J-NVM: Off-Heap Persistent Objects in Java

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

Télécom SudParis
Institut Polytechnique de Paris

The 28th ACM Symposium on Operating Systems Principles
October 26-29, 2021

Non-volatile main memory

new persistent medium (in-between SSD and DRAM)

Durable

resists reboots, power loss

High-density

smallest DIMM = 128 GB

Byte addressable

persistent memory abstraction

High-performance

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

Anatole Lefort et al., J-NVM, SOSP'21.

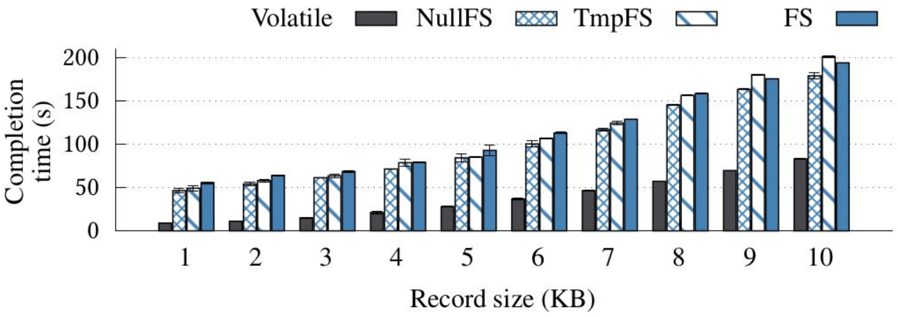
Why Java?

Many data stores & processing frameworks

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

Lack of *efficient* interfaces

- FS/ext4-dax
  - almost as slow as tmpfs
  - dual representation (consistency)
  - cost of marshalling
- PCJ (JNI+PMDK)
  - slower than FS on YCSB benchmark



Varying record size in YCSB-F.

Problematic: Java-native NVMM interface

Anatole Lefort et al., J-NVM, SOSP'21.

Prior works: *internal design*

= [Espresso, AutoPersist, go-pmem]

Challenges	→	**Features**
single data representation		*managed* persistent objects
programming model		orthogonal persistence (`pnew`, `@persistentRoot`)
direct access to NVMM		heavily-modified runtime
durability abstraction		failure-atomic blocks
scalability (large persistent dataset)		?

Anatole Lefort et al., J-NVM, SOSP'21.

Prior works: *internal design*

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

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

heavily-modified runtime

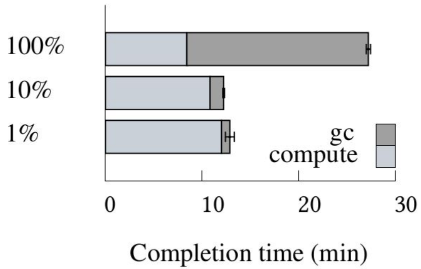
failure-atomic blocks

scalability?

Anatole Lefort et al., J-NVM, SOSP'21.

Prior works: *internal design*

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”.



Varying cache ratio (YCSB-F)

Features

managed persistent objects

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

heavily-modified runtime

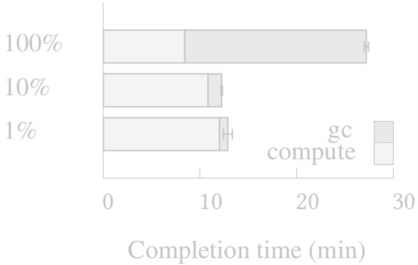
failure-atomic blocks

scalability?

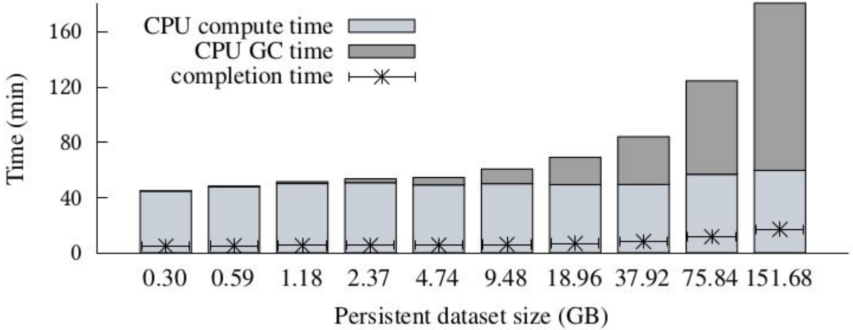
Anatole Lefort et al., J-NVM, SOSP’21.

