AutoPersist: An Easy-To-Use Java NVM Framework Based on Reachability

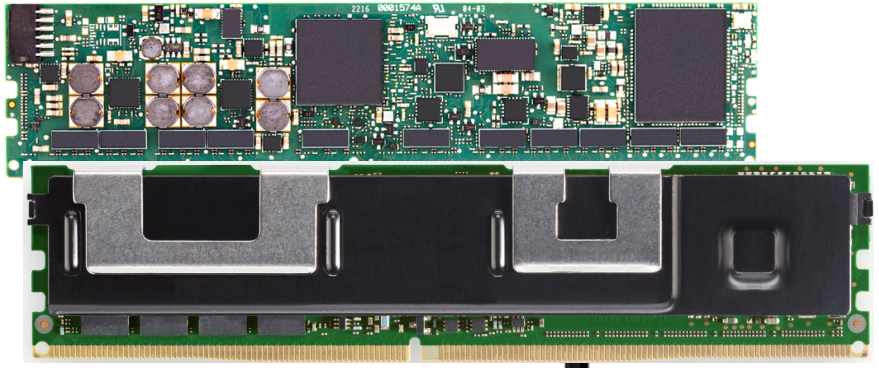
Thomas Shull, Jian Huang, Josep Torrellas
University of Illinois at Urbana-Champaign

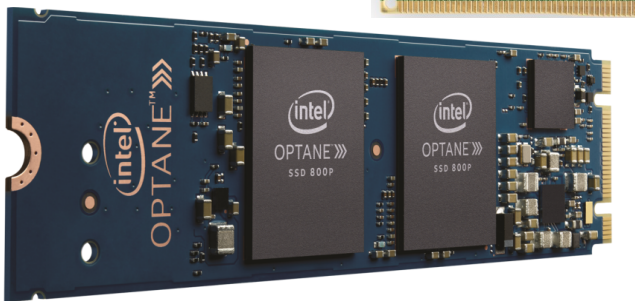
June 24, 2019
PLDI'19 – Memory Management Session

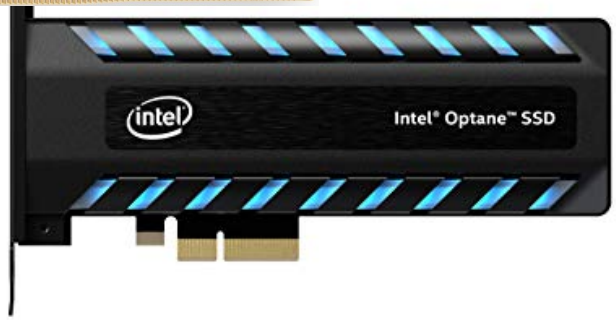
Programming NVM – The Good

- Emerging Non-Volatile Memory (NVM) devices offer an enticing combination of performance, capacity, and persistency
- Applications no longer have to serialize data out to secondary storage for durability
- NVM is byte-addressable

Real NVM Systems







Programming NVM – The Bad

Creating persistent applications in NVM is tricky:

- Volatile processor caches are in the memory hierarchy
- Data must be written back from caches to achieve persistency
  - CLWB (cachline writeback) - force writeback to NVM
  - SFENCE (store fence) - block until writebacks have completed
- Software logging is needed to ensure the failure-atomicity of a collection of writes
  - Hardware only guarantees atomicity at a cacheline granularity
- Solution: use **frameworks** to hide much of the complexity

Presentation Outline

1. Describe current NVM framework landscape
   - Current features
   - Drawbacks
2. Present AutoPersist: an Easy-To-Use Java NVM framework
   - Programming model
   - Programming examples
   - Implementation details
3. Comparison of AutoPersist against existing Java NVM frameworks

Current NVM Frameworks – Features

Currently, the user must perform some of the following actions:

1. Manually identifying persistent objects
2. Add CLWBs and SFENCEs after stores
3. Add logging for failure-atomic regions
4. Using previously marked data structures or libraries

Current NVM Frameworks – Example

```
struct DurableList{
  int element; DurableList* next;
}

DurableList* prepend(DurableList* old_head, int element){
  DurableList* new_head = new DurableList();
  new_head->element = element;

  new_head->next = old_head;


  return new_head;
}
```

Current NVM Frameworks – Example

```
struct DurableList{
  int element; DurableList* next;
}

DurableList* prepend(DurableList* old_head, int element){
  DurableList* new_head = Durable_New DurableList();
  new_head->element = element;
  Clwb(&(new_head->element));
  new_head->next = old_head;
  Clwb(&(new_head->next));
  Sfence();
  return new_head;
}
```

Current NVM Frameworks – Takeaways

Drawbacks of current NVM frameworks:

- Need to identify objects which should be allocated in NVM and also insert CLWBs and SFENCEs
- Easy to introduce bugs
  - Correctness bugs – markings are missing
  - Performance bugs – too many markings
- Difficult for the compiler to perform optimizations
  - Programmer's intention is not visible to the compiler

Contribution: AutoPersist

- Solution to existing shortcomings: design AutoPersist – a new ***easy-to-use*** NVM framework
- Focus on programmability first
  - Rely on compiler optimizations to get high performance
- Make framework tailored to managed languages
  - Effectively leverage managed language runtime features
- AutoPersist significantly outperforms existing Java NVM offerings

AutoPersist NVM Programming Model

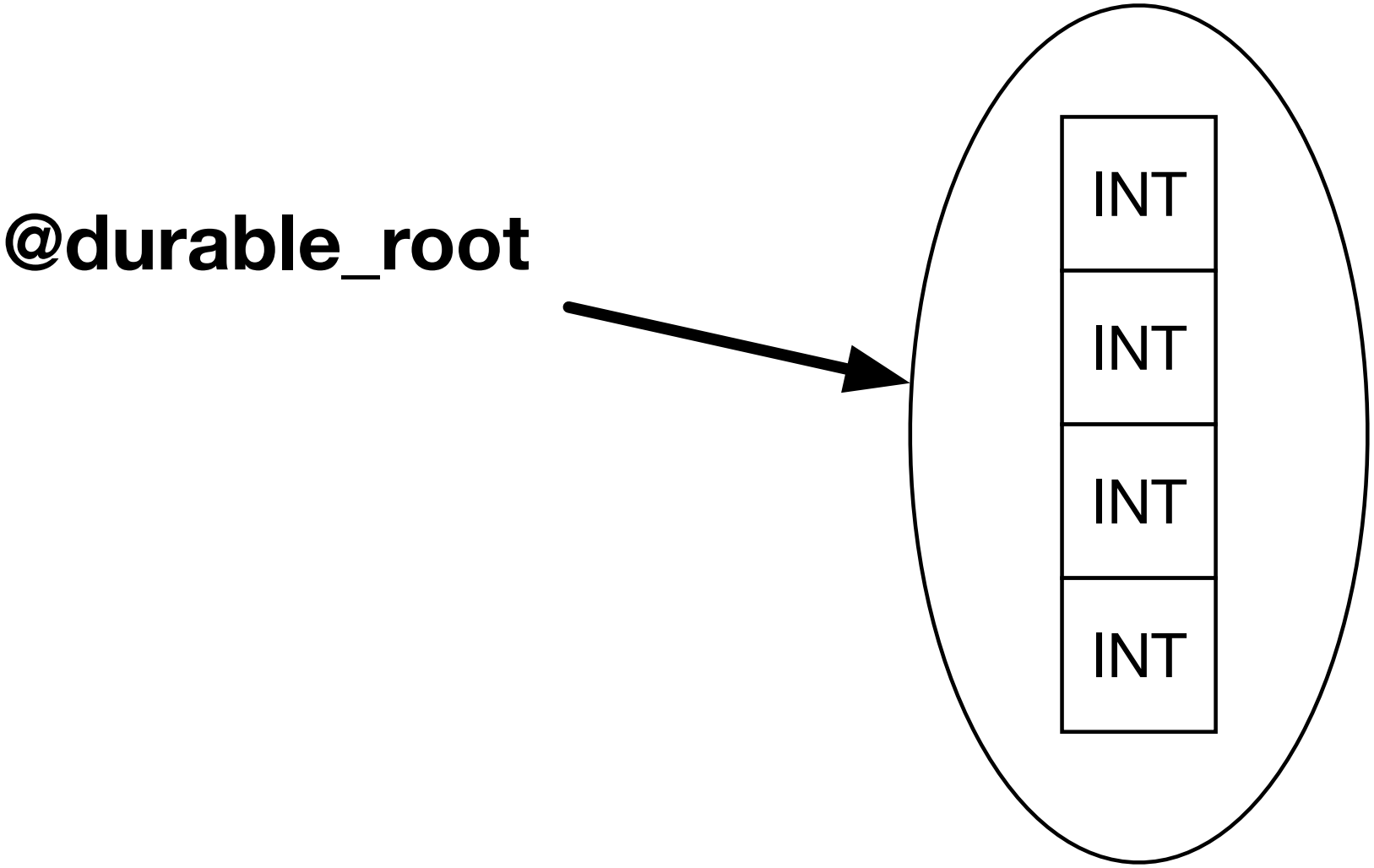
- In the AutoPersist programming model, the user must identify only a ***durable root set*** and ***failure-atomic regions***
  - *Durable root set*: objects directly referred to at recovery time
  - User annotates desired durable roots with the **@durable_root** marking
  - *Failure-atomic regions*: regions which should be atomic from a crash-consistency perspective

