J-NVM: Off-Heap Persistent Objects in Java

Anatole Lefort, Yohan Pipereau, Kwabena Amponsem,
Pierre Sutra, Gaël Thomas

Télécom SudParis
Institut Polytechnique de Paris

Published at SOSP’21

Efficient Support for **Persistent Memory** in **Java**

J-NVM: Off-Heap Persistent Objects in Java

Anatole Lefort, Yohan Pipereau, Kwabena Amponsem,
Pierre Sutra, Gaël Thomas

Télécom SudParis
Institut Polytechnique de Paris

Published at SOSP'21

Efficient Support for **Persistent Memory** in **Java**

J-NVM: Off-Heap Persistent Objects in Java

Anatole Lefort, Yohan Pipereau, Kwabena Amponsem,
Pierre Sutra, Gaël Thomas

Télécom SudParis
Institut Polytechnique de Paris

Published at SOSP'21

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

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

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

e.g., **program variables**

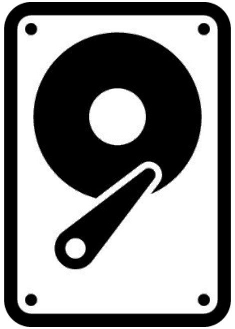
- **Persistent:**
  - **Extended Lifetime:** recalled and reused in subsequent executions.
    - remain consistent even in the wake of a failure.
  - hosted on Storage Devices

e.g., **files' content**

Data Persistence

Common media: **HDD, Flash disks, Tape**

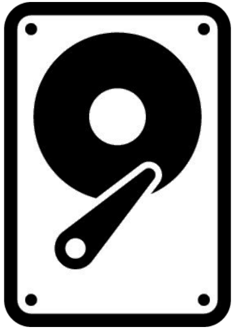
- **Durable:** resist reboots, power loss
- **Large capacity:** at least TBs
- **Slow access latency:** 200μs-15ms

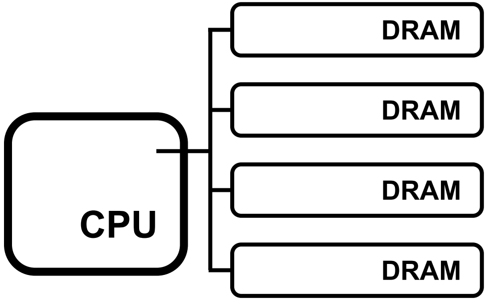


Data Persistence

Common media: **HDD, Flash disks, Tape**

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





CPU

DRAM

DRAM

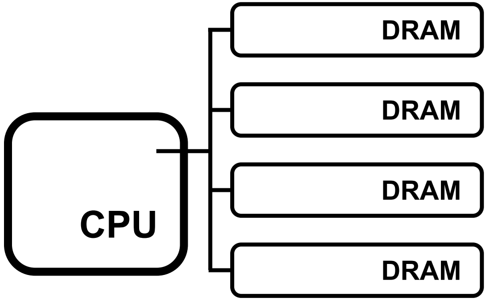
DRAM

DRAM

Data Persistence

Common media: **HDD, Flash disks, Tape**

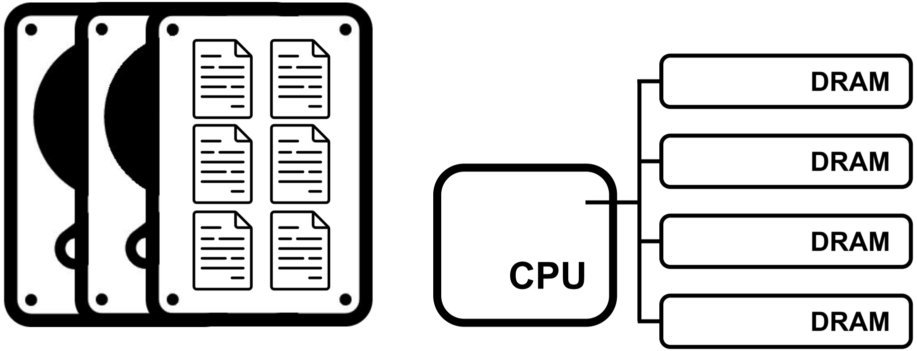
- **Durable:** resist reboots, power loss
- **Large capacity:** at least TBs
- **Slow access latency:** 200µs-15ms



Data Persistence

Common media: **HDD, Flash disks, Tape**

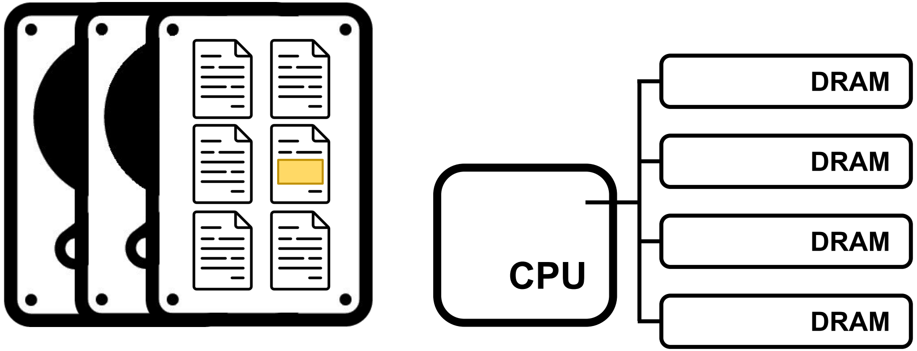
- **Durable:** resist reboots, power loss
- **Large capacity:** at least TBs
- **Slow access latency:** 200µs-15ms



Data Persistence

Common media: **HDD, Flash disks, Tape**

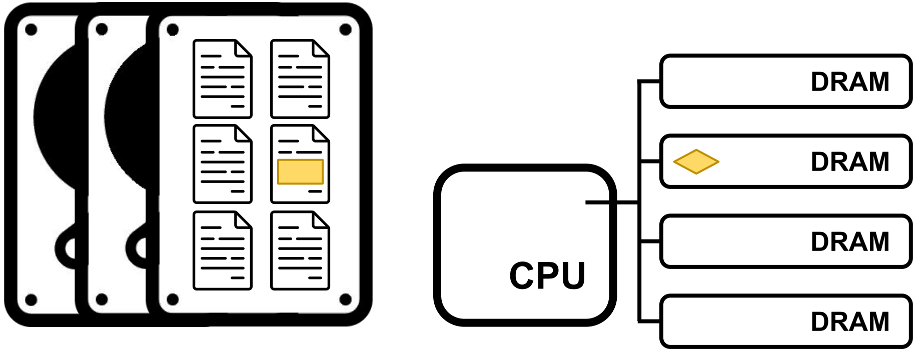
- **Durable:** resist reboots, power loss
- **Large capacity:** at least TBs
- **Slow access latency:** 200µs-15ms



Data Persistence

Common media: **HDD, Flash disks, Tape**

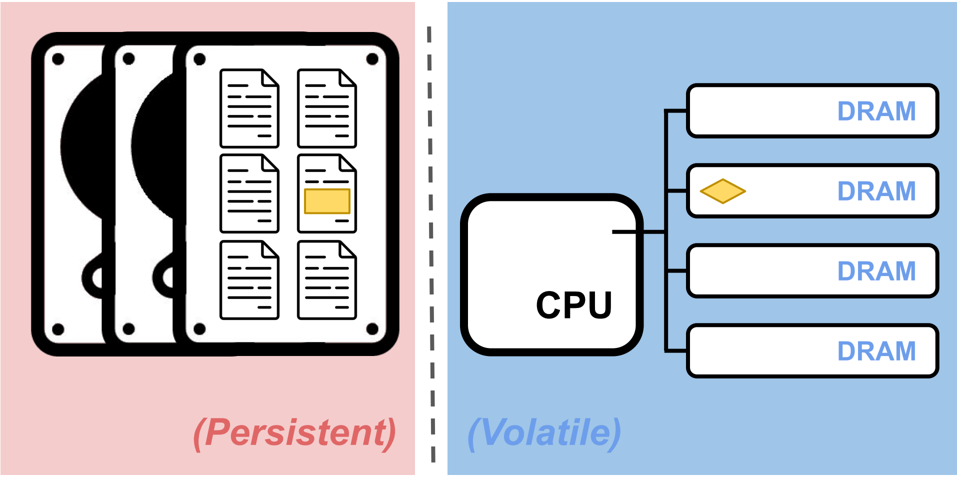
- **Durable:** resist reboots, power loss
- **Large capacity:** at least TBs
- **Slow access latency:** 200µs-15ms