Prior works: *internal design*

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".



Varying cache ratio (YCSB-F)



Increasing dataset (YCSB-F, go-pmem)

Features

managed persistent objects

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

heavily-modified runtime

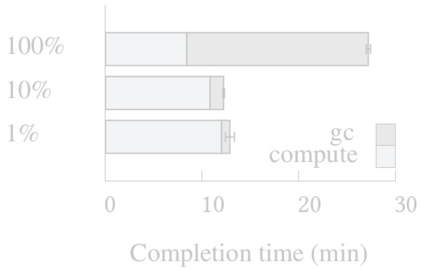
failure-atomic blocks

non-scalable

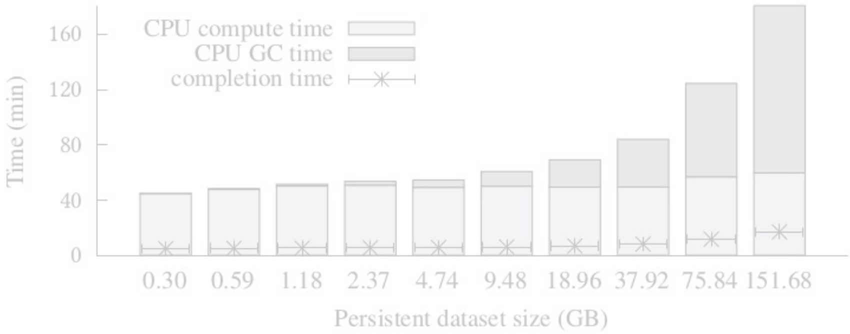
Anatole Lefort et al., J-NVM, SOSP'21.

Prior works: *internal design*

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".



Varying cache ratio (YCSB-F)



Increasing dataset (YCSB-F, go-pmem)

Features

managed persistent objects

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

heavily-modified runtime

failure-atomic blocks

non-scalable

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

Anatole Lefort et al., J-NVM, SOSP'21.

Outline

Introduction

- NVMM
- why Java?
- prior works

System Design

- overview
- programming model
- JPFA
- JPDT

Evaluation

- YCSB benchmark
- recovery

Conclusion

Overview

J-NVM = Off-Heap Persistent Objects

Challenges	→	**Features**
single data representation		*off-heap* persistent objects
programming model		class-centric model (code generator + PDT library)
direct access to NVMM		sun.misc.Unsafe
durability abstraction		failure-atomic blocks + fine-grained
scalability (large persistent dataset)		see evaluation

Anatole Lefort et al., J-NVM, SOSP'21.

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)

Tooling

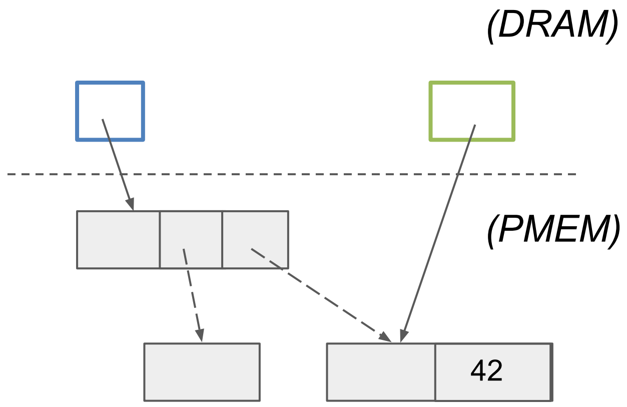
- built-in off-heap memory management for NVMM
- code generator: automatic decoupling for POJOs
- J-PFA: automatic failure-atomic code
- J-PDT: data types + collections for persistent memory
- low-level API (for experts)
- recovery-time GC

Anatole Lefort et al., J-NVM, SOSP'21.