AutoPersist Runtime Responsibilities

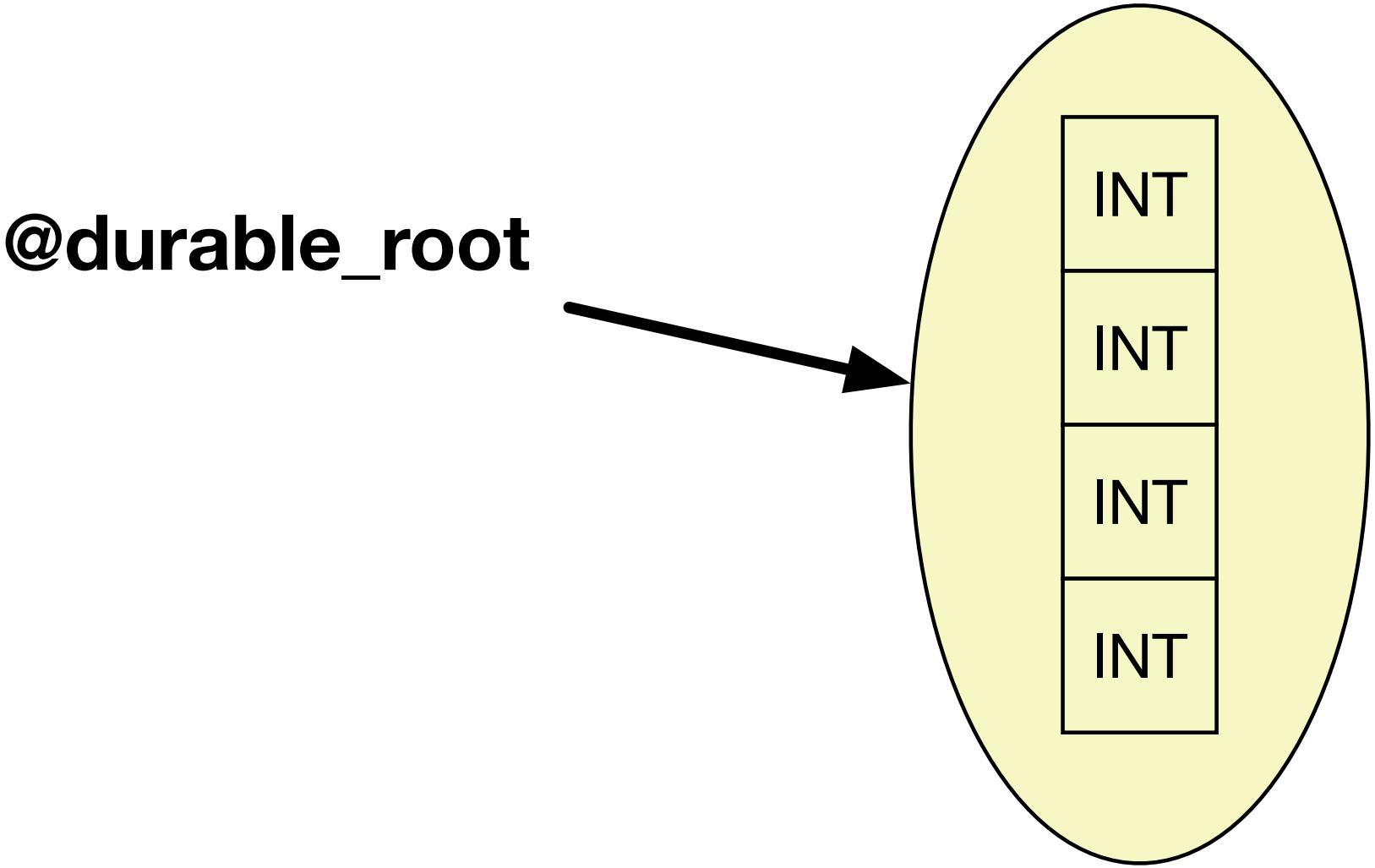
AutoPersist's runtime automatically:

1. Dynamically detects and monitors the *transitive closure* of durable roots
   - Ensures everything reachable from a durable root is in NVM
   - Dynamically moves objects to NVM throughout execution
2. Ensure stores to persistent objects are performed correctly
   - Automatically inserts the needed logging, CLWBs, and SFENCEs

