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

Daixuan Li
UIUC
daixuan2@illinois.edu

Benjamin Reidys
UIUC
breidys2@illinois.edu

Jinghan Sun
UIUC
js39@illinois.edu

Thomas Shull
UIUC
shull1@illinois.edu

Josep Torrellas
UIUC
torrella@illinois.edu

Jian Huang
UIUC
jianh@illinois.edu

ABSTRACT

Byte-addressable, non-volatile memory (NVM) is emerging as a promising technology. To facilitate its wide adoption, employing NVM in managed runtimes like JVM has proven to be an effective approach (i.e., managed NVM). However, such an approach is runtime specific, it lacks a generic abstraction across different managed languages. Similar to the well-known filesystem primitives that allow diverse programs to access the same file via the block I/O interface, managed NVM deserves the same system-wide property for persistent objects across managed runtimes with low overhead.

In this paper, we present UniHeap, a new NVM framework for managing persistent objects. It proposes a unified persistent object model that supports various managed languages, and manages NVM within a shared heap that enables cross-language persistent object sharing. UniHeap reduces the object persistence overhead by managing the shared heap in a log-structured manner and coalescing object updates during the garbage collection. We implement UniHeap as a generic framework and extend it to different managed runtimes that include HotSpot JVM, cPython, and JavaScript engine SpiderMonkey. We evaluate UniHeap with a variety of applications, such as key-value store and transactional database. Our evaluation shows that UniHeap significantly outperforms state-of-the-art object sharing approaches, while introducing negligible overhead to the managed runtimes.

SYSTOR '21, June 14–16, 2021, Haifa, Israel

ACM ISBN 978-1-4503-8398-1/21/06…$15.00
https://doi.org/10.1145/3456727.3463775

CCS CONCEPTS

• **Hardware → Memory and dense storage**; • **Software and its engineering → Object oriented architectures**.

KEYWORDS

Non-volatile Memory, Managed Runtime, Memory Persistency, Persistent Objects

1 INTRODUCTION

Non-volatile memory (NVM), such as phase-change memory (PCM) [49], resistive RAM (ReRAM) [11], NVDIMM [4], and Intel DC persistent memory [27], has become a promising technology that offers near-DRAM speed, scalable storage capacity, and data durability. To facilitate its wide adoption in practice, its management and use in software systems have attracted much attention recently [20, 25, 33, 35, 36, 40, 48].

Specifically, many NVM frameworks and libraries have been developed [14, 17, 19, 24, 42], such as Mnemosyne [56], NVHeaps [15], and Intel PMDK [9]. However, most of them require developers to explicitly specify the persistent data structures in their programs, which significantly increases the development burden. To address this issue, recent researches proposed to integrate NVM into managed runtimes like JVM [12, 52, 57, 60], in which they leverage the runtime system to transparently manage objects in NVM. As the managed languages, such as Java, Python, and JavaScript, have become the most popular programming languages [10, 55], such an approach is becoming pervasive [38]. We define this approach as *managed NVM* in this paper.

Although utilizing managed runtimes to use NVM has proven to be an effective approach to simplify the NVM programming [51–53, 60], state-of-the-art approaches are runtime specific, and lacking an important system-wide property – *persistent object management across managed runtimes*. It is not easy for a Python program to directly access an object persisted by a Java program, as their runtime-specific object format and layout are different.

As system-wide shared resource, managed NVM deserves data sharing, and it provides non-volatility as shared persistent storage does. Similar to the file systems developed for persistent storage [20, 33, 61], which manages data in the format of files, and allow different programs to access shared files with block I/O interface, it is highly desirable to enable diverse managed languages to access shared persistent objects efficiently, which could pave the way for developing managed NVM into a generic approach.

As more applications are developed based on managed runtimes today, they usually use different runtime instance for each individual component. And platform operators also prefer to deploy multiple runtime instances on the same machine to best utilize the compute and memory resources [23, 43, 58]. Take the web service for example, its frontend uses JavaSript runtime while its backend adopts JVM [59].

To achieve persistent object sharing, a straightforward approach is to leverage the persistence layer available in managed runtimes to persist objects to file systems or database. Typical examples include Java Persistence API (JPA) [30] and Java Data Objects (JDO) [29]. However, they cause significant performance overhead [30, 60]. Wegiel et al. [59] proposed to exploit the shared memory to enable the cross-language cross-runtime communication, unfortunately, it does not support NVM. The industry has developed Thrift [39] and Protocol Buffers [6] to facilitate the interoperation across multiple languages, however, they suffer from significant marshalling and unmarshalling overheads [2].

In this paper, we develop a lightweight NVM framework, named UniHeap, for persistent object management across a diversity of managed runtimes. It has a unified persistence layer located between the upper-level runtimes and the underlying managed NVM heap (see Figure 1). This layer has a unified object model that can be extended for various managed languages. It also optimizes the object layout to reduce the persistency overhead, when persisting objects into NVM.

UniHeap manages NVM using shared heaps that store all the persistent objects across the registered managed runtimes. UniHeap organizes the shared heap in a log-structured manner and supports both atomic in-place update and out-of-place update to reduce the persistence overhead. It utilizes a lightweight locking mechanism to manage the concurrent accesses to shared objects. UniHeap conducts coordinated garbage collection (GC) with managed runtimes to ensure the correctness of object cleanups in the shared NVM heap. Overall, we make the following contributions in this paper.

- We propose a generic NVM framework that provides a unified object model to enable efficient persistent object sharing cross diverse managed runtimes within NVM.
- We present a shared NVM heap for managing persistent objects. It manages objects in a log-structured manner and supports both in-place and out-of-place updates to reduce data persistence overhead, while ensuring the crash-safety.
- We develop an efficient GC scheme by decoupling the metadata and data of persistent objects in the NVM heap, and coordinate GC operations with managed runtimes to ensure the correctness of object cleanups.
- We enable UniHeap to support three popular managed runtimes, including HotSpot JVM, cPython, and JavaScript.

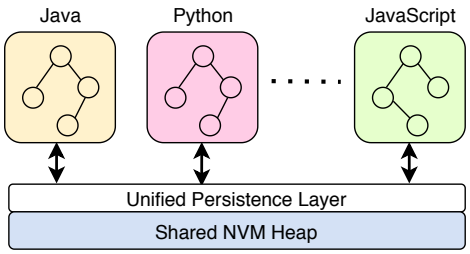


Figure 1: Overview of UniHeap

To evaluate the efficiency of UniHeap, we run a variety of data-intensive applications and typical benchmarks for different managed runtimes, including Yahoo Cloud Service Benchmarks (YCSB) [18] for Java, Python Performance Benchmark Suite [54] for Python, and JetStream2 [31] for JavaScript. Our evaluation demonstrates that UniHeap introduces negligible performance overhead to the runtime systems, compared with state-of-the-art runtime-specific NVM frameworks and persistent object sharing approaches.

