Object-Oriented Recovery for Non-volatile Memory

NACHSHON COHEN, EPFL, Switzerland
DAVID T. AKSUN, EPFL, Switzerland
JAMES R. LARUS, EPFL, Switzerland

New non-volatile memory (NVM) technologies enable direct, durable storage of data in an application's heap. Durable, randomly accessible memory facilitates the construction of applications that do not lose data at system shutdown or power failure. Existing NVM programming frameworks provide mechanisms to consistently capture a running application's state. They do not, however, fully support object-oriented languages or ensure that the persistent heap is consistent with the environment when the application is restarted.

In this paper, we propose a new NVM language extension and runtime system that supports object-oriented NVM programming and avoids the pitfalls of prior approaches. At the heart of our technique is *object reconstruction*, which transparently restores and reconstructs a persistent object's state during program restart. It is implemented in `NVMReconstruction`, a Clang/LLVM extension and runtime library that provides: (i) transient fields in persistent objects, (ii) support for virtual functions and function pointers, (iii) direct representation of persistent pointers as virtual addresses, and (iv) type-specific reconstruction of a persistent object during program restart. In addition, `NVMReconstruction` supports updating an application's code, even if this causes objects to expand, by providing object migration. `NVMReconstruction` also can compact the persistent heap to reduce fragmentation. In experiments, we demonstrate the versatility and usability of object reconstruction and its low runtime performance cost.

CCS Concepts: • **Hardware** → **Non-volatile memory**;

Additional Key Words and Phrases: non-volatile memory, NVM, C++, object-oriented programming, programming model

ACM Reference Format:
Nachshon Cohen, David T. Aksun, and James R. Larus. 2018. Object-Oriented Recovery for Non-volatile Memory. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 153 (November 2018), 22 pages. https://doi.org/10.1145/3276523

1 INTRODUCTION

The three-to-five order-of-magnitude performance gap between durable storage (HDDs and SSDs) and volatile memory (DRAM) has resulted in very different interfaces. Persistent data typically is stored in a database or in a file system and accessed through a high-level abstraction such as a query language or API. Due to the high latency of the storage media, these interfaces favor transferring a large quantity of data at each interaction. Programming languages, in contrast, offer direct access to (transient) data in DRAM.

Non-Volatile Memory (NVM) eliminates the performance gap between durable and volatile storage, but using NVM requires changes to programming models. Technologies such as ReRAM [1, 49], PCM [32, 41], and STT-RAM [26] are a new storage media, albeit with an interface and

Authors' addresses: Nachshon Cohen, EPFL, Lausanne, Switzerland, nachshonc@gmail.com; David T. Aksun, EPFL, Lausanne, Switzerland, david.aksun@epfl.ch; James R. Larus, EPFL, Lausanne, Switzerland, james.larus@epfl.ch.

2475-1421/2018/11-ART153
https://doi.org/10.1145/3276523

performance characteristics similar to DRAM. These memories retain data after a power shutdown and are likely to be widely deployed in servers in the near future.

NVM requires new programming models to ensure that the persistent storage is left in a recoverable state after an unexpected or abrupt program failure. Durable atomic blocks cause a program to atomically transition between consistent states, each of which can be recorded in NVM and used to restart reliably after a crash [4, 9–12, 14, 17, 22, 27, 34, 37, 46, 50, 51].

Capturing a running application's consistent state is, however, only part of recovery. The persistent heap will be used after a restart, which means that it must be put into a state that is consistent in the *new* environment that exists when the application resumes execution. Consider, for example, a persistent key-value store in which each entry contains a pointer to a network connection to a client [36]. The connection is not valid when the program restarts. Recovery should at least discard the old connections to avoid an access to a stale pointer. Sockets, locks, and thread IDs are other, inherently transient fields that are invalid after recovery. These fields are often intermixed with persistent fields in an object, which requires recovery to distinguish the two types of fields and to discard or reinitialize stale, transient values.

