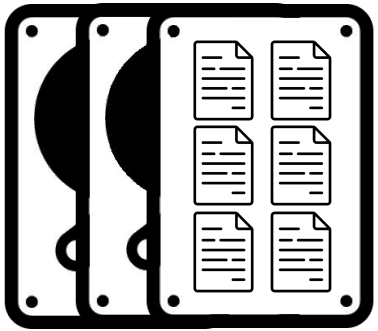
Data Persistence

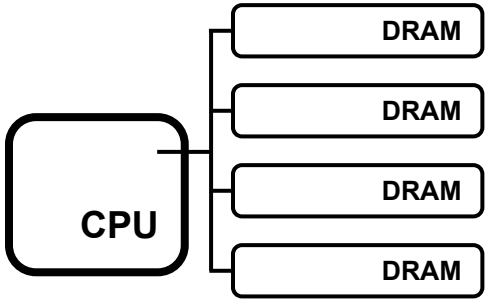
“Persistence is the ability of data to outlive the instance of a program”

“Data remain persistent for the extent of time during which they may be recalled and used by a program”

Common media: **HDD, Flash disks**

- **Durable**: resist reboots, power loss
- **Large capacity:** at least TBs
- **Slow access latency:** 200μs-15ms





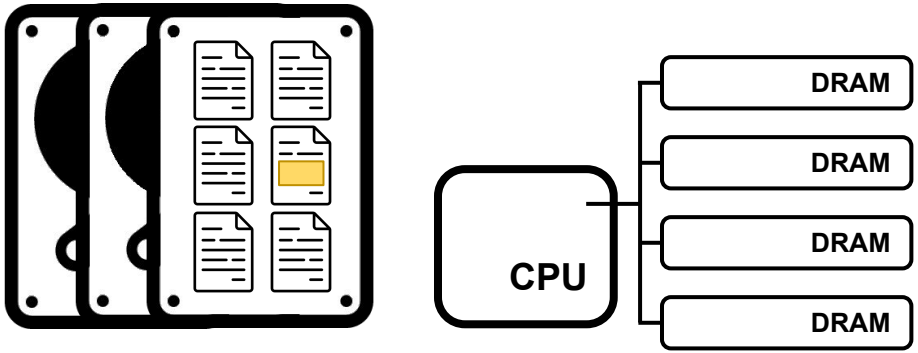
Data Persistence

“Persistence is the ability of data to outlive the instance of a program”

“Data remain persistent for the extent of time during which they may be recalled and used by a program”

Common media: **HDD, Flash disks**

- **Durable**: resist reboots, power loss
- **Large capacity:** at least TBs
- **Slow access latency:** 200μs-15ms



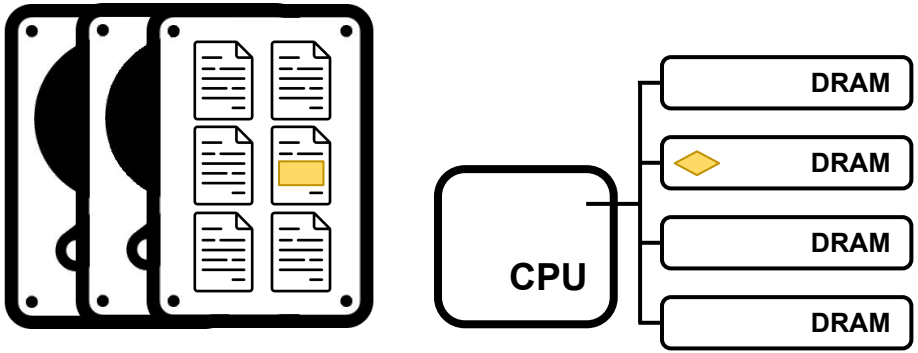
Data Persistence

“Persistence is the ability of data to outlive the instance of a program”

“Data remain persistent for the extent of time during which they may be recalled and used by a program”

Common media: **HDD, Flash disks**

- **Durable**: resist reboots, power loss
- **Large capacity:** at least TBs
- **Slow access latency:** 200µs-15ms



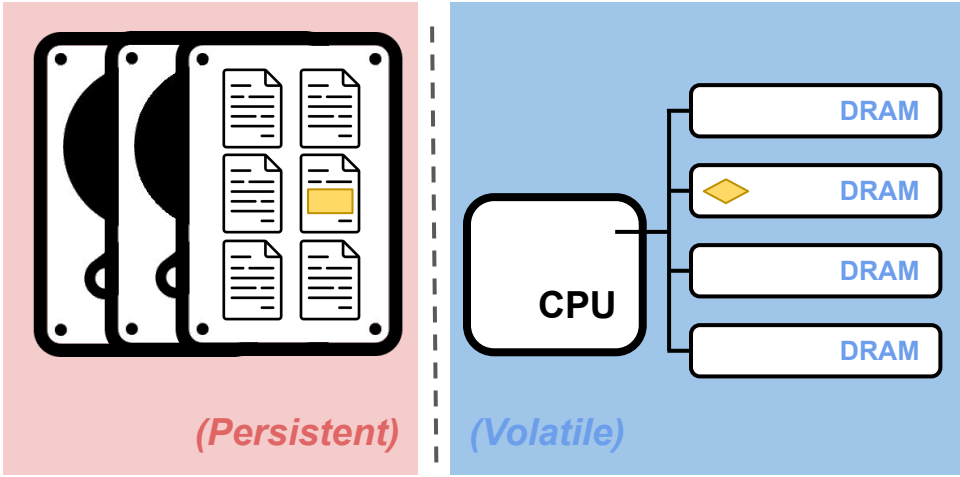
Data Persistence

“Persistence is the ability of data to outlive the instance of a program”

“Data remain persistent for the extent of time during which they may be recalled and used by a program”

Common media: **HDD, Flash disks**

- **Durable**: resist reboots, power loss
- **Large capacity:** at least TBs
- **Slow access latency:** 200µs-15ms



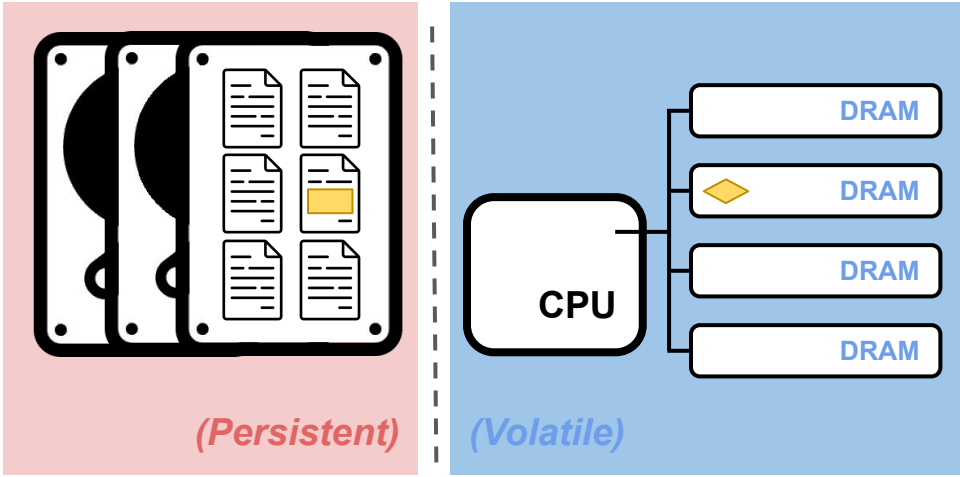
Data Persistence

“Persistence is the ability of data to outlive the instance of a program”

“Data remain persistent for the extent of time during which they may be recalled and used by a program”

Common media: **HDD, Flash disks**

- **Durable**: resist reboots, power loss
- **Large capacity:** at least TBs
- **Slow access latency:** 200µs-15ms



1 - Dual data representation
2 - Expensive I/Os

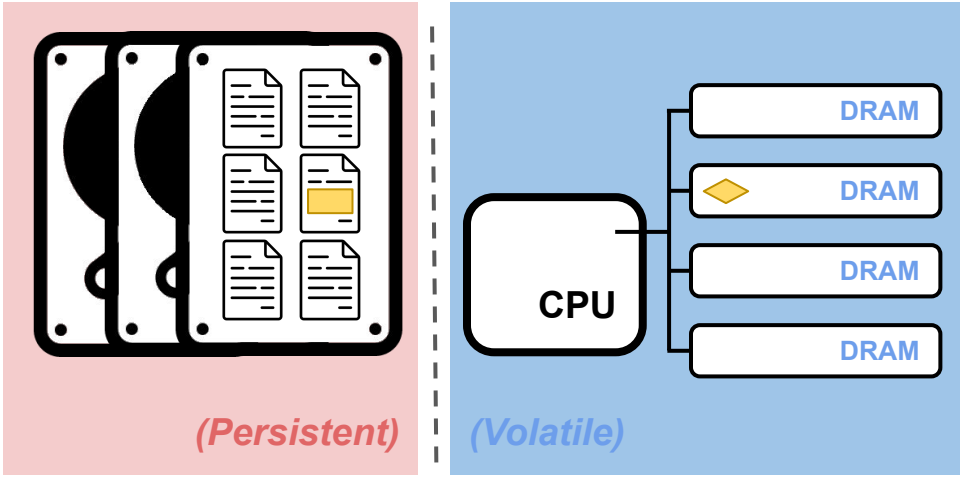
Data Persistence

"Persistence is the ability of data to outlive the instance of a program"

"Data remain persistent for the extent of time during which they may be recalled and used by a program"

Common media: **HDD, Flash disks**

- **Durable**: resist reboots, power loss
- **Large capacity:** at least TBs
- **Slow access latency:** 200µs-15ms



1 - Dual data representation
2 - Expensive I/Os

→

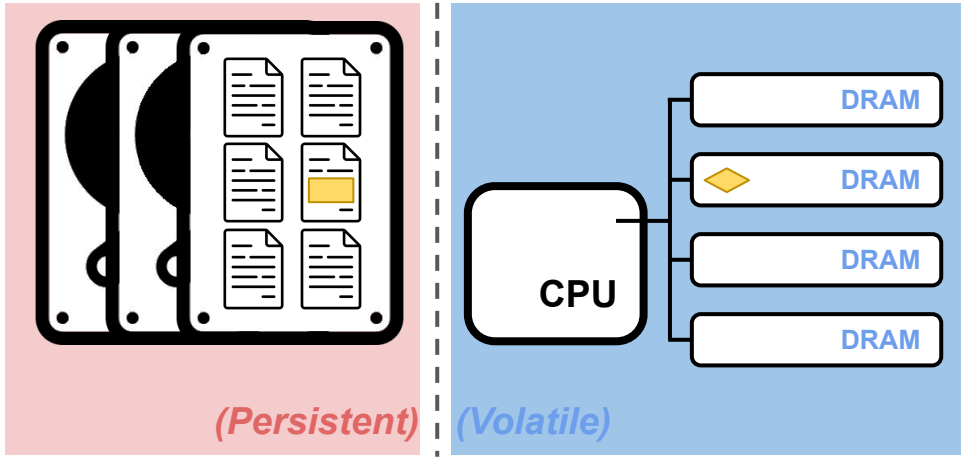
1 - Keep data consistent across media
2 - Trade durability guarantees for performance

Complex Software

Persistent Memory *(PMEM)*

PMEM: A memory device on which data survives power cycles.

What changes with PMEM ?



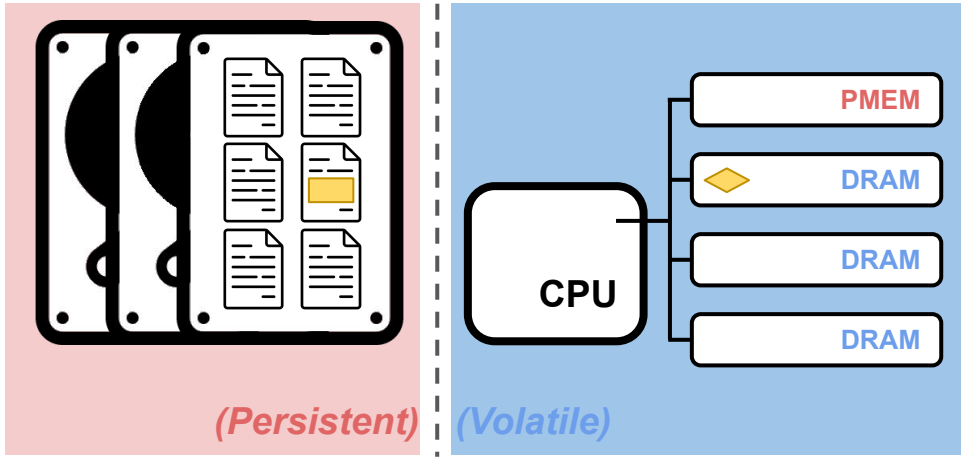
1 - Dual data representation
2 - Expensive I/Os

Persistent Memory *(PMEM)*

PMEM: A memory device on which data survives power cycles.

What changes with PMEM ?