2 BACKGROUND AND MOTIVATION

2.1 Byte-Addressable NVM

Compared with conventional DRAM, NVM provides much higher density, allowing it to have larger storage capacity. NVM provides data durability, which allows it to be used as persistent storage. However, it requires memory persistency to ensure the crash consistency [42, 46, 56].

Ensuring memory persistency is challenging with modern memory hierarchy, as many levels of volatile cache (e.g., volatile processor cache and DRAM) exist between the processor and NVM. With commodity out-of-order processors, the order in which stores are made persistent depends on the order they are evicted from the cache. Therefore, explicit instructions have to be used. For instance, x86-64 processors have the *clwb* instruction [1] to write back a cache line from the processor cache to NVM, and storage fence instruction (*sfence*) to guarantee that a *clwb* instruction completes. However, this will inevitably increase the burden of software development and the complexity of NVM management.

2.2 NVM Management and Programming

To facilitate the use of NVM, both industry and academic community have developed high-level libraries and system

frameworks. They enable programmers to develop durable applications on top of NVM, without dealing with the low-level primitives [9, 14, 15, 17, 19, 24, 42, 48, 56, 61]. Their abstraction for developers is similar, which requires developers to specify the durable data structures and failure-atomic regions for ensuring the crash-safety.

Recently researchers proposed to leverage managed runtimes like JVM to manage NVM [12, 52, 57, 60]. Such an approach is promising to simplify the NVM programming. For instance, AutoPersist [52] only requires developers to explicitly specify the durable root objects, the managed runtime will automatically persist the reachable objects based on the reachability analysis in JVM. As a majority of applications today are developed with managed languages such as Java and Python [10, 55], these prior studies make us believe the runtime-based approach is becoming a pervasive and practical solution for managing NVM.

2.3 Object Management Across Runtimes

The current approach of managing NVM is runtime specific. It lacks an important property: persistent object management across managed runtimes. Similar to the system-wide shared persistent storage whose management software – file systems – enable the data sharing across different programs with the *file* abstraction, the runtime-managed NVM framework is desirable to enable the data sharing with the *persistent object* abstraction, such that the objects created by one managed runtime can be easily accessed by another one.

This property will benefit many real-world application cases, such as web services [23, 58] and data analytics [26, 43, 62]. In these applications, multiple runtime instances access shared data source for further analytics, and different instance may fulfill different functions [7]. For example, (1) a Java program persists its objects in NVM and allows other programs written in a different language such as Python to access them directly (e.g., web services); (2) a Java program and Python program concurrently access shared persistent objects (e.g., shared libraries); or (3) multiple Java programs concurrently access a persistent object (e.g., data analytics).

To achieve the same property, existing managed runtimes persist objects to file systems or database in NVM. However, they suffer from significant software overheads [60]. Typical examples include Java Persistence API (JPA) that uses transactional APIs to persist data [30], and Persistent Collection for Java (PCJ) [5] that manages Java objects in NVM. Both methods have the data transformation procedure, which introduces dramatic performance overhead.

3 UNIHEAP DESIGN

In this paper, we develop a NVM framework, named UniHeap. It enables object persistent and sharing across various managed runtimes in an efficient manner, while preserving the programmability of managed languages for NVM.

Table 1: The mapping of language types in UniHeap.

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

3.1 Design Principles

UniHeap follows three design principles: (1) it should manage runtime objects in NVM efficiently, while ensuring the crash safety for object operations with low data persistence overhead; (2) it should enable data sharing of these persistent objects cross different managed languages without much data transformation overhead; (3) it should provide a simple and unified object model that can easily support a new runtime, which will pave the way for its generic use.

UniHeap provides a unified persistence layer as presented in Figure 1. It has a unified object model (§3.2) that can be mapped to different managed languages. UniHeap stores persistent objects within shared NVM heaps in a log-structured manner for efficiency (§3.3). It enables managed runtimes to directly access the shared NVM heap with simple interfaces (§3.4), while enabling concurrent object accesses cross managed runtimes (§3.5). UniHeap also has efficient GC mechanisms for managing persistent objects in a space efficient manner (§3.6). We discuss each of them as follows.

3.2 Unified Persistence Layer

We design a unified persistence layer (UPL) in UniHeap for two purposes. First, UPL has a language-neutral object model, such that it can be extended to support new managed languages. Second, UPL should facilitate object persistence for managed runtimes to achieve low persistency overhead.

Unified object model and its type system. Unlike recently proposed PCJ for NVM [5], UniHeap does not introduce new type system. UniHeap provides two built-in types: *numeral* type and *reference* type. It does not provide container types, such as *list*, *dict*, and *tuple* in Python, as developer can implement their own container type based on these built-in types. As shown in Table 1, the numeral type includes *char*, *short*, *int*, *long*, *float*, and *double*. For the reference type, the object field stores the pointer to other persistent objects. It is worth noting that *array* is also treated as an object in UniHeap. Thus, its object field can store a pointer to an array. As we provide a transparent type system for managed runtimes, different managed language needs to map their type into the type system of UniHeap. We show the mapping of the popular managed languages that include Java, Python, and JavaScript in Table 1.

Table 2: UniHeap API for object persistence.

API	Description
set_root	declare a durable root in a managed runtime.
get_root	retrieve a durable root in a managed runtime.
atomic_begin	declare the beginning of a persistent region.
atomic_end	commit a persistent region in a managed runtime.

Object persistence. To be compatible with the data persistence approaches in the existing NVM frameworks [52, 60], UniHeap allows developers to label failure-atomic regions as well as enables automatic object persistence with specified root objects (see Table 2). Programmers can use the APIs *atomic_begin* and *atomic_end* to specify a failure-atomic region that provides an all-or-nothing visibility to programmers, as shown in Figure 2 (a). With automatic object persistence enabled by setting the durable root, as presented in Figure 2 (b), all the updates to the objects reachable from a durable root will be persisted in a crash consistency manner. Note that both approaches do not require modifications to the class in the managed languages.

To reduce the object persistence overhead, UniHeap leverages both *atomic update* and *out-of-place update*. For small updates, such as those written to a single data field, UniHeap simply uses atomic instruction and memory fence. UniHeap will capture this type of updates by checking the associated byte code (e.g., *putfield* in JVM).