Persistent fields containing pointers to other durable objects also become invalid if recovery maps the durable heap to a different location in memory. Currently, operating systems do not guarantee that a persistent heap can be mapped by the `mmap` system call to the same address as before a crash. If the mapping changes, pointers to persistent objects become invalid. Furthermore, program code is also not guaranteed to reside at the same locations as before a crash. Most operating systems, in fact, use address space layout randomization (ASLR) to implement the opposite behavior by placing code at a different virtual address for each execution. Disabling ASLR is possible but reduces security. ASLR affects function pointers (C), virtual table pointers (C++), and read-only data (e.g., string literals).

Fixing these problems after a crash is a significant burden on a programmer. To clear and reinitialize transient fields and update pointers, the programmer must iterate over all live durable objects and update pointers. Failure to find an object (or iterating over an object more than once) is often a subtle error. These operations violate encapsulation as the recovery code must be aware of an object's internal structure (and be updated as an object's fields evolve). Recovery cannot be implemented using an object's methods as the virtual table need to be updated before methods are invoked. For these reasons, existing NVM systems generally do not distinguish or reinitialize transient fields and some use self-relative offsets instead of direct pointers [16, 17]. These are more costly and incompatible with standard libraries.[1]

In this paper, we present a C++ language extension for *NVM reconstruction*, the process of reestablishing the consistency of data structures stored in NVM in the environment in which an application is being restarted. `NVMReconstruction` enables a programmer to label a field as transient, so that after a restart, the restored instances of this field will be zeroed. More complex recovery is also supported through type-specific recovery methods that can reinitialize transient fields in arbitrary ways. More importantly, programs written with `NVMReconstruction` use conventional data and code pointers, and the system automatically updates these pointers if the location of the persistent heap or the code segment changes.

`NVMReconstruction` also supports upgrading objects. Durable objects are not discarded when an application terminates and they may live for a long time. However, when an application stops and restarts, it can invoke a newer version of the application code. This upgrade can create a mismatch

[1]For C programs, using offsets requires pervasive code modifications at pointer dereferences. For C++, dereferencing can be hidden by operator overloading, but pointer declaration must be modified. In both cases, offsets are slower than direct references.

between the new code and the objects in the persistent heap. In particular, the new code may append fields to a class, and an object might expand beyond the space allocated to it. `NVMReconstruction` supports code upgrading by relocating an object to a larger space and redirecting pointers to the object's new location.[2]

To reduce fragmentation of the persistent heap, `NVMReconstruction` also implements *offline* NVM compaction. Fragmentation can lower performance and prevent the allocation of large objects. Compaction in a garbage collector executes while a program is running, which requires restrictions on code generation and is difficult to apply to unmanaged languages such as C or C++. Our compaction algorithm is similar to a (fault-tolerant) copying garbage-collector, but it runs *between* executions of an application, so that when compaction is running, the application is not. Thus, compaction imposes no restrictions on code generation.

`NVMReconstruction` consists of a Clang/LLVM extension and a runtime library. The compiler extension implements our C++ language annotations and emits type information for each class and struct. To track runtime types, our extension modifies the existing object format by adding an 8-byte header to hold a type identifier. The runtime also records additional execution-specific information in NVM that it uses for reconstruction and compaction. Reconstruction runs concurrently and lazily with an application, which allows an application to restart and respond quickly, without the long latency from updating the entire persistent heap.

In this paper, we make the following contributions:

(1) The design and implementation of a language extension for C++ that distinguishes transient data from persistent data, enabling object-specific recovery code.
(2) The details of a Clang/LLVM extension that implements the C++ extension and produces a runtime type description for durable objects.
(3) The details of a reconstruction library that performs object-specific recovery and updates pointers.
(4) The design and implementation of a non-concurrent compaction algorithm for upgrading an application's data structures and reducing persistent memory fragmentation.
(5) Demonstration of `NVMReconstruction` using Atlas, an NVM programming framework, and the Echo key-value store, which shows the ease of using `NVMReconstruction` with existing code and its low runtime overhead.

The paper is organized as follows. In Section 2, we describe the need for object-specific recovery in existing systems. In Section 3, we present the `NVMReconstruction` language extension. In Section 4, we describe the implementation of the runtime library. We discuss limitations in Section 5. We demonstrate the approach and measure its performance in Section 6. Finally, in Section 7, we discusses related work and conclude in Section 8.

