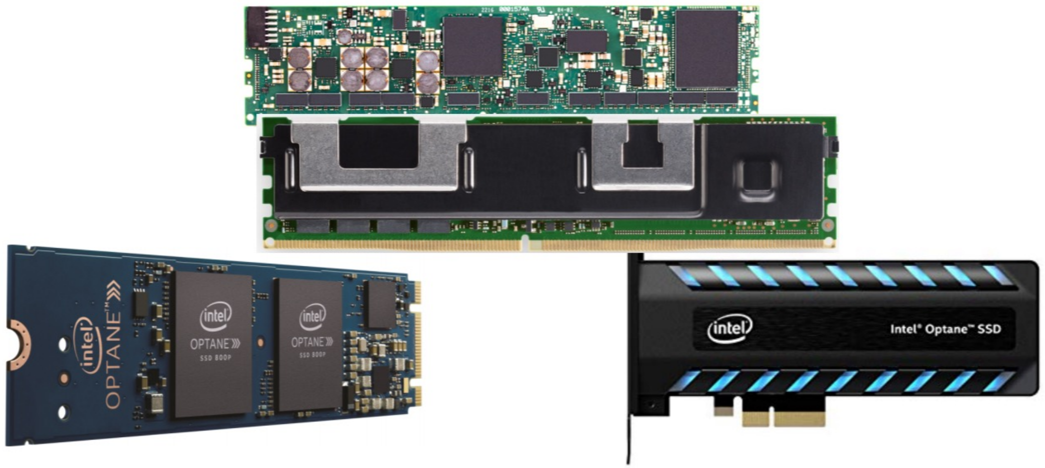
UniHeap : Managing Persistent Objects Across Managed Runtimes for Non-Volatile Memory

Daixuan Li Benjamin Reidys Jinghan Sun
Thomas Shull Josep Torrellas Jian Huang

System Platform Research Group at UIUC
ECE ILLINOIS

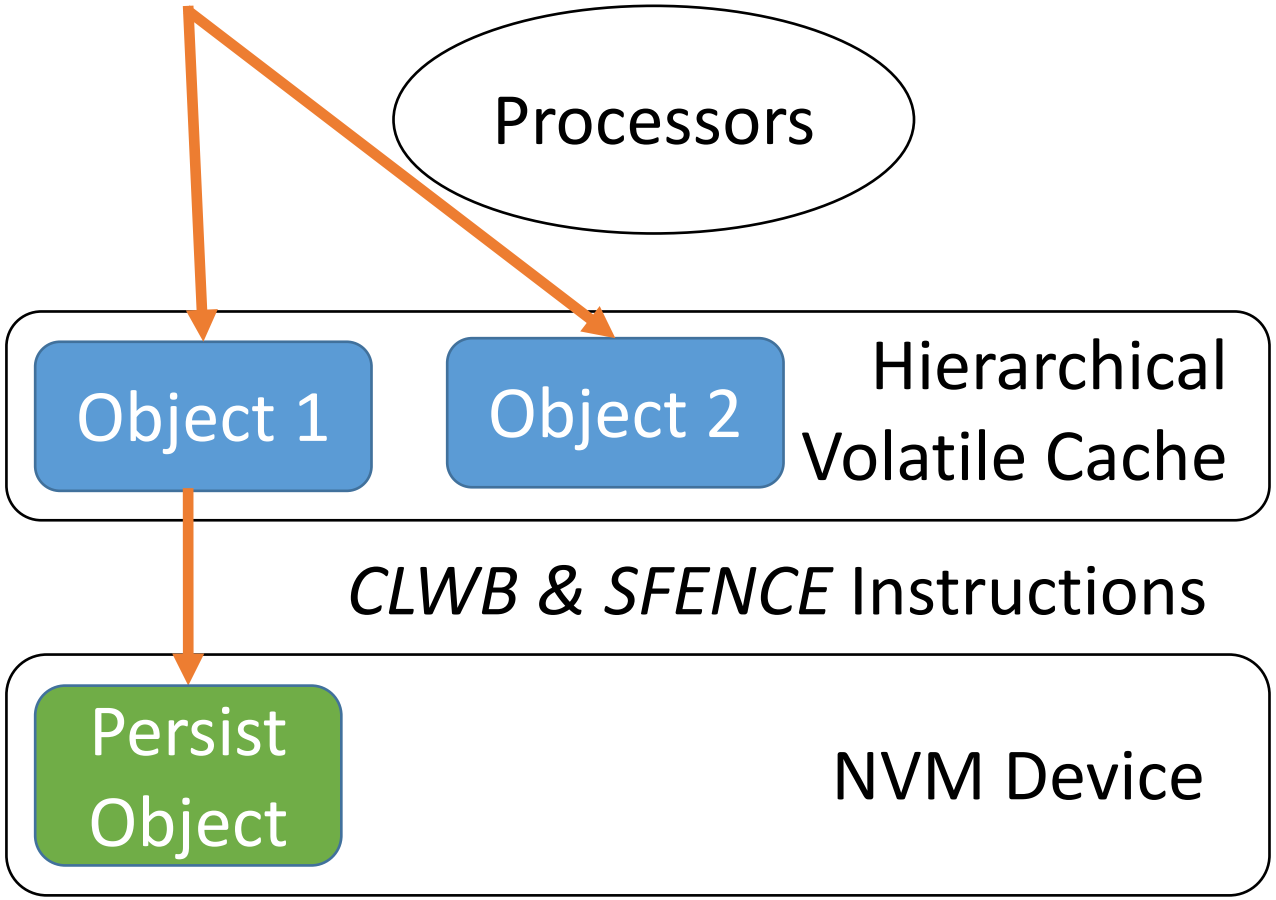
Non-Volatile Memory: Opportunities & Challenges



- Performance & Persistency
- Byte-Addressable
- Data Durability
- Programmability

Programmability Challenge of NVM

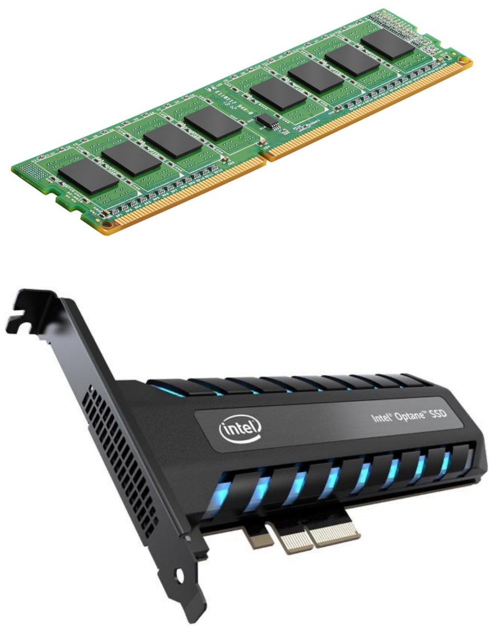
Mark all the persistent object updates in the code



Correctness Problem

Performance Bugs

Leveraging Managed Runtime to Manage NVM



Managed Runtime

python

Java

JavaScript

Hardware Complexity

Managed Data Objects

Popular Programming Models

AutoPersist: An Easy-to-Use NVM Framework


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

Thomas Shull, University of Illinois at Urbana-Champaign, shull1@illinois.edu
Jian Huang, University of Illinois at Urbana-Champaign, jianh@illinois.edu
Josep Torrellas, University of Illinois at Urbana-Champaign, torrella@illinois.edu

Abstract

Byte-addressable, non-volatile memory (NVM) is emerging as a revolutionary memory technology that provides persistency, near-DRAM performance, and scalable capacity. To facilitate its use, many NVM programming models have been proposed. However, most models require programmers to explicitly specify the data structures or objects that should reside in NVM. Such requirement increases the burden on programmers, complicates software development, and introduces opportunities for correctness and performance bugs.

We believe that requiring programmers to identify the data structures that should reside in NVM is untenable. Instead, programmers should only be required to identify *durable roots* – the entry points to the persistent data structures at recovery time. The NVM programming framework should then automatically ensure that all the data structures reachable from these roots are in NVM, and stores to these data structures are persistently completed in an intuitive order.

To this end, we present a new NVM programming framework, named *AutoPersist*, that only requires programmers to identify durable roots. AutoPersist then persists all the data structures that can be reached from the durable roots in an automated and transparent manner. We implement AutoPersist as a thread-safe extension to the Java language and perform experiments with a variety of applications running on Intel Optane DC persistent memory. We demonstrate that AutoPersist requires minimal code modifications, and significantly outperforms expert-marked Java NVM applications.