For updates to a failure-atomic region, UniHeap uses out-of-place update and writes the updates in a log-structured manner. Specifically, UniHeap wraps object updates to the failure-atomic region within a transaction. Instead of using the conventional undo or redo logging approaches that require double writes (i.e., one write for the persistent object update, one write for the undo/redo log) [13, 28, 44, 45], out-of-place update only requires one write to the NVM on the critical path. This is because the old data copies already exist in the NVM and the logging is not required. As we persist the update in a new location, it does not affect the old version, and thus, it inherently supports the atomic data durability. We discuss how these persistent objects are managed and persisted in UniHeap as follows.

3.3 Shared NVM Heap

UniHeap stores persistent objects in shared NVM heaps. It organizes the shared NVM heap into five regions as shown in Figure 3: heap header, plass region, root table region, object region, and log region. The heap header stores the metadata for the corresponding heap, including the heap name (32 bytes) and heap size (8 bytes). The plass region stores all the class descriptors for UniHeap objects. The root table stores all the durable root objects. Each root is a key-value pair with the format of <*root_name*, *root_addr*>. The object region contains an object valid bitmap, which is used by the GC for reclaiming persistent objects in UniHeap (§3.6). Its

```
1  uniheap.atomic_begin();  // Start an atomic region.
2  Car redcar = new Car (...);
3  uniheap.persist(redcar);
4  uniheap.atomic_end();  // Commit a persistent region.
```

(a) An example of using atomic region in UniHeap.

```
1  Car redcar = new Car (...);
2  uniheap.set_root (redcar);  // set durable root.
```

(b) An example of using durable root in UniHeap.

Figure 2: Examples of object persistence in UniHeap.

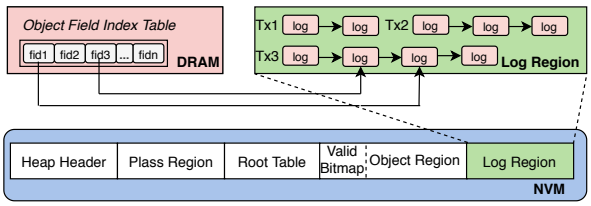


Figure 3: The shared NVM heap structure in UniHeap.

remaining part is organized into numerous fixed-size (16 bytes) chunks, each of which stores an object header. The log region stores the object updates in a log-structured manner.

Out-of-place update to reduce persistency overhead. To facilitate out-of-place update, we decouple the object header from its data (i.e., fields), based on the insight that object header is not frequently updated in a transaction. We store the object header in the object region, and object data in the log region. Once a managed runtime allocates an object from the shared NVM heap (with the APIs discussed in Table 3), UniHeap will bump the pointer in the object region to allocate an object header.

UniHeap uses transactions to ensure data persistence of object updates. UniHeap uses a slab allocator to allocate memory space for corresponding transaction logs. Each durable transaction has a transaction header, which consists of the transaction id (*t_id*), and its next transaction address (*t_next*). Multiple transactions are organized into a linked list. Each transaction can have multiple logs to stores the out-of-place updated data. Each log has a log header, which stores the object id (*l_oid*), object field id (*l_fid*), a valid/invalid bit, and its next log address (*l_next*), as shown in Figure 3.

The out-of-place update approach can reduce write traffic to NVM, however, it requires address remapping to retrieve the latest data for read operation. To address this challenge, UniHeap employs a *per-object* mapping table for address translation. Since each object field has a fixed index value during the object lifetime, UniHeap use the field index to store the address of the latest updates in the log region. The address translation procedure is efficient ($O(1)$). In addition, UniHeap caches the object field index table in the fast DRAM for further performance improvement (see Figure 3).

Garbage collection for the log region. To reduce the storage overhead of the log region, UniHeap garbage collects

Table 3: Data structures and APIs used in UniHeap for interacting with different managed runtimes.

	Name	Description
Data Struc.	heap_info	NVM heap descriptor
	plass_info	object class descriptor
	root_info	root object descriptor
	object_info	object descriptor
Interface	heap operations	load_heap, close_heap
	plass operations	alloc_plass, init_plass, exists_plass
	root operations	set_root, get_root, exists_root
	object operations	alloc_object, init_object, load_field, store_field, set_index
	transaction operations	dtx_begin, dtx_commit, stm_begin, stm_commit, stm_abort, stm_store, stm_load

stale logs with two strategies: *fast* GC and *full* GC. For fast GC, UniHeap periodically scans all the committed transactions and checks whether their logs are valid or not. UniHeap compares the log address with the latest address stored in the object index table. If they do not match with each other, it means the log is invalid. If all logs in one transaction are invalid, UniHeap will free them, and update the next pointer (*t_next*) of the committed transaction.

As for the full GC, it happens when the free space is below a threshold (10% of the entire log region by default). UniHeap accelerates the full GC procedure with thread parallelism. It first identifies the first and last committed transactions, and create multiple GC workers. Each GC worker is assigned with a certain number of transactions. The GC worker scans their assigned transactions. For each committed transaction, it copies all valid logs into a volatile buffer. After finishing scanning all the committed transactions, worker threads use new durable transactions to commit these valid logs. After that, UniHeap will reclaim the log space.

3.4 Persistent Object Sharing

It is challenging to have a unified framework for diversified runtimes, as different runtimes have different properties. To overcome this, UniHeap abstracts the shared heaps with a set of generic interfaces and data structures (see Table 3).

Data structures for persistent object sharing. UniHeap abstracts the shared heap with four core data structures: *heap_info*, *plass_info*, *object_info*, and *root_info*. The *heap_info* data structure that stores NVM heap descriptors (e.g., the heap name and size) is placed in the heap header region (see Figure 3). The *plass_info* is a class descriptor in UniHeap, which is similar to the *klass* in Hotspot JVM. It is stored in the plass region. The *root_info* stores the durable roots specified in managed runtimes, it is placed in the root region of UniHeap. The data structure *object_info* stores the meatadata information of each persistent object (see §3.3).

Interface for accessing shared persistent objects. UniHeap provides a set of interfaces (see Table 3) for managed runtimes to access the shared NVM heap. We classify these interfaces into two categories: language-neutral and language-related. The language-neutral interfaces do not need support from managed runtimes. For example, UniHeap implements the *alloc_obj* interface with its own object allocation policy, which is independent from the managed runtime. For those language-related interfaces that include only *init_plass* and *exists_plass*, they require runtimes to have their own implementations. For instance, when UniHeap initializes a class with *init_plass*, UniHeap requires runtime support to fill up the plass structure, since the class metadata is stored in the language source file (e.g., *.cls* in JVM),

Runtime support for shared NVM heap. To facilitate the interactions with shared NVM heap, UniHeap has a module for each (new) managed runtime. This module is used to support type mapping, object operation, and GC in UniHeap. Each UniHeap module will provide a mapping set to track the object references and durable root objects from the managed runtime, UniHeap will use this information to ensure the GC correctness. We show the development effort required for each managed runtime in §4.