1- add PMEM DIMM



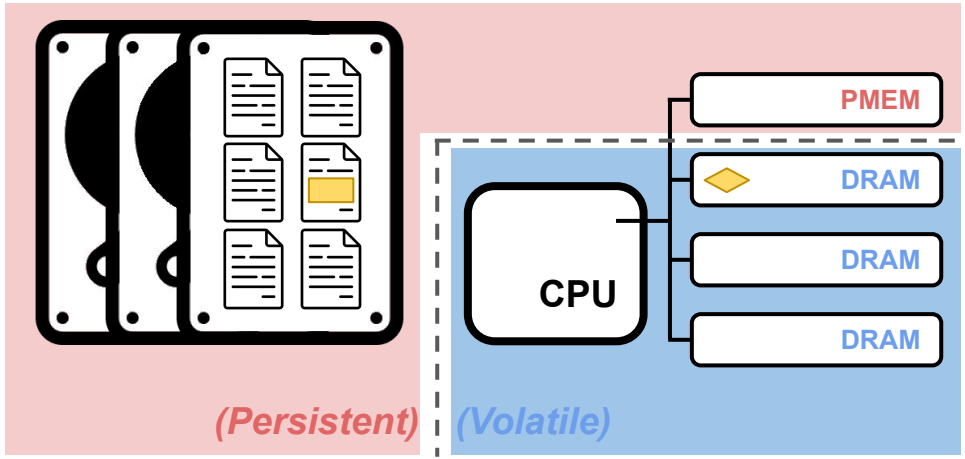
1 - Dual data representation
2 - Expensive I/Os

Persistent Memory *(PMEM)*

PMEM: A memory device on which data survives power cycles.

What changes with PMEM ?

1- add PMEM DIMM,
CPU-attached persistent media



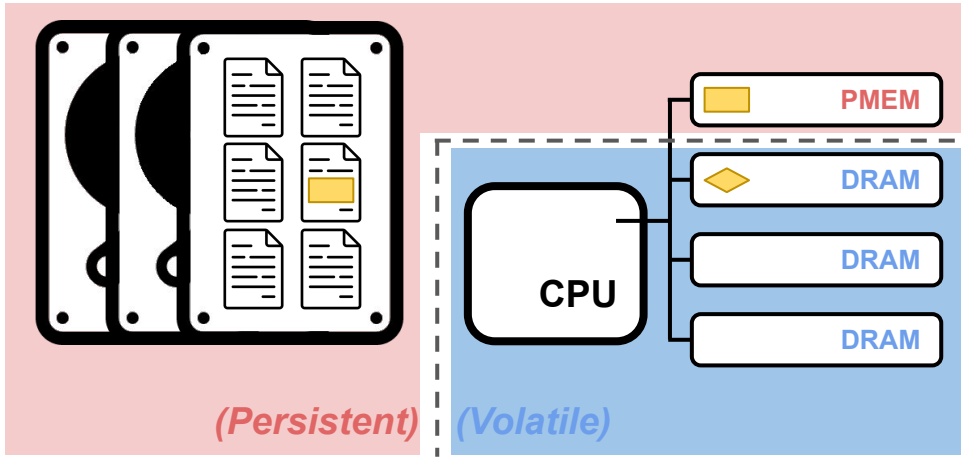
1 - Dual data representation
2 - Expensive I/Os

Persistent Memory *(PMEM)*

PMEM: A memory device on which data survives power cycles.

What changes with PMEM ?

1- add PMEM DIMM,
CPU-attached persistent media
2- move persistent data



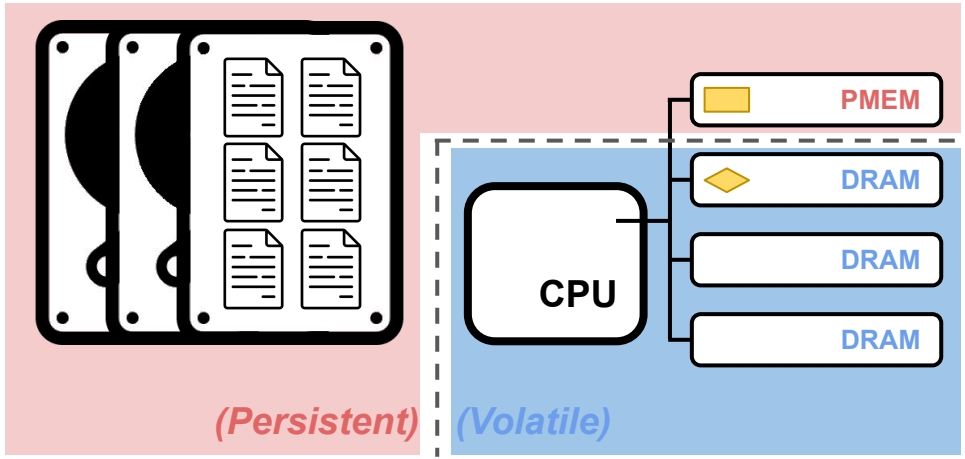
1 - Dual data representation
2 - Expensive I/Os

Persistent Memory *(PMEM)*

PMEM: A memory device on which data survives power cycles.

What changes with PMEM ?

1- add PMEM DIMM,
CPU-attached persistent media
2- move persistent data,
disks become redundant



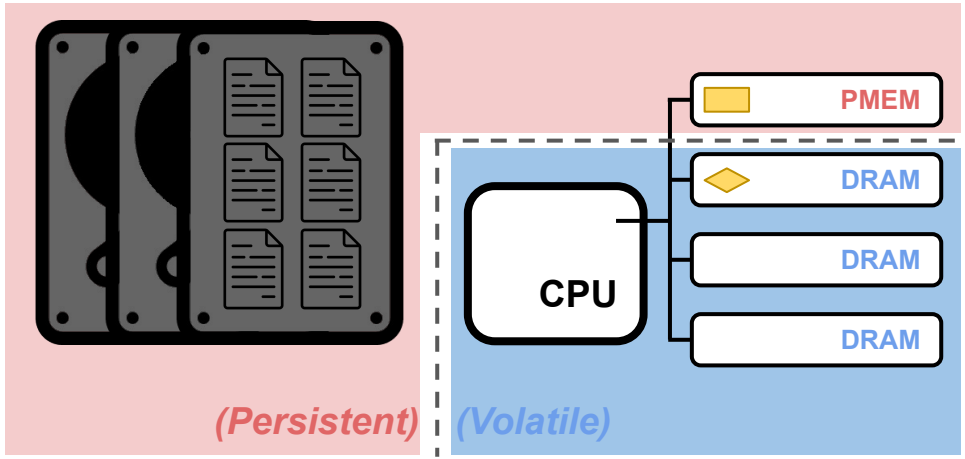
1 - Dual data representation
2 - Expensive I/Os

Persistent Memory *(PMEM)*

PMEM: A memory device on which data survives power cycles.

What changes with PMEM ?

1- add PMEM DIMM,
CPU-attached persistent media
2- move persistent data,
disks become redundant
3- no more disk I/Os



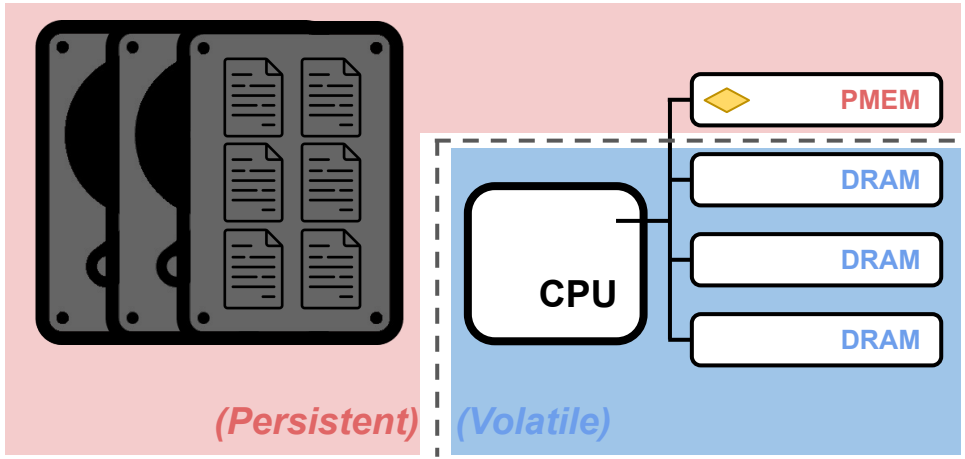
1 - Dual data representation
~~2 - Expensive I/Os~~

Persistent Memory *(PMEM)*

PMEM: A memory device on which data survives power cycles.

What changes with PMEM ?

1- add PMEM DIMM,
CPU-attached persistent media
2- move persistent data,
disks become redundant
3- no more disk I/Os
4- working copy of data is durable



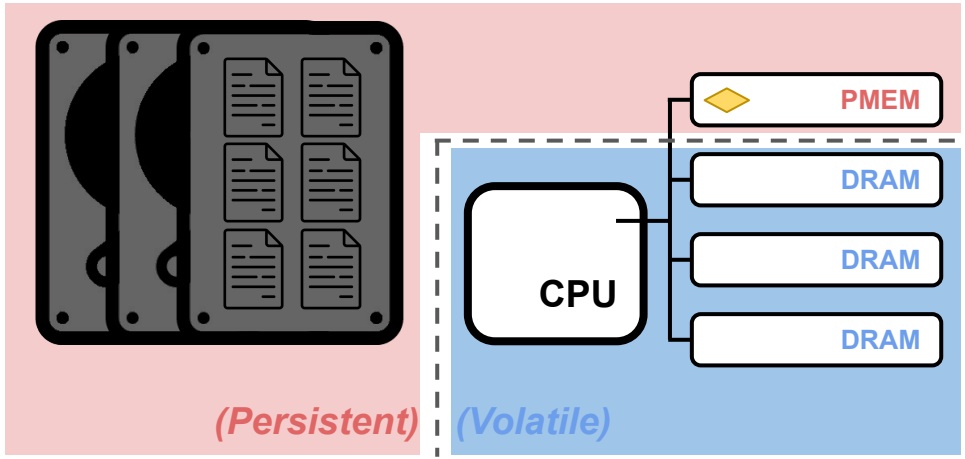
~~**1 - Dual data representation**~~
~~**2 - Expensive I/Os**~~

Persistent Memory *(PMEM)*

PMEM: A memory device on which data survives power cycles.

What changes with PMEM ?

1- add PMEM DIMM,
CPU-attached persistent media
2- move persistent data,
disks become redundant
3- no more disk I/Os
4- working copy of data is durable



~~**1 - Dual data representation**~~
~~**2 - Expensive I/Os**~~

→

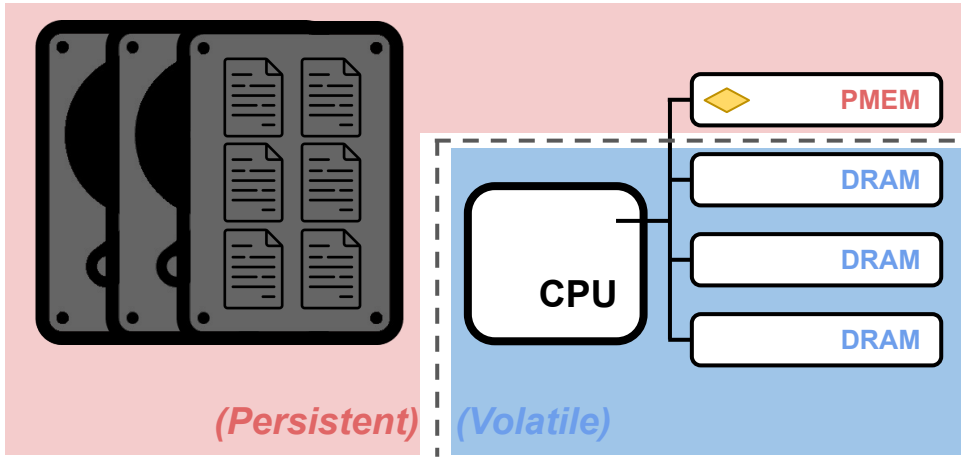
1 - Single data representation
2 - Direct access

Persistent Memory *(PMEM)*

PMEM: A memory device on which data survives power cycles.

What changes with PMEM ?