Data Persistence

Common media: **HDD, Flash disks, Tape**

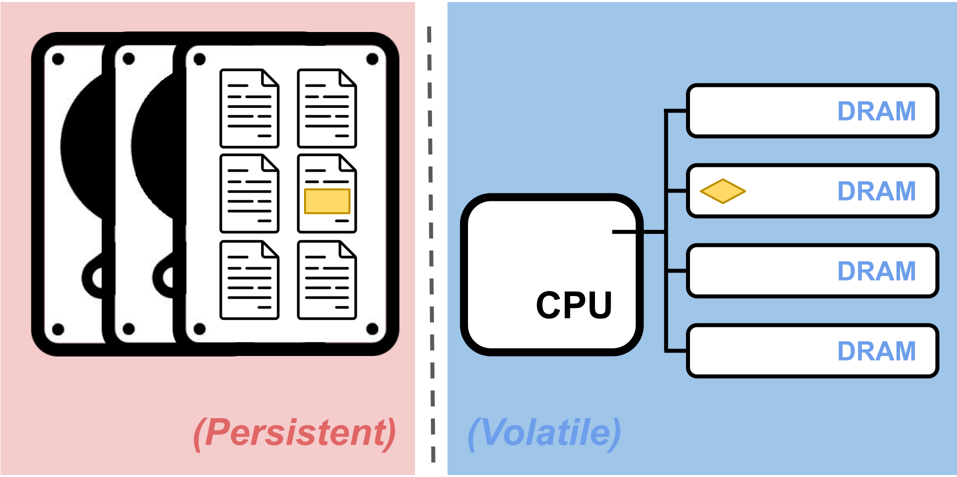
- **Durable:** resist reboots, power loss
- **Large capacity:** at least TBs
- **Slow access latency:** 200µs-15ms