To support object persistence, UniHeap provides a set of APIs (see Table 2) to managed runtimes. Similar to AutoPersist [52], UniHeap provides *set_root* and *get_root* to allow developers to specify the durable roots. UniHeap also provides *atomic_begin* and *atomic_end* to enable developers to specify persistent regions in their programs, in which the atomic durability is guaranteed for object updates.

3.5 Concurrent Access to Objects

As persistent objects could be concurrently updated by different runtimes, it is important to ensure mutual exclusion for concurrent accesses. To achieve this, UniHeap utilizes software transaction memory (STM) [41]. Therefore, programs in different runtime instance can declare their persistent regions with *atomic_begin* and *atomic_end*. Inside the persistent region, UniHeap combines the STM and durable transaction (DTx) to achieve both mutual exclusion and atomic durability for concurrent object accesses.

UniHeap has an interpreter-level implementation of STM. Specifically, UniHeap module will capture the load/store operations against the object fields via hooking the associated bytecode (e.g., *putfield/getfield* in JVM). We adopt an *eager* policy (i.e., *encounter-time locking*) to resolve the conflict of data accesses. STM maintains a global lock array, and each entry owns a lock in the lock array. Each runtime needs to acquire the lock before performing the store operation to an object field. And each runtime will validate the lock during load operation to avoid conflict of data accesses.

The STM in UniHeap is a time-based transaction implementation. UniHeap maintains a global time counter which monotonically increases when a transaction is committed

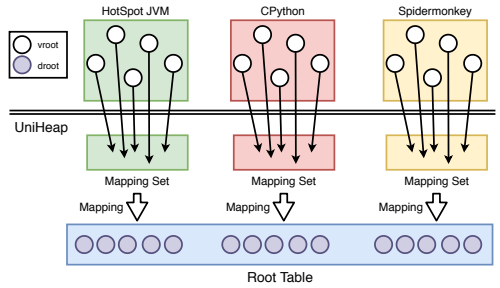


Figure 4: The mapping of durable roots in UniHeap.

successfully. Once the runtime commits the transaction, it persists the execution result by leveraging the durable transaction (DTx). The runtime starts a new durable transaction with the generated global timestamp, allocates logs to persist the execution results, and commits the durable transaction.

If a transaction fails, UniHeap will roll back it to the initial state to restart the execution. Since we build the transaction at the interpreter level, UniHeap needs to roll back the interpreter state in order to re-execute the bytecodes inside the persistent region. Preserving the interpreter state requires the module support in UniHeap, since the interpreter state is language specific. For instance, UniHeap module in JVM stores the runtime thread stack pointer (*sp*), bytecode pointer (*pc*), and local variables (*locals*) at the beginning of the transaction. If STM execution fails, it will restore the saved environment to roll back the interpreter state. And the interpreter will re-execute the bytecode.

3.6 Coordinated GC for Persistent Objects

In §3.3, we discuss the GC for the log region in the shared NVM heap. It is noted that the GC for persistent objects is different, it requires the interaction with managed runtimes to ensure the correctness of object cleanups (see §3.4).

Mark-and-Compact GC in UniHeap. UniHeap uses the mark-and-compact GC to reclaim persistent objects. We have to overcome three challenges. First, as each runtime can have its own reference to a persistent object, UniHeap should be able to track these references efficiently. Second, UniHeap needs to coordinate with multiple runtimes to reach a system-wide safety point, and then stop the world to perform the GC. Third, we should also ensure crash safety at GC.

As managed runtimes use UniHeap to store their persistent objects, UniHeap will track their references and reachability from runtimes. We call these references as *vroot*, with the corresponding of the durable roots (*droots*), as shown in Figure 4. Therefore, UniHeap maintains a mapping set for each runtime to store their *vroots*. As each runtime has different heap layout and object management, the mapping set is tailored for each runtime. Take HotSpot JVM for example, the heap of Hotspot JVM is partitioned in two generations: young region and tenured region. As the object stored in the young region has a shorter lifetime than tenured region, UniHeap uses two hash tables to store the object references for the young and tenured objects, respectively.

As many managed runtimes adopt tracing-based GC, UniHeap reuses their safe point and stop-the-world mechanism. After collecting these *vroots*, UniHeap creates multiple GC worker threads to perform the object reachability analysis. Each thread is assigned with several *droots* and mark lived objects. For any object that is reachable from a durable root, it will be treated as lived object and maintained in the persistent heap. For an object which is no longer reachable from a durable root and it has no reference from any runtime, it will be reclaimed. Therefore, if a user wants to maintain an object in the persistent heap for future use in another runtime, this object should be reachable from a durable root or has a reference from at least one runtime. For these objects hosted in the managed runtime but not reachable from durable roots, they would not be persisted to the unified heap and would be garbage collected by the corresponding managed runtime.

Crash safety in the GC of UniHeap. UniHeap needs to ensure crash-safety during GC. To achieve this, UniHeap will guarantee each step in the GC is idempotent. The GC of UniHeap consists of four phases: (1) marking phase, (2) relocation phase, (3) compaction phase, and (4) clean-up phase. In the marking phase, UniHeap sets the corresponding bit in the valid bitmap to indicate whether the object is live or not. The heap space and layout are not changed. During the relocation phase, UniHeap will conduct a statistical analysis of the shared NVM heap space, allocate a new object region based on the calculation, and use a hash table to store the new addresses of all live objects. After this phase, the new address of each object is determined. During compaction phase, the live objects will be moved to their new addresses. To accelerate the GC procedure, UniHeap uses multiple threads to migrate live objects. At the clean-up phase, old objects will be reclaimed. Note that UniHeap uses *obj_info* as an indirection layer that stores the pointer to the real persistent object. Therefore, during the clean-up phase, UniHeap does not need to modify all reference pointers in all runtimes.

The GC of UniHeap is crash safe by ensuring each GC phase is idempotent. The marking phase is naturally idempotent, since this phase does not change the heap states. As for the compaction phase, we maintain the old object region until the clean-up phase. Therefore, upon a crash or failure, UniHeap can redo the GC during the system recovery.