1- add PMEM DIMM,
CPU-attached persistent media
2- move persistent data,
disks become redundant
3- no more disk I/Os
4- working copy of data is durable



Benefits

1- No more (un)marshalling
2- No need for data caching
3- Faster recovery
4- Lower software complexity

~~**1 - Dual data representation**~~
~~**2 - Expensive I/Os**~~

→

1 - Single data representation
2 - Direct access

Non-volatile main memory *(NVMM)*

new persistent medium (in-between **SSD** and **DRAM**)

Durable

resists reboots, power loss

High-density

smallest DIMM = 128 GB
up to 8x DDR4 capacity

Byte addressable

persistent memory abstraction

High-performance

low latency (seq. read/write ~ 160/90ns)
high bandwidth (up to 8.10GB/s, *2nd gen*)

Intel Optane PMEM, 2019

Non-volatile main memory *(NVMM)*

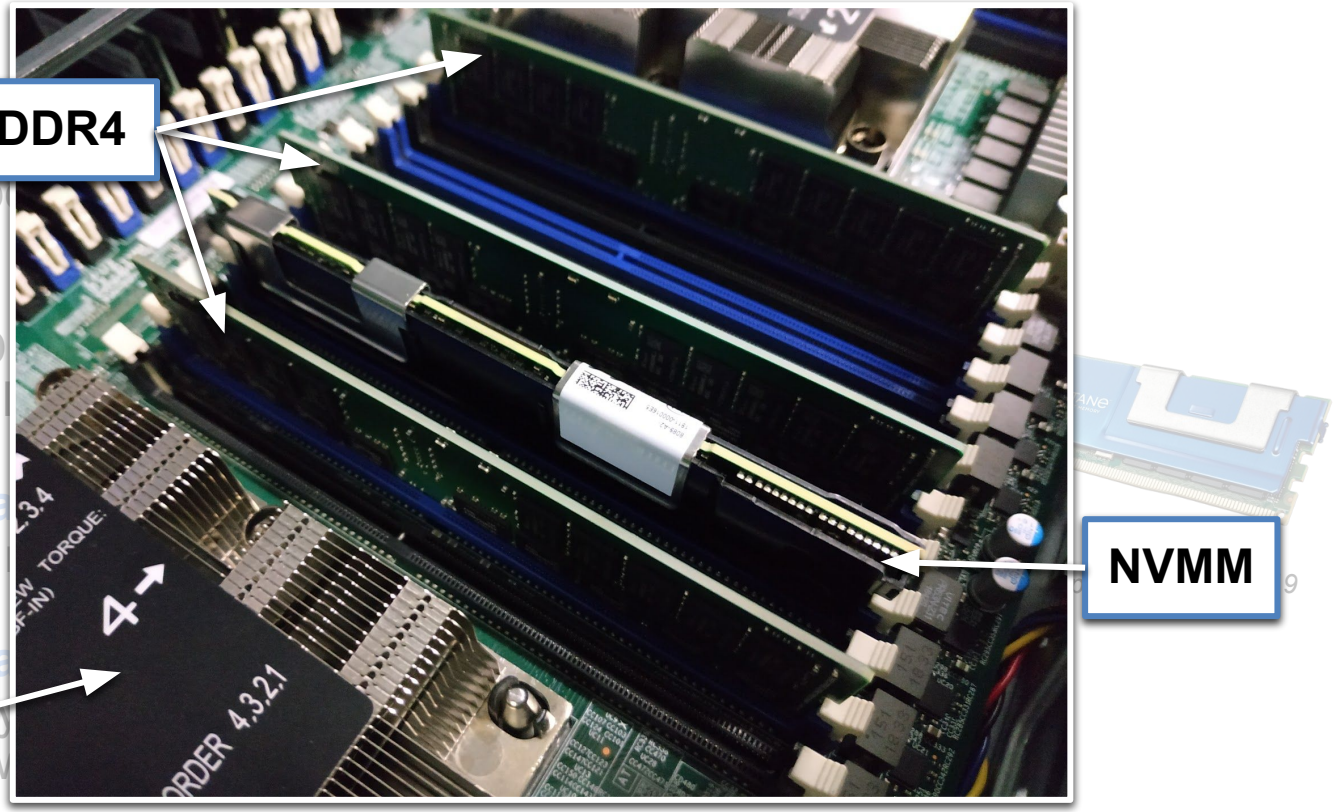
new persistent

Durable
resists reb

High-density
smallest D
up to 8x D

Byte addressa
persistent

High-performa
lo
high bandw



Non-volatile main memory *(NVMM)*

new persistent

Durable

resists rebo

High-density

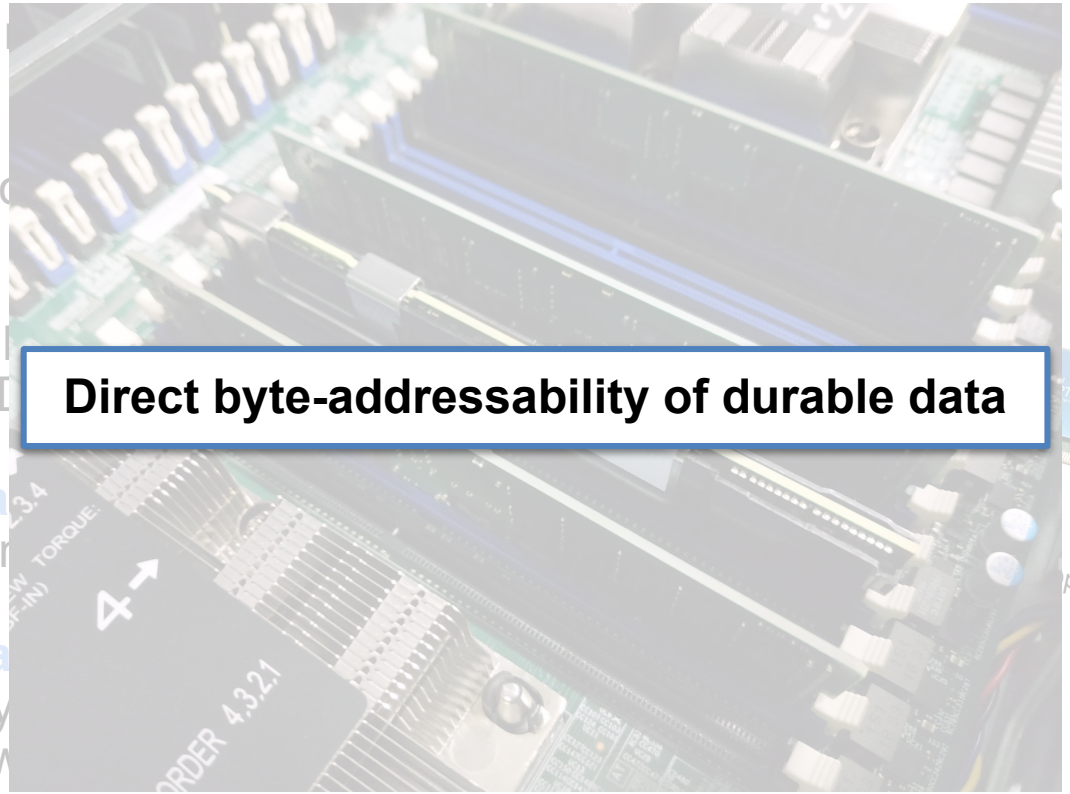
smallest DI
up to 8x DI

Byte addressa

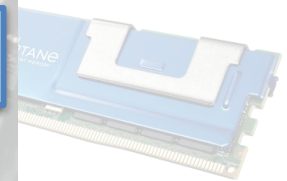
persistent

High-performa

low latency
high bandw



Direct byte-addressability of durable data



ptane PMEM, 2019

Non-volatile main memory *(NVMM)*

new persistent

Durable
resists rebo

High-density
smallest DI
up to 8x DI

Byte addressa
persistent

High-performa
low latency
high bandw

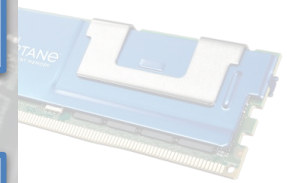
Direct byte-addressability of durable data

=

Dramatic **throughput** and **latency** improvement for persistent data applications

+

Simpler **code** bases with **single data representation** and **no file I/Os**



ptane PMEM, 2019

NVMM - *Usage*

How do we use it ? Storage device compatibility mode (1) ? Persistent Memory (2) ?

(1) File system interface
open/close/read/write/sync

(2) Direct memory access
mmap

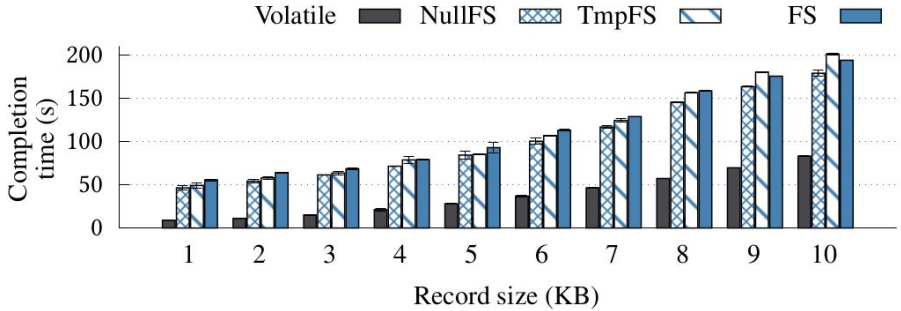
Intel Optane PMEM, 2019

NVMM - *Usage*

How do we use it ? Storage device compatibility mode (1) ?

(1) File system interface

open/close/read/write/sync



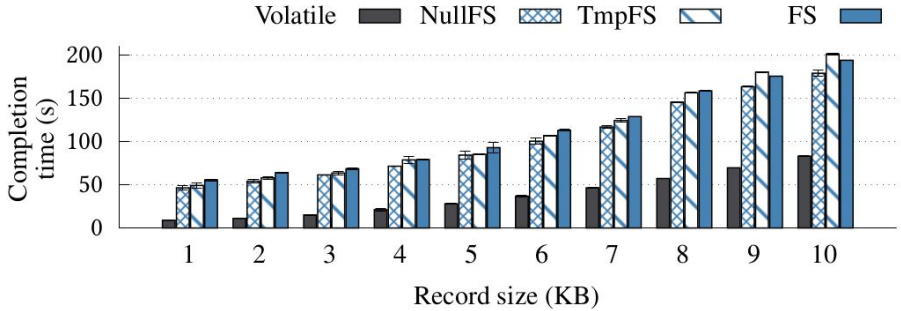
Varying record size in YCSB-F.

NVMM - *Usage*

How do we use it ? Storage device compatibility mode (1) ?

(1) File system interface

open/close/read/write/sync



Varying record size in YCSB-F.

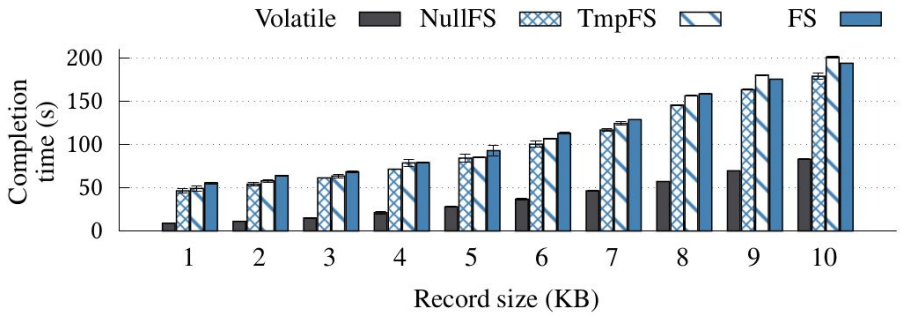
- **Disabling durability** significantly **boost performance**
- **Dummy file systems** are seemingly **identical** to a **PMEM FS**

NVMM - *Usage*

How do we use it ? Storage device compatibility mode (1) ?

(1) File system interface

open/close/read/write/sync



Varying record size in YCSB-F.

Software Bottlenecks:

- dual representation (consistency)
- cost of marshalling

- **Disabling durability** significantly **boost performance**
- **Dummy file systems** are seemingly **identical** to a **PMEM FS**

NVMM - *Usage*

How do we use it ? Persistent Memory (2) ?

(2) Direct memory access

mmap

+ memory **load/store** CPU instructions

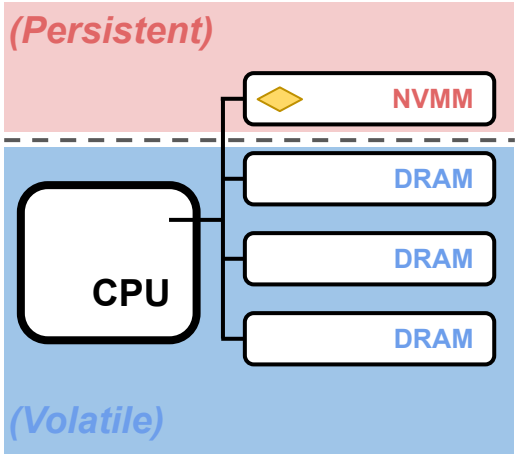
NVMM - *Usage*

How do we use it ? Persistent Memory (2) ?

(2) Direct memory access

mmap

+ memory **load/store** CPU instructions



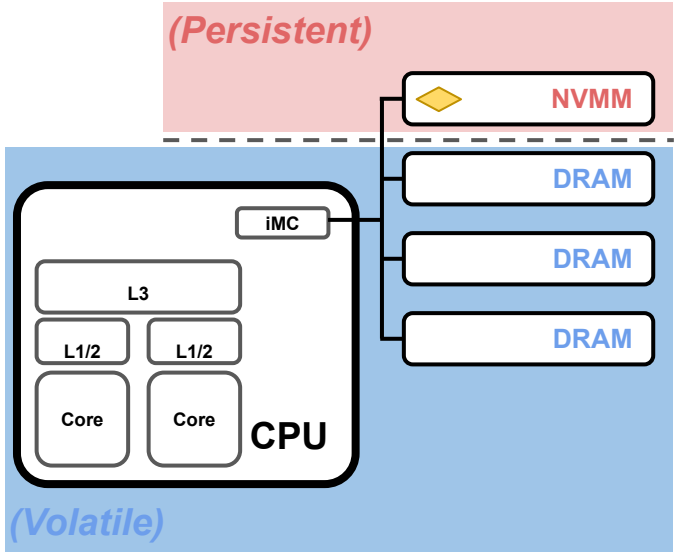
NVMM - *Usage*

How do we use it ? Persistent Memory (2) ?

(2) Direct memory access

mmap

+ memory **load/store** CPU instructions



NVMM - *Usage*

How do we use it ? Persistent Memory (2) ?

