Developing Programmer-Friendly Frameworks for NVM

Thomas Shull
University of Illinois at Urbana-Champaign

July 23, 2019
PIRL'19

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 and ties in naturally to load-store interfaces

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
- Software logging is needed to ensure the failure-atomicity of a collection of writes
- Solution: use **frameworks** to hide much of the complexity

Presentation Outline

1. Explain drawbacks of current NVM frameworks
2. Present some of my work on creating a new high-level Java NVM framework
3. Describe lessons learned throughout my research
4. Highlight several important unsolved problems

Current Techniques for Persistent Applications

- Manual – add explicit assembly instructions and system calls
- Persistent Buffer – persistent memory is exposed as a buffer
- Library Frameworks – provide hooks to make persistent data structures
  - Persistent Memory Development Kit (PMDK) offers C (libpmemobj), C++ (lipmemobj-cpp), and Java (PCJ) bindings
  - Provide support for failure-atomic regions, have some persistent data structures already implemented

Current NVM Frameworks – Example

```
struct durable_list{
  int element;
  struct durable_list* next;
}
struct durable_list* head;\\initialized elsewhere

void insert(int element){
  struct durable_list* node =
    malloc(sizeof(struct durable_list));
  node->element = element;
  node->next = head->next;
  head->next = node;
}
```

Current NVM Frameworks – Example

```
POBJ_LAYOUT_BEGIN(list);
POBJ_LAYOUT_ROOT(list, struct durable_list);
POBJ_LAYOUT_END(list);
struct durable_list{
  int element;
  TOID(struct durable_list) next;
}
TOID(struct durable_list) head; \\initialized elsewhere
void insert(PMEMobjpool *pop, int element){
  TX_BEGIN(pop){
    TOID(struct durable_list) node =
        TX_NEW(struct durable_list);
    D_RW(node)->element = element;
    TX_ADD_FIELD(head, next);
    D_RW(node)->next = D_RO(head)->next;
    D_RW(head)->next = node;
  } TX_END
}
```

Current NVM Frameworks – Example

```
struct durable_list{
  p<int> element;
  persistent_ptr<durable_list> next;
}
persistent_ptr<durable_list> head; \\initialized elsewhere;

void insert(pool_base &pop, int element){
  transaction::run(pop, [&] {
    auto node = make_persistent<durable_list>();
    node->element = element;
    node->next = head->next;
    head->next = node
  });
}
```

Misalignment with High-Level Languages

NVM frameworks are not designed for managed languages like Java, C#, Scala, etc.

- Cannot use existing built-in libraries
  - Built-ins do not contain necessary persistent markings
- Expose low-level features to programmer
  - Does not abstract away enough details

Solution: Create a New NVM Framework

- Solution to existing shortcomings: Design a new 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 Overview

- Call our new framework AutoPersist
  - Framework can automatically perform much of the work needed for creating a persistent application
- AutoPersist currently targets Java
- Implemented within the Maxine JVM
  - Research JVM originally developed by Oracle
  - Allows for fast prototyping of features
- AutoPersist can easily be extended to support other JVM-based languages

Desirable Traits

Three important programmability goals for AutoPersist:

1. Require minimal markings by programmer
2. Making libraries and other pre-existing codes persistent should be simple
3. Failure-atomic support should be provided and need only minimal markings