CCS Concepts • **Hardware** → **Non-volatile memory**; • **Software and its engineering** → **Just-in-time compilers; Source code generation.**

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6712-7/19/06...$15.00
https://doi.org/10.1145/3314221.3314608

Keywords Java, Non-Volatile Memory, JIT Compilation

ACM Reference Format:
Thomas Shull, Jian Huang, and Josep Torrellas. 2019. AutoPersist: An Easy-To-Use Java NVM Framework Based on Reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3314221.3314608

1 Introduction

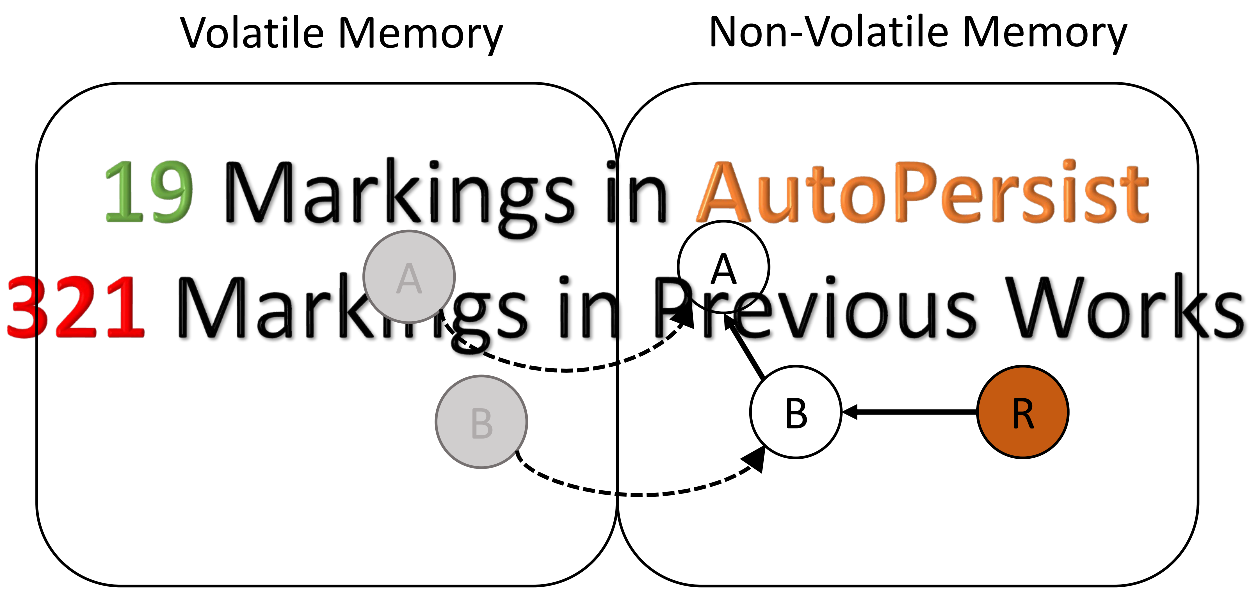
There have recently been significant technological advances towards providing fast, byte-addressable non-volatile memory (NVM), such as Intel 3D XPoint [37], Phase-Change Memory (PCM) [52], and Resistive RAM (ReRAM) [10]. These memory technologies promise near-DRAM performance, scalable memory capacity, and data durability, which offer great opportunities for software systems and applications.

To enable applications to take advantage of NVM, many NVM programming frameworks have been proposed, such as Intel PMDK [6], Mnemosyne [60], NVHeaps [21], Espresso [62], and others [20, 23, 25, 35, 48]. While the underlying model to ensure data consistency [20, 50] varies across frameworks, all of these frameworks share a common trait: they require the programmer to explicitly specify the data structures or objects that should reside in NVM. This limitation results in substantial effort from programmers, and introduces opportunities for correctness and performance bugs due to the increased programming complexity [53]. Moreover, it limits the ability of applications to use existing libraries.

We believe that requiring users to identify all the data structures or objects that reside in NVM is unreasonable. Instead, the user should only be required to identify the *durable roots*, which are the named entries into durable data structures at recovery time. Given this input, the NVM framework should then automatically ensure that all the data structures reachable from these durable roots are in NVM.

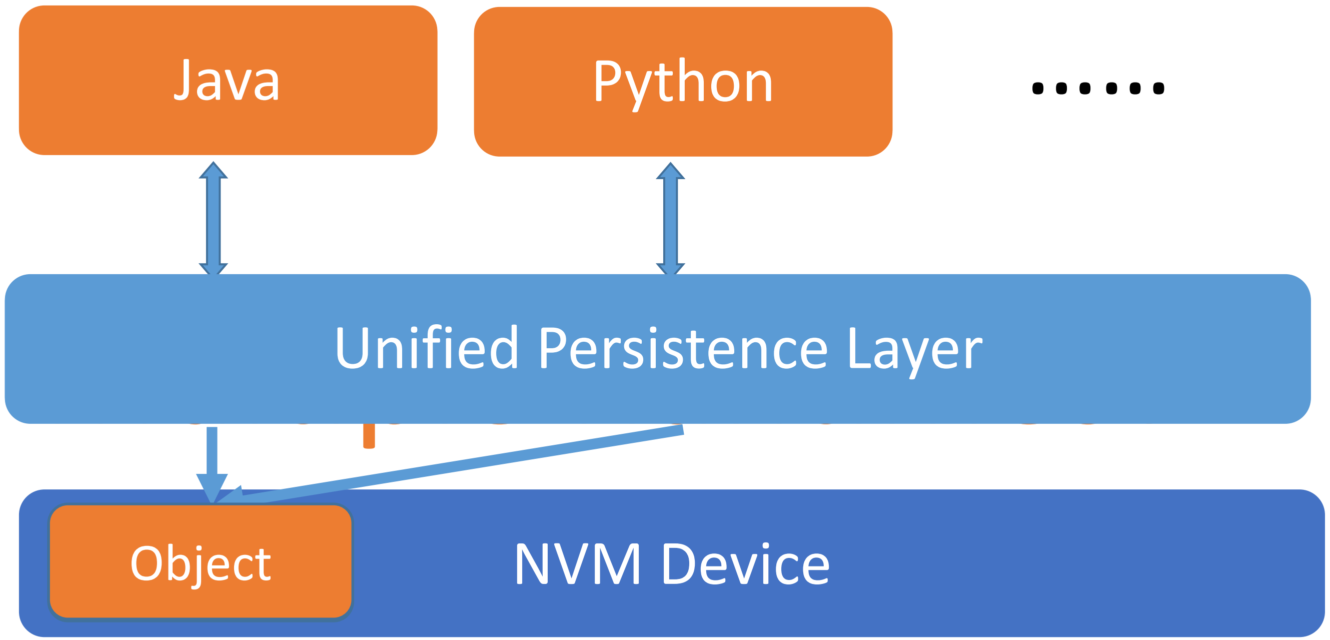
In this paper, we present a new NVM programming framework named *AutoPersist* that only requires programmers to identify the set of durable roots. While most NVM frameworks are implemented in C or C++, we choose to implement AutoPersist as an extension to the Java language. As is common for managed languages, Java already provides transparent support for object movement in memory, as well as high-level semantics for programmers.

```
@durable_root
public static KeyValueStore kv;
```



Programmability Improvement with Specifying Durable Roots

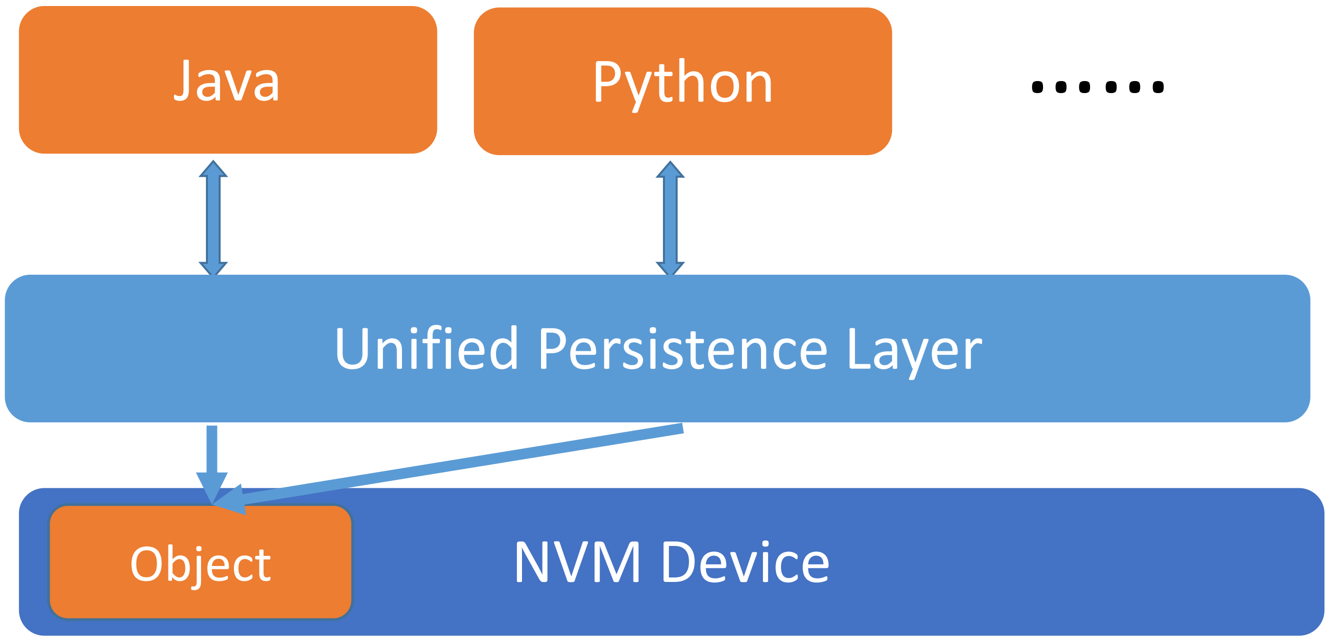
Managing Persistent Object Across Runtimes is Desirable



File System Enable Data Sharing with **File** Abstraction.

Managing Persistent Object Across Runtimes is Desirable

Enable Data Sharing with **Persistent Object** Abstraction.



File System Enable Data Sharing with **File** Abstraction.

Managing Persistent Object Across Runtimes is Desirable

Web Service

Java

python

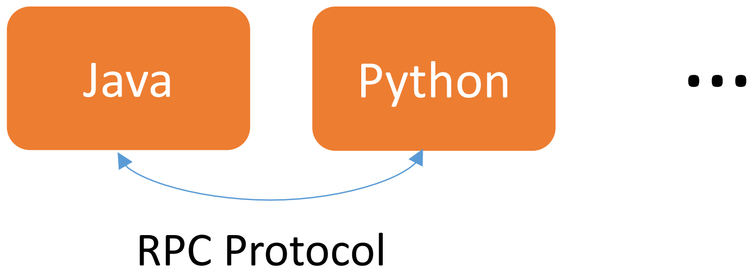
Shared Libraries

Data Analytics

Sharing persistent objects across multiple runtimes is Needed.