Programming model - *persistent objects*

Persistent object is

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



Alive when reachable (from persistent root)

```
Map root = JNVM.root();
Simple s = root.get(“Simple”);
s.setX(42);
```

Class-centric model

- safe references thanks to the type system

Anatole Lefort et al., J-NVM, SOSP’21.

Programming model - *life cycle*

Constructor

- allocate NVMM
- attach persistent data structure

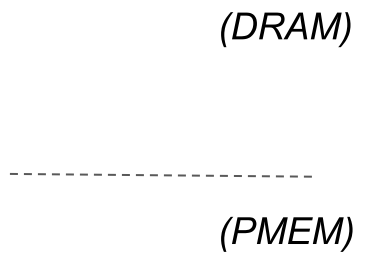
Re-Constructor

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

Destructor

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

```
Simple s = new Simple(42);
```



Anatole Lefort et al., J-NVM, SOSP'21.

Programming model - *life cycle*

Constructor

- allocate NVMM
- attach persistent data structure

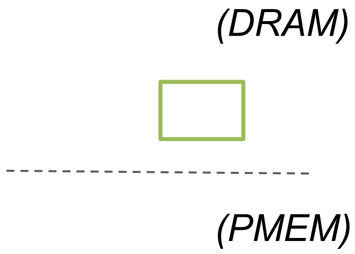
Re-Constructor

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

Destructor

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

```
Simple s = new Simple(42);
```



Anatole Lefort et al., J-NVM, SOSP’21.

Programming model - *life cycle*

Constructor

- allocate NVMM
- attach persistent data structure

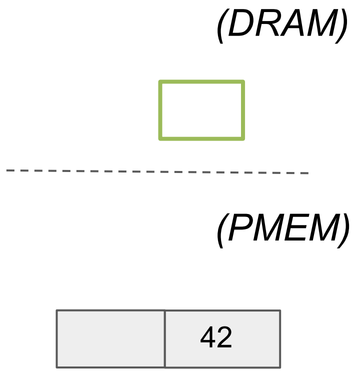
Re-Constructor

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

Destructor

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

```
Simple s = new Simple(42);
```



Anatole Lefort et al., J-NVM, SOSP’21.

Programming model - *life cycle*

Constructor

- allocate NVMM
- attach persistent data structure

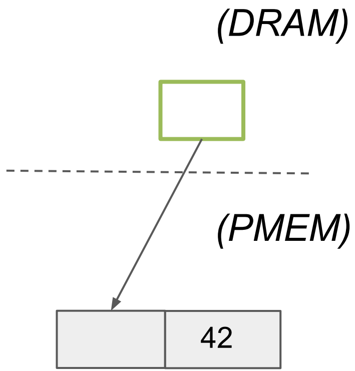
Re-Constructor

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

Destructor

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

```
Simple s = new Simple(42);
```



Anatole Lefort et al., J-NVM, SOSP'21.

Programming model - *life cycle*

Constructor

- allocate NVMM
- attach persistent data structure

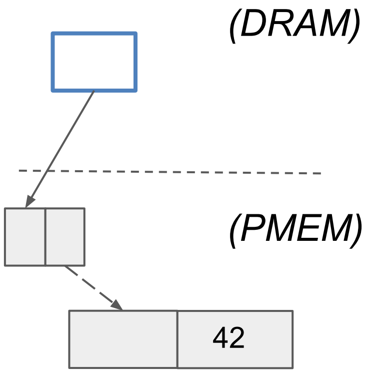
Re-Constructor

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

Destructor

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

```
Simple s = t.getSimple();
```



Anatole Lefort et al., J-NVM, SOSP'21.

Programming model - *life cycle*

Constructor

- allocate NVMM
- attach persistent data structure

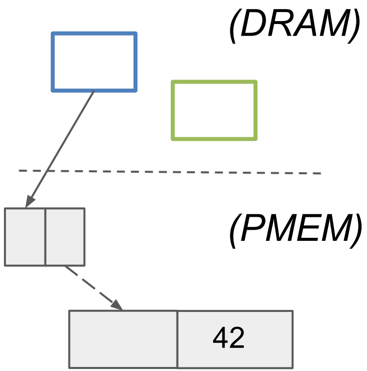
Re-Constructor

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

Destructor

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

```
Simple s = t.getSimple();
```



Anatole Lefort et al., J-NVM, SOSP’21.