New NVM Framework Highlights

- In AutoPersist, 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

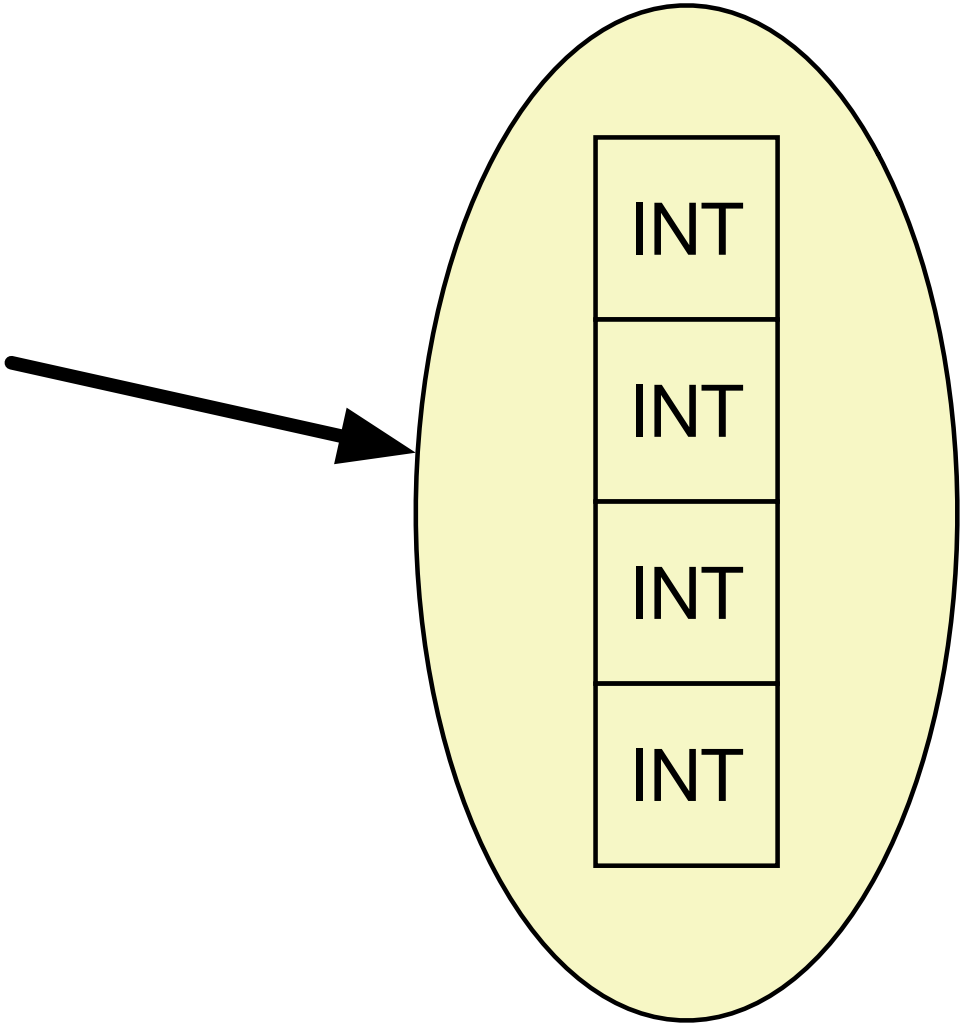
AutoPersist Runtime Responsibilities

AutoPersist's runtime automatically:

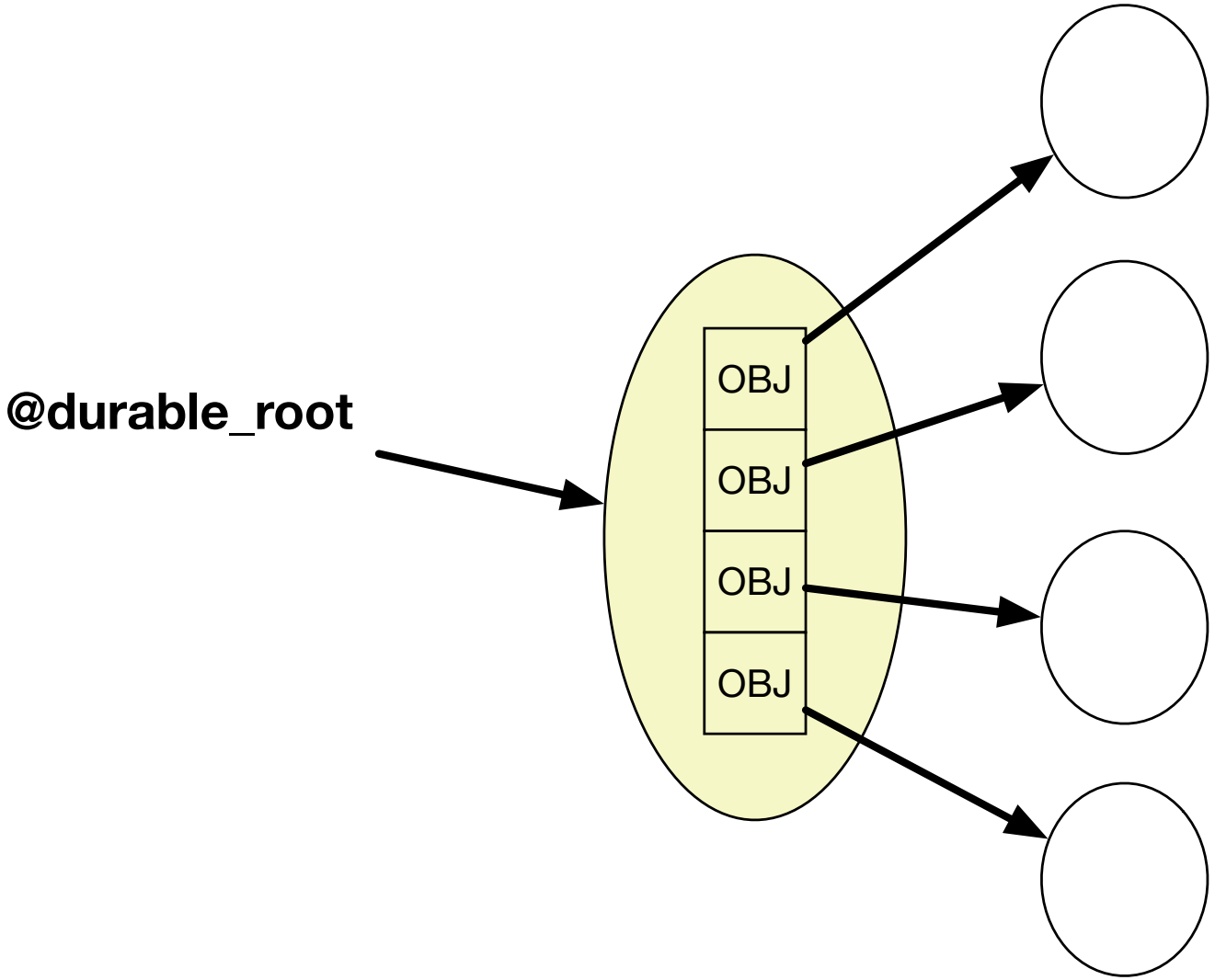
1. Dynamically detects and monitors the *transitive closure* of durable roots
   - Ensures everything reachable from a durable root is in NVM
   - Dynamically moves objects to NVM throughout execution
2. Ensure stores to persistent objects are performed correctly
   - Automatically inserts the needed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