Data Persistence

Common media: **HDD, Flash disks, Tape**

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

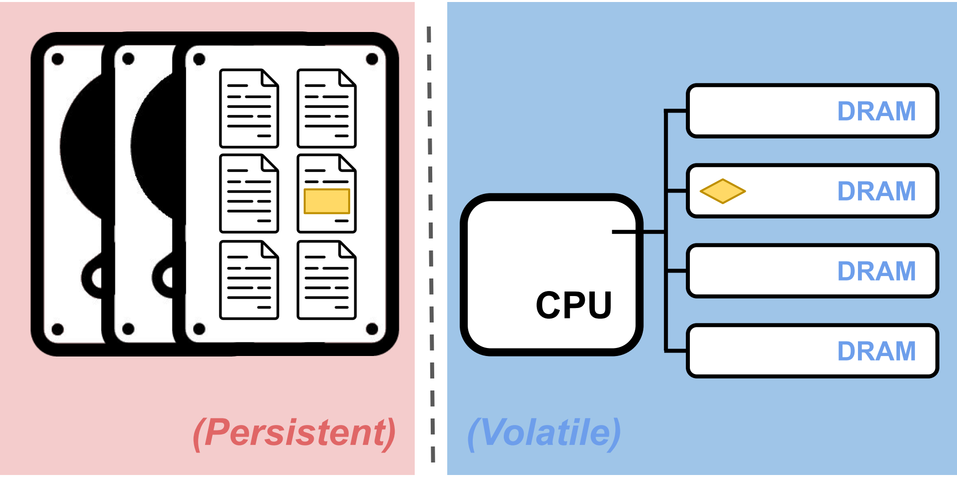


1 - Dual data representation
2 - Expensive I/Os

Data Persistence

Common media: **HDD, Flash disks, Tape**

- **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 System Software

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

Intel Optane PMEM, 2019

High-performance

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

2-3x slower than **DRAM**
3 order of magnitude faster than **SSD**

Non-volatile main memory *(NVMM)*

new persistent

Durable

resists reb

High-density

smallest D

up to 8x D

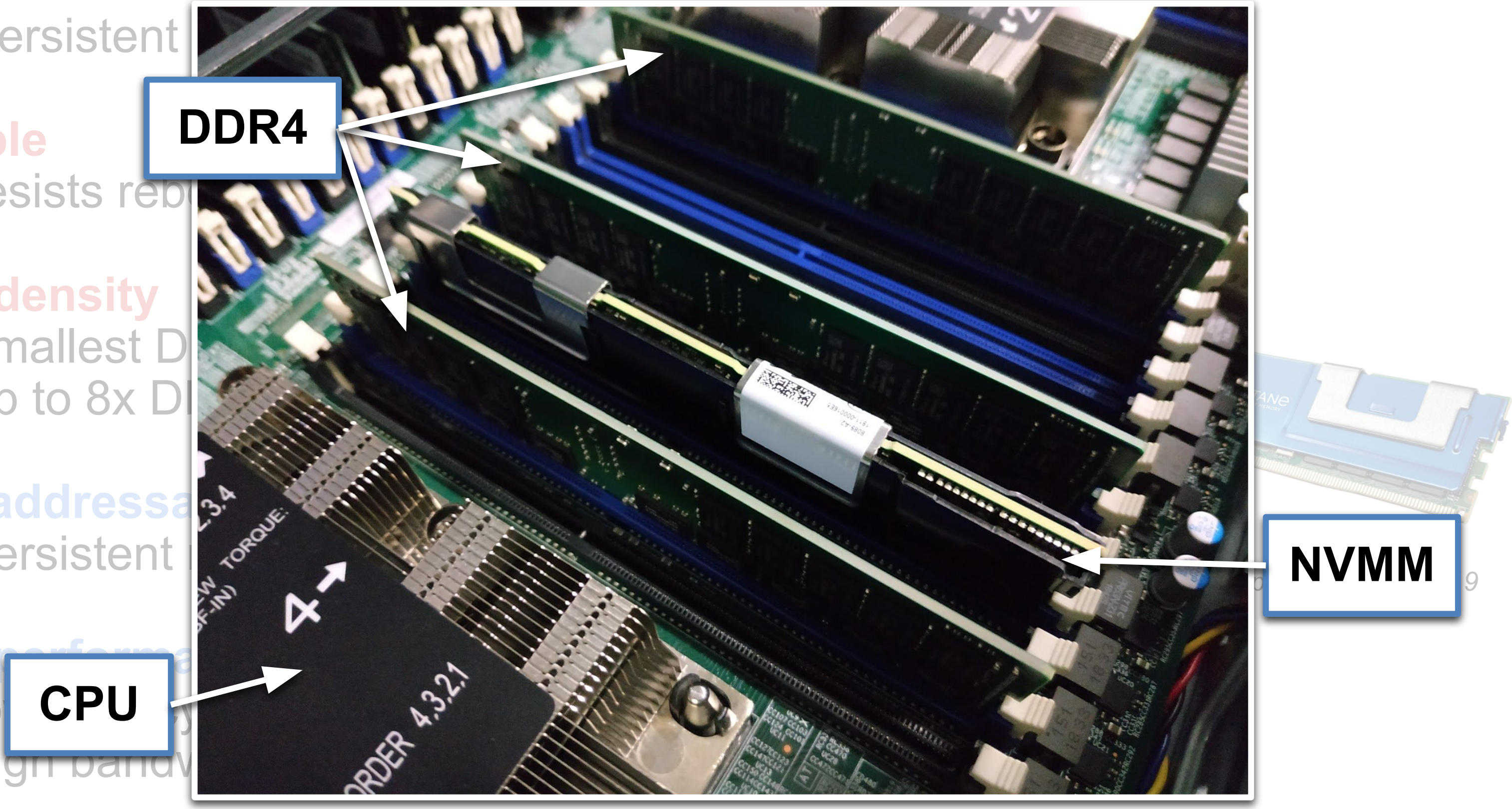
Byte addressa

persistent

High-

lo

high bandv



Non-volatile main memory *(NVMM)*

new persistent

Durable

resists rebo

High-density

smallest DI

up to 8x DD

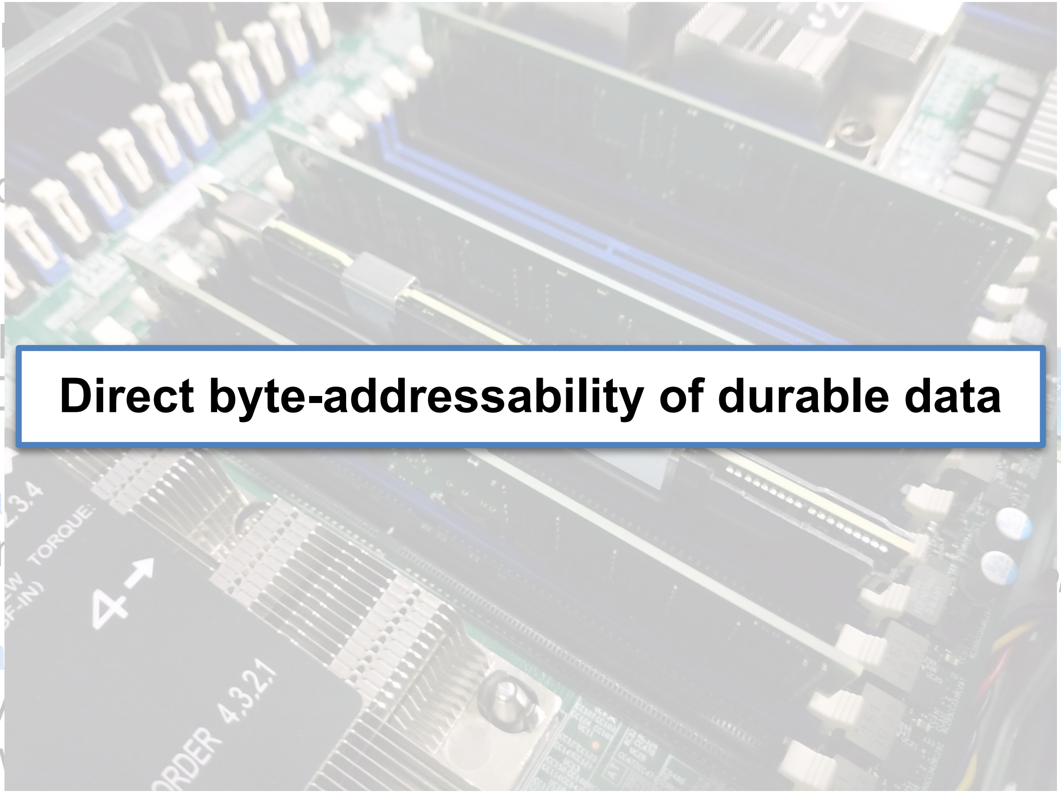
Byte addressa

persistent r

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

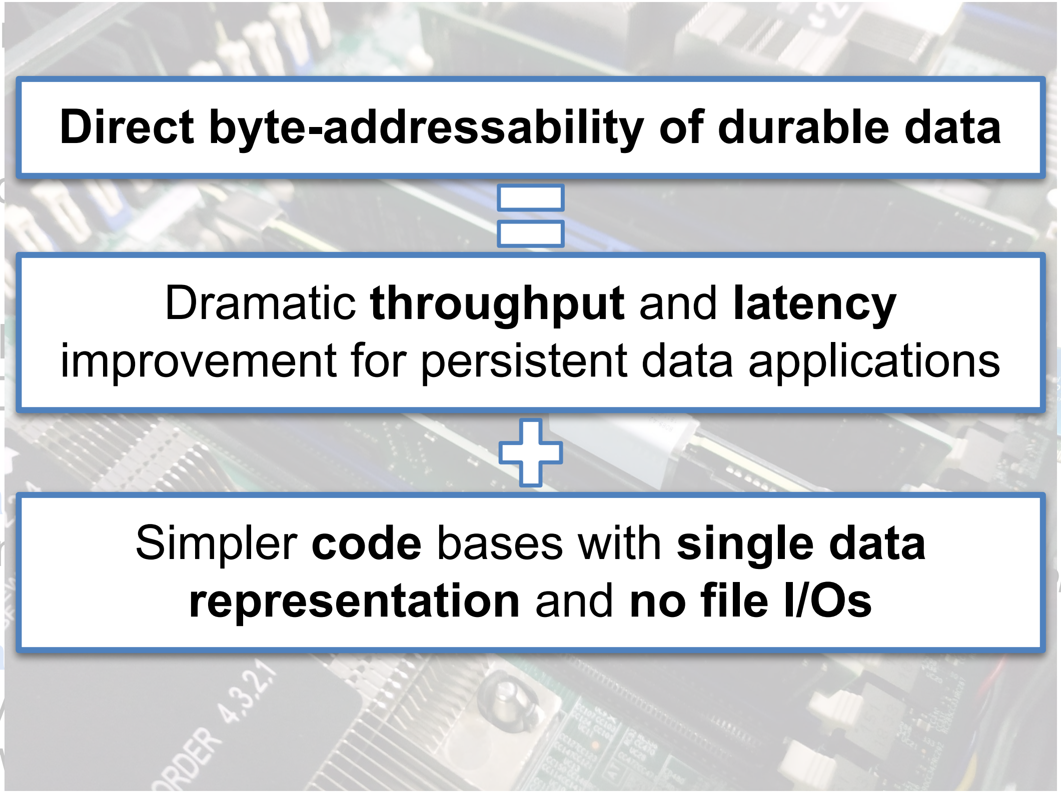
smallest DI
up to 8x DD

Byte addressa

persistent r

High-performa

low latency
high bandw



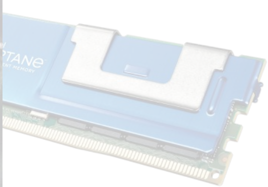
Direct byte-addressability of durable data

=

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

+

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



ptane PMEM, 2019

Persistent Memory *(PMEM)*

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

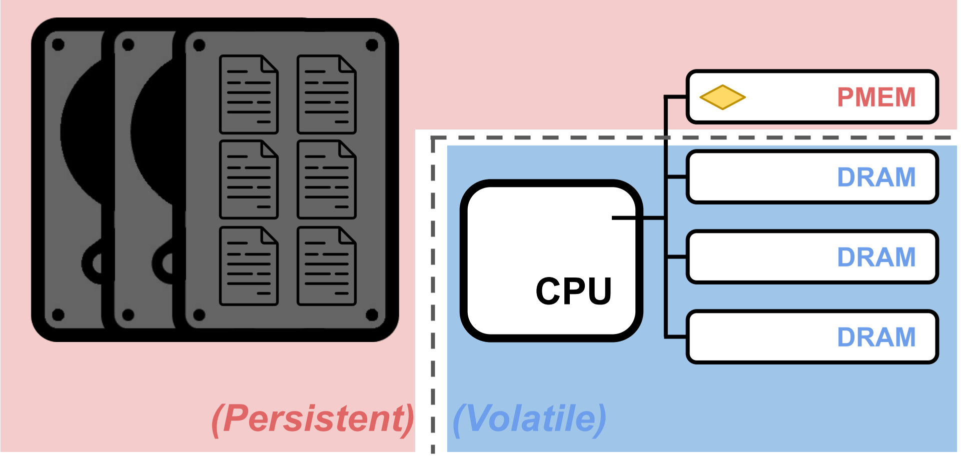
What changes with PMEM ?