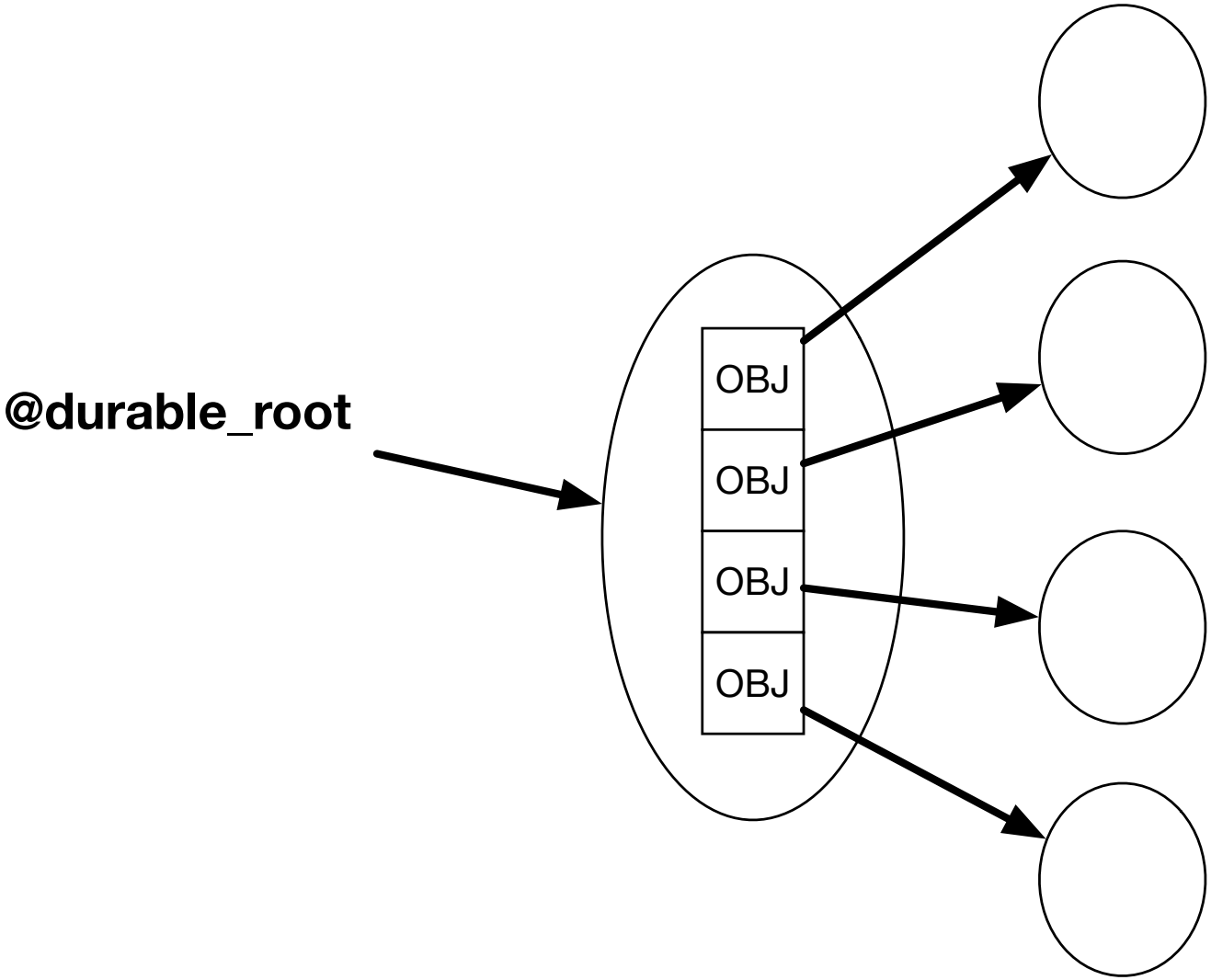
Dynamically Moving Objects to NVM



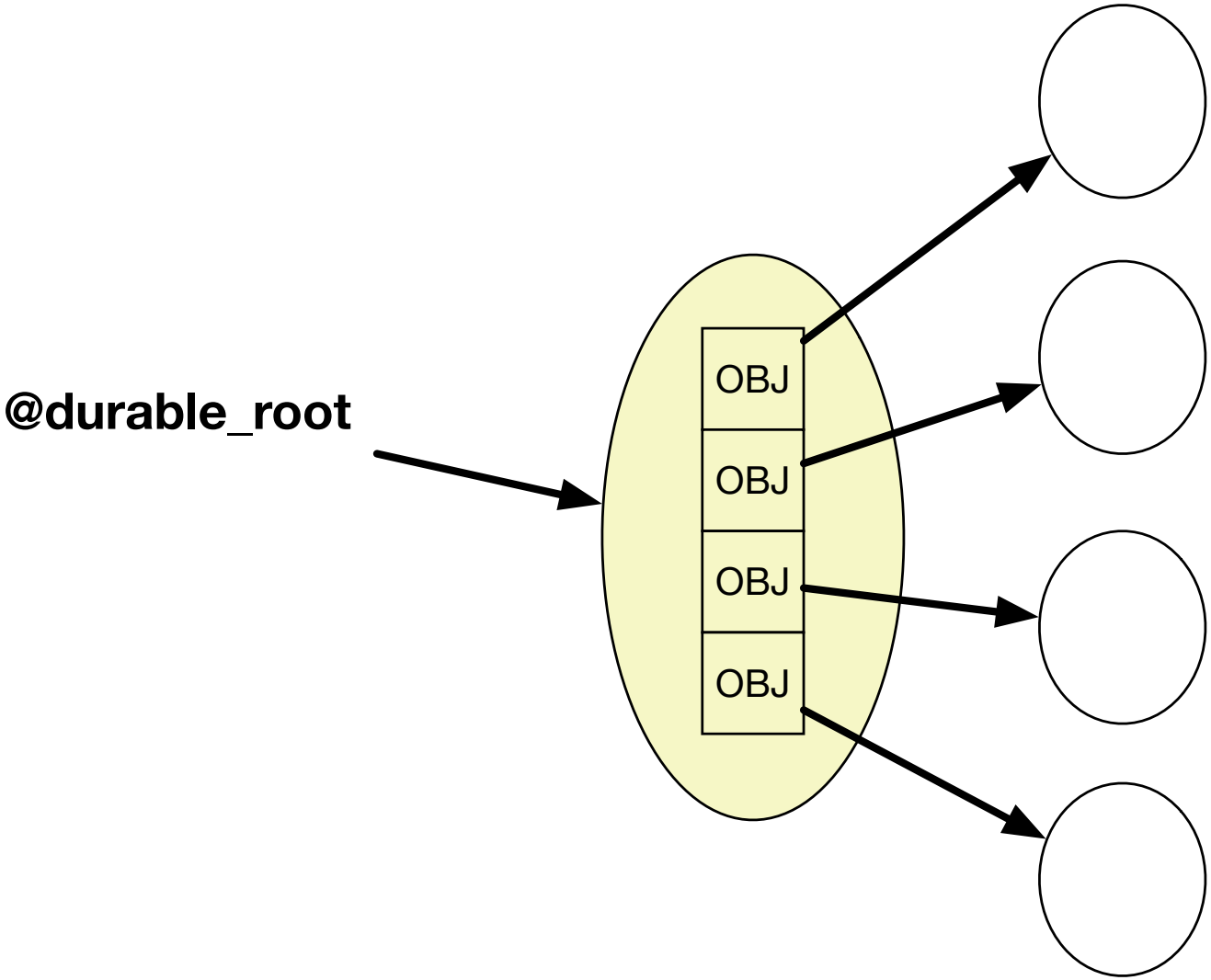
Dynamically Moving Objects to NVM



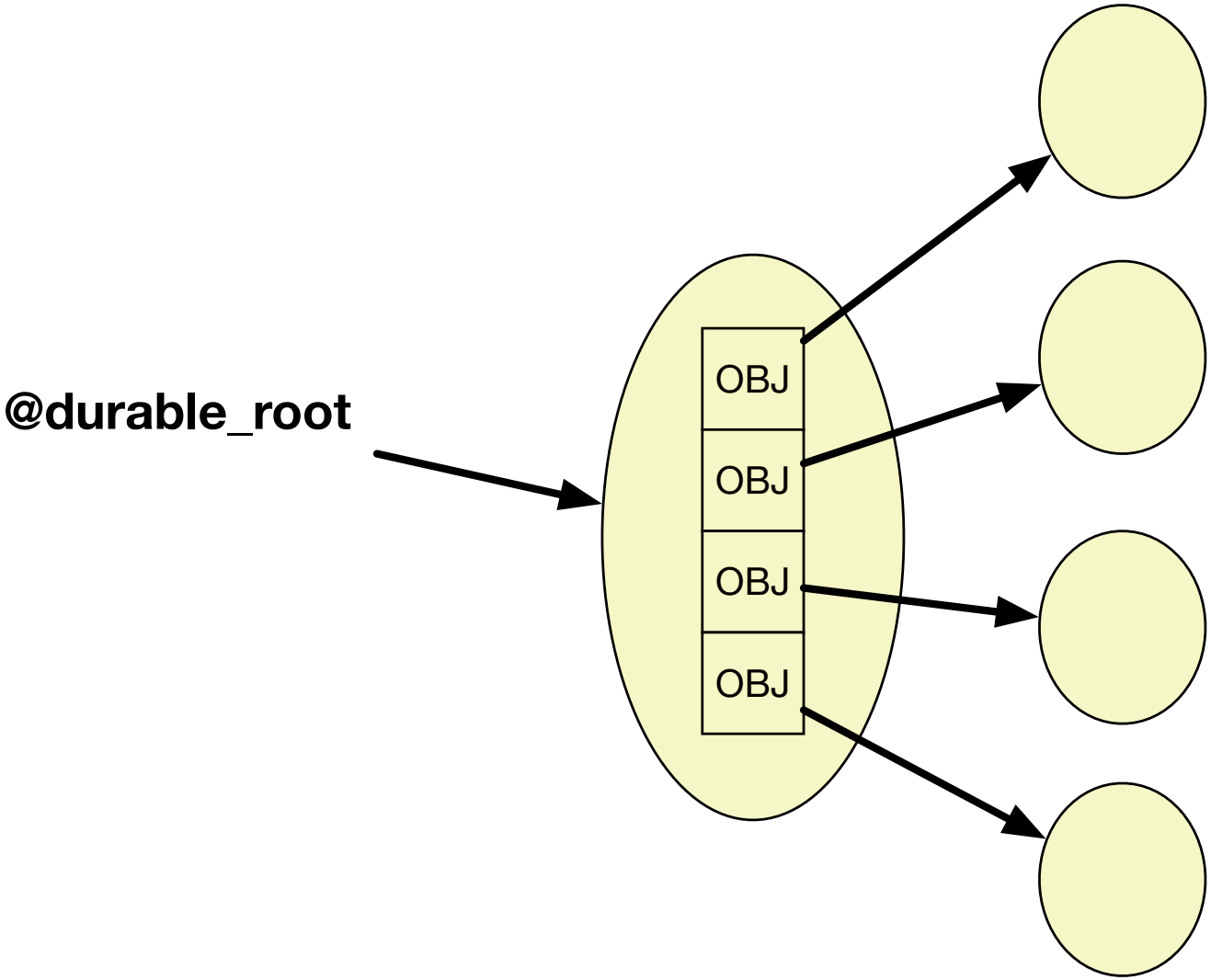
Dynamically Moving Objects to NVM



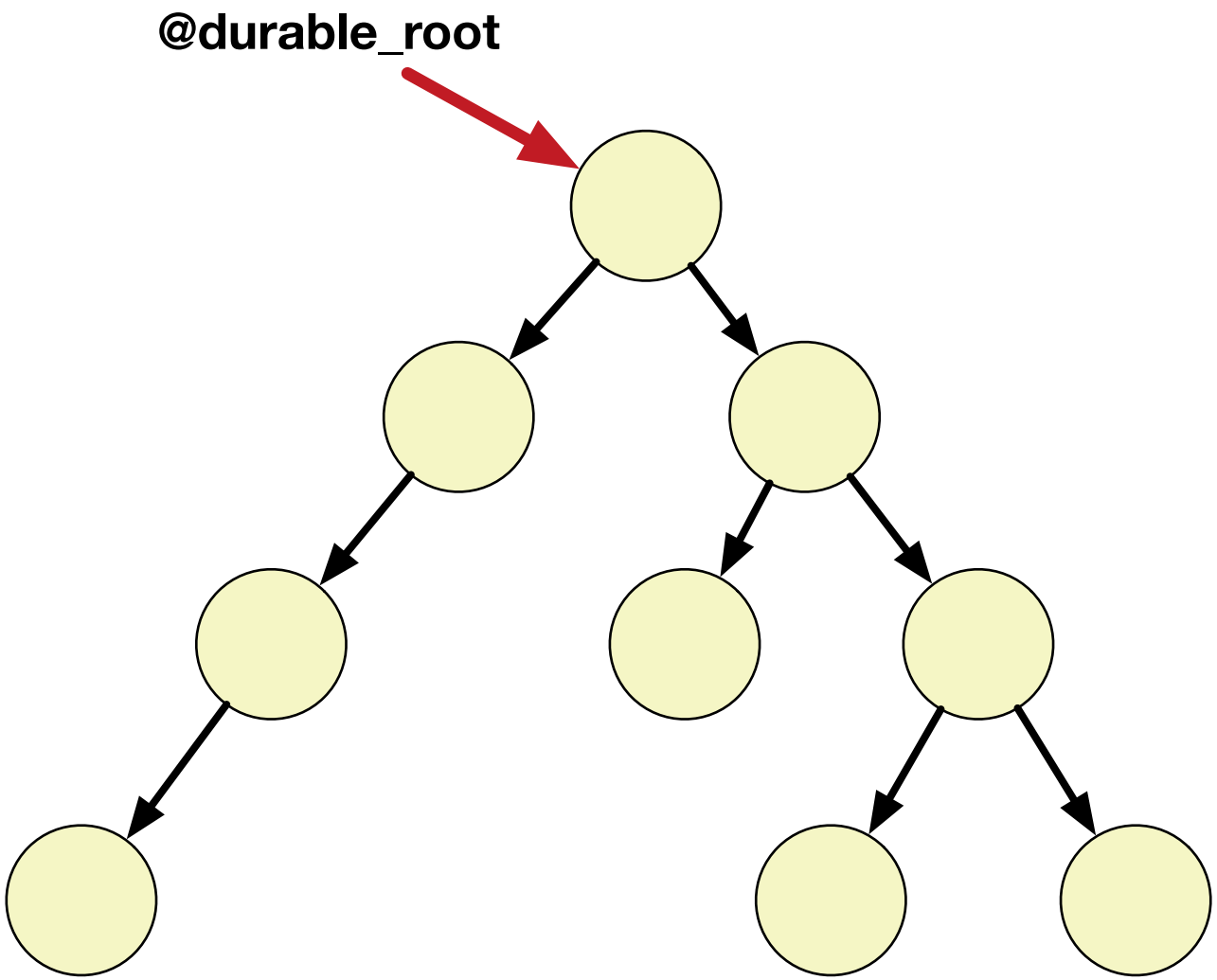
Dynamically Moving Objects to NVM



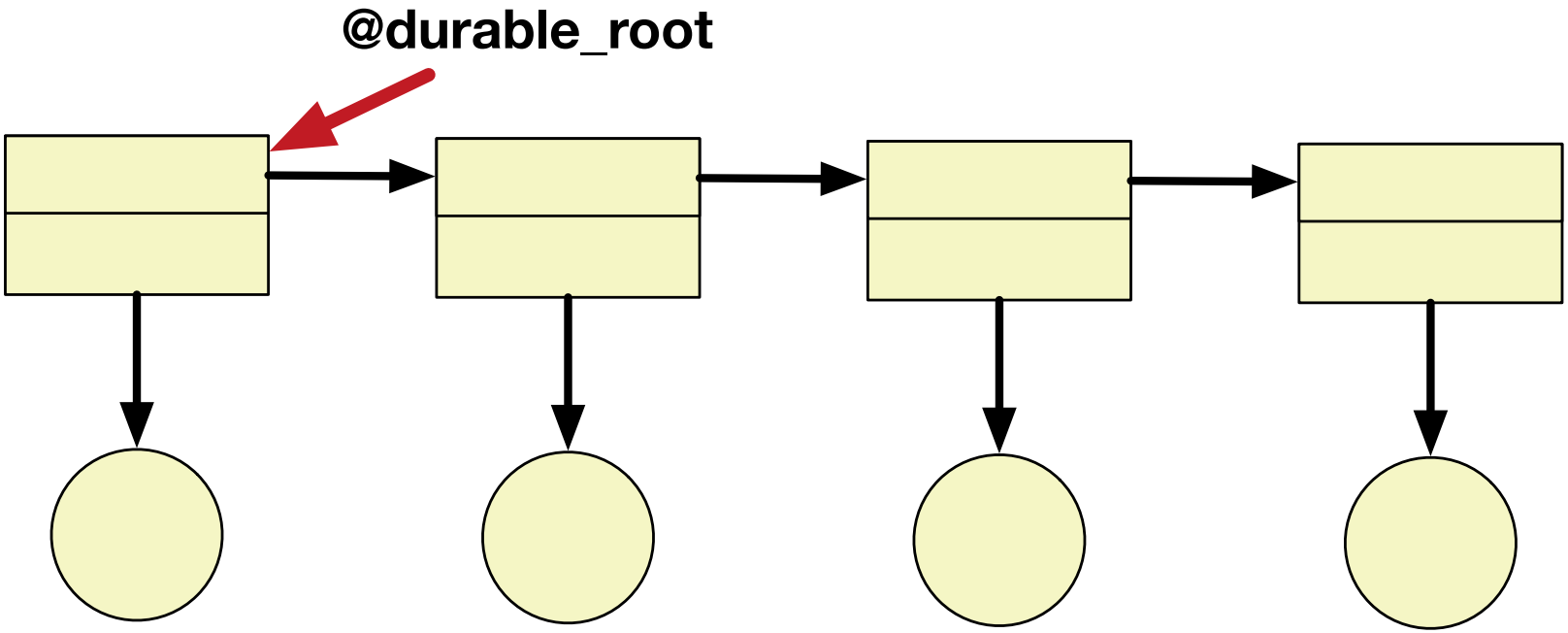
Dynamically Moving Objects to NVM