Identifying Persistent Data Structures

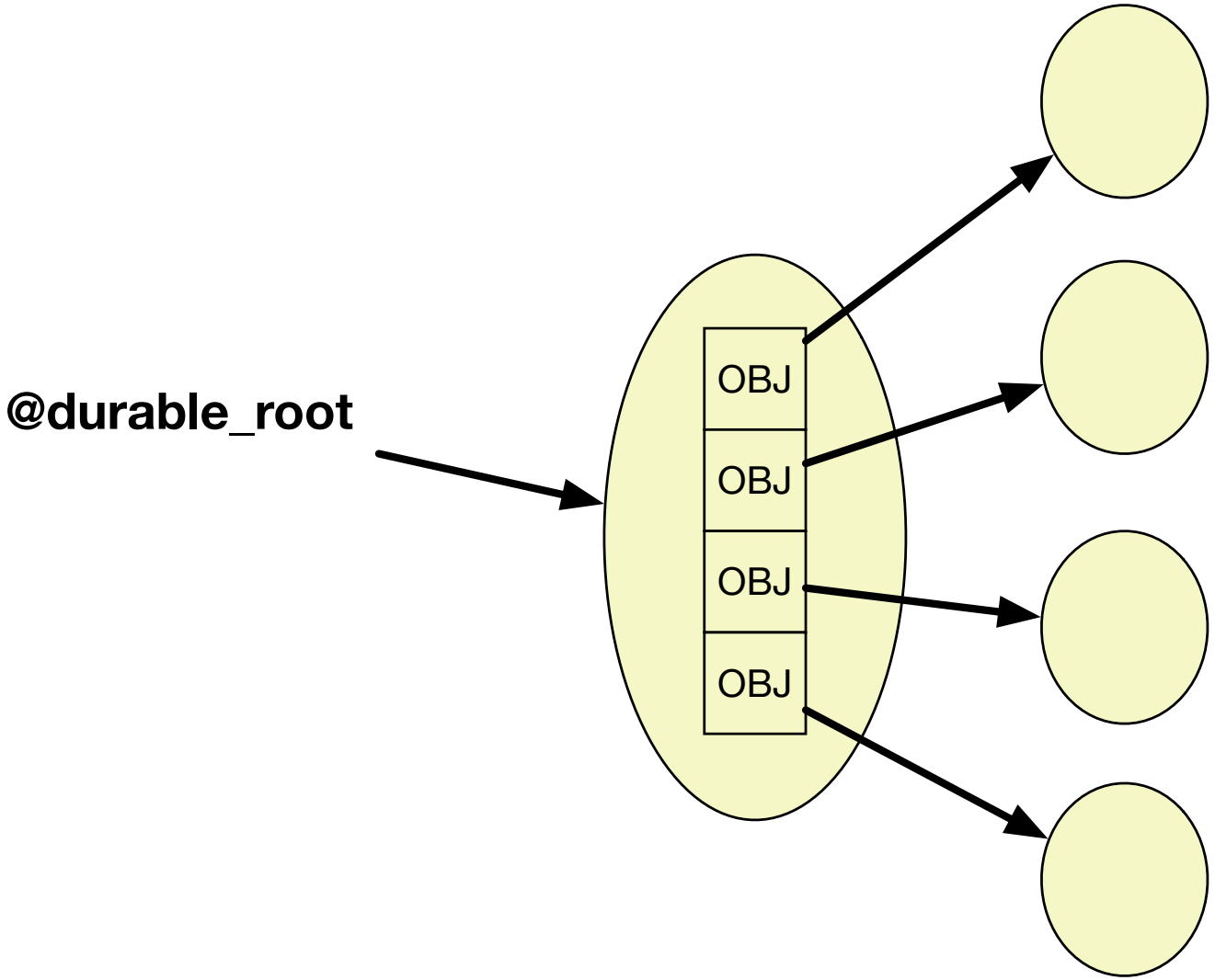
@durable_root



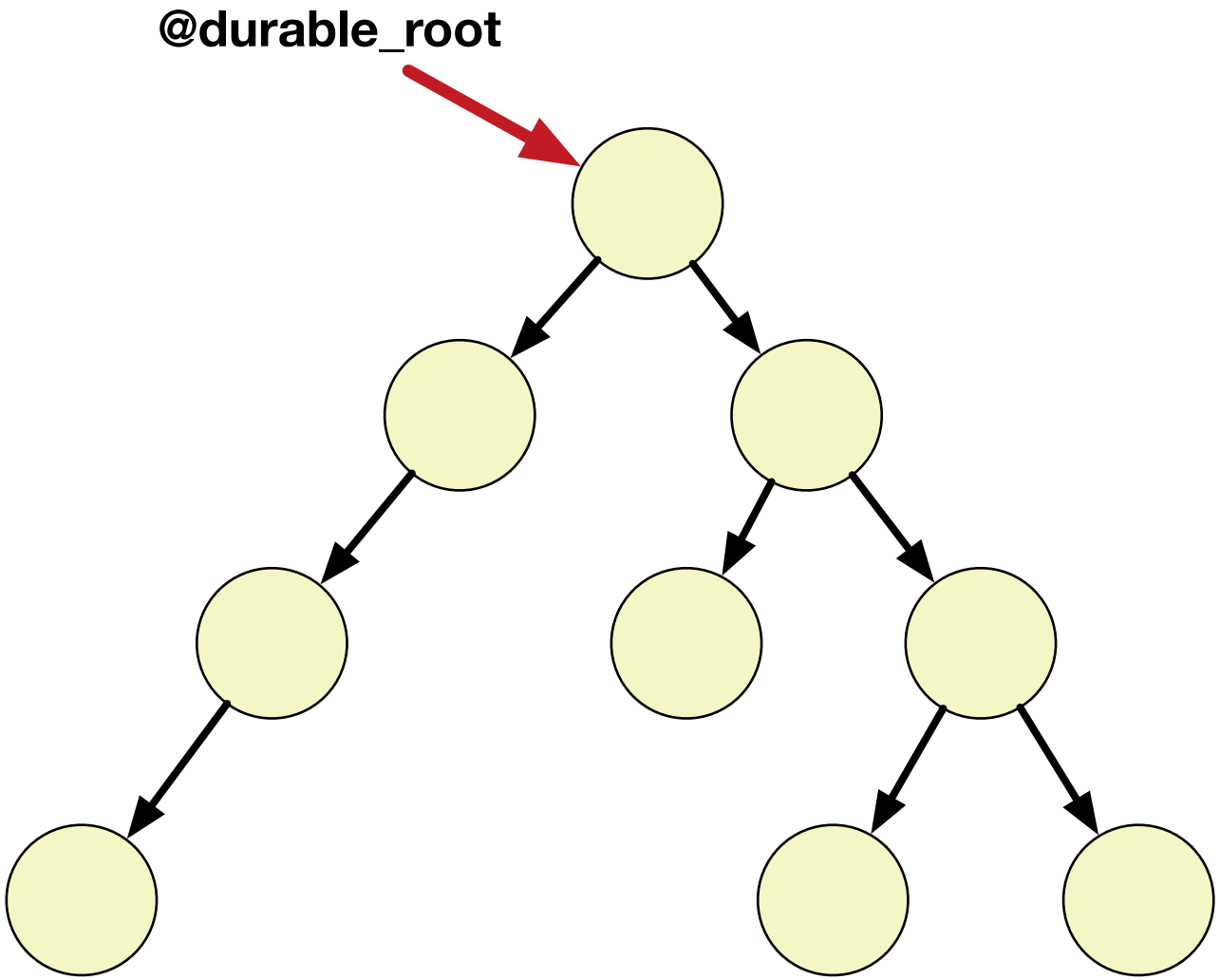
Identifying Persistent Data Structures



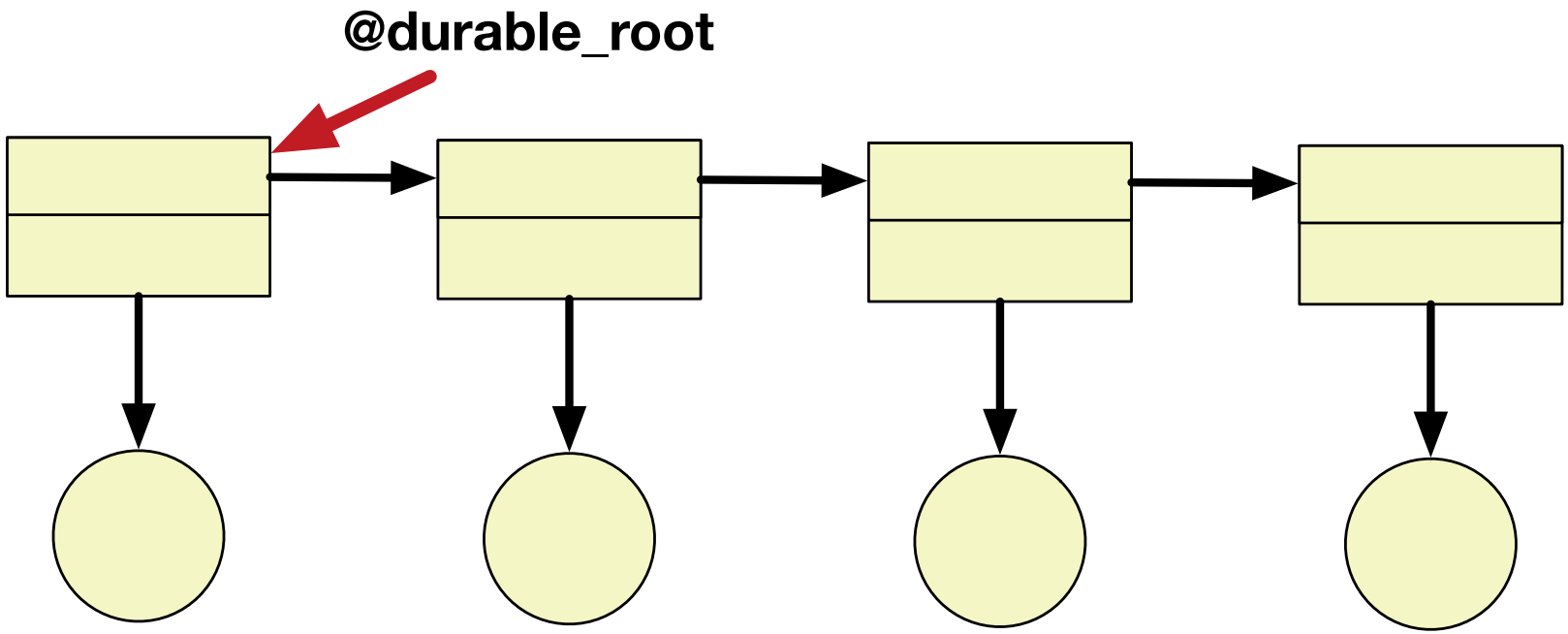
Identifying Persistent Data Structures



Identifying Persistent Data Structures



Identifying Persistent Data Structures



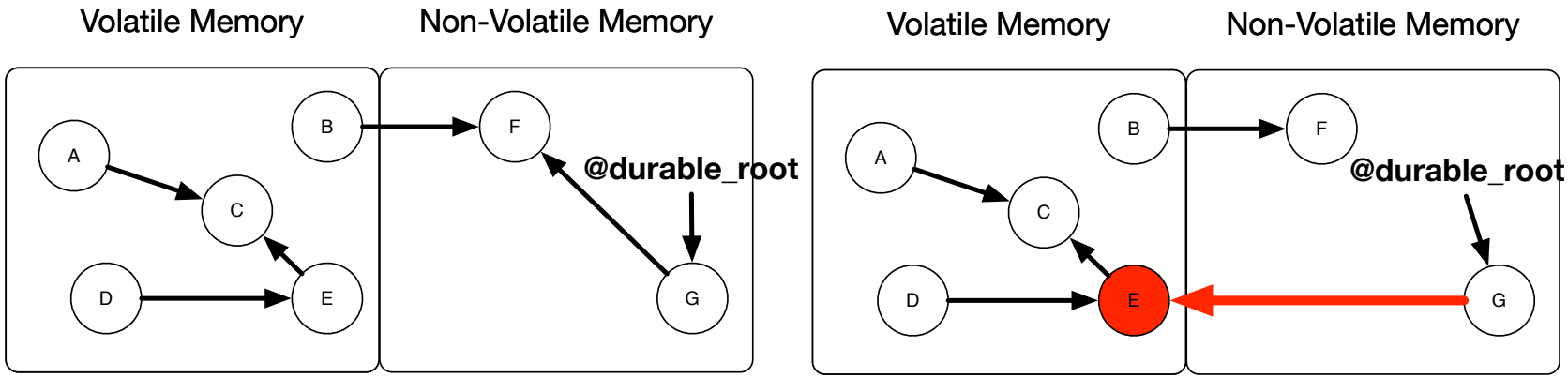
Programming in AutoPersist – Example

```
class DurableList{
  @durable_root
  public static DurableList head; \\initialization elsewhere

  private int element;
  private DurableList next;

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

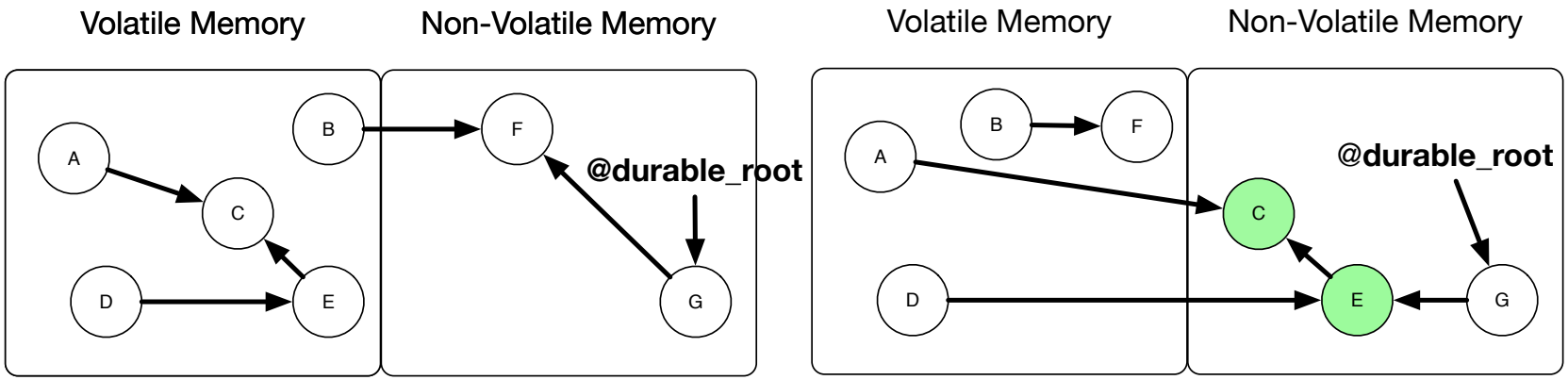
Dynamically Moving Objects to NVM



(a) Initial Heap State

(b) Model Violation

Dynamically Moving Objects to NVM



(a) Initial Heap State

(b) Runtime Maintaining Correct Heap State

Ensuring Stores are Made Persistent

AutoPersist Persistency Model:

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

Implementing New Model

- Model requires many guarded actions before accesses
  - Storing to **@durable_root**?