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

NVMM - *Usage*

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

(1) File system interface

open/close/read/write/sync

⇒ “SSD emulation”

(2) Direct memory access

mmap

⇒ memory-mapped file

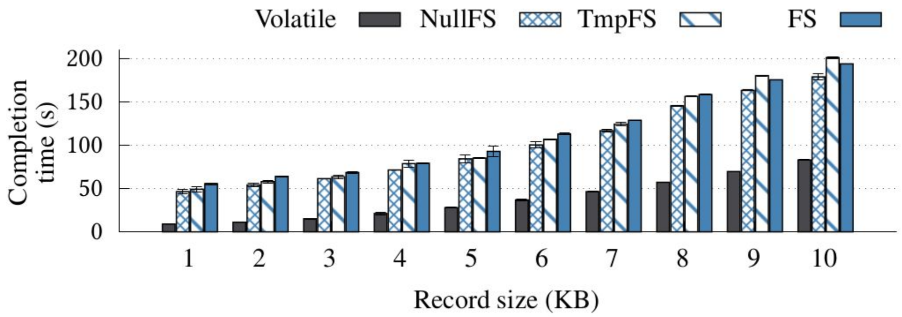
Intel Optane PMEM, 2019

NVMM - *Usage*

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

(1) File system interface

open/close/read/write/sync



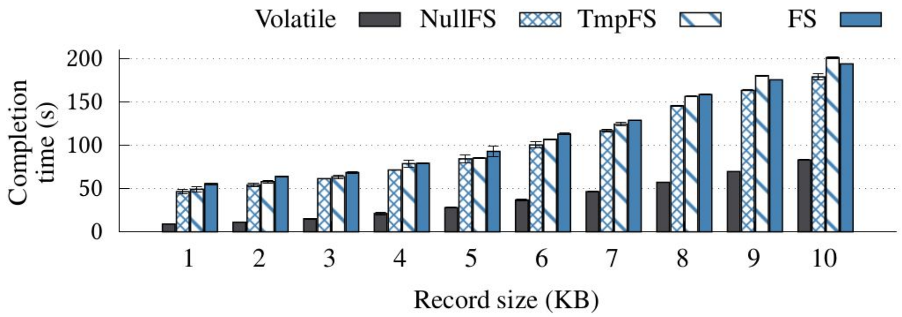
Varying record size in YCSB-F.

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*

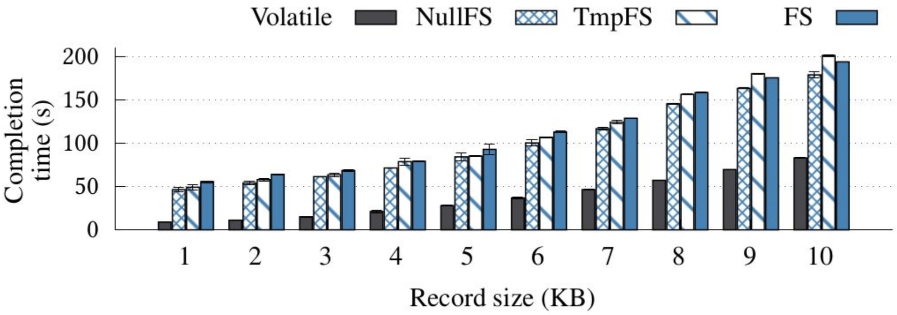
How do we use it ? Storage device compatibility mode (1) ?

(1) File system interface

open/close/read/write/sync

Software Bottlenecks:

- dual representation (consistency)
- cost of marshalling



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 ? Persistent Memory **(2)** ?*

(2) Direct memory access

mmap

+ memory **load/store** CPU instructions

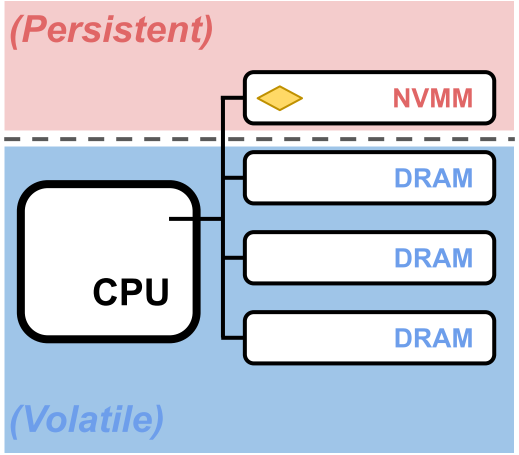
NVMM - *Usage*

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

(2) Direct memory access

mmap

+ memory **load/store** CPU instructions



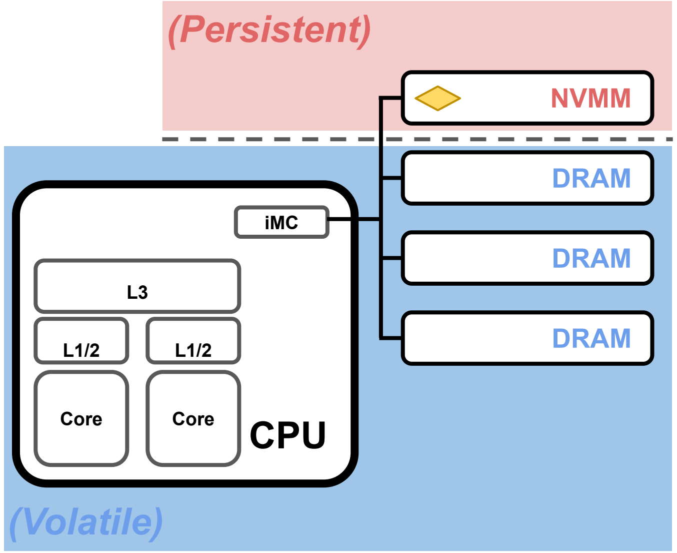
NVMM - *Usage*

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) ?

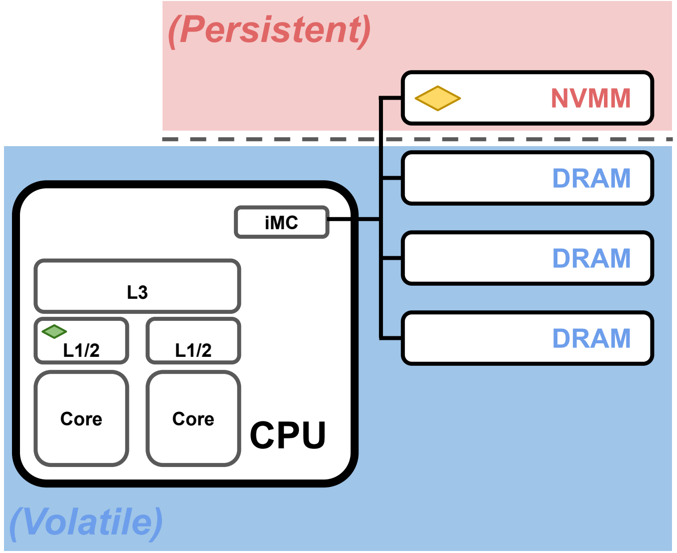
(2) Direct memory access

mmap

+ memory **load/store** CPU instructions

+ special **flush/fence** CPU instructions

(*manually control cache line eviction order*)



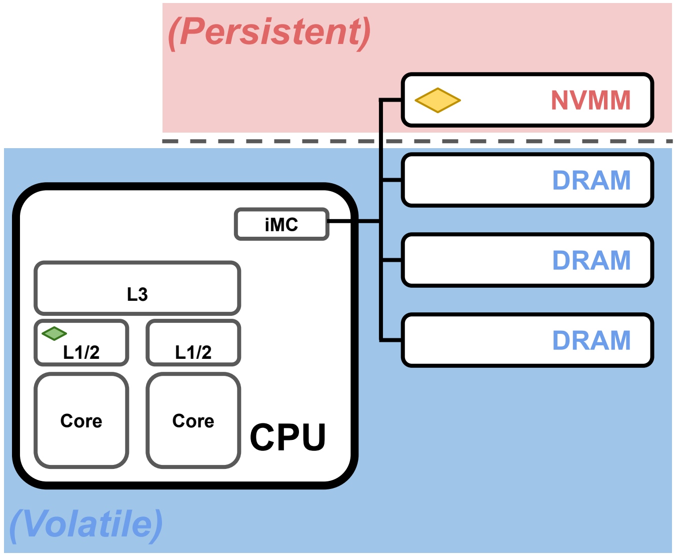
NVMM - *Usage*

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

(2) Direct memory access

mmap

+ memory **load/store** CPU instructions



+ 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 <u>native code</u> (*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

Espresso, go-pmem: ***pnew***

```
String a = new String(“toto”);
String b = pnew String(“titi”);
```

AutoPersist: ***@persistentRoot***

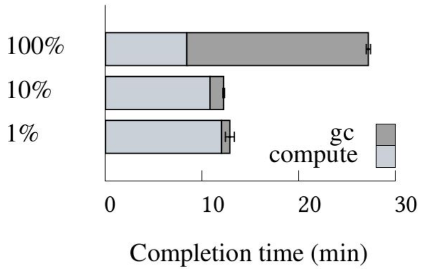
```
@persistentRoot
static List<String> root = new List<>();
…
String a = new String(“toto”);
String b = new String(“titi”);
root.add(b);
```