Programming model - *life cycle*

Constructor

- allocate NVMM
- attach persistent data structure

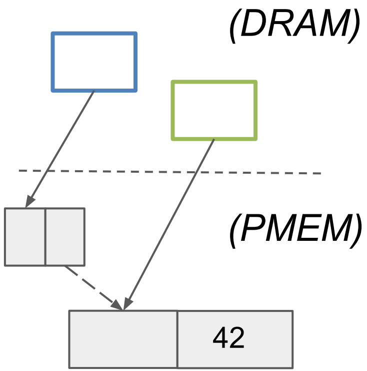
Re-Constructor

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

Destructor

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

```
Simple s = t.getSimple();
```



Anatole Lefort et al., J-NVM, SOSP'21.

Programming model - *life cycle*

Constructor

- allocate NVMM
- attach persistent data structure

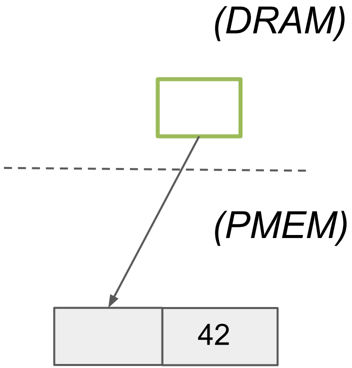
Re-Constructor

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

Destructor

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

```
JNVM.free(s);
```



Anatole Lefort et al., J-NVM, SOSP'21.

Programming model - *life cycle*

Constructor

- allocate NVMM
- attach persistent data structure

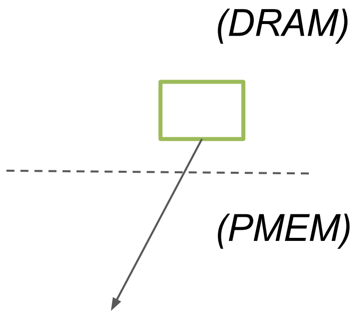
Re-Constructor

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

Destructor

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

```
JNVM.free(s);
```



Programming model - *life cycle*

Constructor

- allocate NVMM
- attach persistent data structure

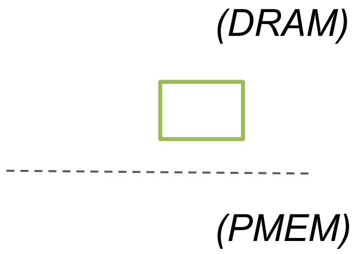
Re-Constructor

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

Destructor

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

```
JNVM.free(s);
```



Programming model - *life cycle*

Constructor

- allocate NVMM
- attach persistent data structure

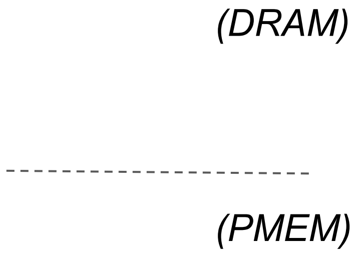
Re-Constructor

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

Destructor

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

```
JNVM.free(s);
```



Anatole Lefort et al., J-NVM, SOSP'21.

Programming model - *code generator*

```java
@Persistent(fa="non-private")
class Simple {
  PString msg;
  int x;
  transient int y;

  Simple(int x) {
    this.x = x;
    this.msg = new PString("Hello, NVMM!");
  }

  void inc() { x++; }
}
```

```java
// transformed code
class Simple implements PObject {
  transient int y;
  long addr; // persistent data structure

  Simple(int x) {
    JNVM.faStart();
    this.addr = JNVM.alloc(getClass(), size());
    setX(x);
    setMsg(new PString("Hello, NVMM!"));
    JNVM.faEnd();
  }

  Simple(long addr) {
    this.addr = addr;
    this.resurrect();
  }
}
```

Goals

- class-wide off-heap layout
- generate constructor, re-constructor
- replace (non-transient) field accesses
- wrap non-private methods