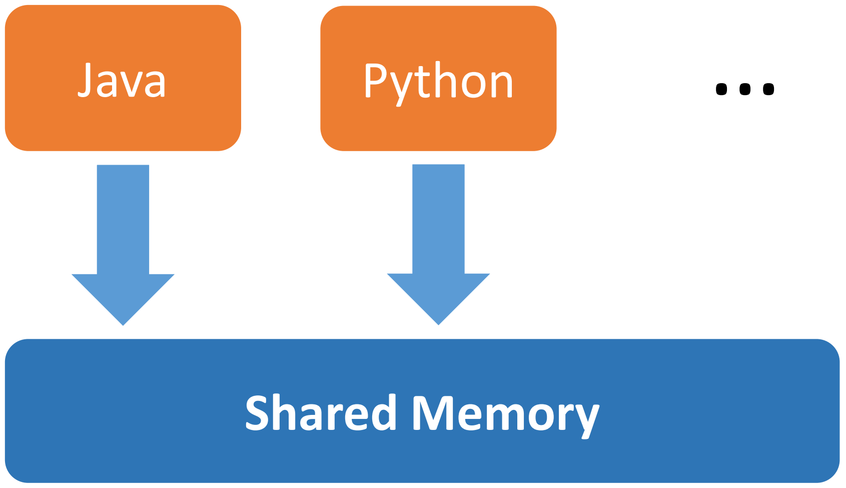
State-of-the-Art Object Sharing Approaches

- Thrift/Protobuf:



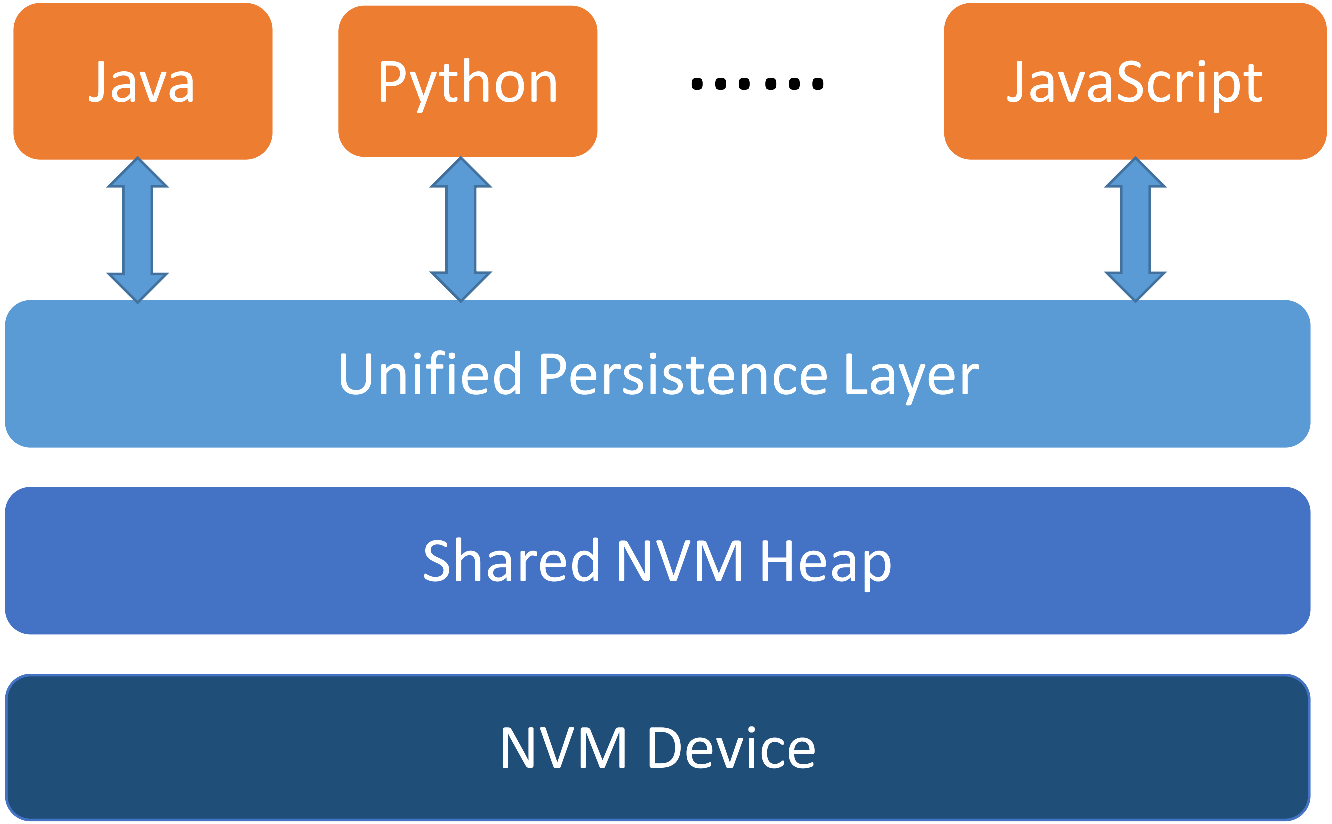
Serialization Overhead

- Shared Memory:

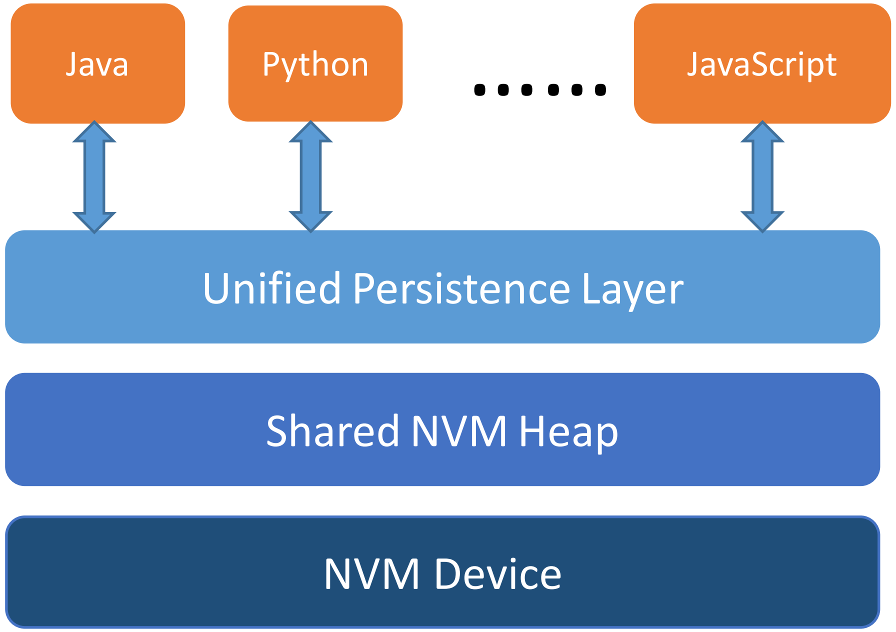


Does not support NVM

UniHeap: Managing Persistent Objects Across Runtimes

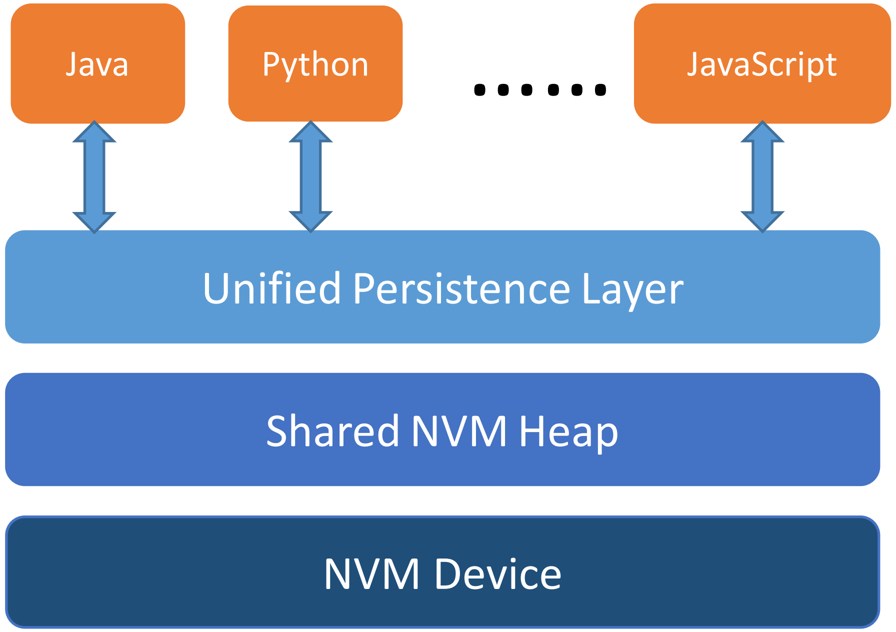


Challenges of Persistent Object Management Across Runtimes



- Unified Object Model
- Persistent and Crash-Safe Implementation
- Efficient and Correct GC