  - Storing to an object reachable from a **@durable_root**?
  - In a failure-atomic region?

Solution:

1. Add extra object header word to contain persistent state and metadata
2. Extend the semantics of several JVM bytecodes to perform the necessary checks and guarded actions

Optimizing Implementation

Our implementation collects profiling information to limit the performance overhead:

- Limit check overhead
- Preallocate objects in NVM

Evaluation Environment

- Modify Maxine version 2.0.5
- 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

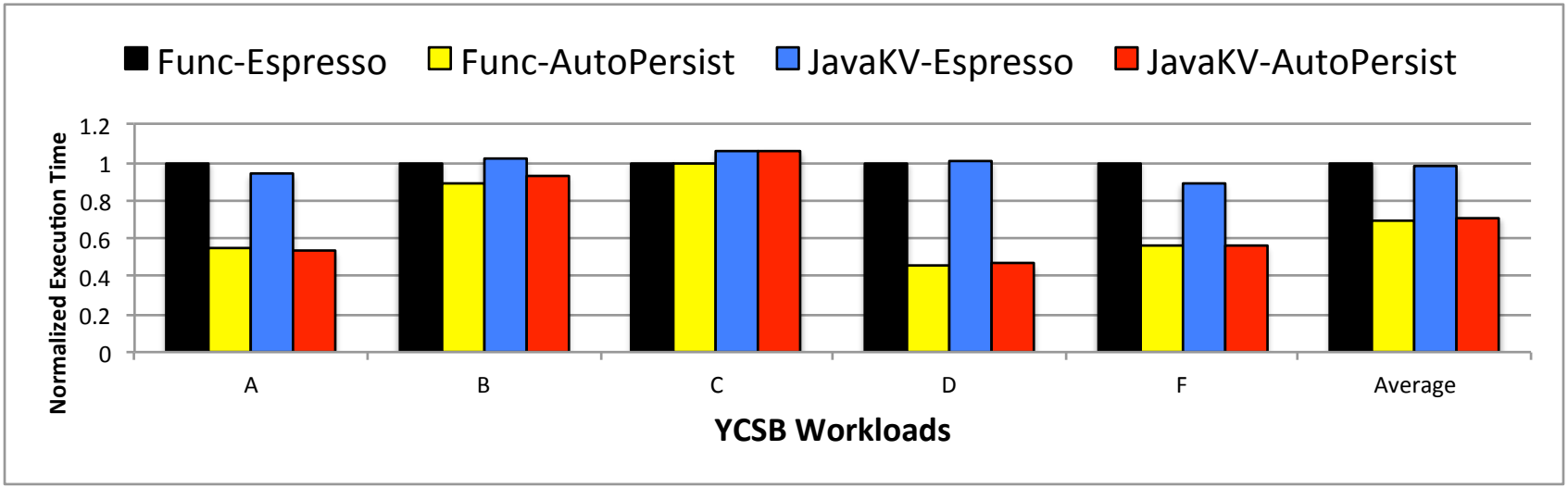
- KVStore: Key-Value Store Server
  - Modify implementation to use AutoPersist-annotated data structures
  - Test with two data structures
    - **JavaKV**: B^+ Tree implementation
    - **Func**: Functional Hashmap
  - Evaluate KVStore performance running YCSB workloads
- Kernels: simple data structures
  - 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

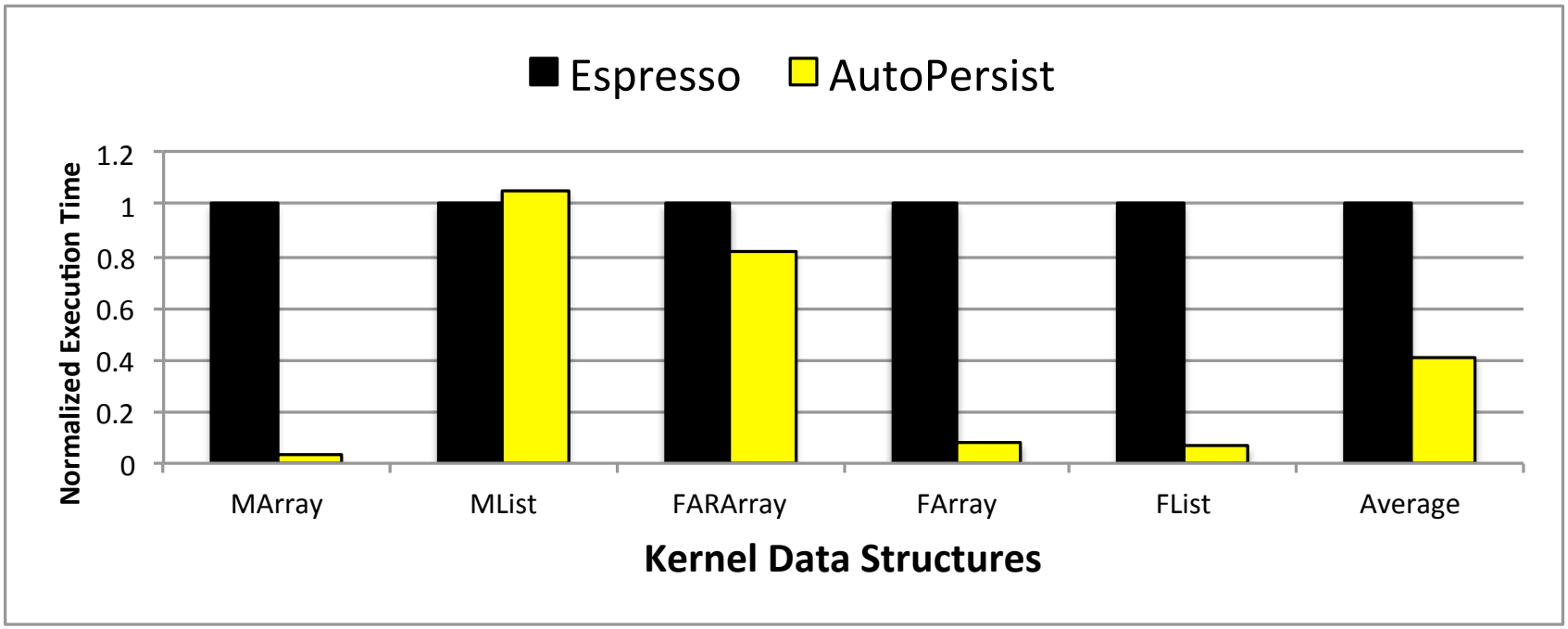


- On average, AutoPersist is **31%** and **28% faster** than Espresso for the Func and JavaKV implementations