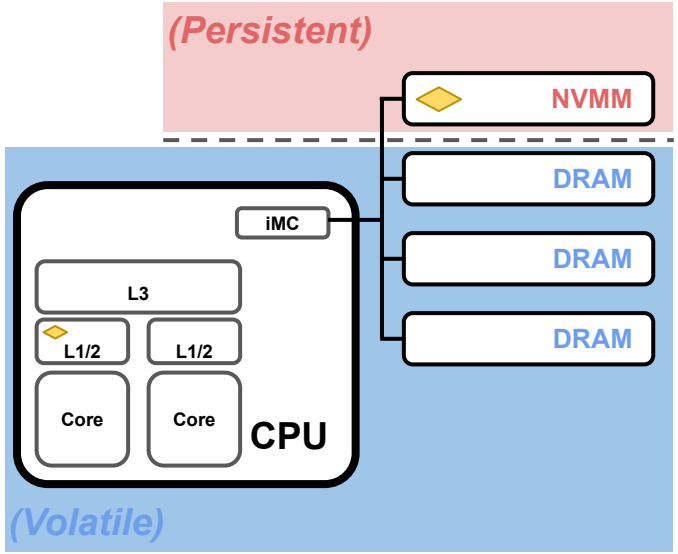
(2) Direct memory access

mmap

+ memory **load/store** CPU instructions

data update path

*1- **pull data** in CPU **caches***



NVMM - *Usage*

How do we use it ? Persistent Memory (2) ?

(2) Direct memory access

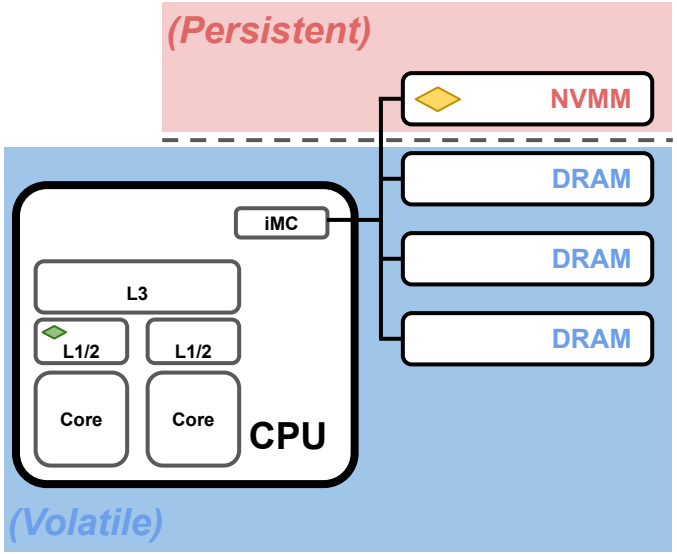
mmap

+ memory **load/store** CPU instructions

data update path

*1- **pull data** in CPU **caches***

*2- **update data** in CPU **caches first,***



NVMM - *Usage*

How do we use it ? Persistent Memory (2) ?

(2) Direct memory access

mmap

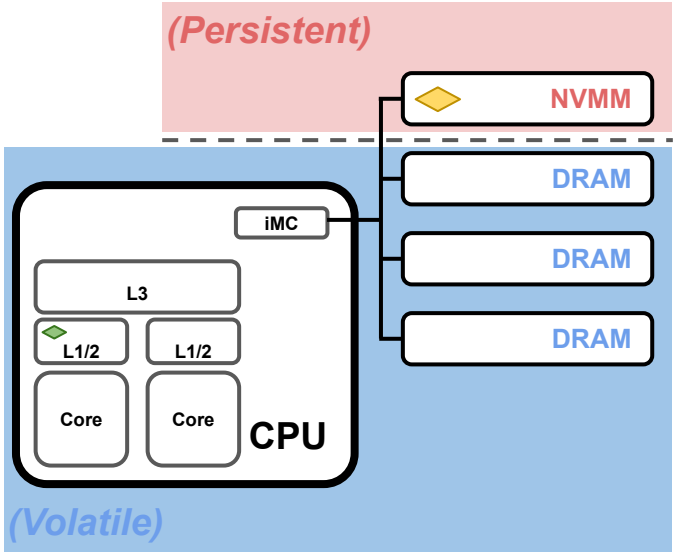
+ memory **load/store** CPU instructions

data update path

*1- **pull data** in CPU **caches***

*2- **update data** in CPU **caches first,***

NVMM** version becomes **stale



NVMM - *Usage*

How do we use it ? Persistent Memory (2) ?

(2) Direct memory access

mmap

+ memory **load/store** CPU instructions

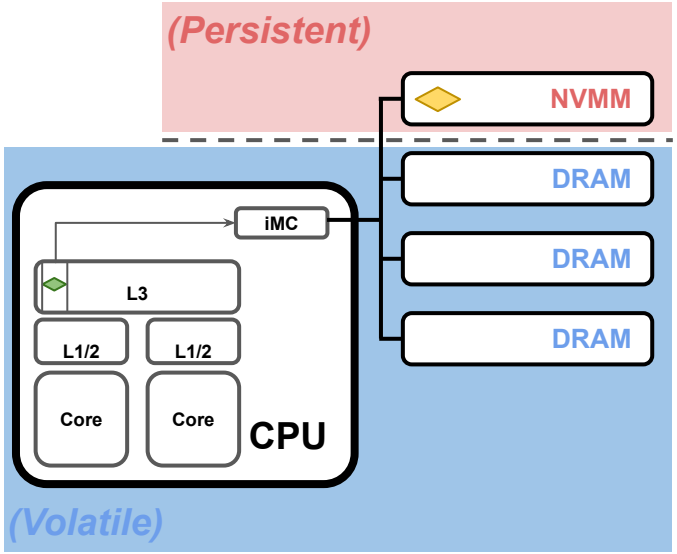
data update path

*1- **pull data** in CPU **caches***

*2- **update data** in CPU **caches first,***

NVMM** version becomes **stall

3-** Hardware dictates **when** and **in which order** data are **evicted** from **caches



NVMM - *Usage*

How do we use it ? Persistent Memory (2) ?

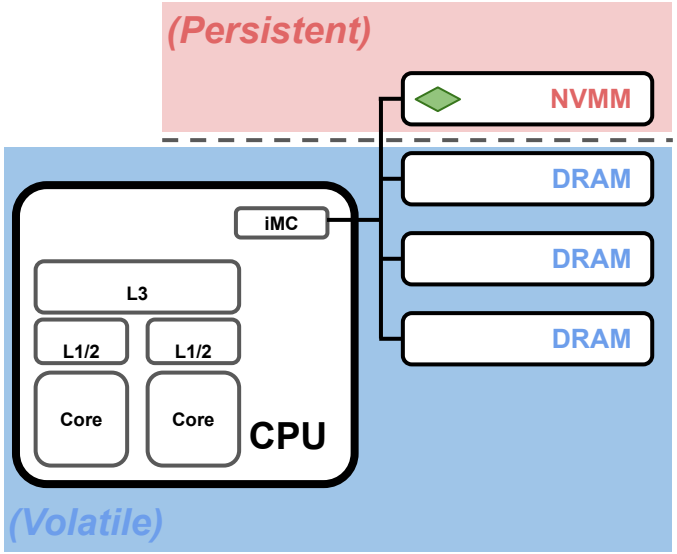
(2) Direct memory access

mmap

+ memory **load/store** CPU instructions

data update path

*1- **pull data** in CPU **caches***
*2- **update data** in CPU **caches first,** **NVMM** version becomes **stall***
3-** Hardware dictates **when** and **in which** **order** data are **evicted** from **caches
***4-** update **eventually** reaches NVMM*



NVMM - *Usage*

How do we use it ? Persistent Memory (2) ?

(2) Direct memory access

mmap

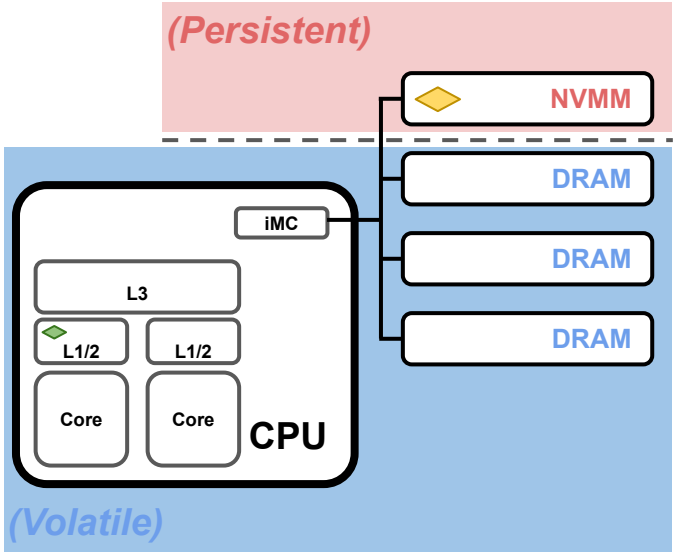
+ memory **load/store** CPU instructions

Version in **caches** (◆) diverged from **NVMM** (◆)

*Is it always **safe** to recover (◆) **after a crash** ?*
*How to keep **data crash-consistent** ?*

data update path

*1- **pull data** in CPU **caches***
*2- **update data** in CPU **caches first,** **NVMM** version becomes **stall***
*3- Hardware dictates **when** and **in which** **order** data are **evicted** from **caches***
*4- update **eventually** reaches NVMM*



NVMM - *Usage*

How do we use it ? Persistent Memory (2) ?

(2) Direct memory access

mmap

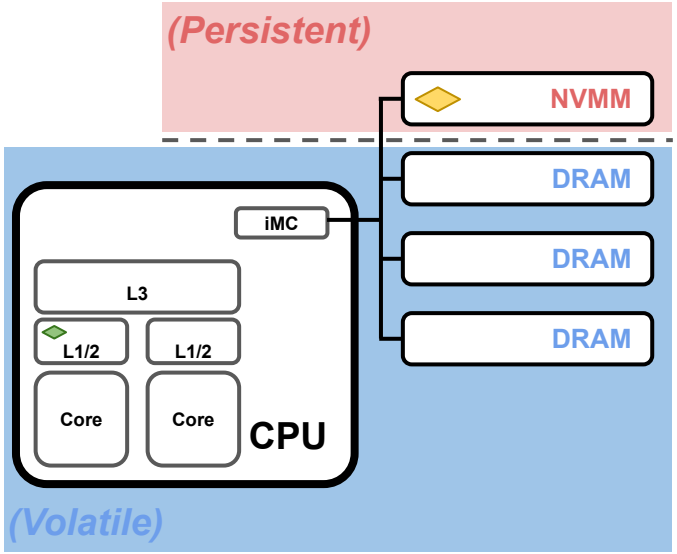
+ memory **load/store** CPU instructions

Version in **caches** (◆) diverged from **NVMM** (◆)

*Is it always **safe** to recover (◆) **after a crash** ?*
*How to keep **data crash-consistent** ?*

data update path
*1- **pull data** in CPU **caches***
*2- **update data** in CPU **caches first,** **NVMM** version becomes **stall***
*3- Hardware dictates **when** and **in which** **order** data are **evicted** from **caches***
*4- update **eventually** reaches NVMM*

+ special **flush/fence** CPU instructions
(*manually control cache line eviction order*)



NVMM - *Usage*

How do we use it ? Persistent Memory (2) ?

(2) Direct memory access

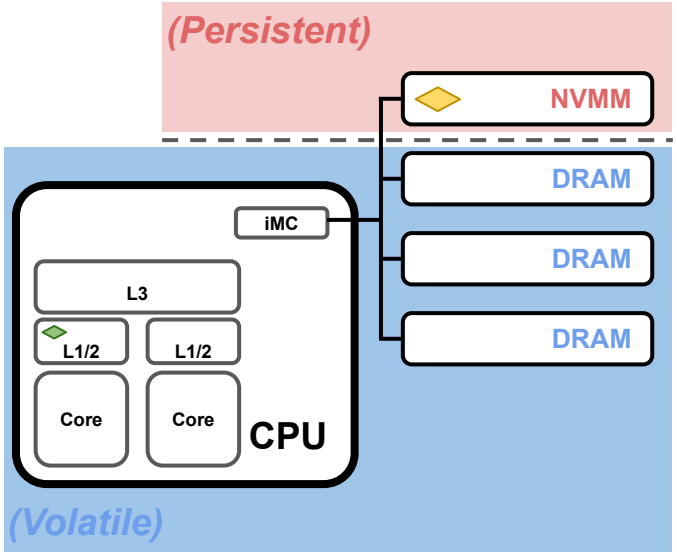
mmap

+ memory **load/store** CPU instructions

Version in **caches** (◆) diverged from **NVMM** (◆)

*Is it always **safe** to recover (◆) **after a crash** ?*
*How to keep **data crash-consistent** ?*

data update path
*1- **pull data** in CPU **caches***
*2- **update data** in CPU **caches first,** **NVMM** version becomes **stall***
*3- Hardware dictates **when** and **in which** **order** data are **evicted** from **caches***
*4- update **eventually** reaches NVMM*



+ special **flush/fence** CPU instructions
(*manually control cache line eviction order*)

⇒ *Too low-level programming*
⇒ *Brittle reasoning about crash-consistency*

NVMM - *Usage*

*How do we use it ? Persistent Memory **(2)** ?*

(2) **Direct memory access** ~ *the easy way*

mmap

+ fitting **programming abstractions** (e.g. Intel's PMDK)

⇒ *ensure data **crash-consistency***

⇒ *aid data **recovery***

NVMM - *Usage*

How do we use it ? Persistent Memory (2) ?

(2) Direct memory access ~ *the easy way*

mmap

+ fitting **programming abstractions** (e.g. Intel's PMDK)

⇒ *ensure data **crash-consistency***

⇒ *aid data **recovery***

challenges:

_ support for **failure-atomicity** abstractions ?

_ **persistent memory allocator** ?

_ **persistent pointers** in ephemeral process address space ?

NVMM - *Usage*

How do we use it ? Persistent Memory (2) ?

(2) Direct memory access ~ *the easy way*

mmap

+ fitting **programming abstractions** (e.g. Intel's PMDK)

⇒ *ensure data **crash-consistency***

⇒ *aid data **recovery***

challenges:

_ support for **failure-atomicity** abstractions ?