Challenges of Persistent Object Management Across Runtimes



Unified Object Model

Persistent and Crash-Safe Implementation

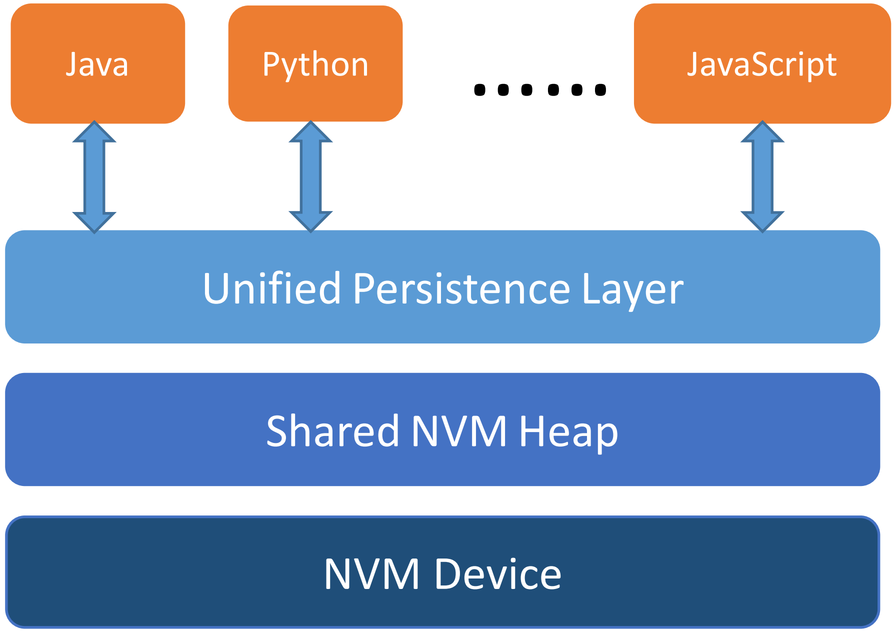
Efficient and Correct GC

Unified Object Model and Type System

Uniheap	char	short	int	long	float	double	reference
Java	boolean, byte	char	int	long	float	double	reference, array
Python	-	-	int	long	float	-	list, dict, tuple
JavaScript	boolean	-	num	num	num	num	array

Two built-in types: **numeral type** and **reference type**

Challenges of Persistent Object Management Across Runtimes

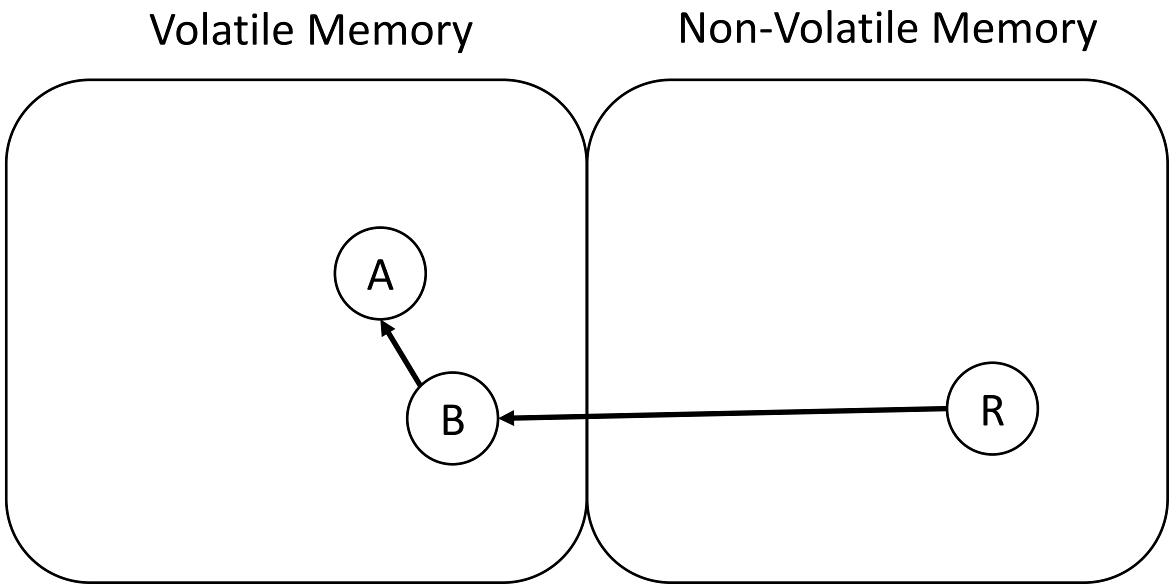


- Unified Object Model
- Persistent and Crash-Safe Implementation
- Efficient and Correct GC

Compatible with Automated Data Persistence Approach

Durable Root

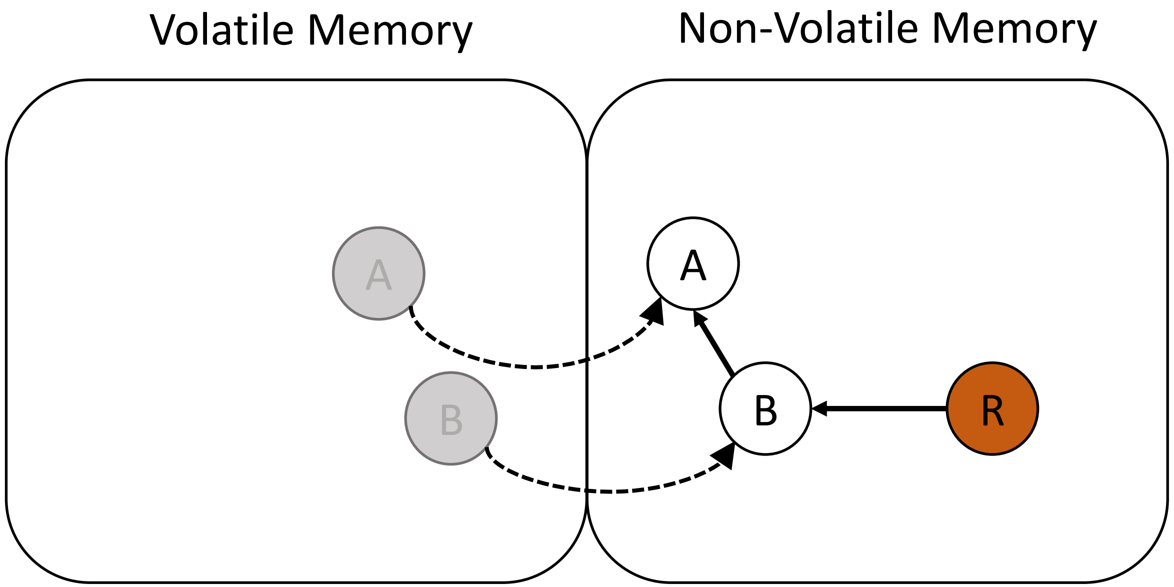
set_root



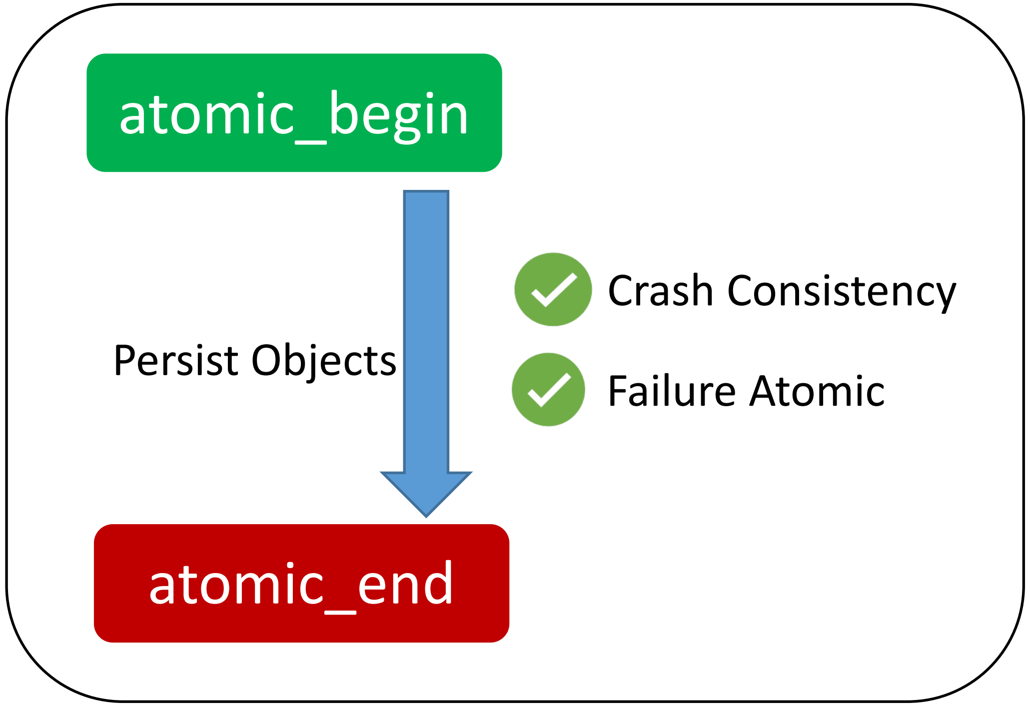
Compatible with Automated Data Persistence Approach