Prior art: *internal design*

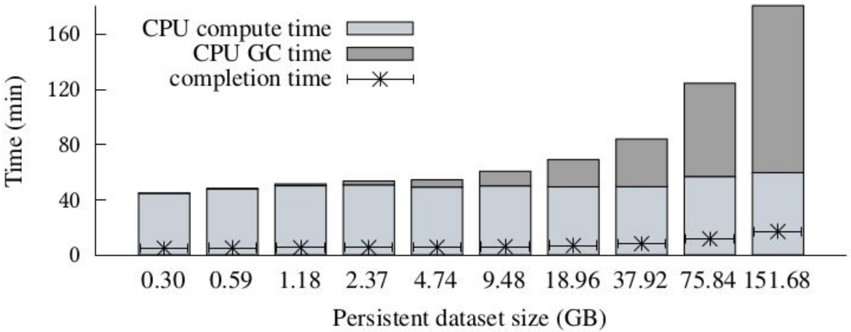
= [Espresso, AutoPersist, go-pmem]

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]

Features

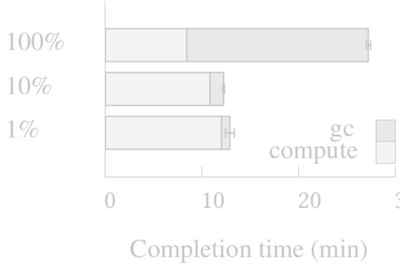
- Garbage collectors do not scale to large persistent datasets

managed persistent objects

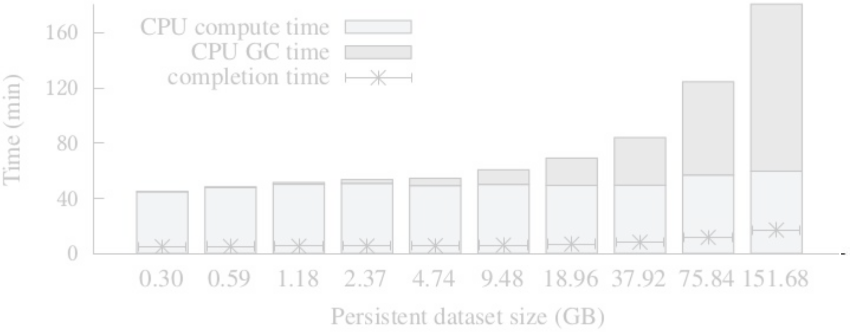
non-scalable

heavily-modified JVM

= 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]

Features

managed persistent objects

non-scalable

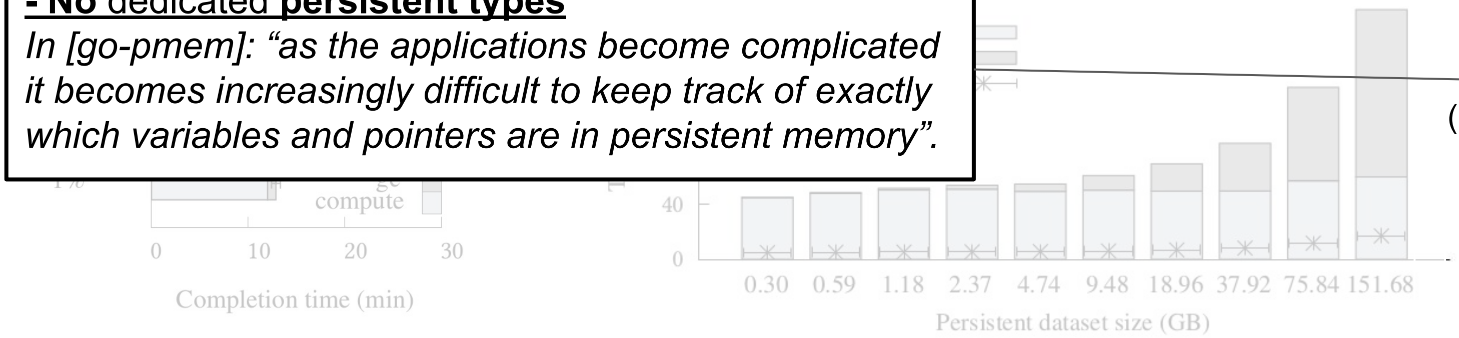
heavily-modified JVM

orthogonal persistence
(pnew, @persistentRoot)

- 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".



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*

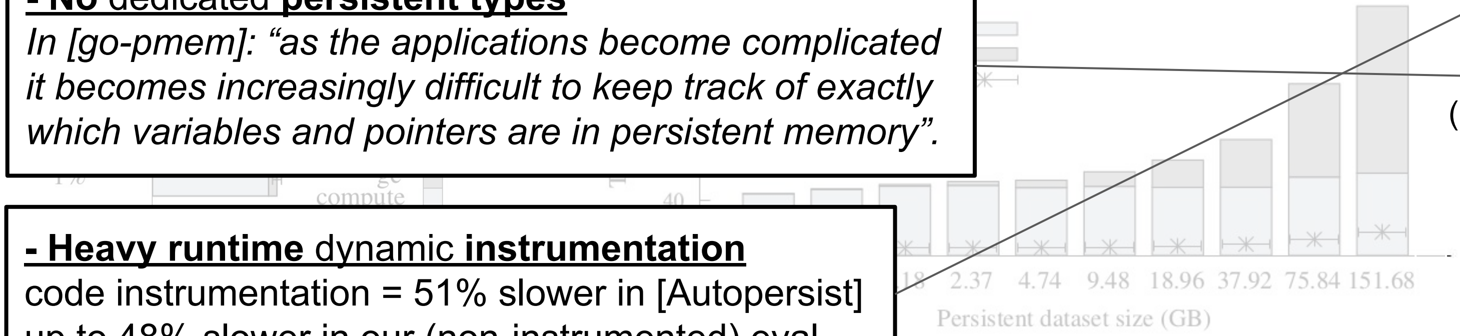
= [Espresso, AutoPersist, go-pmem]

- 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.



dataset

(YCSB-F, Infinispan with 80GB dataset)

(YCSB-F, go-pmem)

GC cost outweighs the benefits of large DRAM caches

Features

managed persistent objects

non-scalable

heavily-modified JVM

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

failure-atomic blocks

Outline

(1) Introduction

- data persistence
- persistent memory
- NVMM
- why Java?
- prior art

(2) System Design of J-NVM

- contribution overview
- ~~demo~~
- key idea
- programming model
  - persistent objects
  - code generator
- J-PFA
- J-PDT

(3) Evaluation

- YCSB benchmark
- recovery

(4) Conclusion

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()

Overview

J-NVM = Off-Heap Persistent Objects

A Java Library for PMEM

- novel persistent objects + off-heap crash-consistent memory management
- **code generator:** automatic decoupling for POJOs
- **J-PFA:** automatic failure-atomic code
- **J-PDT:** data types + collections for persistent memory