_ **persistent memory allocator** ?

_ **persistent pointers** in ephemeral process address space ?

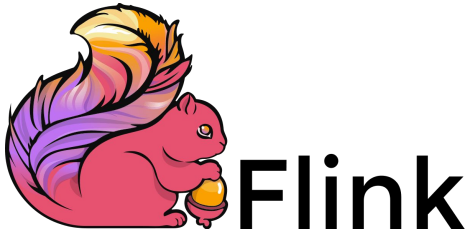
Language-level NVMM programming **help & support**

Why **Java**? - *greatest programming language (seriously)*

Managed language - first released in the 90s - still an industry standard.

Many data stores & processing frameworks:

- *Spark, Hadoop, Kafka, Flink, Cassandra, HBase, Elasticsearch, etc.*



Why **Java**? - *(lack of) NVMM support*

Nearly all **NVMM libraries** and **tools** support only native code (*C, C++*)

Lack of *efficient* interfaces :

(1) File System [ext4-dax]

- *storage device compatibility mode* (*cf. slide 6*)

(2) Intel's PMDK through the Java Native Interface (**JNI**) [PCJ]

- *native library with compatibility layer*
  - slower than **FS** on YCSB benchmark (*cf. evaluation*)

Problem statement: No Java-native NVMM interface

Prior art: *internal design*

= [Espresso, AutoPersist, go-pmem]

Managed persistent Java objects

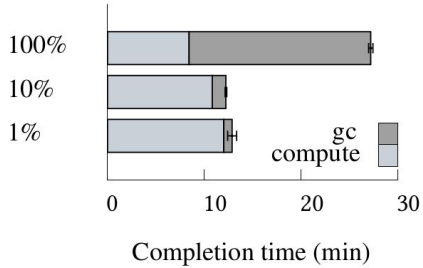
= extend JVM to manage persistent memory

Prior art: *internal design*

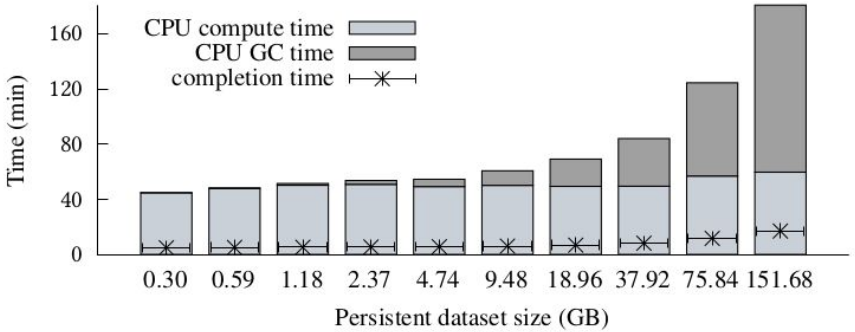
= [Espresso, AutoPersist, go-pmem]

Managed persistent Java objects

= extend JVM to manage persistent memory



Exp1: *Varying cache ratio*
(YCSB-F, Infinispan with 80GB dataset)



Exp2: *Increasing dataset*
(YCSB-F, go-pmem)

GC cost outweighs the benefits of large DRAM caches

Prior art: *internal design*

= [Espresso, AutoPersist, go-pmem]

- Garbage collectors do not scale to large persistent datasets

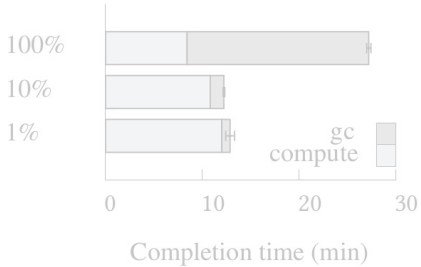
= extend JVM to manage persistent memory

Features

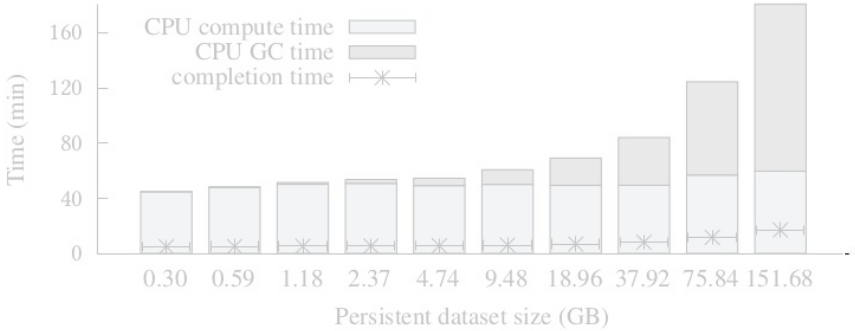
managed persistent objects

non-scalable

heavily-modified JVM



Exp1: *Varying cache ratio*
(YCSB-F, Infinispan with 80GB dataset)



Exp2: *Increasing dataset*
(YCSB-F, go-pmem)

GC cost outweighs the benefits of large DRAM caches

Prior art: *internal design*

= [Espresso, AutoPersist, go-pmem]

- Garbage collectors do not scale to large persistent datasets

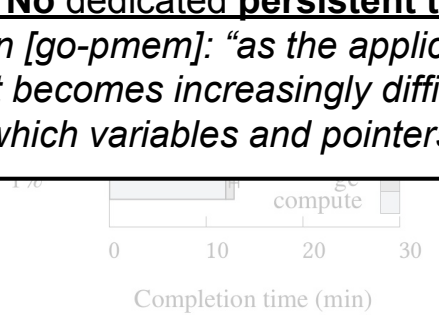
= extend JVM to manage persistent memory

- No dedicated persistent types

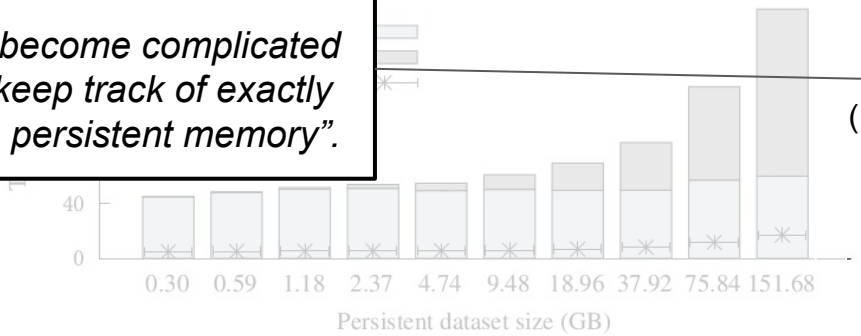
In [go-pmem]: "as the applications become complicated it becomes increasingly difficult to keep track of exactly which variables and pointers are in persistent memory".

Features

- *managed* persistent objects
  - non-scalable
  - heavily-modified JVM
- orthogonal persistence (`pnew`, `@persistentRoot`)



Exp1: *Varying cache ratio (YCSB-F, Infinispan with 80GB dataset)*



Exp2: *Increasing dataset (YCSB-F, go-pmem)*

GC cost outweighs the benefits of large DRAM caches

Prior art: *internal design*

= [Espresso, AutoPersist, go-pmem]

- Garbage collectors do not scale to large persistent datasets

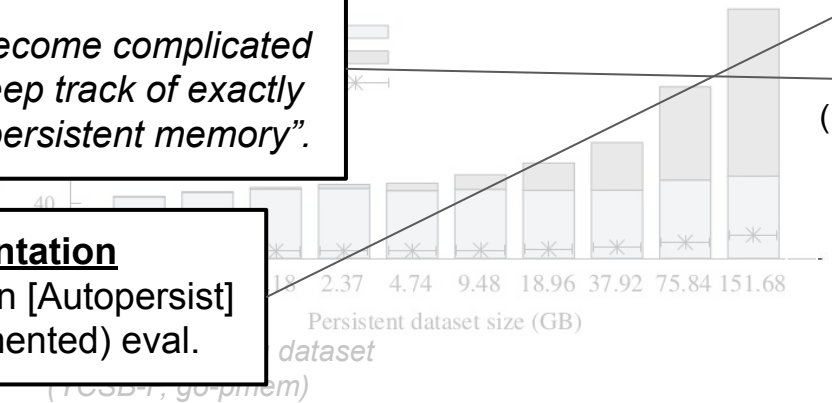
= extend JVM to manage persistent memory

- No dedicated persistent types

In [go-pmem]: "as the applications become complicated it becomes increasingly difficult to keep track of exactly which variables and pointers are in persistent memory".

- Heavy runtime dynamic instrumentation

code instrumentation = 51% slower in [Autopersist]
up to 48% slower in our (non-instrumented) eval.



Features

managed persistent objects

non-scalable

heavily-modified JVM

orthogonal persistence
(`pnew`, `@persistentRoot`)

failure-atomic blocks

GC cost outweighs the benefits of large DRAM caches

Contribution Overview

J-NVM - Off-Heap Persistent Objects

Challenges

Persistent Memory

(1) *single data representation*

(2) *direct access to NVMM*

(3) *crash-consistency*

Java

(4) *object-oriented idioms*

(5) explicit persistent types

(6) *persistent memory management*
with low overhead and scalable to large heaps

Contribution Overview

J-NVM - Off-Heap Persistent Objects

	Challenges	→	Features
Persistent Memory	(1) *single data representation*		(1, 6) ***off-heap*** persistent objects
	(2) *direct access to NVMM*		(2) sun.misc.Unsafe
	(3) *crash-consistency*		(3) failure-atomic blocks + fine-grained
Java	(4) *object-oriented idioms*		(4) persistent java objects + PDT library
	(5) explicit persistent types		(5) **class-centric** model + code generator
	(6) *persistent memory management* *with low overhead and scalable to large heaps*		(6) **recovery-time** GC (*no online GC*) **explicit** free()

Demo - *the application*

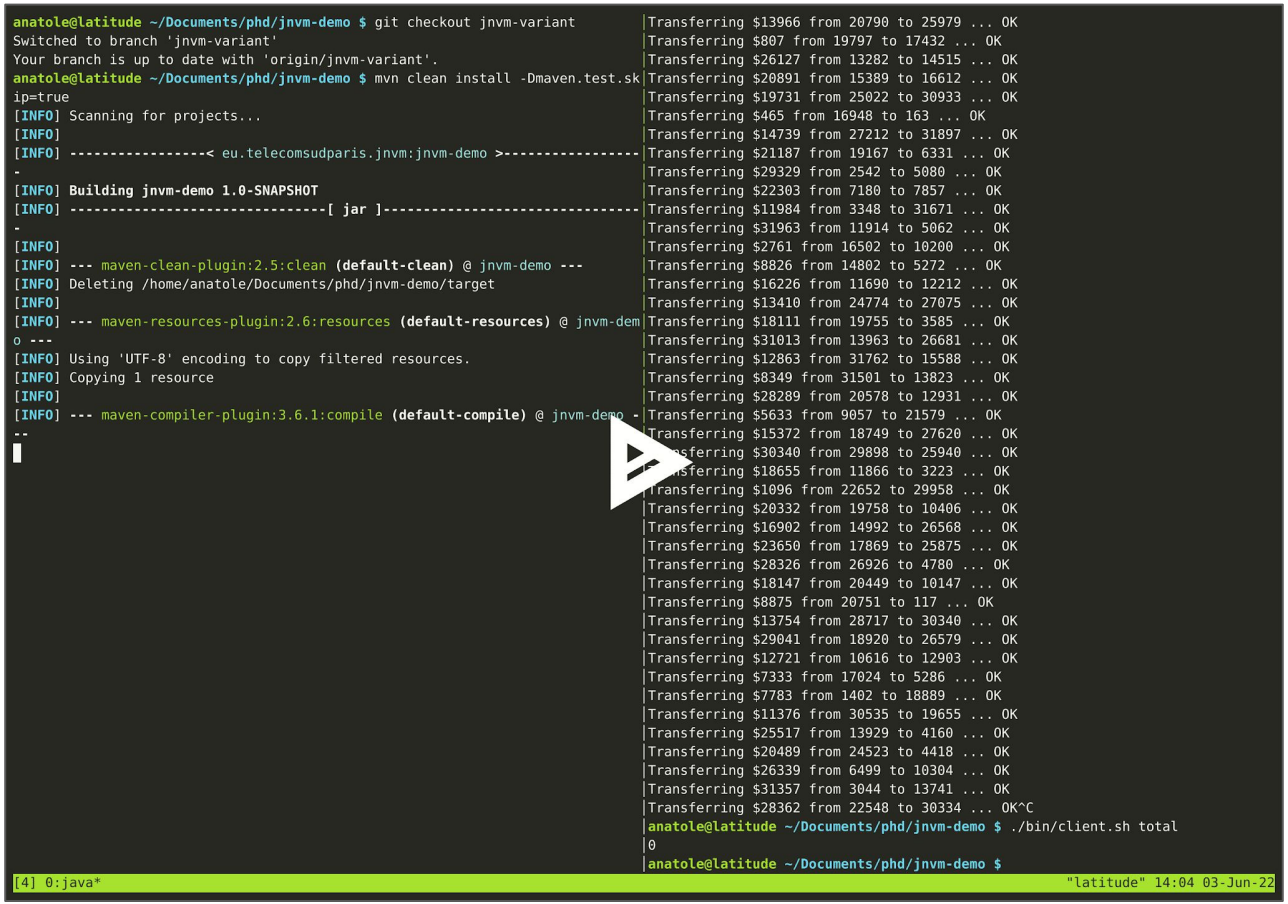
A Simple Bank:

Server

Bank
-*accounts*: Map<String, Account>
+performTransfer(String *from*, String *to*, long *amount*) +createAccount(String *id*, long *initialDeposit*)

Account
-*id*: Integer -*balance*: Long
+transferTo(Account *dest*, long *amount*)

Demo - *It's showtime !*



Outline

(1) Introduction
- data persistence
- persistent memory
- NVMM
- why Java?
- prior art
- contribution overview

(2) System Design of J-NVM
- demo
- key idea
- programming model
  - persistent objects
  - code generator
- J-PFA
- J-PDT

(3) Evaluation
- YCSB benchmark
- recovery

(4) Conclusion

Overview

J-NVM = Off-Heap Persistent Objects

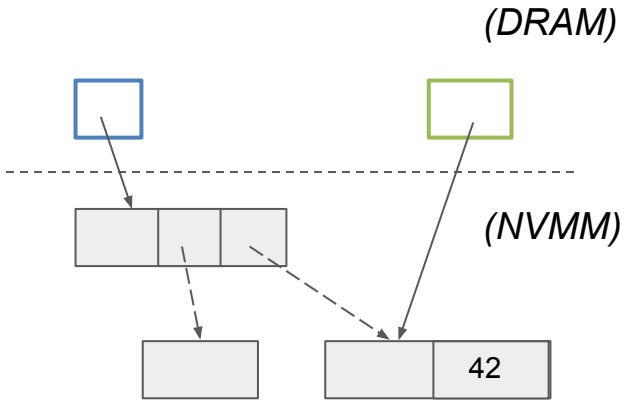
Key idea
each persistent object is ***decoupled*** into

- *a persistent data structure*: ***unmanaged***, allocated off-heap (NVMM)
- *a proxy*: ***managed***, allocated on-heap (DRAM)

Programming model - *persistent objects*

Persistent object is

- *a persistent data structure*
  - holds object fields
- *a proxy*
  - holds object methods
  - implements `PObject` interface
  - intermediates access to pers. data structure
  - instantiated lazily (low GC pressure)



Alive when reachable (from persistent root)

Class-centric model

- safe references thanks to the type system

```
Map root = JNVM.root();
Bank b = root.get("Bank");
Account a = b.find("toto");
a.setBalance(42);
```

Programming model - *life cycle*

Constructor
- allocate NVMM
- attach persistent data structure

Re-Constructor
- re-attach proxy
- re-build soft state via resurrect()

Destructor
- explicit **JNVM**.free() to reclaim NVMM
- detach proxy
- ready to be GCed

```
Account a = new Account(“toto”, 42);
```

(DRAM)

(NVMM)

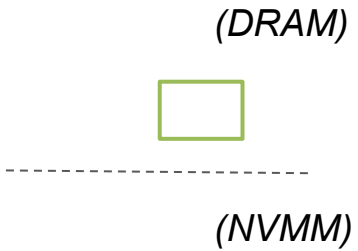
Programming model - *life cycle*

Constructor
- allocate NVMM
- attach persistent data structure

Re-Constructor
- re-attach proxy
- re-build soft state via resurrect()

Destructor
- explicit **JNVM**.free() to reclaim NVMM
- detach proxy
- ready to be GCed

```
Account a = new Account(“toto”, 42);
```



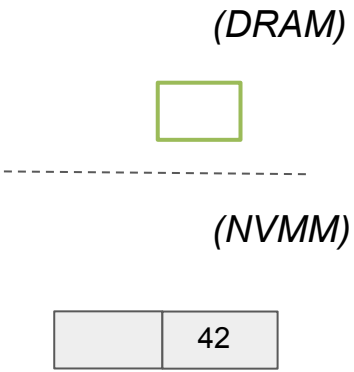
Programming model - *life cycle*

Constructor
- allocate NVMM
- attach persistent data structure

Re-Constructor
- re-attach proxy
- re-build soft state via resurrect()

Destructor
- explicit **JNVM**.free() to reclaim NVMM
- detach proxy
- ready to be GCed

```
Account a = new Account(“toto”, 42);
```



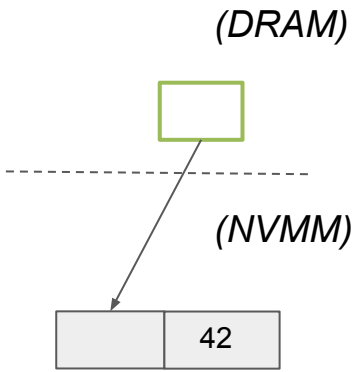
Programming model - *life cycle*

Constructor
- allocate NVMM
- attach persistent data structure

Re-Constructor
- re-attach proxy
- re-build soft state via resurrect()

Destructor
- explicit **JNVM**.free() to reclaim NVMM
- detach proxy
- ready to be GCed

```
Account a = new Account(“toto”, 42);
```



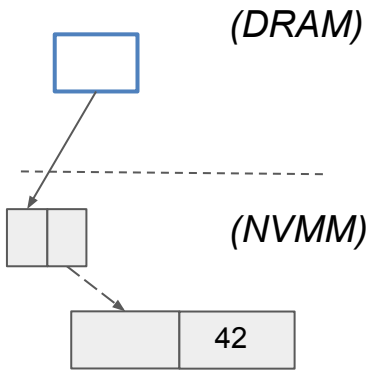
Programming model - *life cycle*

Constructor
- allocate NVMM
- attach persistent data structure

Re-Constructor
- re-attach proxy
- re-build soft state via resurrect()

Destructor
- explicit **JNVM**.free() to reclaim NVMM
- detach proxy
- ready to be GCed

```
Account a = bank.find("toto");
```



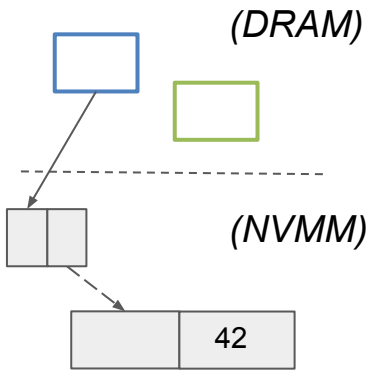
Programming model - *life cycle*

Constructor
- allocate NVMM
- attach persistent data structure

Re-Constructor
- re-attach proxy
- re-build soft state via resurrect()

Destructor
- explicit **JNVM**.free() to reclaim NVMM
- detach proxy
- ready to be GCed

```
Account a = bank.find("toto");
```



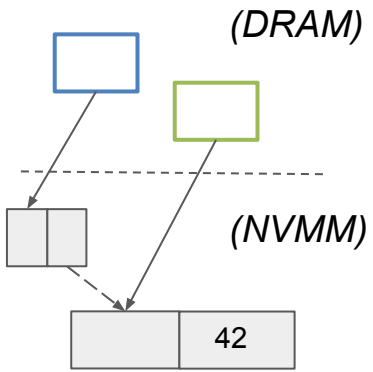
Programming model - *life cycle*

Constructor
- allocate NVMM
- attach persistent data structure

Re-Constructor
- re-attach proxy
- re-build soft state via resurrect()

Destructor
- explicit **JNVM**.free() to reclaim NVMM
- detach proxy
- ready to be GCed

```
Account a = bank.find("toto");
```



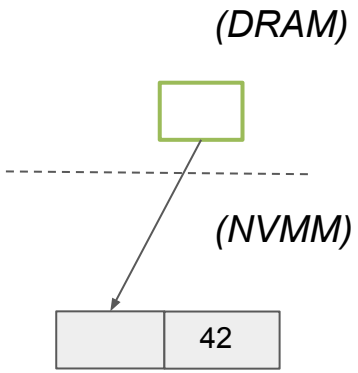
Programming model - *life cycle*

Constructor
- allocate NVMM
- attach persistent data structure

Re-Constructor
- re-attach proxy
- re-build soft state via resurrect()

Destructor
- explicit **JNVM**.free() to reclaim NVMM
- detach proxy
- ready to be GCed

```
JNVM.free(a);
```



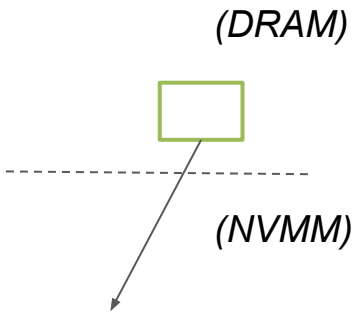
Programming model - *life cycle*

Constructor
- allocate NVMM
- attach persistent data structure

Re-Constructor
- re-attach proxy
- re-build soft state via resurrect()

Destructor
- explicit **JNVM**.free() to reclaim NVMM
- detach proxy
- ready to be GCed

```
JNVM.free(a);
```



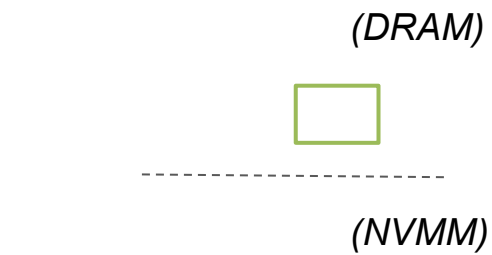
Programming model - *life cycle*

Constructor
- allocate NVMM
- attach persistent data structure

Re-Constructor
- re-attach proxy
- re-build soft state via resurrect()

Destructor
- explicit **JNVM**.free() to reclaim NVMM
- detach proxy
- ready to be GCed

```
JNVM.free(a);
```



Programming model - *life cycle*

Constructor
- allocate NVMM
- attach persistent data structure

Re-Constructor
- re-attach proxy
- re-build soft state via resurrect()

Destructor
- explicit **JNVM**.free() to reclaim NVMM
- detach proxy
- ready to be GCed

```
JNVM.free(a);
```

(DRAM)

(NVMM)

Overview

J-NVM = Off-Heap Persistent Objects

Tooling

- built-in off-heap memory management for NVMM
- **code generator:** automatic decoupling for POJOs
- **J-PFA:** automatic failure-atomic code
- **J-PDT:** data types + collections for persistent memory
- low-level API (for experts)
- recovery-time GC

Programming model - *code generator*

Goals

(1) compute class-wide ***off-heap layout***
(2) replace (non-transient) **field accesses**
(3) generate ***constructor***, ***re-constructor***
(4) ***FA-wrap*** non-private methods

```
@Persistent(fa="non-private")
class Account {
  PString name;
  int balance;
  transient int y;

  Account(String name, int balance) {
    this.name = new PString(id);
    this.balance = balance;
  }

  void transferTo(Account dest, int amount) {
    this.balance -= amount;
    dest.balance += amount;
  }
}
```

Programming model - *code generator*

Goals

(1) compute class-wide ***off-heap layout***
(2) replace (non-transient) **field accesses**
(3) generate ***constructor***, ***re-constructor***
(4) ***FA-wrap*** non-private methods

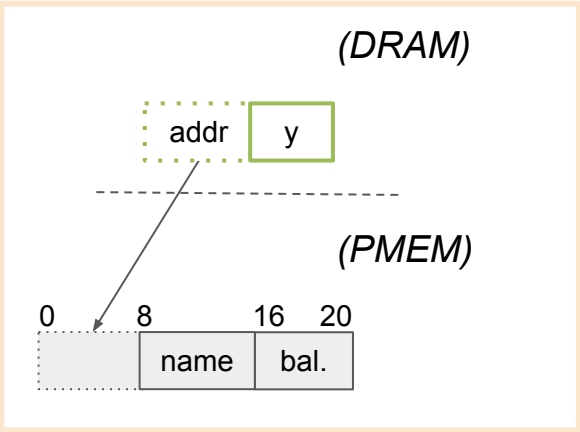
```
@Persistent(fa="non-private")
class Account {
  PString name;
  int balance;
  transient int y;

  Account(String name, int balance) {
    this.name = new PString(id);
    this.balance = balance;
  }

  void transferTo(Account dest, int amount) {
    this.balance -= amount;
    dest.balance += amount;
  }
}
```

Programming model - *code generator*

Implementation

Goals

(1) compute class-wide ***off-heap layout***
(2) replace (non-transient) **field accesses**
(3) generate ***constructor***, ***re-constructor***
(4) ***FA-wrap*** non-private methods



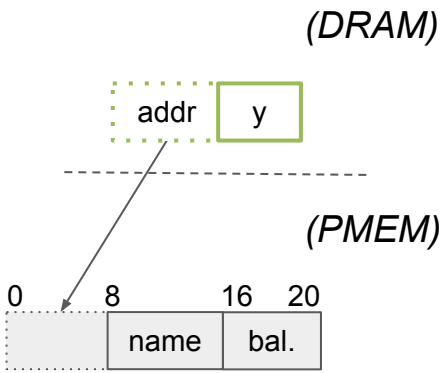

```
@Persistent(fa="non-private")
class Account {
  PString name;
  int balance;
  transient int y;

  Account(String name, int balance) {
    this.name = new PString(id);
    this.balance = balance;
  }

  void transferTo(Account dest, int amount) {
    this.balance -= amount;
    dest.balance += amount;
  }
}
```

Programming model - *code generator*

Implementation

Goals

(1) compute class-wide ***off-heap layout***
  (a) *remove persistent attributes*
(2) replace (non-transient) **field accesses**
(3) generate ***constructor***, ***re-constructor***
(4) ***FA-wrap*** non-private methods