3.7 NVM Heap Management

In UniHeap, we have a server proxy managing all NVM heaps. Each UniHeap module for each managed runtime has a thread to communicate with the server proxy via UNIX domain socket. UniHeap minimizes the inter-process communication during the runtime execution. In particular, UniHeap

module only communicates with UniHeap server proxy at the runtime initialization and GC procedure. For the runtime initialization, UniHeap module interacts with the server proxy to allocate shared memory, load the shared NVM heap, and initialize other system states. Once UniHeap starts GC, all runtimes need to report their *vroots* to the UniHeap server proxy. UniHeap also coordinates with runtimes to stop the world through this communication channel.

4 UNIHEAP IMPLEMENTATION

In this section, we describe the UniHeap implementation details. We will first discuss the implementation of the main framework. And then, we will present the module implementation for each individual managed runtime, including HotSpot JVM, cPython, and SpiderMonkey.

Main Framework Implementation. We implement UniHeap system prototype with 9,163 lines of C programming code. The UniHeap framework is implemented as a shared library, including UniHeap interface and GC for the log region. As for the shared NVM heap, UniHeap uses the memory-mapped interface with Direct Access (DAX) [61] enabled for fast access to the NVM device. Each runtime module will dynamically load this shared library at the runtime initialization. The UniHeap server proxy implementation has 5,732 lines of C code, including process communication and GC.

Hotspot JVM Implementation. We modify a few OpenJDK8u212-b03-0 components to support UniHeap. We modify the Hotspot template interpreter to implement our transaction mechanism. We add hook functions in the bytecodes for object field accesses (i.e., *putfield* and *getfield*), and array element access (e.g., *arrayload* and *arraystore*). We also modify the parallel scavenge GC to support our GC for persistent objects. Specifically, we add the remembered sets [8] support in the GC procedure. After the GC copies all objects to the *from* region, the runtime module reports all the remaining live objects as *vroots* to UniHeap.

cPython Implementation For the Python runtime, we use the open-source cPython 2.7.15. We add the hook functions in the corresponding bytecodes for object accesses, such as *LIST_APPEND*, *LOAD_LOCALS*, *LOAD_ATTR*, *STORE_LOCAL*, and *STORE_GLOBAL*. Since cPython uses a reference counter-based GC rather than a tracing-based one, special effort is required to obtain the object references. In cPython, a module is used to implement a generational GC on top of the reference counter based GC, in which object deallocation happens when its reference count drops to zero. Therefore, we can obtain *vroots* by tracking the object references from their allocation to deallocation.

JavaScript Implementation. Similar to HotSpot and cPython, UniHeap adds hook functions to the relevant bytecodes for object accesses in SpiderMonkey with version 52. They include *JSOP_GETLOCAL*, *JSOP_SETLOCAL*, *JSOP_GETNAME*, *JSOP_SETNAME*, and others. The GC of SpiderMonkey is similar to that of HotSpot JVM. Their major difference is that the marking phase in SpiderMonkey occurs incrementally. But the way of tracking *vroots* in SpierMonkey is similar to that of HotSpot JVM. The SpiderMonkey runtime module reports its *vroots* when copying them to a new heap during the GC.

5 EVALUATION

Our evaluation shows that (1) UniHeap performs better than state-of-art approaches of persistent object sharing (§5.2); (2) It can scale the persistent object sharing as we increase the number of managed runtimes (§5.3). (3) UniHeap enables persistent object sharing across different runtimes without introducing much performance overhead (§5.4 and §5.5);

5.1 Experimental Setup

We conduct our evaluation on an Intel machine configured with real NVM devices. It has two Intel Xeon processors, and each processor has 24 cores. It has eight Intel Optane DC persistent memory modules with a total capacity of 1TB (8×128GB), and 384GB of DDR4 DRAM. The server runs Fedora 27 with the Linux kernel version 4.15. We modify the OpenJDK8u212-b03-0, CPython2.7.15, and SpiderMonkey-52 runtimes, respectively (see § 4), and use UniHeap to manage shared persistent objects. We use *libpmem* [9] to create shared NVM heaps and utilize *DAX* technique to enable direct memory access to NVM DIMMs. The shared heap is configured as 20GB by default. We compare UniHeap with two state-of-the-art data persistence approaches used in managed runtime systems.

- **ManualPersist:** It requires developers to manually identify the objects that should be allocated in NVM [60]. When persisting objects to NVM, *clwb* and *mfence* instructions are called to enforce the memory persistency.
- **AutoPersist:** It requires developers to only specify the durable root objects, the runtime will automatically persist the objects reachable from the durable roots [52].

We follow the automatic object persistence technique in AutoPersist, and enable it in cPython and SpiderMonkey runtimes for fair comparison. Although ManualPersist and AutoPersist approaches were developed for supporting NVM in managed runtimes, they do not have the feature of sharing persistent objects across managed runtimes. Therefore, we also compare UniHeap with state-of-the-art object sharing approaches: Thrift [39] and shared memory [59].

- **Thrift:** It is a scalable cross-language RPC protocol developed by Facebook [39]. Similar to other cross-language

Table 4: Benchmarks used in our evaluation.

Runtime	Benchmark	Description
Java	YCSB A	Recent actions in a user session
	YCSB B	Photo tagging application
	YCSB C	User profile cache
	YCSB D	User status updates
	YCSB F	Read-modify-update in user database
Python	bm_nqueens	N Queens solver
	bm_nbody	N body benchmark
	bm_chameleon	Template rendering
	bm_chaos	Chaos game like fractals
	bm_logging	Benchmarks on the logging module
JavaScript	gcc_loop	GCC and LLVM vectorization tuning
	float_mm	Floating point matrix multiplication
	quicksort	Quicksort
	hashset	Hash table operations

protocols, such as ProtoBuf [6], Thrift requires serialization/deserialization when it generates template codes. Recent studies [2, 59] show that Thrift performs much better than other similar techniques.

- **SharedMem:** It enables object sharing across Java and Python runtimes through shared memory [59]. To enable the NVM support in SharedMem, we use the libpmem [9], such that the objects stored in the shared memory space will be persisted in NVM.

We evaluate UniHeap against the aforementioned baselines with a variety of synthetic workloads and benchmarks. For different managed runtimes, we use different dedicated benchmarks that were developed for testing each individual runtime. We describe them as follows.

- **Java Benchmarks:** We use the key-value store QuickCached [47] and transactional database H2 [3]. QuickCached is a pure Java implementation of Memcached that uses persistent data structures to store key-value pairs. H2 is a popular transactional database written in Java language. For both QuickCached and H2, we use Yahoo Cloud Services Benchmark (YCSB) [18] to populate the backend storage and generate cloud workloads with different read/write ratio following the Zipfian distribution.
- **Python Benchmarks:** We use Python Performance Benchmark Suite [54], which is a performance benchmark suite developed for emulating real-world workloads. It has a variety of benchmarks that include Chaos games, template rendering, and event logging, as shown in Table 4.
- **JavaScript Benchmarks:** For JavaScript runtime, we use JetStream2 [31], a JavaScript benchmark suite for web applications. It covers a variety of common operations executed in JavaScrpt-based applications, such as matrix multiplication, quicksort, and hash table operations.