- Significant performance improvements for update-heavy workloads (A, D, F)

Explaining AutoPersist's Performance Advantages

- In Java, the machine-level layout of objects is hidden from the programmer
- Makes it expensive for programmer 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

Papers About Our Model & Framework

Motivating the need for new Java-specific NVM programming model – **ManLang'18**: Defining a High-level Programming Model for Emerging NVRAM Technologies

How to limit our framework's runtime check overhead on volatile objects – **VEE'19**: QuickCheck: Using Speculation to Reduce the Overhead of Checks in NVM Frameworks

How our framework dynamically moves objects between DRAM and NVM – **PLDI'19**: AutoPersist: An Easy-To-Use Java NVM Framework Based on Reachability

Additional Topics Covered in Papers

- 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
- Additional evaluation – (H2, Intel's `pmemkv`)

Experiences with NVM – Programming

- Manually identifying persistent objects is hard
  - Easy to fail to mark a field
  - I was dealing with very simple applications and still made multiple mistakes!
- Easy to miss inserting a CLWB and SFENCE

Experiences with NVM – Performance

- CLWB and SFENCEs are expensive
  - It is crucial to minimize their occurance
- Hard to minimize their occurance within libraries
  - libmemobj: performs much software analysis to minimize # of CLWBs
  - libmemobj-cpp: performs per-field logging and lives with the consequences

Takeaway – Need Language-Level Support

- Language-Level Support is necessary
  - Need to be able to persistently use existing built-in data structures and API
- Shouldn't have to declare which fields are persistent or when to persist them
- Built-in support for failure-atomic regions is a must

Takeaway – Need Compiler Support

- Must consider compiler opportunties
- Libraries are not good enough
  - Especially as performance of NVM improves one cannot get away with runtime software tricks
  - Compiler cannot optimize CLWB and SFENCE pragmas well
- Compilers can potentially effectively reduce overhead of crash-consistency