Dynamically Moving Objects to NVM



Dynamically Moving Objects to NVM



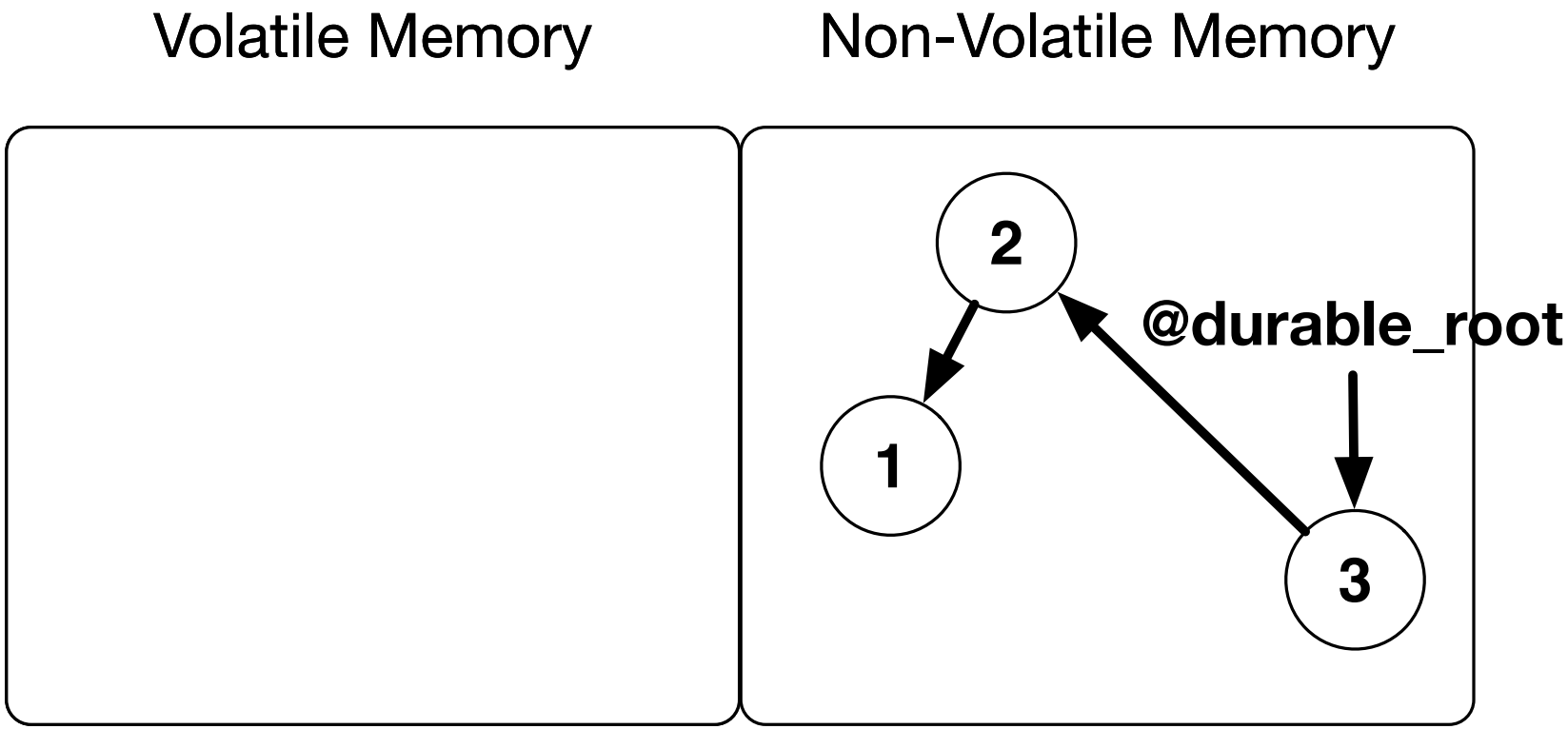
Programming in AutoPersist – Example

```
class DurableList{
  private int element; private DurableList next;

  public static DurableList head;

  public static void prepend(int element){
    DurableList newHead = new DurableList();
    newHead.element = element;
    newHead.next = head;