Durable Root

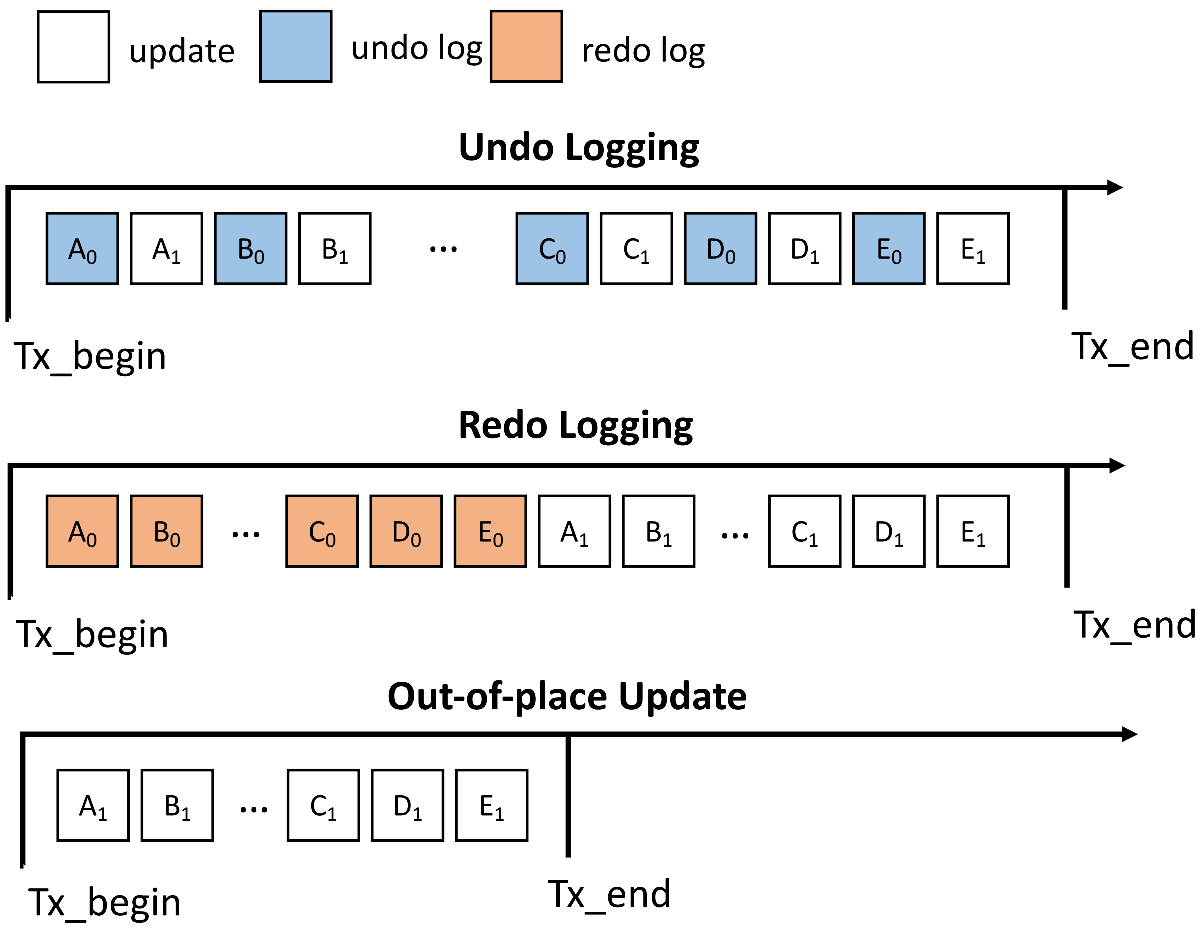
set_root



Atomic Region



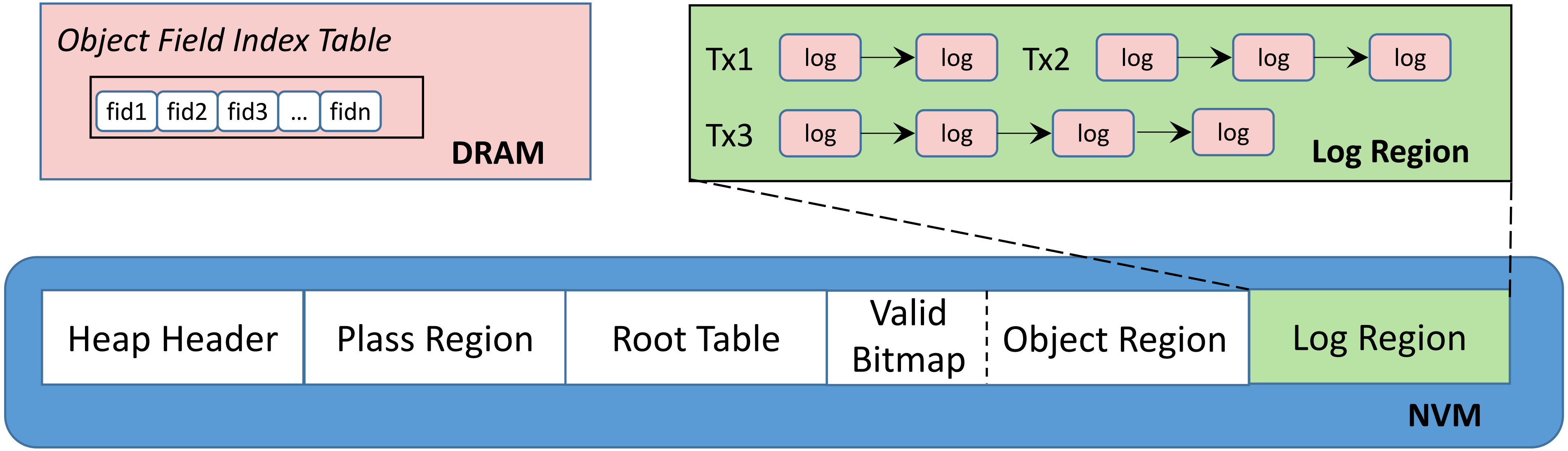
The Persistent Overhead of Managing Persistent Objects



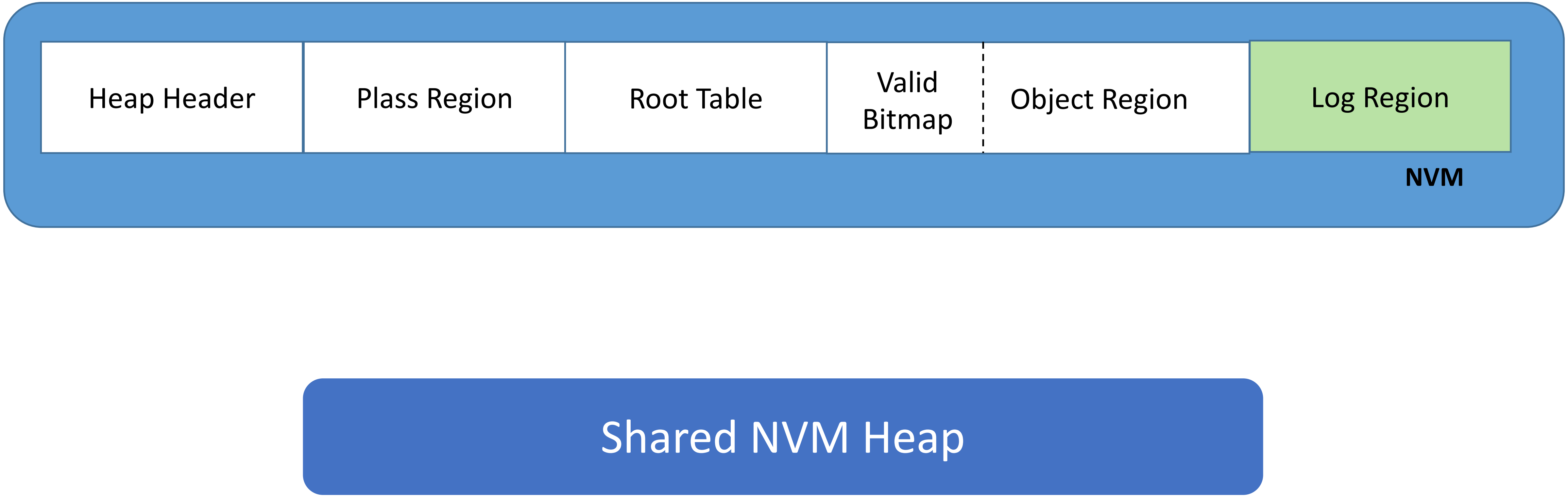
Redo and undo logs bring duplicate write overhead

Reduce Logging Overhead with Atomic Update and Out-of-Place Update

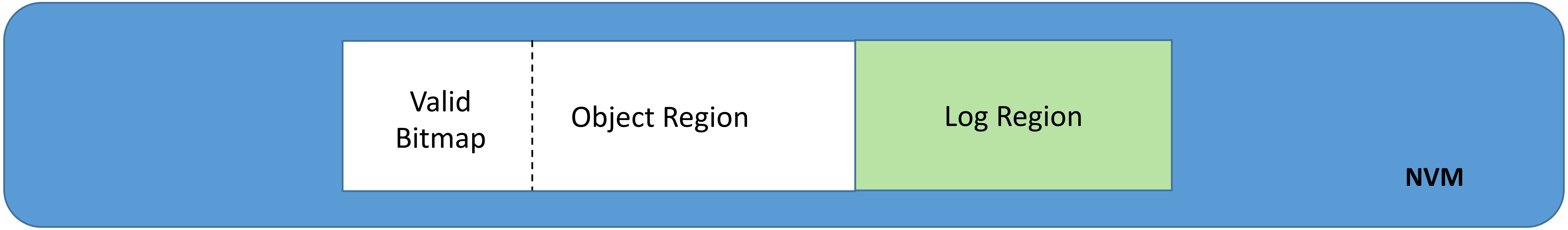
Managing Persistent Objects in A Log-Structured Manner