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*

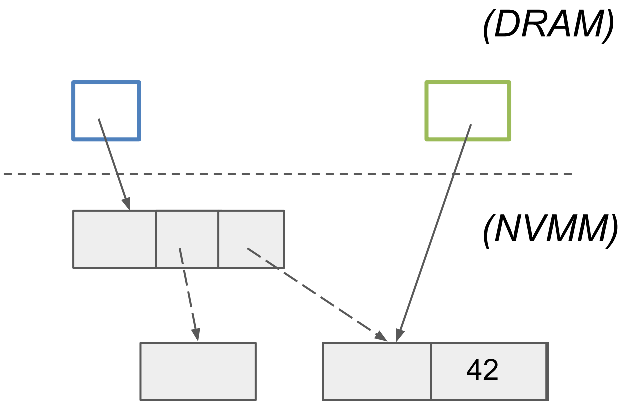
Persistent object is

- *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

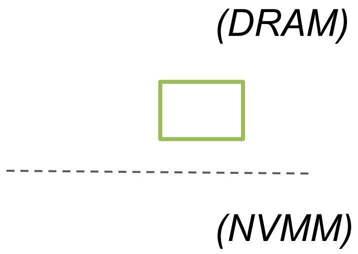
Re-Constructor

- 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

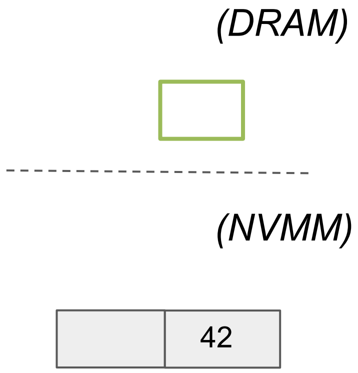
Re-Constructor

- 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

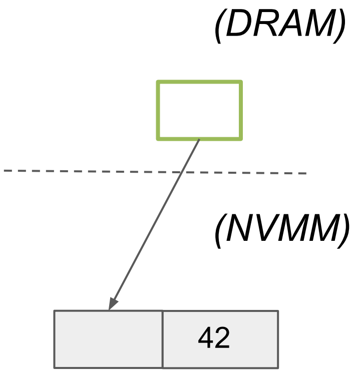
Re-Constructor

- 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

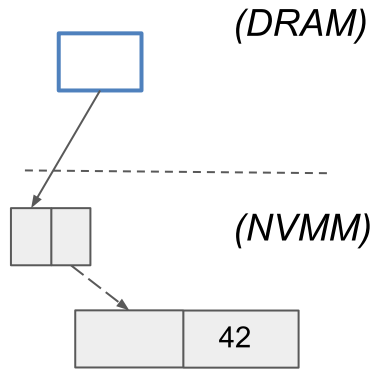
Re-Constructor

- 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

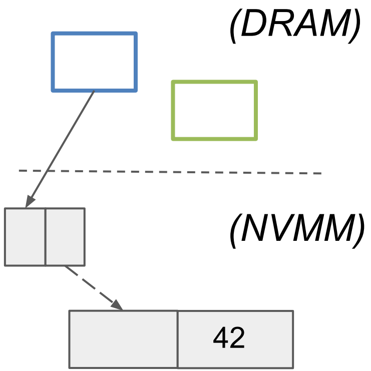
Re-Constructor

- 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*

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

Constructor

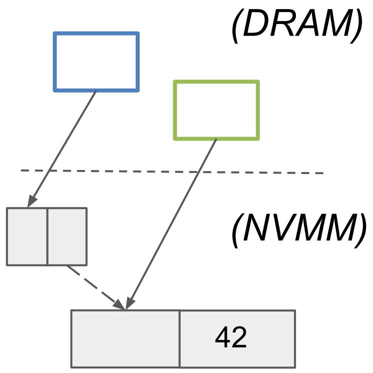
- allocate NVMM
- attach persistent data structure

Re-Constructor

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

Destructor

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



Programming model - *life cycle*

Constructor

- allocate NVMM
- attach persistent data structure

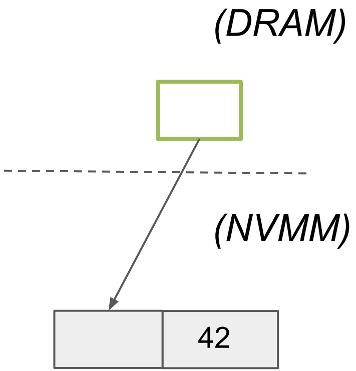
Re-Constructor

- 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

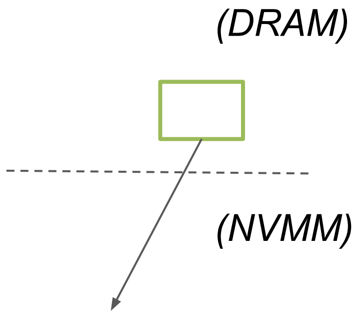
Re-Constructor

- 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

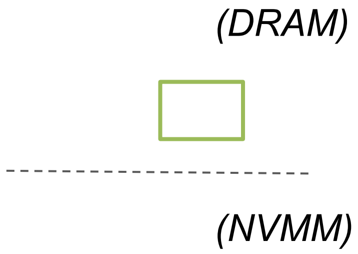
Re-Constructor

- 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

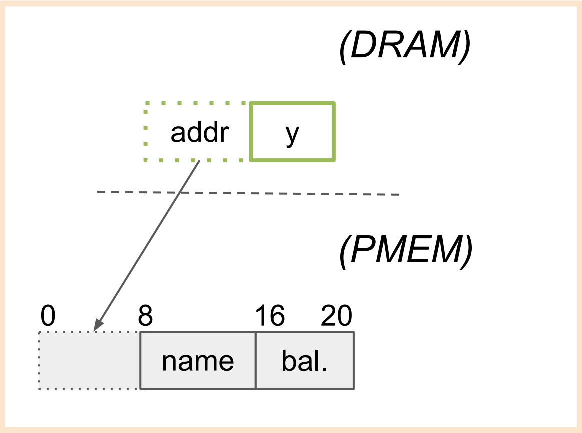
(1) compute class-wide ***off-heap layout***
(2) replace (non-transient) **field accesses**
(3) generate ***constructor***, ***re-constructor***
(4) ***FA-wrap*** non-private methods

```
@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*

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

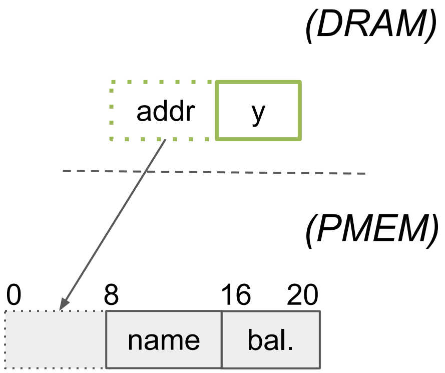


```
@Persistent(fa="non-private")
class Account {
  PString name;
  int balance;
  transient int y;

  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***
  (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*

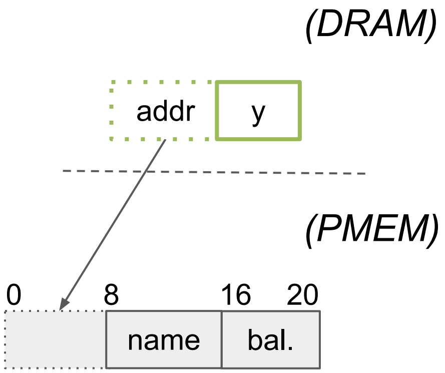
Goals

(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 (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;
  }
…
```

```
// 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);
  }
…
```

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*

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.name = new PString(id);
    this.balance = balance;
  }

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

```
// 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);
  }
…
```

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();
  }
…
```

J-PFA

Automatic crash-consistent update

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

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

Log new, free and updates

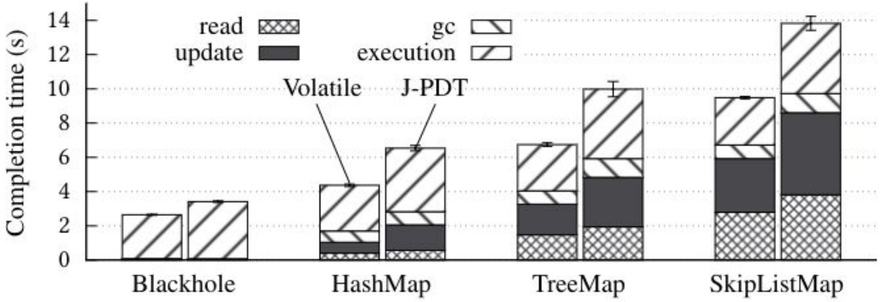
granularity = a block of PMEM

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

J-PDT

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

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



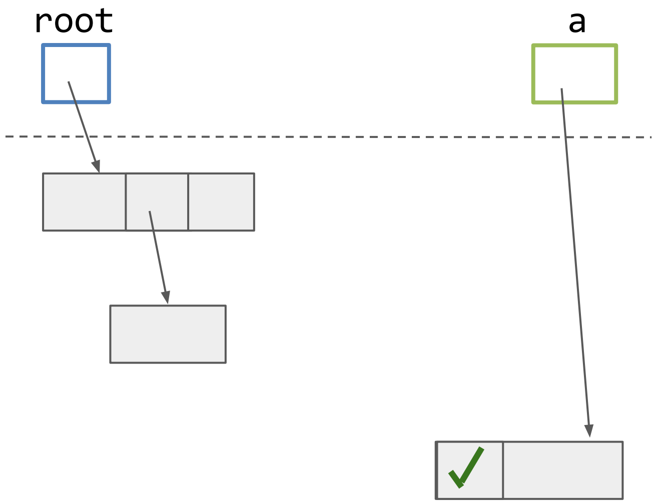
Persistent vs Volatile data types *(YCSB-A)*