5.2 Performance of Shared Object Accesses

We first examine the performance benefit of UniHeap on persistent object sharing. We compare UniHeap with state-of-the-art object sharing approaches Thrift and SharedMem, as described in § 5.1. We first run microbenchmarks. We

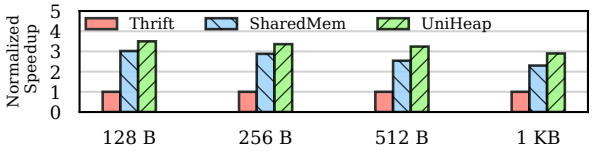


Figure 5: Performance comparison of different persistent object sharing approaches.

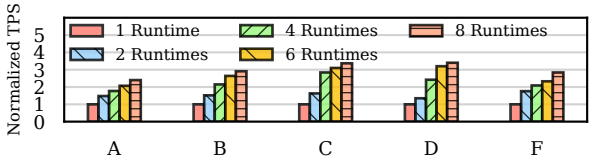


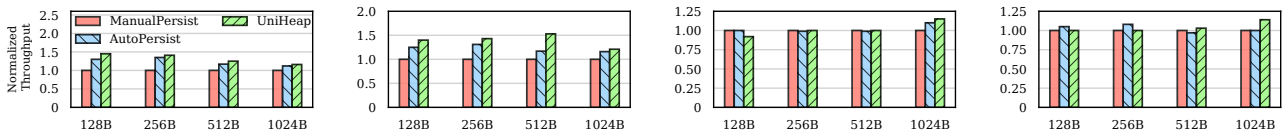
Figure 6: Scalability of UniHeap.

run one HotSpot JVM and one SpiderMonkey runtime on UniHeap concurrently. Within the HotSpot JVM, we run the microbenchmark to generate 10K persistent objects sequentially. We vary the object size from 128 bytes to 1KB. For each persistent object, we share it with the SpiderMonkey runtime with different approaches. As shown in Figure 5, both UniHeap and SharedMem perform 2.3-3.4× faster than Thrift. This is because Thrift has to conduct the serialization and deserialization operations, while former ones use shared memory techniques to achieve object sharing. As we increase the object size, Thrift improves its efficiency, however, it still performs much worse than the shared memory approach.

We now use a real-world benchmark to demonstrate the interoperability across different managed runtimes through UniHeap. We run the YCSB in the HotSpot JVM against a QuickCached service. We first populate the key-value store with 1 million key-value pairs, such that these key-value pairs will be stored in the shared NVM heap in UniHeap. And then, we run another YCSB client in the SpiderMonkey runtime. We use the YCSB workload C (read-only) to issue query requests against the key-value store. We compare UniHeap with the SharedMem solution, and find that UniHeap performs 21.7% better than SharedMem. This is because SharedMem does not support NVM, but its shared memory space can be mapped to the NVM through *libpmem*. However, it still performs worse than UniHeap, due to the lack of data persistence optimizations.

5.3 Scalability Benefit of UniHeap

We now evaluate the scalability of UniHeap. To facilitate our evaluation, we use the H2 database as the back-end persistent storage service. We first run the YCSB client in one HotSpot JVM runtime, and load 1 million key-value pairs. After that, we increase the number of YCSB clients, and each YCSB client will run in an individual HotSpot JVM. As shown in Figure 6, for all the YCSB workloads we evaluate, their throughput will increase as we increase the number of JVM runtimes. This demonstrates that the mechanisms designed for concurrent



(a) Sequential Write (b) Random Write (c) Sequential Read (d) Random Read

Figure 7: Performance of supporting HotSpot JVM in UniHeap.

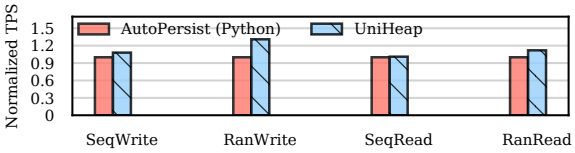


Figure 8: Performance of cPython in UniHeap.

Figure 9: Performance of SpiderMonkey in UniHeap.

data accesses to persistent objects in UniHeap can scale to support multiple managed runtimes.

5.4 Synthetic Workloads

We now compare UniHeap with each individual (native) managed runtime that does not provide the function of persistent object sharing. In the experiment, we generate 1 million objects within the managed runtime. After that, we perform sequential/random read and update operations against these objects. We vary the object size from 128 bytes to 1 KB.

In UniHeap, we wrap the load and store operation against the object fields with a durable transaction. In ManualPersit, we use *clwb* and *mfence* instructions after each object field update to ensure the data persistence. In AutoPersist, we use its *failure-atomic region* to wrap each object update and use the undo logging by default to guarantee the crash safety.

UniHeap does not introduce additional performance overhead, as shown in Figure 7. For writes, UniHeap outperforms ManualPersist and AutoPersist by 10.2% and 13.4% on average, respectively. For reads, UniHeap does not introduce much performance overhead, although it has to check an indirection layer for the log-structured memory design.

AutoPersist outperforms ManualPersist, because it has less cacheline flushes and memory fence operations. Since AutoPersist relies on the runtime system to persist objects, it has the knowledge of the runtime system (e.g., object layout and reachability). UniHeap follows the design principle of AutoPersist, which manages the memory persistency at runtime. However, UniHeap enables persistent object sharing.

Both AutoPersist and ManualPersist employ logging to ensure crash safety, which causes duplicate writes to NVM for each persistent write. UniHeap develops the log-structured memory to achieve both strong data persistence and low write traffic. This is especially helpful for random writes, since the log-structured memory design will turn the random data accesses into sequential pattern by appending each update into the log (see Figure 3).

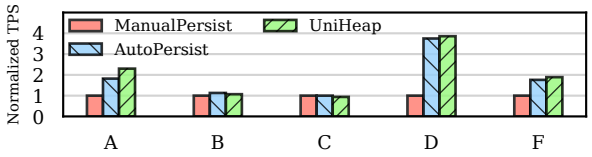


Figure 10: Performance of running HotSpot JVM with UniHeap, when using QuickCached+YCSB.