    head = newHead;
  }
}
```

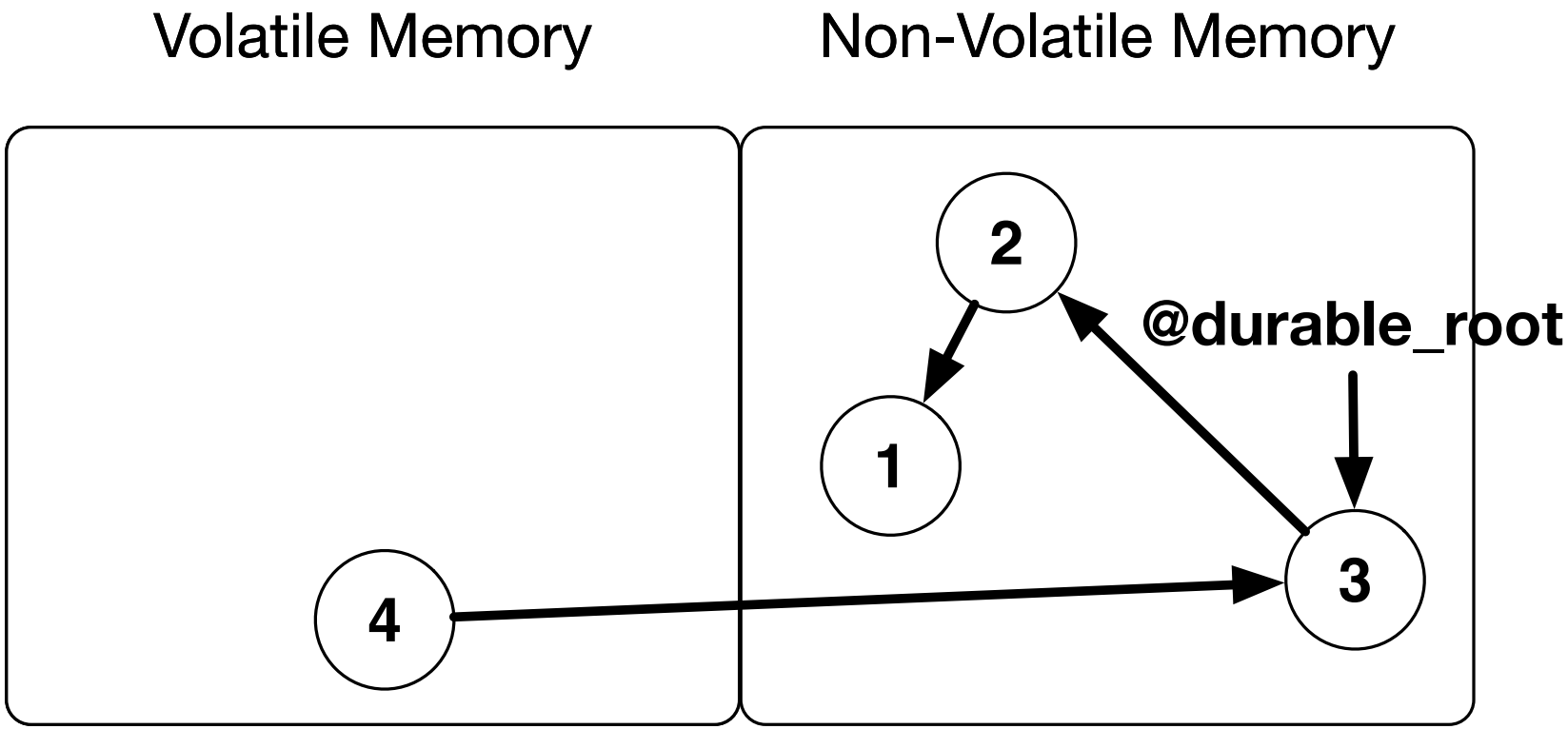
Programming in AutoPersist – Example

```
class DurableList{
  private int element; private DurableList next;
  @durable_root
  public static DurableList head;

  public static void prepend(int element){
    DurableList newHead = new DurableList();
    newHead.element = element;
    newHead.next = head;
    head = newHead;
  }
}
```

