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
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)
Non-volatile main memory

new persistent medium (in-between SSD and DRAM)

Durable
resists reboots, power loss

High-density
smallest DIMM = 128 GB

Direct byte-addressability of durable data

Byte addressable
persistent memory abstraction

High-performance
low latency (seq. read/write ~ 160/90ns)
high bandwidth (up to 8.10GB/s, 2nd gen)
Non-volatile main memory

new persistent medium (in-between SSD and DRAM)

Durable
resists rebo

High-density
smallest DIMM

Byte addressable
persistent memory abstraction

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

Direct byte-addressability of durable data

1- Dramatic **throughput** and **latency** improvement for persistent data applications
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)

Direct byte-addressability of durable data

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

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

J-NVM, NVMW’22.
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

Problematic: Java-native NVMM interface
Prior works: *internal design*

= [Espresso, AutoPersist, go-pmem]

**Managed Persistent Objects**

- Infinispan datastore in-memory cache ratio
  - 100%
  - 10%
  - 1%

**Features**
- managed persistent objects
- orthogonal persistence (pnew, @persistentRoot)
- heavily-modified runtime
- failure-atomic blocks
- non-scalable

Garbage collectors can not scale to large persistent datasets

**Fixed dataset size - 80GB on heap for 100% cache**

**Varying cache ratio (YCSB-F)**

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

**Infinispan datastore in-memory cache ratio**

**Time (min)**

**CPU compute time**

**CPU GC time**

**completion time**

**Persistent dataset size (GB)**

0.30 0.59 1.18 2.37 4.74 9.48 18.96 37.92 75.84 151.68
Prior works: *internal design*

= [Espresso, AutoPersist, go-pmem]

**Managed Persistent Objects**

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
- non-scalable

---

Garbage collectors can not scale to large persistent datasets
Prior works: *internal design*

= [Espresso, AutoPersist, go-pmem]

Managed Persistent Objects

*Features*

- *managed* persistent objects
- orthogonal persistence (pnew, @persistentRoot)
- heavily-modified runtime
- failure-atomic blocks
- non-scalable

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”.

code instrumentation = made whole JVM 51% slower in [Autopersist]

In [go-pmem]: Increasing dataset (YCSB-F, go-pmem)

Completion time (min)

Fixed dataset size - 80GB on heap for 100% cache

Varying cache ratio (YCSB-F)

Garbage collectors cannot scale to large persistent datasets

*J-NVM, NVMW’22.*
### Overview

J-NVM = Off-Heap Persistent Objects

<table>
<thead>
<tr>
<th><strong>Challenges</strong></th>
<th><strong>Features</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>single data representation</td>
<td><em>off-heap</em> persistent objects</td>
</tr>
<tr>
<td>programming model</td>
<td>class-centric model</td>
</tr>
<tr>
<td>direct access to NVMM</td>
<td>(code generator + PDT library)</td>
</tr>
<tr>
<td>durability abstraction</td>
<td>sun.misc.Unsafe</td>
</tr>
<tr>
<td>scalability (large persistent dataset)</td>
<td>failure-atomic blocks + fine-grained</td>
</tr>
<tr>
<td></td>
<td>see evaluation</td>
</tr>
</tbody>
</table>

---

*J-NVM, NVMW’22.*
Outline

Introduction
- NVMM
- why Java?
- prior works
- overview

System Design
- programming model
  - persistent objects
  - code generator
- JPFA
- JPDT

Evaluation
- YCSB benchmark
- recovery

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
  - implement PObject interface
  - intermediate 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

```java
Map root = JNVM.root();
Simple s = root.get("Simple");
s.setX(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

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

(DRAM)

------------------------
(PMEM)

*J-NVM, NVMW'22.*
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

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

(DRAM)

---------------------------------------------

(PMEM)
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);
```

(DRAM)

(PMEM)

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

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

(DRAM)

(PMEM)
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

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

(DRAM)

(PMEM)

```
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

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

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

![Diagram](image.png)
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

```java
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

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

*(DRAM)*

*(PMEM)*
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

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

(DRAM)

(PMEM)
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

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

(DRAM)

------------------

(PMEM)
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

@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

// 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();
    }
}
@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
Programming model - code generator

Goals

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

@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();
    }

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

    J-NVM, NVMW'22.
Goals
- class-wide off-heap layout
- generate constructor, re-constructor
- replace (non-transient) field accesses
- wrap non-private methods

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

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

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

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

// 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
- generate or transform field accessors

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 errMsg() {
    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);}
```
Automatic crash-consistent update
usage = \texttt{JNVM.faStart()} \textit{some code} \texttt{JNVM.faEnd()}

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

Log new, free and updates
granularity = a block of PMEM

Do \textit{not} log updates to “new” persistent objects
(e.g. allocated within the FA-block)
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();
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();
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();
Outline

Introduction
- NVMM
- why Java?
- prior works
- overview

System Design
- programming model
  - persistent objects
  - code generator
- JPFA
- JPDT

Evaluation
- YCSB benchmark
- recovery

Conclusion
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

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 but common for durable data
- limited code re-use but safer programming model