Managing Persistent Objects in A Log-Structured Manner

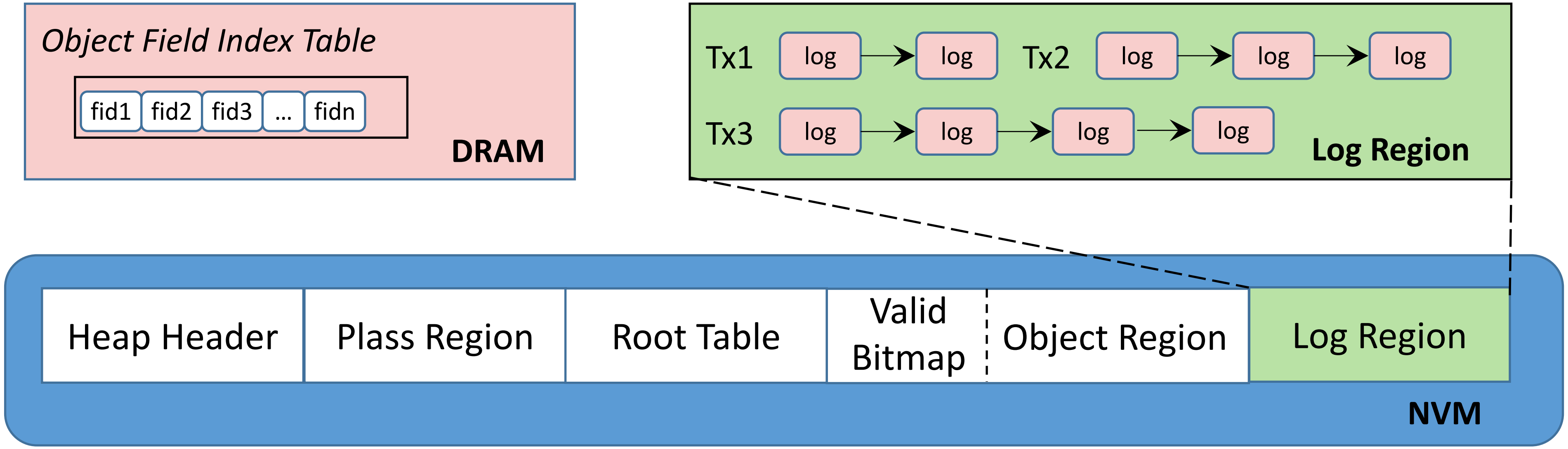


Managing Persistent Objects in A Log-Structured Manner

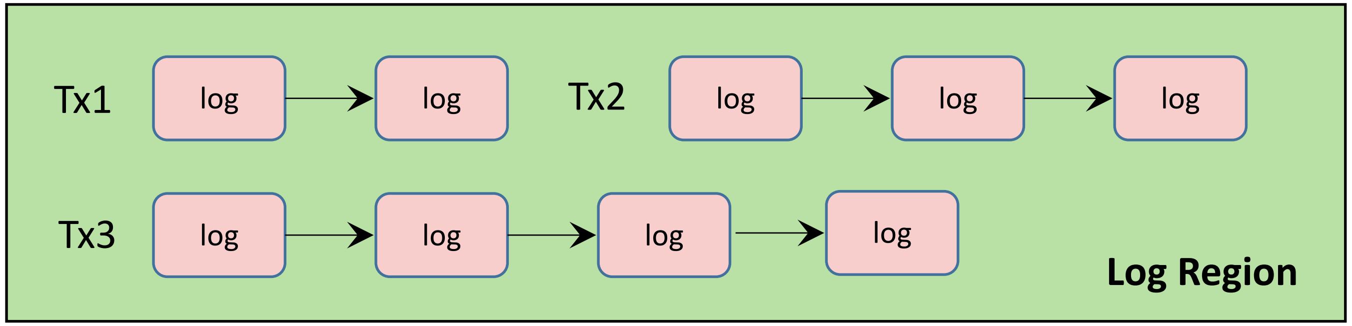


Decoupling the Data and Metadata of Objects

Managing Persistent Objects in A Log-Structured Manner



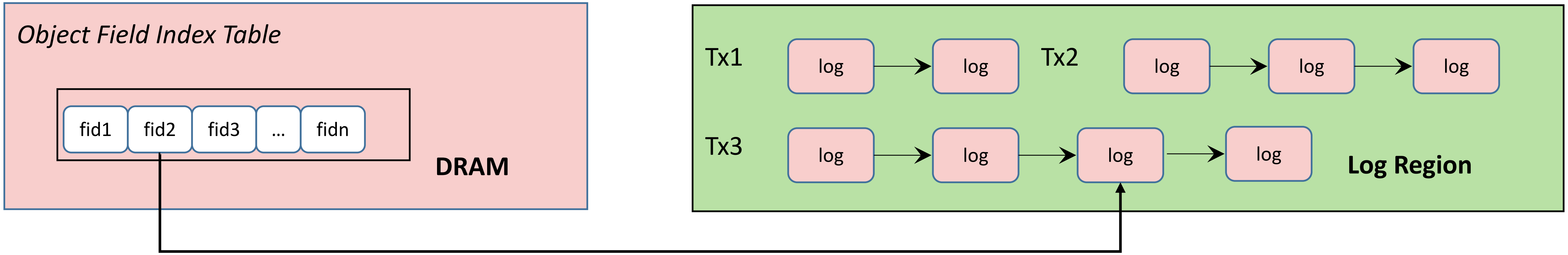
Managing Persistent Objects in A Log-Structured Manner



- Crash Safety
- Concurrent Access
- Garbage Collection

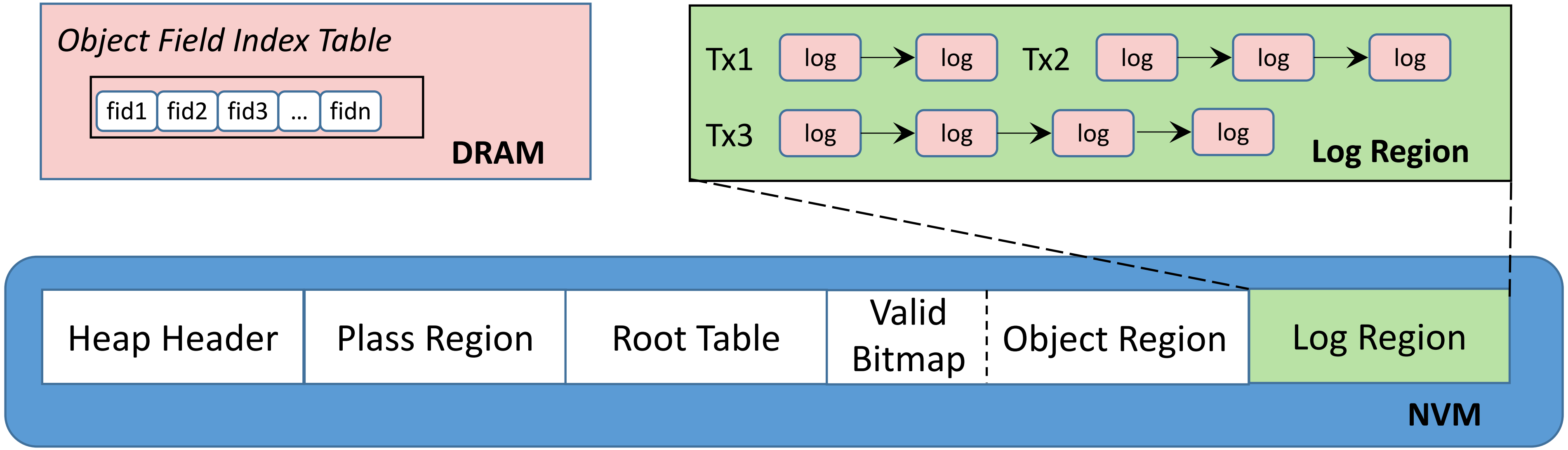
Transaction-based Out-of-place Object updates

Managing Persistent Objects in A Log-Structured Manner



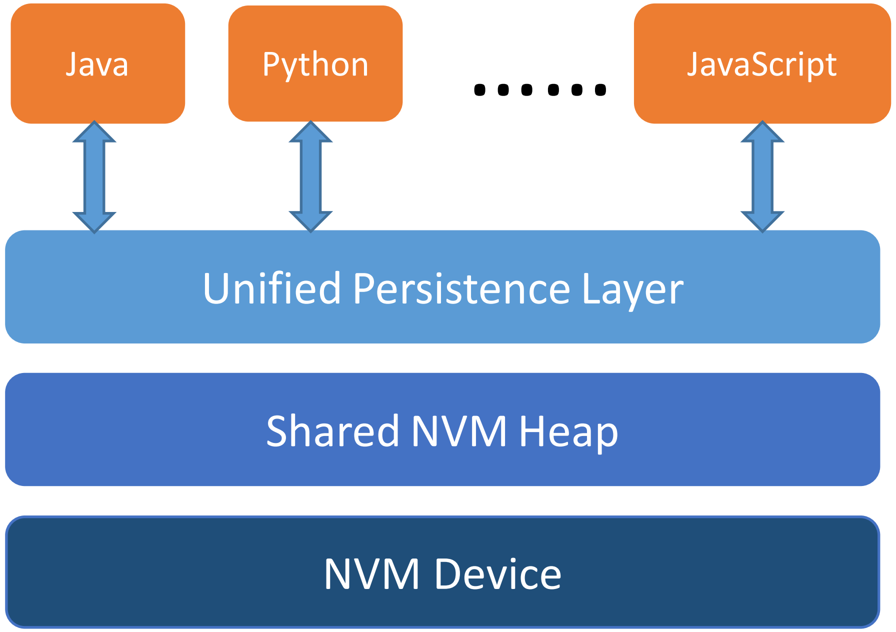
Address Remapping with Cached Index Table