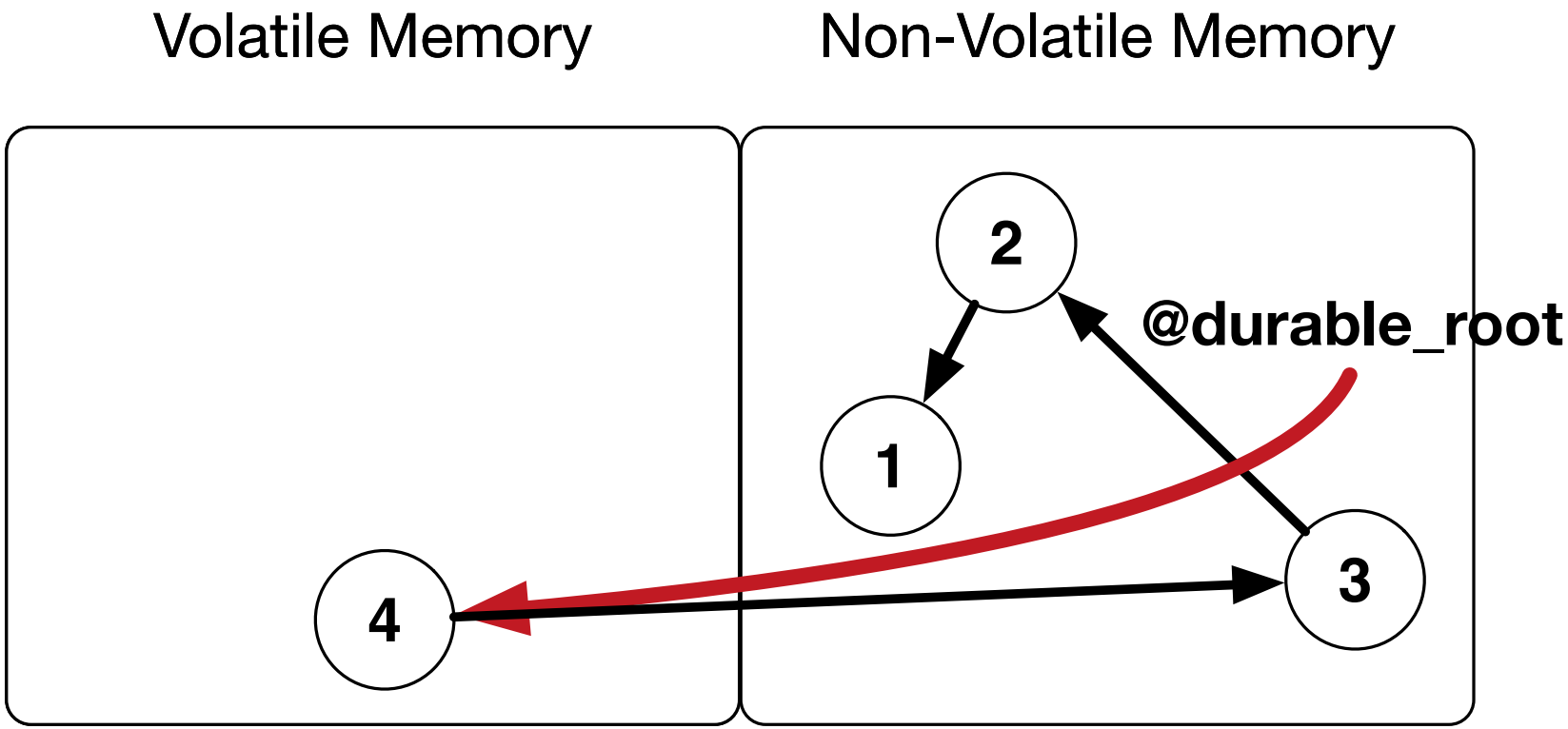
Dynamically Moving Objects to NVM



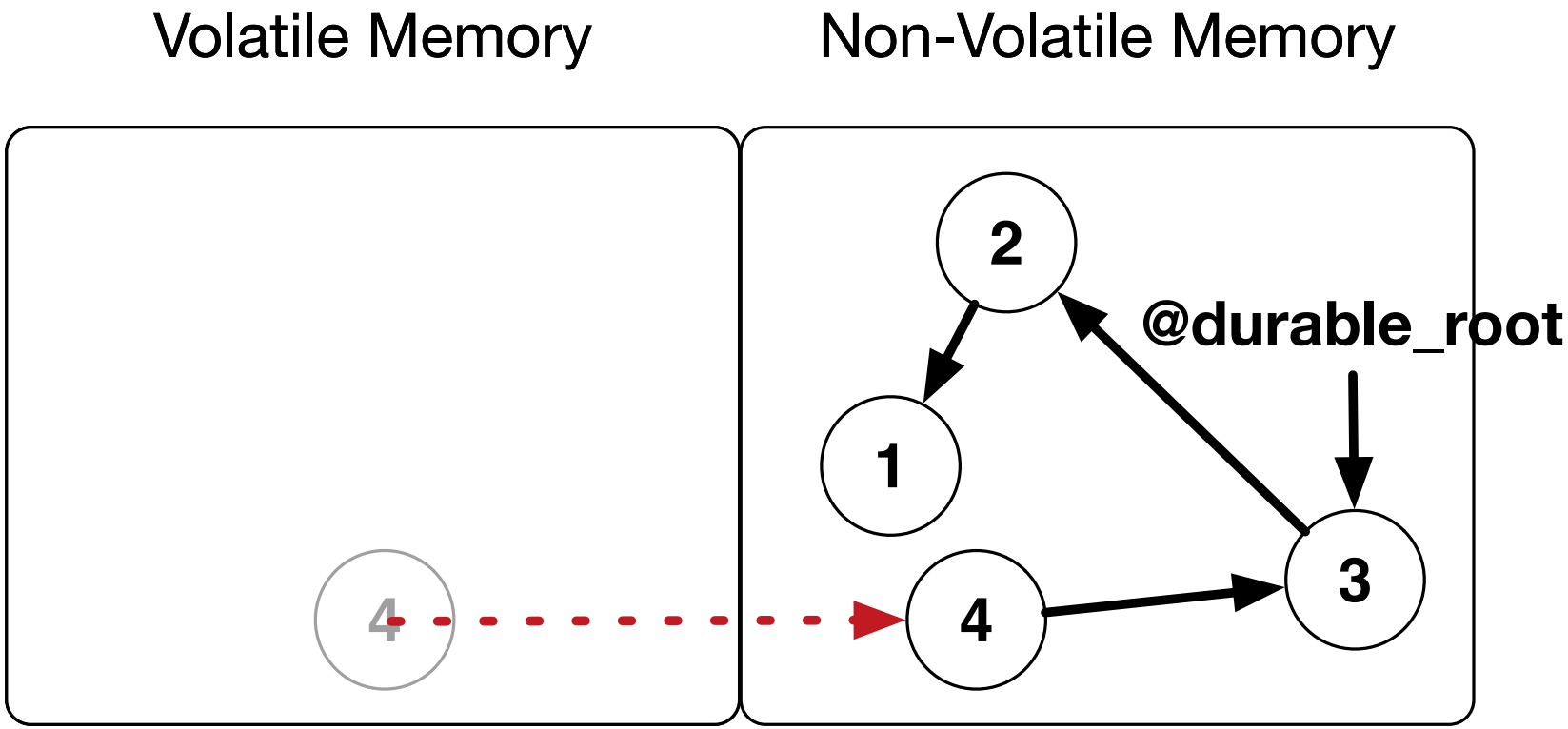
Dynamically Moving Objects to NVM



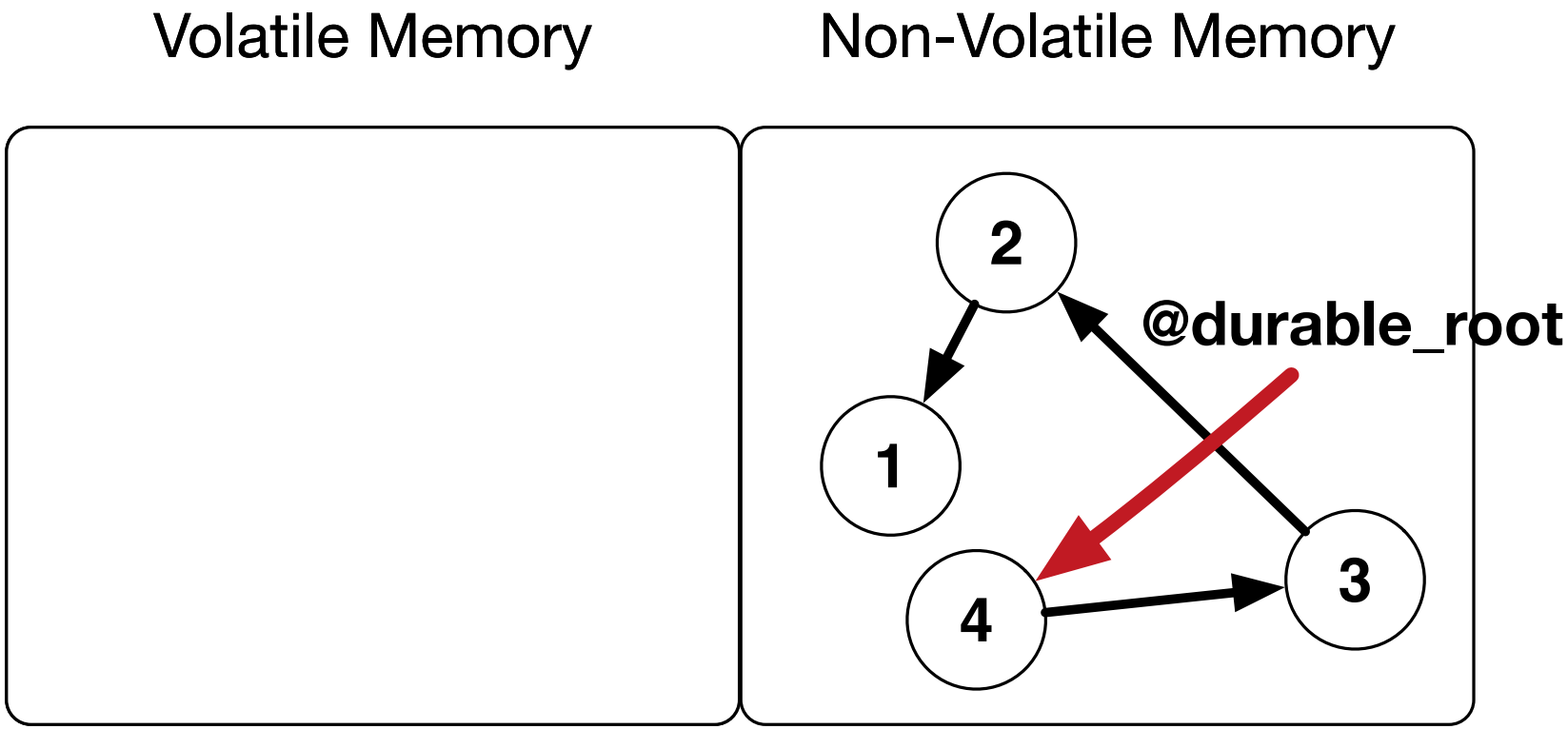
Dynamically Moving Objects to NVM



Dynamically Moving Objects to NVM



Dynamically Moving Objects to NVM



Ensuring Stores are Made Persistent

AutoPersist Persistency Model:

- Epoch Persistency
  - Used within failure-atomic regions
  - Undo logging is performed for atomicity
- Sequential Persistency
  - Used everywhere else
  - Inserts CLWBs and SFENCEs after each store

AutoPersist Persistency – Example

```
**Elements in persistent list**
DurableList a = ...
DurableList b = ...
DurableList c = ...

a.element = 10;


c.element = 11;


b.element = 12;
```

AutoPersist Persistency – Example

```
**Elements in persistent list**
DurableList a = ...
DurableList b = ...
DurableList c = ...

a.element = 10;
Clwb(a.element); Sfence

c.element = 11;
Clwb(c.element); Sfence

b.element = 12;
Clwb(b.element); Sfence
```

AutoPersist Persistency – Example

```
**Elements in persistent list**
DurableList a = ...
DurableList b = ...
DurableList c = ...

a.element = 10;
~~Clwb(a.element); Sfence~~

c.element = 11;
~~Clwb(c.element); Sfence~~

b.element = 12;
~~Clwb(b.element); Sfence~~
```

AutoPersist Persistency – Example

```
**Persistent Int Array**
int[] parray = ...