Takeaway:

- around 50% slower than volatile data types on DRAM

Low-level interface

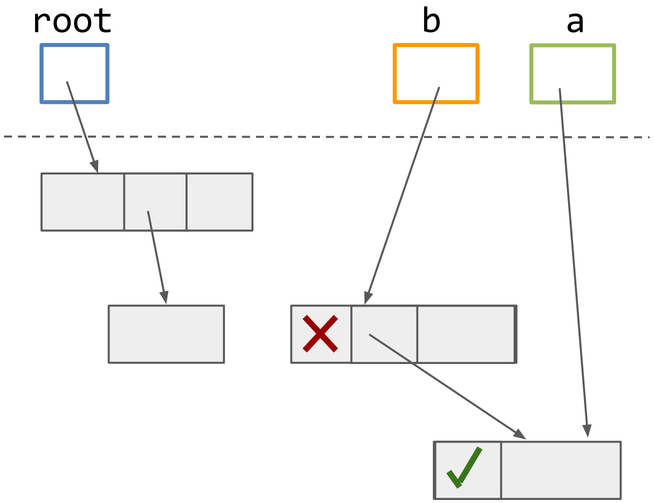
- unsafe.{pwb,pfence, psync}
- NVMM block allocator
- recovery time GC (à la Makalu)
- validation = 1 bit in object header
  - makes atomic reclamation easier
  - allows deferring object liveness
  - interpreted on recovery
    to reclaim reachable invalid objects



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

Low-level interface

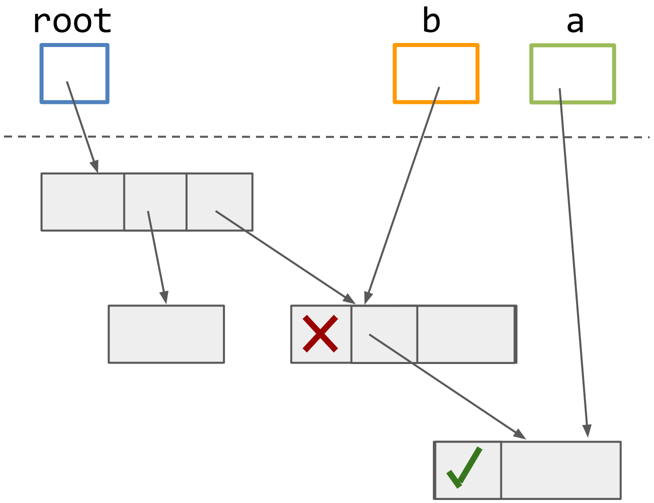
- unsafe.{pwb,pfence, psync}
- NVMM block allocator
- recovery time GC (à la Makalu)
- validation = 1 bit in object header
    - makes atomic reclamation easier
    - allows deferring object liveness
    - interpreted on recovery
      to reclaim reachable invalid objects



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

Low-level interface

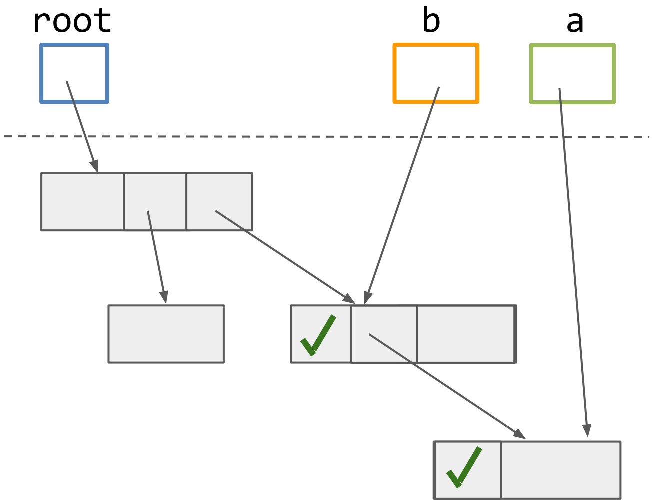
- unsafe.{pwb,pfence, psync}
- NVMM block allocator
- recovery time GC (à la Makalu)
- validation = 1 bit in object header
  - makes atomic reclamation easier
  - allows deferring object liveness
  - interpreted on recovery
    to reclaim reachable invalid objects



```
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

(2) System Design of J-NVM

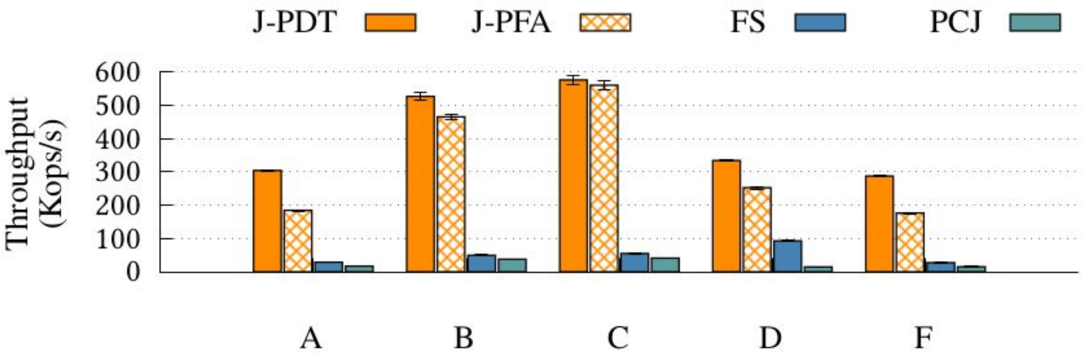
- demo
- key idea
- programming model
  - persistent objects
  - code generator
- J-PFA
- J-PDT

(3) Evaluation

- YCSB benchmark
- recovery

(4) Conclusion

YCSB Benchmark



Durable backends for Infinispan:

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

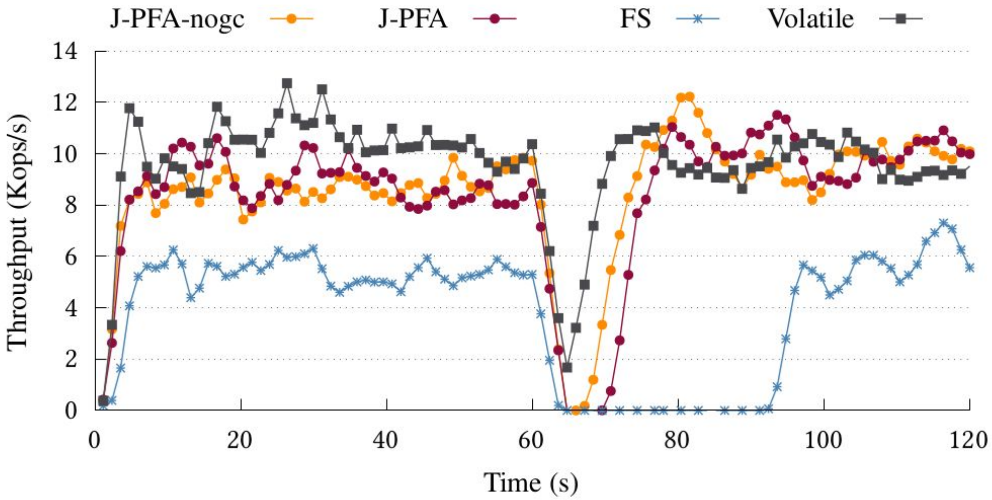
Hardware used:

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

<u>Contribution</u> = 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 *<u>but</u>* common for durable data
- limited code re-use *<u>but</u>* safer programming model

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

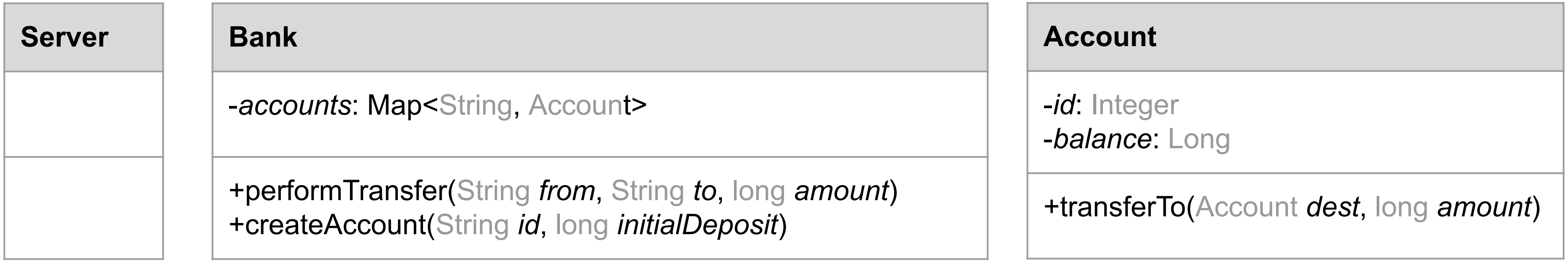
(3) Evaluation

- YCSB benchmark
- recovery

(4) Conclusion

Demo - *the application*

A Simple Bank:



Server

Bank

-*accounts*: Map<String, Account>

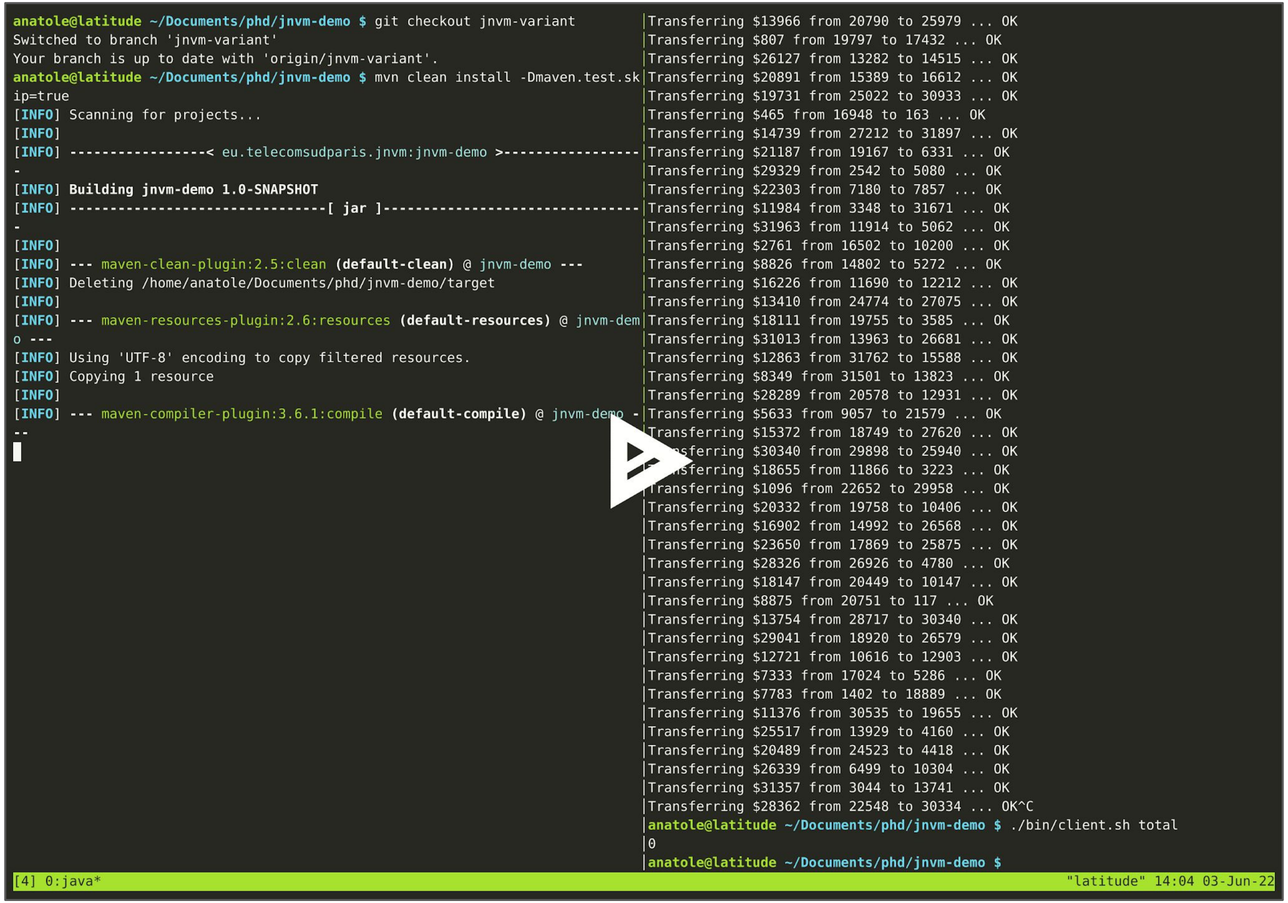
+performTransfer(String *from*, String *to*, long *amount*)
+createAccount(String *id*, long *initialDeposit*)

Account

-*id*: Integer
-*balance*: Long

+transferTo(Account *dest*, long *amount*)

Demo - *It’s showtime !*