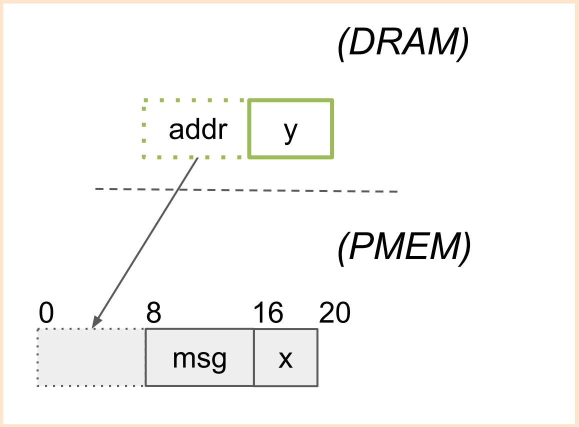
Anatole Lefort et al., J-NVM, SOSP'21.

Programming model - *code generator*

```
@Persistent(fa="non-private")
class Simple {
  PString msg;
  int x;
  transient int y;

  Simple(int x) {
    this.x = x;
    this.msg = new PString("Hello, NVMM!");
  }

  void inc() { x++; }
}
```



Goals

- class-wide off-heap layout
- generate constructor, re-constructor
- replace (non-transient) field accesses
- wrap non-private methods

Anatole Lefort et al., J-NVM, SOSP'21.

Programming model - *code generator*

```java
@Persistent(fa="non-private")
class Simple {
  PString msg;
  int x;
  transient int y;

  Simple(int x) {
    this.x = x;
    this.msg = new PString("Hello, NVMM!");
  }

  void inc() { x++; }
}
```

```java
// transformed code
class Simple implements PObject {
  transient int y;
  long addr; // persistent data structure

  Simple(int x) {
    JNVM.faStart();
    this.addr = JNVM.alloc(getClass(), size());
    setX(x);
    setMsg(new PString("Hello, NVMM!"));
    JNVM.faEnd();
  }

  Simple(long addr) {
    this.addr = addr;
    this.resurrect();
  }
}
```

Goals

- class-wide off-heap layout
- generate constructor, re-constructor
- replace (non-transient) field accesses
- wrap non-private methods

Anatole Lefort et al., J-NVM, SOSP'21.

Programming model - *code generator*

```
@Persistent(fa="non-private")
class Simple {
  PString msg;
  int x;
  transient int y;

  Simple(int x) {
    this.x = x;
    this.msg = new PString("Hello, NVMM!");
  }

  void inc() { x++; }
}
```

```
// transformed code
class Simple implements PObject {
  transient int y;
  long addr; // persistent data structure

  Simple(int x) {
    JNVM.faStart();
    this.addr = JNVM.alloc(getClass(), size());
    setX(x);
    setMsg(new PString("Hello, NVMM!"));
    JNVM.faEnd();
  }

  void inc() {
    JNVM.faStart();
    setX(getX()++);
    JNVM.faEnd();
  }
}
```

Goals

- class-wide off-heap layout
- generate constructor, re-constructor
- replace (non-transient) field accesses
- wrap non-private methods

Anatole Lefort et al., J-NVM, SOSP'21.

Programming model - *code generator*

```java
@Persistent(fa="non-private")
class Simple {
  PString msg;
  int x;
  transient int y;

  Simple(int x) {
    this.x = x;
    this.msg = new PString("Hello, NVMM!");
  }

  void inc() { x++; }
}
```

```java
// transformed code
class Simple implements PObject {
  transient int y;
  long addr; // persistent data structure

  Simple(int x) {
    JNVM.faStart();
    this.addr = JNVM.alloc(getClass(), size());
    setX(x);
    setMsg(new PString("Hello, NVMM!"));
    JNVM.faEnd();
  }

  void inc() {
    JNVM.faStart();
    setX(getX()++);
    JNVM.faEnd();
  }
}
```

Goals

- class-wide off-heap layout
- generate constructor, re-constructor
- replace (non-transient) field accesses
- wrap non-private methods

Anatole Lefort et al., J-NVM, SOSP'21.