Managing Persistent Objects in A Log-Structured Manner



Efficient and crash-safe persistent object management

Challenges of Persistent Object Management Across Runtimes

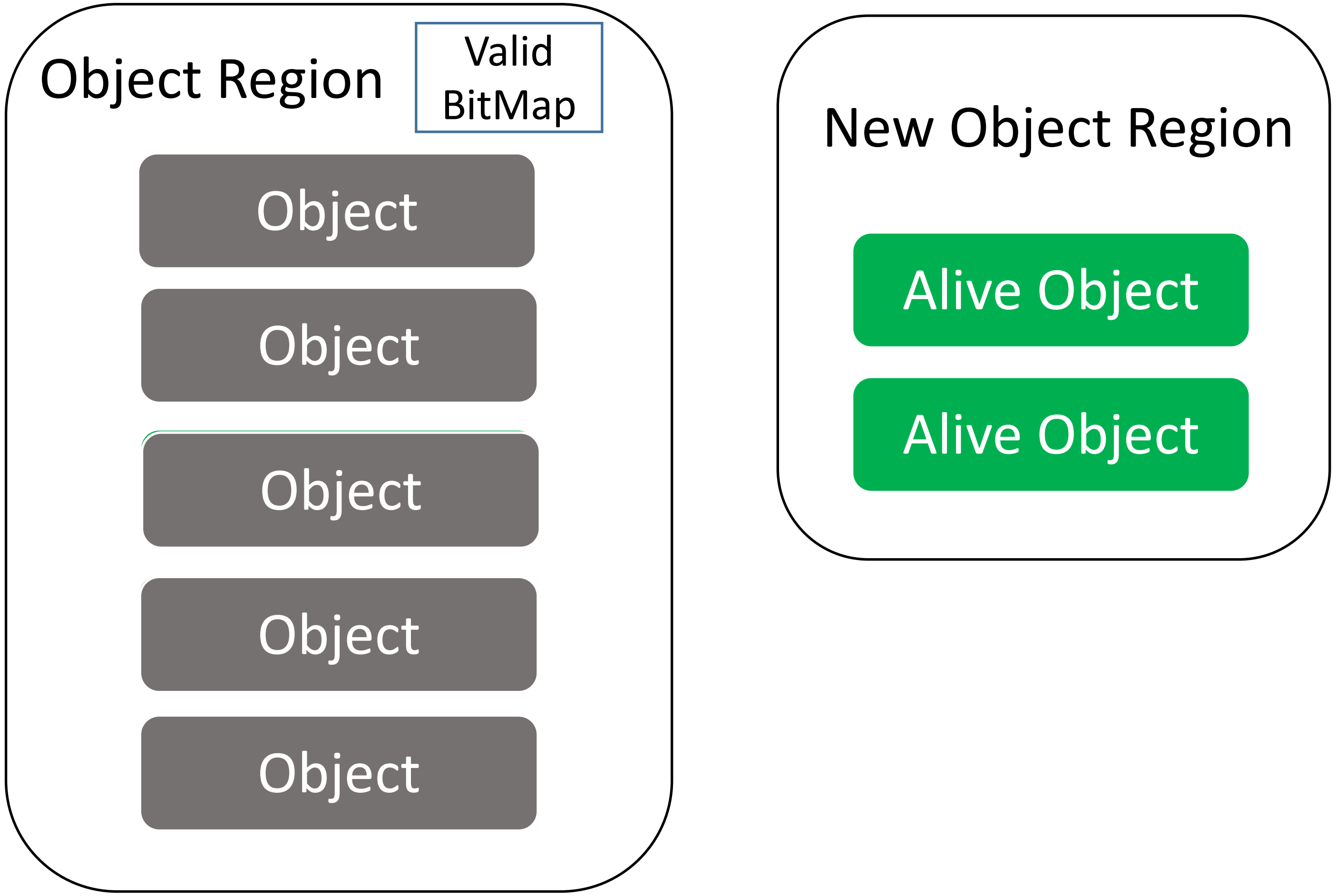


Unified Object Model

Persistent and Crash-Safe Implementation

Efficient and Correct GC

Garbage Collection of UniHeap



- Marking phase
- Relocation phase
- Compaction phase
- Clean-up phase

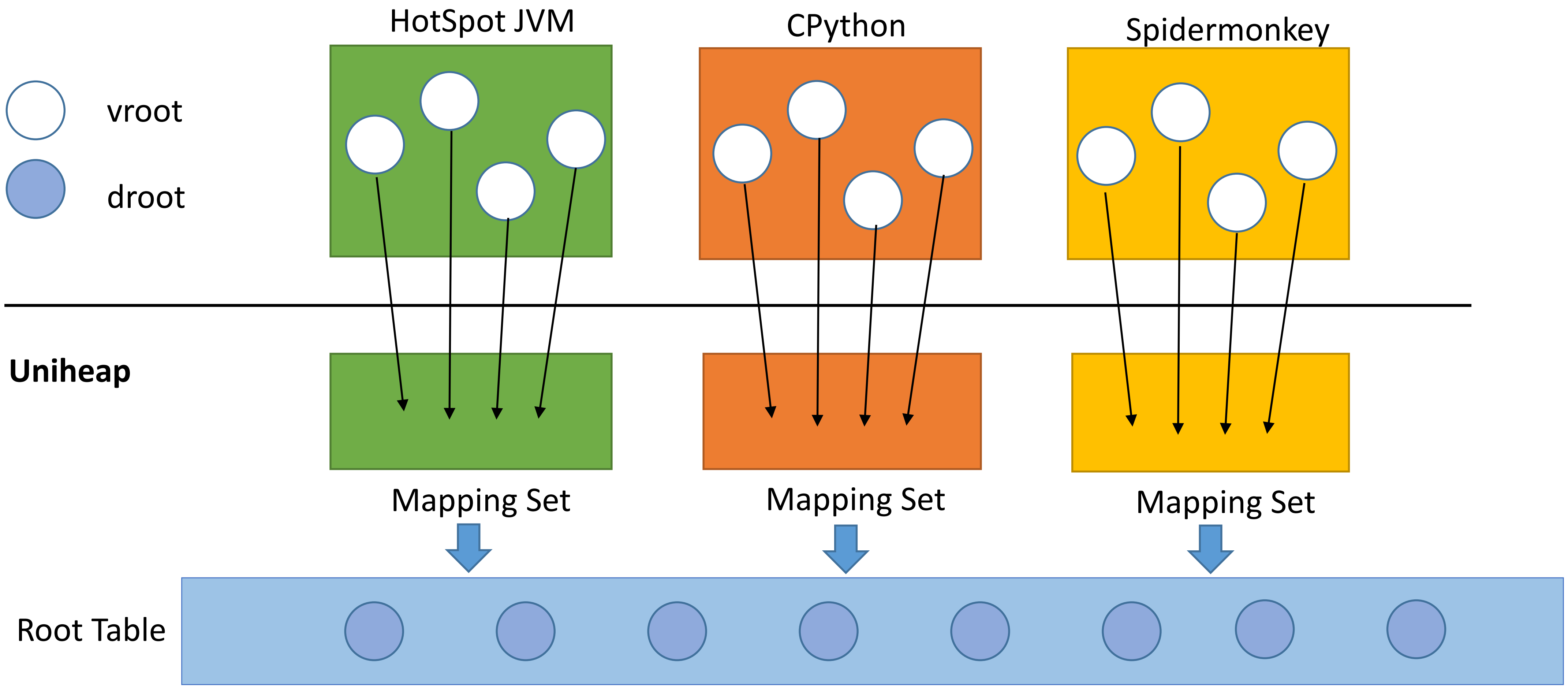
Garbage Collection of UniHeap

- Marking phase — ✔ **Naturally Crash-Safe**
- Relocation phase
- Compaction phase
- Cleanup phase

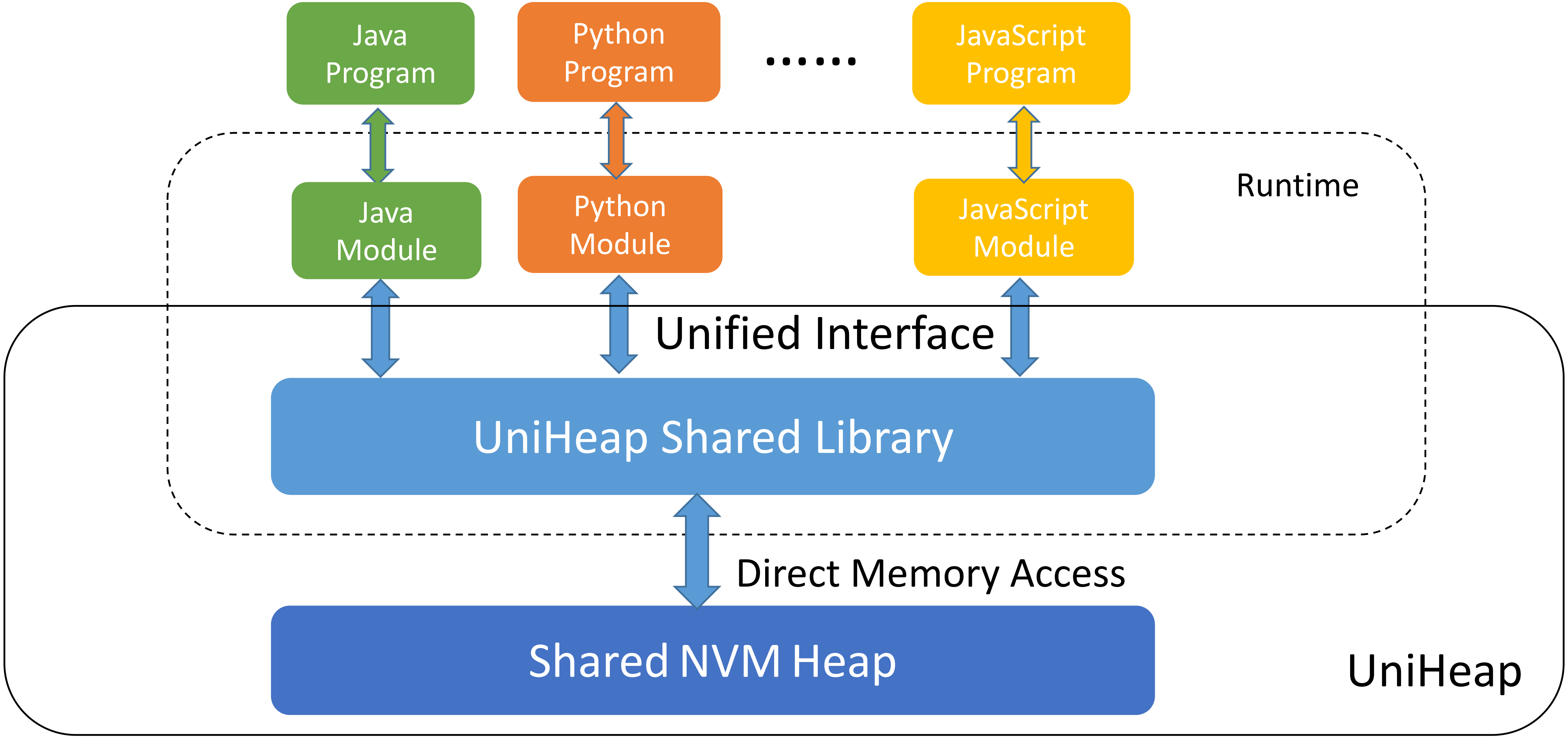
(Relocation, Compaction, Cleanup phases) ✔ **Keep old data until Clean up Phases**