```
anatole@latitude ~/Documents/phd/jnvm-demo $ git checkout jnvm-variant
Switched to branch 'jnvm-variant'
Your branch is up to date with 'origin/jnvm-variant'.
anatole@latitude ~/Documents/phd/jnvm-demo $ mvn clean install -Dmaven.test.sk
ip=true
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------< eu.telecomsudparis.jnvm:jnvm-demo >----------------
-
[INFO] Building jnvm-demo 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]--------------------------------
-
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ jnvm-demo ---
[INFO] Deleting /home/anatole/Documents/phd/jnvm-demo/target
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ jnvm-dem
o ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.6.1:compile (default-compile) @ jnvm-demo -
--
```

```
Transferring $13966 from 20790 to 25979 ... OK
Transferring $807 from 19797 to 17432 ... OK
Transferring $26127 from 13282 to 14515 ... OK
Transferring $20891 from 15389 to 16612 ... OK
Transferring $19731 from 25022 to 30933 ... OK
Transferring $465 from 16948 to 163 ... OK
Transferring $14739 from 27212 to 31897 ... OK
Transferring $21187 from 19167 to 6331 ... OK
Transferring $29329 from 2542 to 5080 ... OK
Transferring $22303 from 7180 to 7857 ... OK
Transferring $11984 from 3348 to 31671 ... OK
Transferring $31963 from 11914 to 5062 ... OK
Transferring $2761 from 16502 to 10200 ... OK
Transferring $8826 from 14802 to 5272 ... OK
Transferring $16226 from 11690 to 12212 ... OK
Transferring $13410 from 24774 to 27075 ... OK
Transferring $18111 from 19755 to 3585 ... OK
Transferring $31013 from 13963 to 26681 ... OK
Transferring $12863 from 31762 to 15588 ... OK
Transferring $8349 from 31501 to 13823 ... OK
Transferring $28289 from 20578 to 12931 ... OK
Transferring $5633 from 9057 to 21579 ... OK
Transferring $15372 from 18749 to 27620 ... OK
Transferring $30340 from 29898 to 25940 ... OK
Transferring $18655 from 11866 to 3223 ... OK
Transferring $1096 from 22652 to 29958 ... OK
Transferring $20332 from 19758 to 10406 ... OK
Transferring $16902 from 14992 to 26568 ... OK
Transferring $23650 from 17869 to 25875 ... OK
Transferring $28326 from 26926 to 4780 ... OK
Transferring $18147 from 20449 to 10147 ... OK
Transferring $8875 from 20751 to 117 ... OK
Transferring $13754 from 28717 to 30340 ... OK
Transferring $29041 from 18920 to 26579 ... OK
Transferring $12721 from 10616 to 12903 ... OK
Transferring $7333 from 17024 to 5286 ... OK
Transferring $7783 from 1402 to 18889 ... OK
Transferring $11376 from 30535 to 19655 ... OK
Transferring $25517 from 13929 to 4160 ... OK
Transferring $20489 from 24523 to 4418 ... OK
Transferring $26339 from 6499 to 10304 ... OK
Transferring $31357 from 3044 to 13741 ... OK
Transferring $28362 from 22548 to 30334 ... OK^C
anatole@latitude ~/Documents/phd/jnvm-demo $ ./bin/client.sh total
0
anatole@latitude ~/Documents/phd/jnvm-demo $
```

```
[4] 0:java*                                                                              "latitude" 14:04 03-Jun-22
```