**removing the front of array**


for(int i = 1; i < parray.length; i++){

    parray[i-1] = parray[i];

}
```

AutoPersist Persistency – Example

```
**Persistent Int Array**
int[] parray = ...

**removing the front of array**

Create Undo Log
for(int i = 1; i < parray.length; i++){
    log original value and location
    parray[i-1] = parray[i];
    writeback parray index
}
Clear Undo Log
```

AutoPersist Persistency – Example

```
**Persistent Int Array**
int[] parray = ...

**removing the front of array**
BeginFailureAtomic
Create Undo Log
for(int i = 1; i < parray.length; i++){
    Log original value and location
    parray[i-1] = parray[i];
    Writeback parray index
}
Clear Undo Log
EndFailureAtomic
```

Modifying JVM Runtime

To provide its runtime support, AutoPersist:

1. Extends the semantics of several JVM bytecodes to perform AutoPersist's runtime actions
2. Adds an extra object header word to contain persistent state metadata

Evaluation Environment

- Modify Maxine 2.0.5
  - Research JVM originally developed by Oracle
- Run on system containing two 24-core Intel® second generation Xeon® Scalable processors (codenamed Cascade Lake)
- System has 128GB Intel Optane DC persistent memory modules
- Compare against Espresso [Wu et al. ASPLOS'18]
  - Persistent objects must be explicitly allocated in NVM
  - CLWBs and SFENCEs must be manually inserted into the code