Crash Safety of GC

Coordinated GC Across Managed Runtimes



Put It All Together



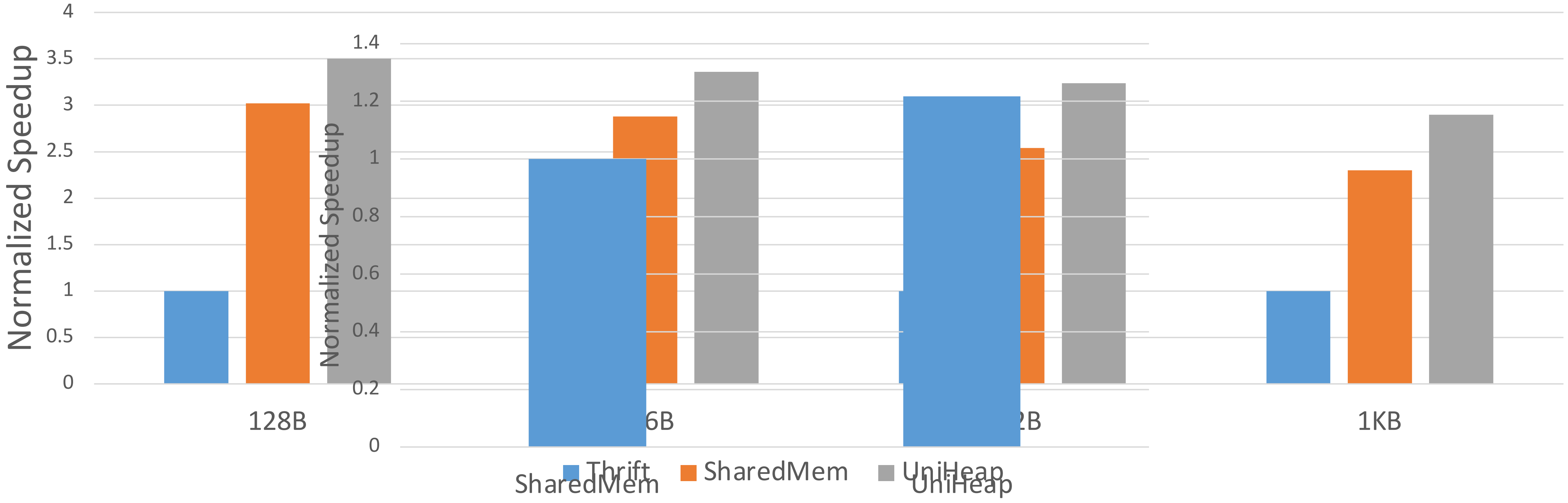
Experiment Setup

- **CPU:** 24-core Intel 2nd Xeon
- **NVM:** 8 * 128GB Intel Optane DC

Benchmarks

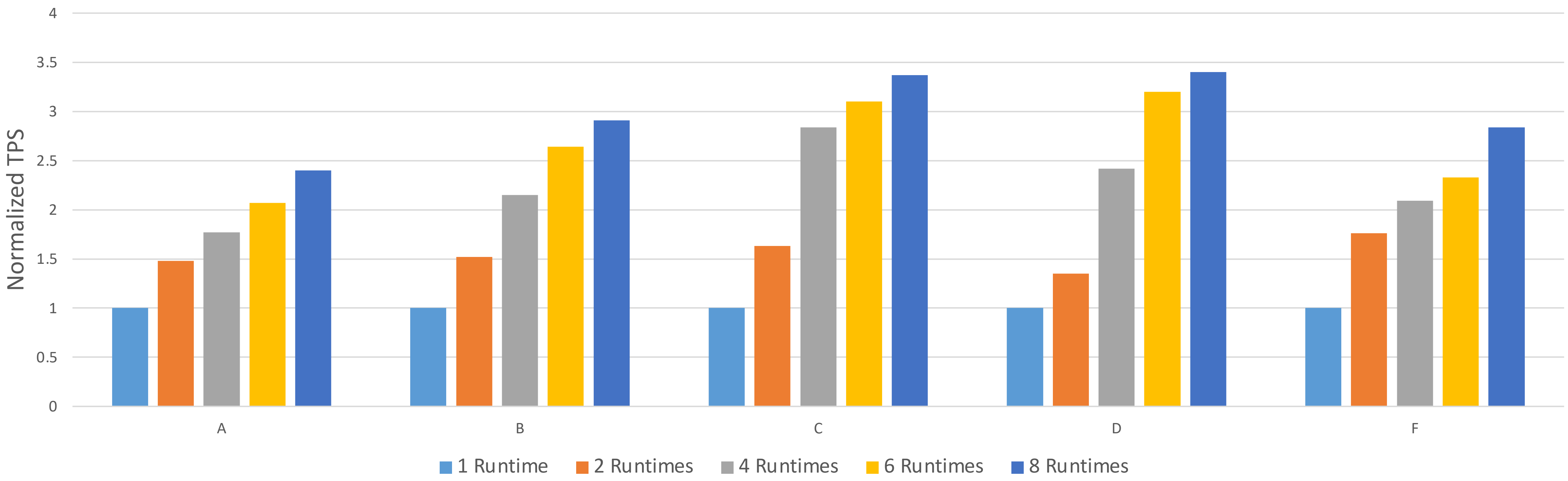
- **Java:** YCSB over QuickCached and H2
- **Python:** Python Performance Benchmark Suite
- **JavaScript:** JetStream2

Performance of Persistent Object Sharing



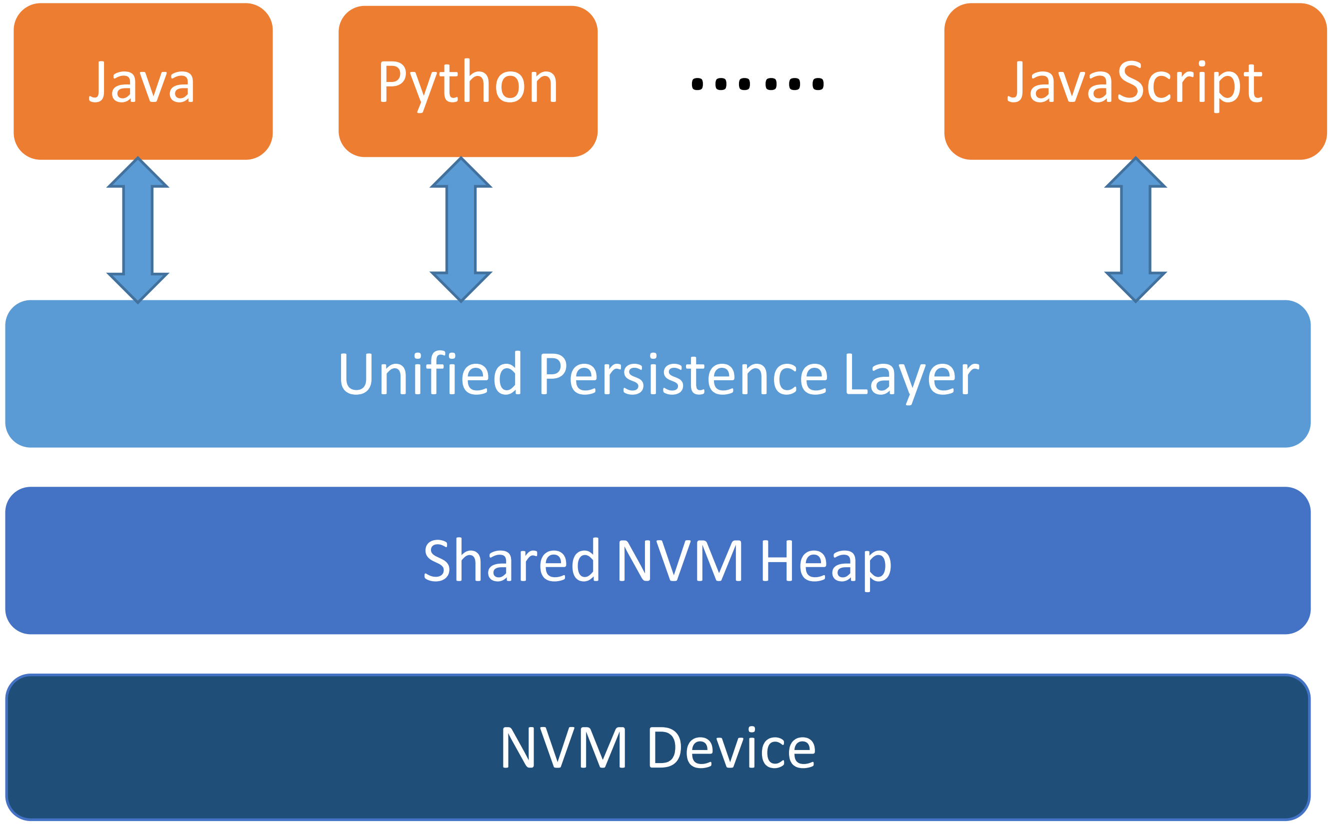
UniHeap outperforms existing approach by 1.2x - 3.4x

Scalability of UniHeap



UniHeap can scale to support multiple managed runtimes.

UniHeap Summary



Thanks!

Daixuan Li

daixuan2@Illinois.edu

Benjamin Reidys   Jinghan Sun   Thomas Shull

Josep Torrellas   Jian Huang