Programming model - *code generator*

```java
@Persistent(fa="non-private")
class Simple {
  PString msg;
  int x;
  transient int y;

  Simple(int x) {
    this.x = x;
    this.msg = new PString("Hello, NVMM!");
  }

  void inc() { x++; }
}
```

```java
// transformed code (continued)
  long addr; // the persistent data structure
  long size() { return 12; }

  PString getMsg() { return (PString)
          JNVM.readPObject(addr, 0); }

  void setMsg(PString v) {
          JNVM.writePObject(addr, 0, v); }

  int getX() {return JNVM.readInt(addr, 8);}

  void setX(int v) {JNVM.writeInt(addr, 8, v);}
```

Goals

- class-wide off-heap layout
- generate constructor, re-constructor
- replace (non-transient) field accesses
- wrap non-private methods
- generate or transform field accessors

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)

Anatole Lefort et al., J-NVM, SOSP’21.

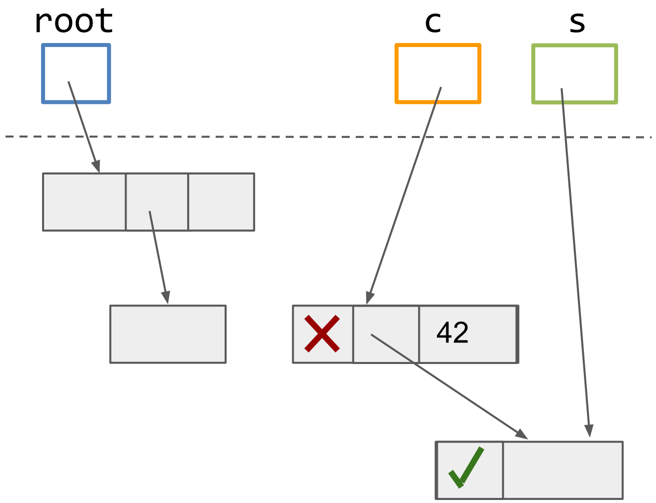
J-PDT + Low-level interface

J-PDT

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

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



```
Complex c = new Complex(s);
root.put(“Complex”, c);
c.validate();
```

Anatole Lefort et al., J-NVM, SOSP’21.

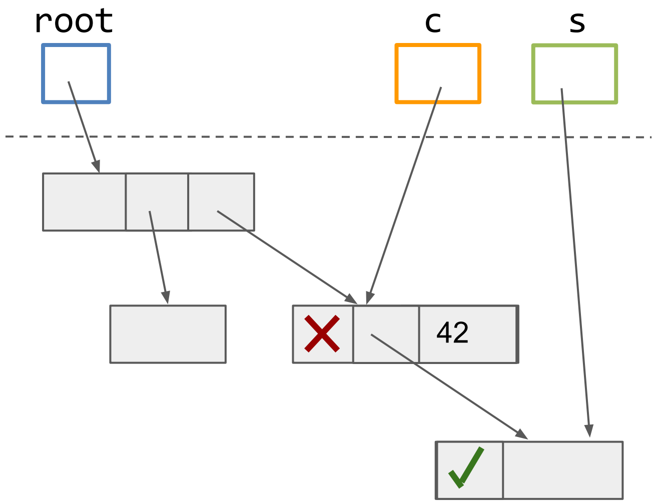
J-PDT + Low-level interface

J-PDT

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

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



```
Complex c = new Complex(s);
root.put("Complex", c);
c.validate();
```

Anatole Lefort et al., J-NVM, SOSP'21.

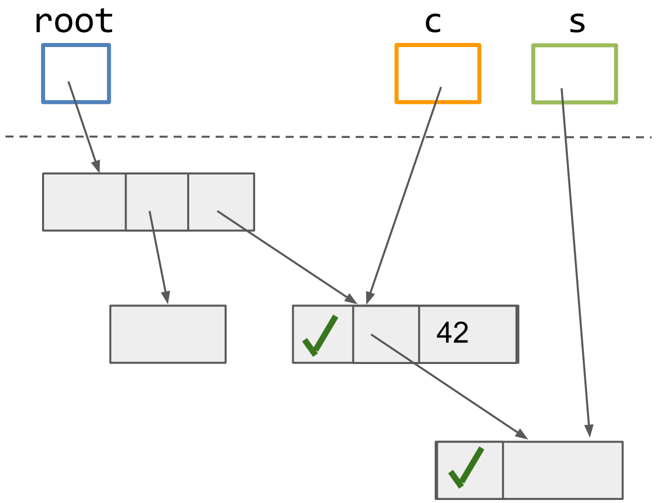
J-PDT + Low-level interface

J-PDT

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

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



```
Complex c = new Complex(s);
root.put("Complex", c);
c.validate();
```