  - Identify situations where CLWBs can be coalesced or removed
  - Move around CLWBs and SFENCEs to have minimal impact
  - Reduce the overhead of logging

Multi-Threaded Programmability Concerns (1)

- Support for failure-atomic regions is a must
- Problem: what about the undo logs of two threads race
  - For example, programmer wraps multiple critical sections into same failure-atomic region
- Big question: is this a correct or incorrect program?
- In current frameworks ⇒ incorrect program
  - Can consider expanding the data-race free programming model into persistency-race free
- Other option: incorporate notion of persistent race detection and rollback into failure-atomic regions
  - Persistency-flavored software transactional memory

Multi-Threaded Programmability Concerns (2)

- Alternative: use lock-free algorithms
- Problem: how to implement this
  - Hardware synchronization primitives are for *visibility*, not persistency
- Solution: to be determined
  - Wait for hardware primitives
  - Take software measures
    - Wrap atomic load/store instructions
    - Reader can always persist before taking action
    - Tagging can be used to identify the object state
- If using managed language with NVM support, then the implementation can change as (if) new hardware features become available

General Reachability Problems

- Some data within NVM may inherently be transient
  - Locks - should be reinitialized across restarts
- Other objects may have application-specific characteristics
  - e.g. Timers, File Handles, Sockets
  - Can provide default behaviors, but must have way for application developer to intervene
- Threads: contain a combination of state and data
  - Three options: **resume** from current spot, **restart** from beginning, **forget** – do not restart
    - Resume: snapshots likely too much overhead
    - Restart: useful if thread is performing cron-like task
    - Forget: right behavior if thread is performing transient task
  - Once again, application developer should provide input

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
- Language-integrated frameworks have much promise for both programmability and performance
- Many problems still need to be solved