We also run the synthetic workloads with cPython and SpiderMonkey runtimes. For these runtimes, we have our own implementation of AutoPersist [52]. As shown in Figure 8 and Figure 9, we set the object size to 1KB, UniHeap provides the similar performance behavior as HotSpot JVM. This demonstrates that UniHeap does not introduce much performance overhead to each individual managed runtime.

5.5 Real-World Applications

In this section, we use real applications to evaluate UniHeap.

Comparison with HotSpot JVM. We first evaluate UniHeap against the HotSpot JVM with ManualPersist and AutoPersist enabled, respectively. As described in §5.1, We run QuickCached as the back-end storage service and store its persistent data structures in NVM. We run YCSB as the client and issue put/get requests against the QuickCached service.

We show the experimental results in Figure 10. UniHeap outperforms AutoPersist by up to 26.3% for the workloads with intensive writes, such as workload A, D, and F. This is because UniHeap utilizes the log-structured memory design and stores the object updates in the log. This can significantly reduce the number of memory fence operations, as we ensure the memory persistency for object updates. As for the comparison between AutoPersist and ManualPersist, we obtain the similar conclusion as discussed in [52]. AutoPersist leverages the runtime profiling and optimization to reduce the data persistence overhead. However, AutoPersist uses undo log for ensuring the data durability and atomicity. UniHeap further improves this mechanism by using the out-of-place update in the log region (see Figure 3).

To further understand the performance benefit of UniHeap, we sample the number of *mfence* operations incurred

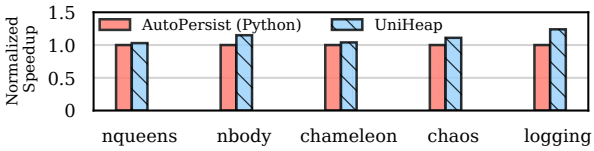


Figure 11: Performance of running cPython on UniHeap with Python Performance Benchmark Suite.

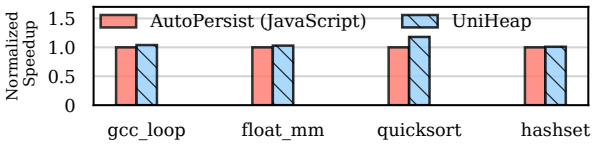


Figure 12: Performance of running SpiderMonkey on UniHeap with JetStrem2 benchmarks.

at runtime, when we issue 1K requests against the key-value store. As shown in Table 5, UniHeap significantly reduces the number of *mfence* operations. Since the runtime will incur extra operations on persistent objects, AutoPersist requires more *mfence* to ensure the data persistence.

Comparison with cPython. We now evaluate UniHeap against the cPython with AutoPersist enabled. We use the Python Performance Benchmark Suite to run various Python benchmarks within the managed runtime. As shown in Figure 11, UniHeap does not introduce extra performance overhead for all the benchmarks. It outperforms the cPython runtime by 24.1%, when running the write-intensive *bm_logging* benchmark. As for the benchmarks *bm_nqueens* and *bm_chameleon*, UniHeap performs slightly better than cPython, as they generate mixed read/write operations.

Comparison with SpiderMonkey. For the comparison with SpiderMonkey, we run JetStream2 benchmarks that were designed specifically to evaluate the JavaScript runtime engine. We demonstrate the experimental results in Figure 12. UniHeap introduces trivial extra performance overhead. For the workloads *gcc_loop* and *float_mm* that are less memory intensive, the performance of UniHeap is similar to that of running SpiderMonkey independently. For the workload *quicksort* that generates intensive writes, UniHeap performs better because of the reduced data persistence overhead.

Overall, as we run each managed runtime on UniHeap, its performance is better than that of running each managed runtime individually. This shows the low performance overhead of UniHeap as a generic NVM runtime framework.

6 RELATED WORK

Programming Frameworks for NVM. To facilitate the use of NVM, many NVM frameworks have been proposed [9, 14–17, 19, 24, 34, 42, 56, 60]. All of these works require the programmer to explicitly specify the data structures that should reside in NVM. This limitation requires significant effort from programmers, potentially causing correctness and performance bugs in NVM programs. Recent studies such as AutoPersist [52] and Espresso [60] proposed to exploit the managed runtime to manage NVM, which simplifies the NVM programming. However, these runtime-based solutions are mainly developed based on JVM, which cannot work as a generic programming framework. Our work UniHeap extends this thread of research to other managed languages such as Python and JavaScript, while enabling efficient persistent object sharing across different runtimes.

Table 5: Number of incurred *mfence* operations.

Benchmarks	AutoPersist	UniHeap
Workload A	1798	18
Workload B	398	4
Workload C	0	0
Workload D	331	6
Workload F	1898	22

Reduce Memory Persistency Overhead. NVM requires memory persistency operations to ensure crash consistency [42, 46, 56]. However, ensuring memory persistency with modern memory hierarchies is challenging. To alleviate memory persistency overhead, prior studies proposed both software [22, 25, 37, 56] and hardware [21, 32, 50] techniques to reduce the overhead of logging in atomic regions and transactions. UniHeap manages persistent objects in a log-structured manner, which avoids double writes and significantly reduce the logging overhead.

Persistent Object Sharing. As system-wide shared resources like persistent storage, NVM deserves efficient data sharing across different managed runtimes. JVM has developed persistence layers to persist objects to file systems or database, including Java Persistence API (JPA) [30] and Java Data Objects (JDO) [29]. However, they still suffer from significant object serialization/deserialization overhead. The most efficient approach for enabling cross-language communication is the shared memory [59]. However, the existing study did not support NVM and data persistence. To the best of our knowledge, UniHeap is the first work that enables persistent object sharing across various managed runtimes.

7 CONCLUSION

We present UniHeap, a unified persistent object management for different managed runtimes. UniHeap is driven by the trend that managed runtime would be the new abstraction to manage NVM. UniHeap provides a more generic NVM programming framework by enabling persistent object sharing across various managed runtimes.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and shepherd Jonathan Balkind for their helpful comments and feedback. This work was partially supported by the NSF grant CNS-1850317, CCF-1919044, and CNS-1763658.

REFERENCES

[1] 2015. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-\instruction-set-reference-manual-325383.pdf.

[2] 2019. Apache Thrift vs Protocol Buffers vs Fast Buffers. https://www.eprosima.com/index.php/resources-all/performance/apache-thrift-vs-protocol-buffers-vs-fast-buffers.

[3] 2019. H2 Database Engine. https://www.h2database.com

[4] 2019. NVDIMM. https://en.wikipedia.org/wiki/NVDIMM.

[5] 2019. PCJ. https://github.com/pmem/pcj.

[6] 2019. Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data. https://developers.google.com/protocol-buffers/.