Anatole Lefort et al., J-NVM, SOSP'21.

Outline

Introduction

- NVMM
- why Java?
- prior works

System Design

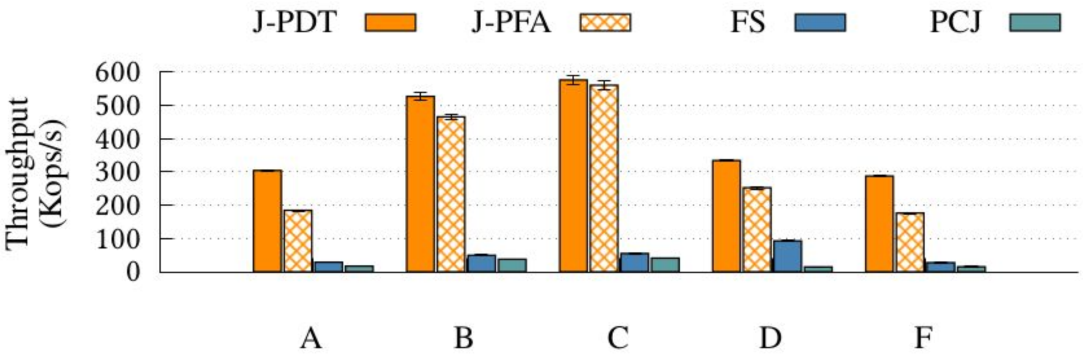
- overview
- programming model
- JPFA
- JPDT

Evaluation

- YCSB benchmark
- recovery

Conclusion

YCSB Benchmark



Durable backends for Infinispan:

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

Hardware used:

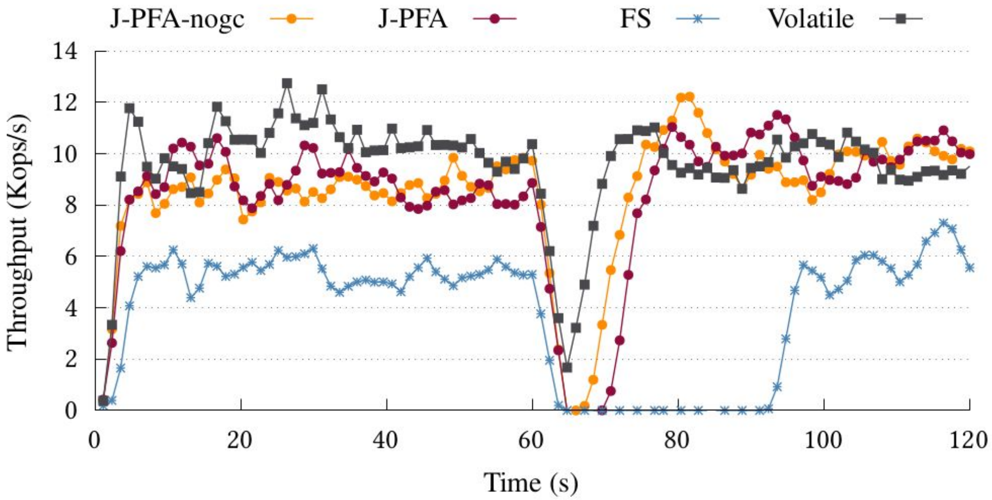
4 Intel CLX 6230 HT 80-core
128GB DDR4,
4x128GB Optane (gen1)

Takeaways:

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

Anatole Lefort et al., J-NVM, SOSP'21.

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)

Anatole Lefort et al., J-NVM, SOSP'21.

Conclusion

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)

Cons:

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