Evaluation Applications

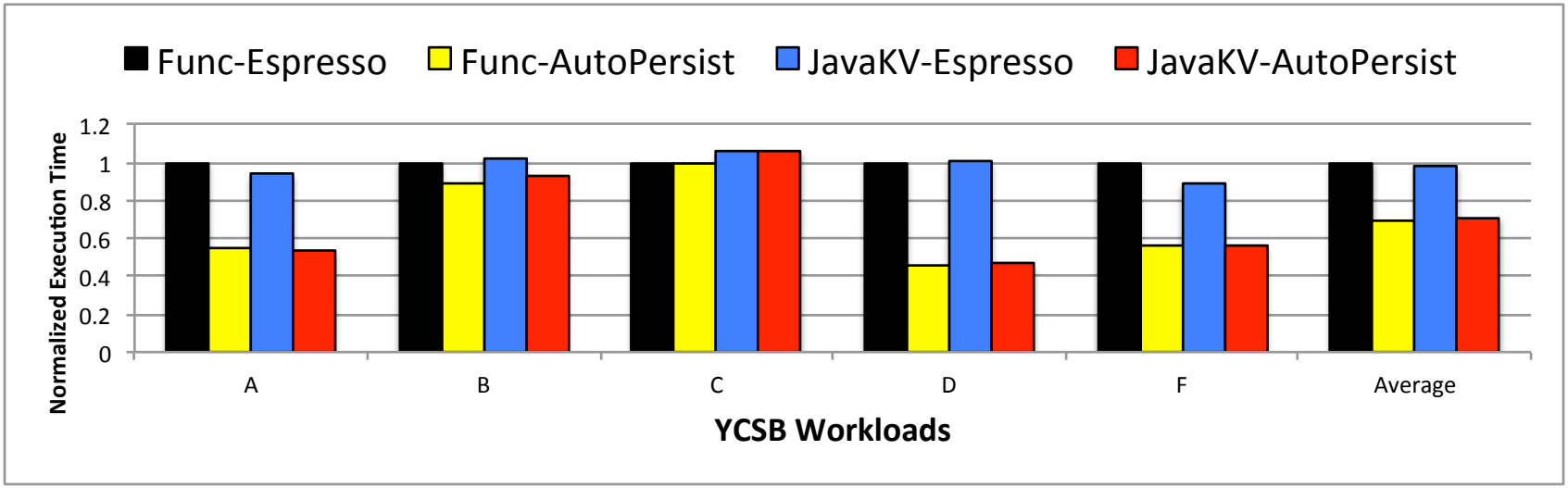
- KVStore: Key-Value Store Server
  - Modify implementation to use AutoPersist-annotated data structures
  - Test with two data structures
    - **JavaKV**: B^+ Tree implementation
    - **Func**: Functional Hashmap
  - Evaluate KVStore performance running YCSB workloads
- Kernels: simple data structures
  - Collection of different arraylist and linked list implementations
  - Run random sequence of get, set, update, and delete operations on each data structure

Comparison of # Marks Needed

Framework	KVStore Applications		
	Func	JavaKV	Total
AutoPersist	4	6	10
Espresso	55	45	100

Framework	Kernels					
	MArray	MList	FARArray	FArray	FList	Total
AutoPersist	1	1	5	1	1	9
Espresso	49	48	63	47	14	321

AutoPersist Results – KVStore

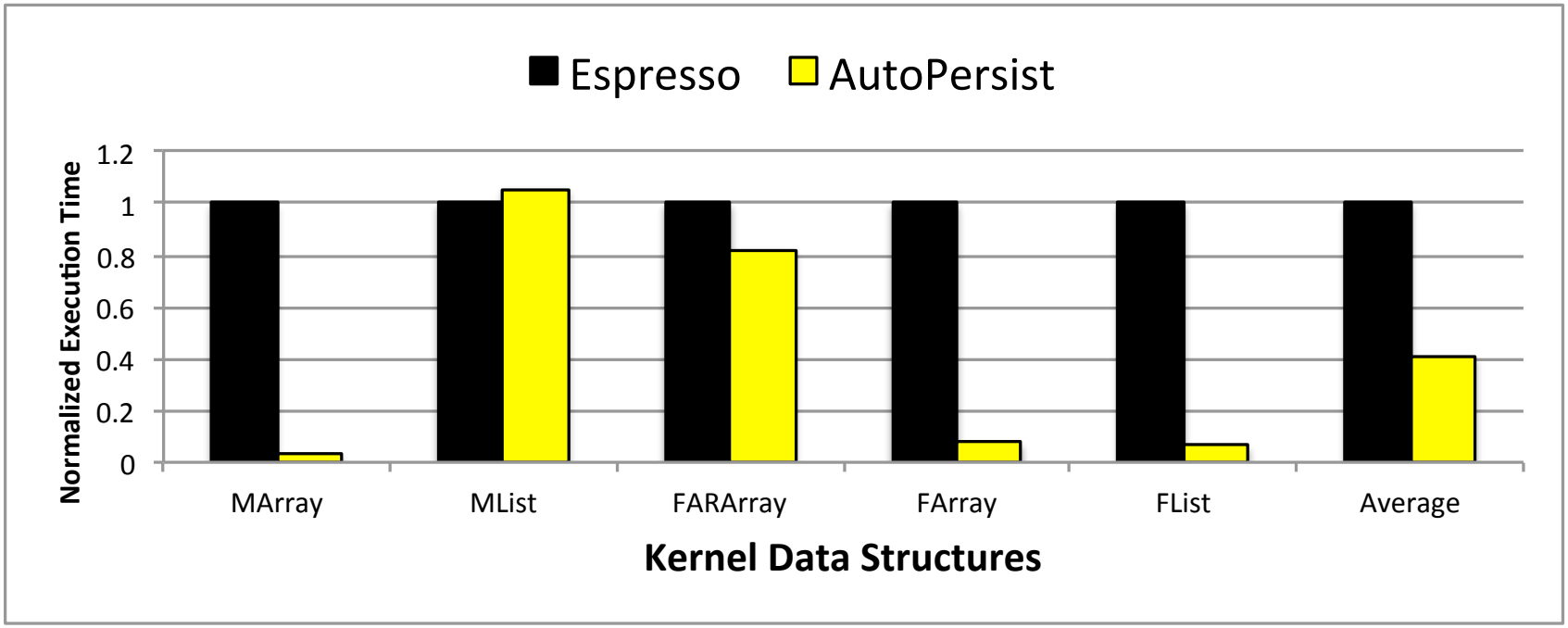


- On average, AutoPersist is **31%** and **28% faster** than Espresso for the Func and JavaKV implementations
- Significant performance improvements for update-heavy workloads (A, D, F)

Explaining AutoPersist's Performance Advantages

- In Java, the machine-level layout of objects is hidden from the programmer
- Makes it expensive to persist entire objects in Espresso
  - Must conservatively insert CLWBs to each field name
- In AutoPersist, when an object becomes reachable from a **@durable_root**, then the object is made persistent efficiently
  - Because AutoPersist knows the object's internal layout, only needs to insert one CLWB per cacheline
- **Takeaway**: AutoPersist is able to limit the number of CLWBs needed when initially persisting objects

AutoPersist Results – Kernels



- On average, AutoPersist is **59% faster** than Espresso
- MArray, FArray, and FList create many new objects for insertions
  - Highlights AutoPersist's efficient object persisting mechanism

Also in the Paper

- More details about AutoPersist's programming model (Introspection API, Recovery API)
- In depth explanation of how to ammend the JVM bytecode semantics
- Algorithms for how to concurrently move objects to NVM in a thread-safe manner
- Profiling techniques to improve AutoPersist's performance
- Additional evaluation – (H2, Intel's `pmemkv`)

AutoPersist Summary

- NVM Challenge: difficult to program
- Solution: introduce AutoPersist
  - Needs minimal markings
  - Leverages the JVM runtime to automatically perform much of the complicated actions associated with NVM
  - Outperforms existing Java NVM frameworks by minimizing # of CLWBs when persisting complete objects