[7] 2019. Why are multiple programming languages used in the development of one product or piece of software. https://softwareengineering.stackexchange.com/questions/370135/why-are-multiple-programming-languages-used-in-the-development-of-one-product-or.

[8] 2021. 10 Garbage-First Garbage Collector Tuning. https://stackoverflow.com/questions/61936621/what-is-remembered-set-in-g1-algorithms-used-for.

[9] 2021. Persistent Memory Development Kit. http://pmem.io/pmdk/

[10] 5 Popular Programming Languages and Their Uses. 2017. https://medium.com/@CarolPelu/5-popular-programming-languages-and-their-\uses-22af241de35b.

[11] H. Akinaga and H. Shima. 2010. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proc. IEEE* 98, 12 (Dec 2010), 2237–2251. https://doi.org/10.1109/JPROC.2010.2070830

[12] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-Rationing Garbage Collection for Hybrid Memories. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. Philadelphia, PA.

[13] Miao Cai, Chance Coats, and Jian Huang. 2020. Hoop: Efficient Hardware-Assisted Out-of-Place Update for Non-Volatile Memory. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA'20)*. Virtual Event.

[14] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[15] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[16] Nachshon Cohen, David T. Aksun, and James R. Larus. 2018. Object-oriented recovery for non-volatile memory. *Proceedings of the ACM on Programming Languages,* Vol. 2, OOPSLA (2018), 153:1–153:22. https://doi.org/10.1145/3276523

[17] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. Big Sky, Montana, USA.

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. [n.d.]. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*.

[19] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. 2016. NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing* (Kyoto, Japan) *(HPDC '16)*. ACM, New York, NY, USA, 125–136. https://doi.org/10.1145/2907294.2907303

[20] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Huntsville, Ontario, Canada).

[21] K. Doshi, E. Giles, and P. Varman. 2016. Atomic persistence for SCM with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 77–89. https://doi.org/10.1109/HPCA.2016.7446055

[22] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-free Regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 46–61. https://doi.org/10.1145/3192366.3192367

[23] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-Performance Cross-Language Interoperability in a Multi-Language Runtime. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS'15)*. Pittsburgh, PA, USA.

[24] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, New York, NY, USA, 468–482. https://doi.org/10.1145/3064176.3064204

[25] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2015. NVRAM-Aware Logging in Transaction Systems. In *Proceedings of the VLDB Endowment (VLDB'15)*. Kohala Coast, Hawaii.

[26] Jian Huang, Xuechen Zhang, and Karsten Schwan. 2015. Understanding Issue Correlations: A Case Study of the Hadoop System. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC'15)*. Kohala Coast, Hawaii.

[27] Intel. 2018. 3D XPoint: A Breakthrough in Non-Volatile Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[28] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of 21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. Atlanta, GA.

[29] Java Data Objects. 2006. https://en.wikipedia.org/wiki/Java_Data_Objects.

[30] Java Persistence API. 2013. https://en.wikipedia.org/wiki/Java_Persistence_API.

[31] JetStream2. 2019. https://browserbench.org/JetStream/.

[32] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 361–372. https://doi.org/10.1109/HPCA.2017.50

[33] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*

(SOSP'19) (Huntsville, Ontario, Canada).

[34] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-level Persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 481–493. https://doi.org/10.1145/3079856.3080229

[35] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Huntsville, Ontario, Canada).

[36] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Huntsville, Ontario, Canada).

[37] Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. 2018. Persistent Memory Transactions. arXiv:cs.DC/1804.00701

[38] Mario Wolczko, Non-Volatile Memory and Java. 2019. https://medium.com/@mwolczko/non-volatile-memory-and-java-7ba80f1e730c.

[39] Mark Slee and Aditya Agarwal and Marc Kwiatkowski. 2007. Thrift: Scalable Cross-Language Services Implementation. *White Paper* (2007).

[40] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. Lausanne, Switzerland.

[41] Software Transactional Memory. 2019. https://en.wikipedia.org/wiki/Software_transactional_memory.

[42] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 135–148. https://doi.org/10.1145/3037697.3037730

[43] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: Connecting Managed Heaps in Distributed Big Data Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. Williamsburg, VA, USA.

[44] Tri M. Nguyen and David Wentzlaff. 2018. PiCL: a software-transparent, persistent cache log for nonvolatile main memory. In *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. Fukuoka, Japan, 178–190.

[45] Yuanjiang Ni, Jishen Zhao, Heiner Litz, Daniel Bittman, and Ethan L. Miller. 2019. SSP: Eliminating Redundant Writes in Failure-Atomic NVMRAMs via Shadow Sub-Paging. In *Proceedings of 52st International Symposium on Microarchitecture (MICRO'19)*. Columbus, OH.

[46] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*. Minneapolis, Minnesota, USA.

[47] QuickCached. 2019. https://github.com/QuickServerLab/QuickCached.

[48] Dulloor Subramanya Rao, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*. 15:1–15:15.

[49] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. H. Chen, H. L. Lung, and C. H. Lam. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4.5 (July 2008), 465–479. https://doi.org/10.1147/rd.524.0465

[50] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. ACM, New York, NY, USA, 178–190. https://doi.org/10.1145/3123939.3124539

[51] Thomas Shull, Jian Huang, and Josep Torrellas. 2018. Defining a High-level Programming Model for Emerging NVRAM Technologies. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes* (Linz, Austria) *(ManLang '18)*. ACM, New York, NY, USA, Article 11, 7 pages. https://doi.org/10.1145/3237009.3237027

[52] 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)*. Phoenix, AZ.

[53] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. QuickCheck: Using Speculation to Reduce the Overhead of Checks in NVM Frameworks. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'19)*. Providence, Rhode Island.

[54] The Python Performance Benchmark Suite. 2017. https://pyperformance.readthedocs.io/.

[55] Top 10 Programming Languages of the World - 2019 to Begin with. 2019. https://www.geeksforgeeks.org/top-10-programming-languages-\of-the-world-2019-to-begin-with/.

[56] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)* (Newport Beach, California, USA).

[57] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Multu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. Phoenix, AZ.

[58] Michael Wegiel and Chandra Krintz. 2008. XMem: Type-Safe, Transparent, Shared Memory for Cross-Runtime Communication and Coordination. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. Tucson, AZ.

[59] Michal Wegiel and Chandra Krintz. 2010. Cross-language, type-safe, and transparent object sharing for co-located managed runtimes. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10)*. Reno/Tahoe, Nevada.

[60] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Espresso: Brewing Java For More Non-Volatility with Non-volatile Memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. Williamsburg, VA, USA.

[61] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*. 323–338.

[62] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. Boston, MA.