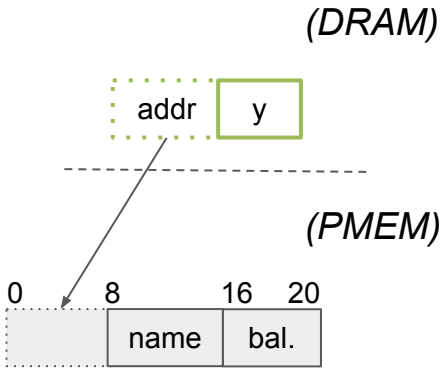

```
// transformed code (decompiled)
class Account {

  transient int y;

  Account(String name, int balance) {
    this.name = new PString(id);
    this.balance = balance;
  }

  void transferTo(Account dest, int amount) {
    this.balance -= amount;
    dest.balance += amount;
  }
}
```

Programming model - *code generator*

Implementation

Goals

(1) compute class-wide ***off-heap layout***
  (a) remove persistent attributes
  *(b) add **"addr"** field*
(2) replace (non-transient) **field accesses**
(3) generate ***constructor***, ***re-constructor***
(4) ***FA-wrap*** non-private methods




```
// transformed code (decompiled)
class Account {
  long addr; // persistent data structure
  transient int y;

  Account(String name, int balance) {
    this.name = new PString(id);
    this.balance = balance;
  }

  void transferTo(Account dest, int amount) {
    this.balance -= amount;
    dest.balance += amount;
  }
}
```

Programming model - *code generator*

Goals

(1) compute class-wide ***off-heap layout***
  (a) *remove persistent attributes*
  (b) *add "addr" field*
  (c) *generate or transform field* ***getters/setters***
(2) replace (non-transient) **field accesses**
(3) generate ***constructor***, ***re-constructor***
(4) ***FA-wrap*** non-private methods

```
// transformed code (continued)
  PString getName() {
    return JNVM.readPObject(addr, 0);
  }
  protected void setName(PString v) {
    JNVM.writePObject(addr, 0, v);
  }

  int getBalance() {
    return JNVM.readInt(addr, 8);
  }
  void setBalance(int v) {
    JNVM.writeInt(addr, 8, v);
  }
…
```

```
// transformed code (decompiled)
class Account {
  long addr; // persistent data structure
  transient int y;

  Account(String name, int balance) {
    this.name = new PString(id);
    this.balance = balance;
  }

  void transferTo(Account dest, int amount) {
    this.balance -= amount;
    dest.balance += amount;
  }
…
```

Programming model - *code generator*

Goals

(1) compute class-wide ***off-heap layout***
  (a) *remove persistent attributes*
  (b) *add "addr" field*
  (c) *generate or transform field* ***getters/setters***
(2) replace (non-transient) **field accesses**
(3) generate ***constructor***, ***re-constructor***
(4) ***FA-wrap*** non-private methods

```
// transformed code (continued)
  PString getName() {
    return JNVM.readPObject(addr, 0);
  }
  protected void setName(PString v) {
    JNVM.writePObject(addr, 0, v);
  }

  int getBalance() {
    return JNVM.readInt(addr, 8);
  }
  void setBalance(int v) {
    JNVM.writeInt(addr, 8, v);
  }
…
```

```
// transformed code (decompiled)
class Account implements PObject {
  long addr; // persistent data structure
  transient int y;

  Account(String name, int balance) {
    this.name = new PString(id);
    this.balance = balance;
  }

  void transferTo(Account dest, int amount) {
    this.balance -= amount;
    dest.balance += amount;
  }
…
```

1- Use `JNVM` static helpers

Programming model - *code generator*

Goals

(1) compute class-wide ***off-heap layout***
  (a) *remove persistent attributes*
  (b) *add "addr" field*
  (c) *generate or transform field* ***getters/setters***
(2) replace (non-transient) **field accesses**
(3) generate ***constructor***, ***re-constructor***
(4) ***FA-wrap*** non-private methods

```
// transformed code (continued)
  PString getName() {
    return JNVM.readPObject(addr, 0);
  }
  protected void setName(PString v) {
    JNVM.writePObject(addr, 0, v);
  }

  int getBalance() {
    return JNVM.readInt(addr, 8);
  }
  void setBalance(int v) {
    JNVM.writeInt(addr, 8, v);
  }
…
```

```
// transformed code (decompiled)
class Account implements PObject {
  long addr; // persistent data structure
  transient int y;

  Account(String name, int balance) {
    this.name = new PString(id);
    this.balance = balance;
  }

  void transferTo(Account dest, int amount) {
    this.balance -= amount;
    dest.balance += amount;
  }
…
```

1- Use JNVM static helpers with field offsets

Programming model - *code generator*

Implementation

Goals

(1) compute class-wide ***off-heap layout***
  (a) *remove persistent attributes*
  (b) *add "addr" field*
  (c) *generate or transform field* ***getters/setters***
(2) replace (non-transient) ***field accesses***
(3) generate ***constructor***, ***re-constructor***
(4) ***FA-wrap*** non-private methods

```
// transformed code (continued)
  PString getName() {
    return JNVM.readPObject(addr, 0);
  }
  protected void setName(PString v) {
    JNVM.writePObject(addr, 0, v);
  }

  int getBalance() {
    return JNVM.readInt(addr, 8);
  }
  void setBalance(int v) {
    JNVM.writeInt(addr, 8, v);
  }
…
```

```
// transformed code (decompiled)
class Account implements PObject {
  long addr; // persistent data structure
  transient int y;

  Account(String name, int balance) {
    this.name = new PString(id);
    this.balance = balance;
  }

  void transferTo(Account dest, int amount) {
    this.balance -= amount;
    dest.balance += amount;
  }
…
```

1- Use `JNVM` static helpers with field offsets
2- internal setter for *final* fields

Programming model - *code generator*

Goals

(1) compute class-wide ***off-heap layout***
  (a) *remove persistent attributes*
  (b) *add "addr" field*
  (c) *generate or transform field* ***getters/setters***

(2) replace (non-transient) **field accesses**

(3) generate ***constructor***, ***re-constructor***

(4) ***FA-wrap*** non-private methods

```
// transformed code (decompiled)
class Account implements PObject {
  long addr; // persistent data structure
  transient int y;

  Account(String name, int balance) {
    this.setName(new PString(id));
    this.setBalance(balance);
  }

  void transferTo(Account dest, int amount) {
    this.setBalance(getBalance() - amount);
    dest.setBalance(dest.getBalance() + amount);
  }
…
```

Programming model - *code generator*

Goals

(1) compute class-wide ***off-heap layout***
  (a) *remove persistent attributes*
  (b) *add "addr" field*
  (c) *generate or transform field* ***getters/setters***
(2) replace (non-transient) **field accesses**
(3) generate ***constructor***, ***re-constructor***
(4) ***FA-wrap*** non-private methods

```
// transformed code (decompiled)
class Account implements PObject {
  long addr; // persistent data structure
  transient int y;

  Account(String name, int balance) {
    this.addr = JNVM.alloc(getClass(), size());
    this.setName(new PString(id));
    this.setBalance(balance);
  }

  void transferTo(Account dest, int amount) {
    this.setBalance(getBalance() - amount);
    dest.setBalance(dest.getBalance() + amount);
  }
…
```

```
// transformed code (continued)
  Account(long addr) {
    this.addr = addr;
    this.resurrect();
  }
…
```

Programming model - *code generator*

Goals

(1) compute class-wide ***off-heap layout***
  (a) *remove persistent attributes*
  (b) *add "addr" field*
  (c) *generate or transform field* ***getters/setters***
(2) replace (non-transient) **field accesses**
(3) generate ***constructor***, ***re-constructor***
(4) ***FA-wrap*** non-private methods

```
// transformed code (decompiled)
class Account implements PObject {
  long addr; // persistent data structure
  transient int y;

  Account(String name, int balance) {
    JNVM.faStart();
    this.addr = JNVM.alloc(getClass(), size());
    this.setName(new PString(id));
    this.setBalance(balance);
    JNVM.faEnd();
  }

  void transferTo(Account dest, int amount) {
    JNVM.faStart();
    this.setBalance(getBalance() - amount);
    dest.setBalance(dest.getBalance() + amount);
    JNVM.faEnd();
  }
...
```

Programming model - *code generator*

Goals

(1) compute class-wide ***off-heap layout***
  (a) *remove persistent attributes*
  (b) *add "addr" field*
  (c) *generate or transform field* ***getters/setters***
(2) replace (non-transient) **field accesses**
(3) generate ***constructor***, ***re-constructor***
(4) ***FA-wrap*** non-private methods

Tool implementation

(1) Bytecode-to-bytecode transformer
(2) post-compilation Maven plugin

```
// transformed code (decompiled)
class Account implements PObject {
  long addr; // persistent data structure
  transient int y;

  Account(String name, int balance) {
    JNVM.faStart();
    this.addr = JNVM.alloc(getClass(), size());
    this.setName(new PString(id));
    this.setBalance(balance);
    JNVM.faEnd();
  }

  void transferTo(Account dest, int amount) {
    JNVM.faStart();
    this.setBalance(getBalance() - amount);
    dest.setBalance(dest.getBalance() + amount);
    JNVM.faEnd();
  }
...
```

Automatic crash-consistent update

usage = **JNVM**.faStart() *some code* **JNVM**.faEnd()

Per-thread persistent redo-log (inspired by Romulus)

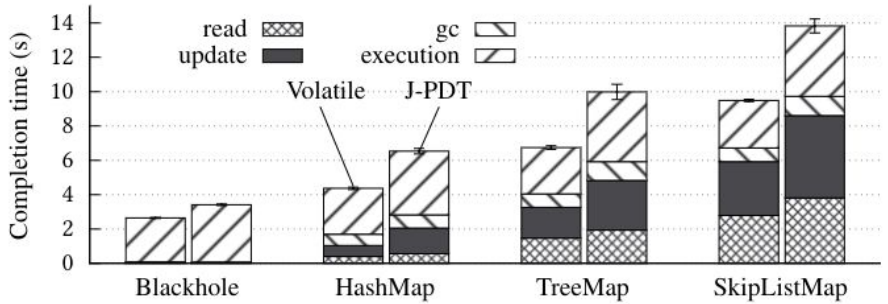
Log new, free and updates

granularity = a block of PMEM

Do *not* log updates to “new” persistent objects
(e.g. allocated within the FA-block)

Persistent **D**ata **T**ypes

- drop-in replacement for (part of) JDK
  e.g., string, native array, map.



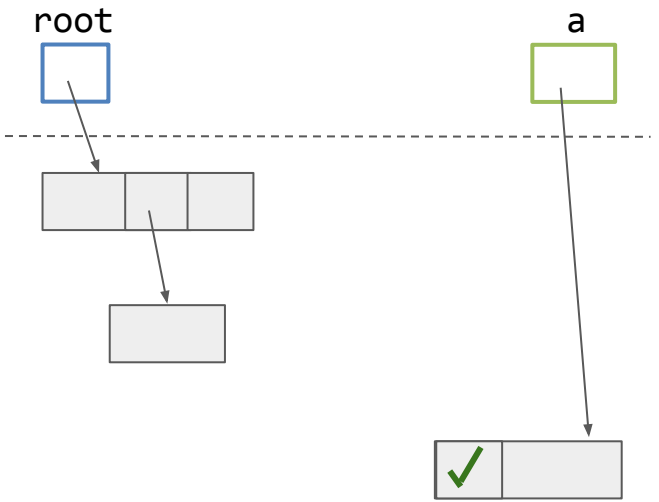
Persistent vs Volatile data types *(YCSB-A)*

Takeaway:

- around 50% slower than volatile data types on DRAM

Low-level interface

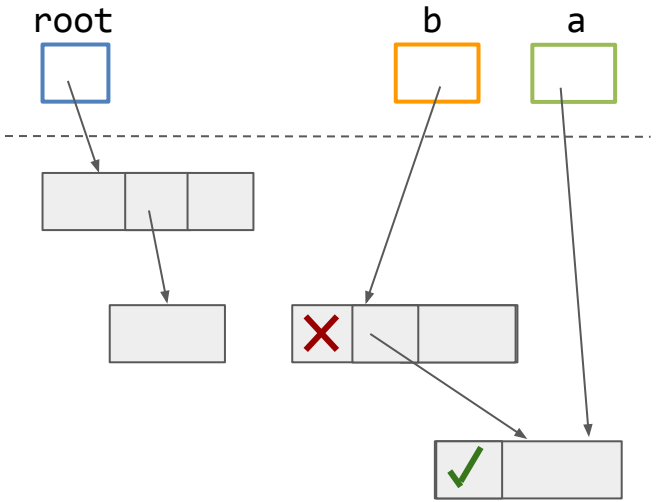
- unsafe.{pwb,pfence, psync}
- NVMM block allocator
- recovery time GC (à la Makalu)
- validation = 1 bit in object header
  - makes atomic reclamation easier
  - allows deferring object liveness
  - interpreted on recovery
    to reclaim reachable invalid objects




```
List<Account> a = randAcc(100);
Bank b = new Bank(a);
root.put("Bank", b);
b.validate();
```

Low-level interface

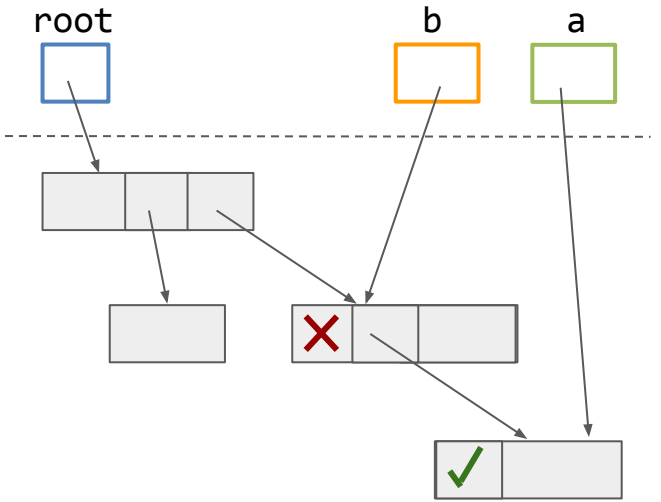
- unsafe.{pwb,pfence, psync}
- NVMM block allocator
- recovery time GC (à la Makalu)
- validation = 1 bit in object header
  - makes atomic reclamation easier
  - allows deferring object liveness
  - interpreted on recovery
    to reclaim reachable invalid objects




```
List<Account> a = randAcc(100);
Bank b = new Bank(a); //Not atomic
root.put("Bank", b);
b.validate();
```

Low-level interface

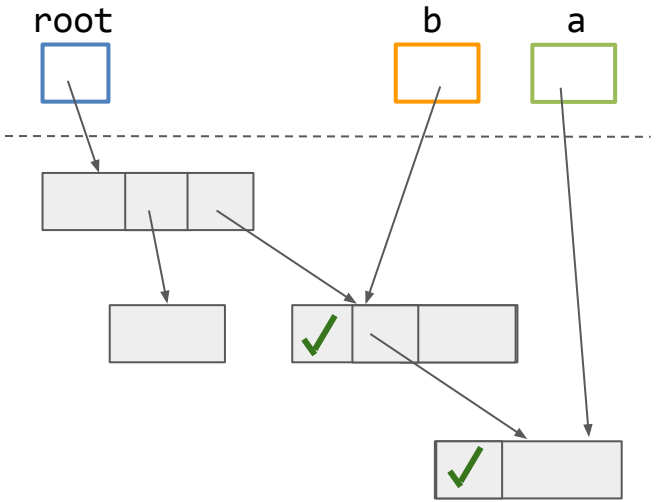
- unsafe.{pwb,pfence, psync}
- NVMM block allocator
- recovery time GC (à la Makalu)
- validation = 1 bit in object header
  - makes atomic reclamation easier
  - allows deferring object liveness
  - interpreted on recovery
    to reclaim reachable invalid objects




```
List<Account> a = randAcc(100);
Bank b = new Bank(a);
root.put(“Bank”, b); //Not atomic
b.validate();
```

Low-level interface

- unsafe.{pwb,pfence, psync}
- NVMM block allocator
- recovery time GC (à la Makalu)
- validation = 1 bit in object header
  - makes atomic reclamation easier
  - allows deferring object liveness
  - interpreted on recovery
    to reclaim reachable invalid objects




```
List<Account> a = randAcc(100);
Bank b = new Bank(a);
root.put("Bank", b);
b.validate();
```

Outline

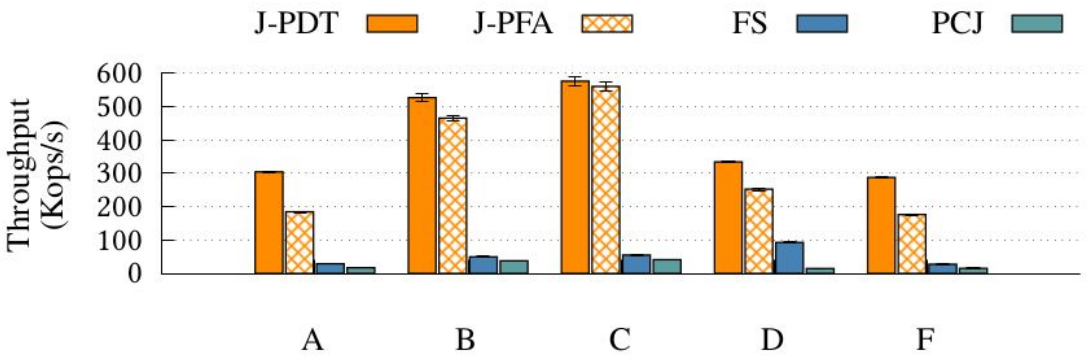
(1) Introduction
- data persistence
- persistent memory
- NVMM
- why Java?
- prior art
- contribution overview
- demo

(2) System Design of J-NVM
- key idea
- programming model
  - persistent objects
  - code generator
- J-PFA
- J-PDT

(3) Evaluation
- YCSB benchmark
- recovery

(4) Conclusion

YCSB Benchmark



Durable backends for Infinispan:

- PCJ = HashMap from Persistent Collections Java (JNI + PMDK)
- FS: ext4-dax

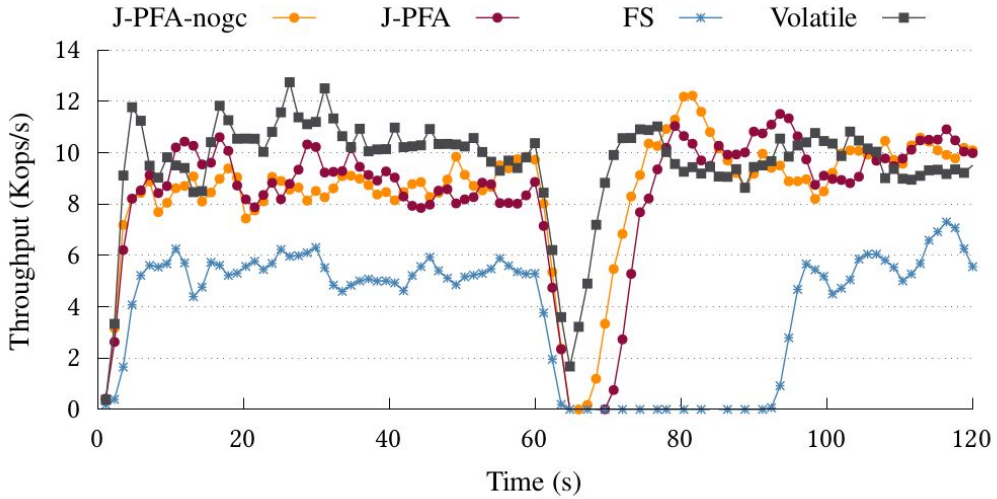
Hardware used:

4 Intel CLX 6230 HT 80-core
128GB DDR4,
4x128GB Optane (gen1)

Takeaways:

- J-NVM up to *10.5x* (resp. *22.7x)* than FS (resp. PCJ)
- no need for volatile cache

Recovery



TPC-B like benchmark
10M accounts (140 B each)
client-server setting
SIGKILL after 1 min

Takeaways:

- J-NVM is more than *5x* faster to recover than FS
- no-need for graph traversal in some cases (e.g., only FA blocks)

Conclusion

Contribution = J-NVM: off-heap persistent objects

Each persistent object is composed of
- *a persistent data structure*: unmanaged, allocated off-heap (NVMM)
- *a proxy*: managed, allocated on-heap (DRAM)

Pros:
- unique data representation (no data marshalling)
- recovery-time GC (not at runtime, does not scale)
- consistently faster than external designs (JNI, FS)

+ automagic tool
+ library ~ no runtime changes

Cons:
- explicit free *but* common for durable data
- limited code re-use *but* safer programming model

Conclusion

Related Work

- Persistence History
- Early PMEM
- NVMM challenges
- NVMM Programming abstractions
- Persistence in Java
- NVMM in Java

Big summary table p.100:

Synoptic view
& J-NVM positioning

	Support		Failure-atomicity			Applicability			Efficiency					
	Model type	Language	Impl.	Granu.	Persistency	Static types	FASE	PDT	Dual rep.	Marsh.	NVM perf.	Heap mgmt	Heap size	Recovery
***File systems* (§2.5.2)**														
Ext4-DAX [2]	file system	any	user	page	imm.	no	no	no	yes	yes	− − −	N/A	N/A	− − −
NOVA [331]	file system	any	CoW	page	imm./buff.	no	no	no	yes	yes	−−	N/A	N/A	− − −
Mashona [156]	library	Java (native)	append	word	imm.	no	no	no	yes	yes	−	N/A	N/A	− − −
***Persistent data type* (§2.7.1)**														
SOFT [353]	data type	C++	log-free	op.	d. lin.	yes	no	yes	no	no	+ + +	+ + +	+ + +	++
***General-purpose constructions* (§2.7.2)**														
Capsules [57]	method	C++	dual-copy+CAS	word	detect. rec.	no	no	user	yes	no	++	N/A	N/A	+ + +
RECIPE [207]	method	C++	log-free	op.	imm.	no	no	user	no	no	++	N/A	N/A	+ + +
PMwCAS [314]	method+library	C++	log-free	word	imm.+linea.	no	no	user	no	no	++	N/A	N/A	+ + +
MOD [158]	method+data types	C++	shadow pag.+CAS	op.	imm.	no	no	yes	no	no	++	N/A	N/A	+ + +
NVTraverse [139]	method+library	C++	log-free	op.	d. lin.	no	no	user	no	no	++	N/A	N/A	+ + +
PRONTO [231]	library	C++	sem. log+chkpt.	op.	d. lin.	no	yes	no	yes	no	++	N/A	N/A	−
CX-PUC [103]	library	C++	2N replicas+CAS	object	d. lin.	yes	yes	no	no	no	+	++	++	+ + +
Montage [318]	library	C++	CoW+epoch reclam.	payload[1]	buff. d. lin.	yes	yes	no	no	no	+ + +	+ + +	+ + +	++
Mirror [140]	library	C++	log-free atomics	word	d. lin.	yes	no	yes	yes	no	+ + +	+ + +	+ + +	+
PREP-UC [93]	library	C++	2 replicas+ sem. log	cache[2]	(buff.) d. lin.	yes	yes	no	yes	no	++	+ + +	++	++
Tracking [52]	method	C++	log-free	op.	detect. rec.	no	no	user	no	no	++	N/A	N/A	+ + +
PCOMB [130]	method	unspe.	CoW+CAS	object	detect. rec.	no	no	user	no	no	+ + +	++	+ + +	+ + +
ResPCT [192]	library	C	InCLL(undo+epoID)	epoch	buff. d. lin.	yes	yes	no	no	no	+ + ++	++	++	+ + +
***Persistent transactional memory* (§2.7.3)**														
Romulus [102]	library	C++	dual-copy+vola. redo	word	d. lin.	yes	yes	no	no	no	+	+ + +	++	−−
OneFile [266]	library	C++	redo	word	d. lin.	yes	yes	no	no	no	+	++	+ + +	+ + +
Redo-PTM [103]	library	C++	$N+1$ replicas+CAS	word	d. lin.	yes	yes	no	no	no	+	++	++	+ + +
Crafty [142]	library	C	undo	word	buff.	no	yes	no	no	no	N/A	++	+ + +	+ + +
***Persistent heap managers* (§2.7.5, §2.8.3, §2.8.4)**														
Mnemosyne [309]	library+compiler	C or C++	redo	word	imm./buff.	no	yes	no	no	no	+	++	+ + +	+ + +
Atlas [83]	library+compiler	C	undo	word	buff.	no	yes	no	no	no	+	++	+ + +	+ + +
PMDK [19]	library	C & C++	undo	object	imm.	yes	yes	no	no	no	+	+	+ + +	+ + +
NVthreads [167]	library	C	CoW+redo	page	buff.	no	yes	no	no	no	N/A	++	+ + +	++
NV-heaps [92]	library	C++	undo	object	d. lin.	yes	yes	no	no	no	N/A	++	+ + +	+ + +
Makalu [63]	library (allocator)	C	undo	16B	imm.	no	no	no	no	no	+	++	+ + +	++
Ralloc [76]	library (allocator)	C++	user	8B	recoverable	no	no	no	no	no	+ + +	+ + +	+ + +	++
PCJ [16]	library	Java (JNI)	undo	object	imm.	yes	yes	yes	yes	yes	− − −	− − −	+ + +	+ + +
LLPL [20]	library	Java (JNI)	undo	object	imm.	yes	yes	no	yes	yes	− − −	− − −	+ + +	+ + +
MDS [298]	library	Java (JNI)	none	cache[2]	flush-on-fail	yes	yes	no	no	yes	−−	++	++	+ + +
Espresso [328]	compiler+runtime	Java (native)	undo	word	imm.	no	yes	no	no	no	+	+	− − −	+
AutoPersist [287]	compiler+runtime	Java (native)	undo	word	imm.	no	yes	no	no	no	−	−	− − −	+
Go-PMEM [143]	compiler+runtime	Go	undo	word	imm.	no	yes	no	no	no	+	+	− − −	+
***This thesis* (§4.3, §4.5, §4.6)**														
J-NVM [209]	library	Java (native)	user	user	user	yes	yes	yes	no	no	+ + +	+ + +	+ + +	+ + +
J-PDT [209]	library	Java (native)	log-free	word	imm.	yes	yes	yes	no	no	+ + +	+ + +	+ + +	++
J-PFA [209]	library	Java (native)	redo	block	imm.	yes	yes	yes	no	no	++	+ + +	+ + +	+ + +

Table 2.4: Comparison of various NVMM programming support libraries and systems.

[1] payload: NVMM allocation payloads returned by the allocator.
[2] cache: the whole CPU cache hierarchy is invalidated and flushed to (persistent) memory.

Future work

- Data interoperability
- Test tools
- Killer App:
  - NVMM use for serverless apps (FaaS)

Outline

(1) Introduction
- data persistence
- persistent memory
- NVMM
- why Java?
- prior art
- contribution overview

(2) System Design of J-NVM
- demo
- key idea
- programming model
  - persistent objects
  - code generator
- J-PFA
- J-PDT

(3) Evaluation
- YCSB benchmark
- recovery

(4) Conclusion