2 PRESERVING OBJECT SEMANTICS ACROSS INVOCATIONS

In this section, we discuss the challenges in using objects to store durable data. For each of these difficulties, we also describe existing solutions and enumerate their shortcomings.

In our data model, durable data is composed of *objects*. Each object is an instance of a *class type* that corresponds to a class or struct definition in C++. For simplicity, consider two consecutive executions of an application. In the first execution, denoted the *initial execution*, the application allocates objects in durable storage, then reads and modifies them. In the second execution, denoted *re-execution*, the application again uses the objects allocated in the initial execution. As we are

[2] Other changes, such as deleting or reordering fields, are beyond this work and require application-specific modifications [35].

primarily concerned with the behavior during re-execution, we also call this the *current* execution and call the initial execution (or a prior execution) the *old* execution.

Some data residing in NVM can be *execution-specific* or *transient*: it is relevant only to a specific execution and is invalid in subsequent executions. For such data, we use the term *stale* data for data constructed during the old execution and *current* data for data constructed during the current execution.

2.1 Transient Fields

Traditional durable storage interfaces, such as databases, only support durable data and offer no means to distinguish transient data that exists for a single execution of a program. The database system uses transient data, such as locks, but these details are hidden by the storage abstraction. In contrast, when using an NVM heap, durable and transient data will be intermixed.

Consider, for example, locks. A lock is not reusable after the termination of a program. Still, it is a common programming practice to put a lock in an object and to use it to synchronize accesses [13]. Separating the lock from the data, for example in a hash table, would introduce overhead and reduce program performance and scalability. Sockets, process (or thread) IDs, and file descriptors are similar transient data that are typically kept with an object.

Another example is transient pointers that occur when data in DRAM is pointed to by a field in a durable object. Moreover, to increase the performance of an application, a programmer can convert a persistent field to a transient field, thus reducing the number of flush operations required to update the field in NVM. These fields could be statistical summaries or any data that is recomputable from durable data. For example, in a persistent queue based on a linked list [20], the tail of the queue is never flushed to NVM as it can be recovered easily. In this case, the tail pointer is a transient field even though it is semantically durable and resides in a persistent queue object.

The presence of transient data in a durable object creates two problems. First, after re-execution, an application might *incorrectly* assume that a transient field is still usable. Second, even if it is possible to detect that a value is invalid, every method must explicitly check if every durable object is well-formed. This introduces runtime overhead and creates unnecessarily complex code.

Coburn et al. [13] advocate preventing durable objects from holding pointers to DRAM. They also propose *generation locks* that are considered "uninitialized" if the lock's generation number differs from the NVM heap's generation number. At the start of each re-execution, the global execution number is incremented, effectively releasing all NVM-resident locks from the previous execution. Although this method is practical, it prevents a programmer from using system-provided locks, such as `pthread_mutex`, or it requires *persistent* wrappers to add a generation number to these locks. This solution also does not address the problem of scalar transient data, such as a process ID, or semantically durable data, such as the tail of a persistent queue, though a similar approach might address these cases as well.

A recent report [36] found that the intermingling of transient and durable data is common in real applications and that separating these types of fields requires a significant effort.

2.2 Heap and Code Pointer Relocation

Persistent objects containing direct pointers to other persistent objects or to code are vulnerable to the relocation of the heap and of the code segments. We discuss each of these cases separately and consider existing solutions.

Heap Relocation. During an application's initial execution, the NVM library creates a DAX (direct access) [33] memory-mapped file backed by NVM memory. The memory contents reside in NVM and are mapped (using `mmap` system call) into the virtual address space of the application. The data, like other abstractions in Linux, has a name in the file system, so that it can be found and reused. During re-execution, the heap "file" is opened and mapped into the application's virtual address space.

The layout of an address space can change when the operating system or libraries are updated or when a program runs under a debugger, profiler, or another runtime tool. The operating system can remap the NVM memory to a different location in virtual memory than the one in the initial execution. This behavior is explicitly permitted by the semantics of mmap, even if a program specifies a desired virtual address, and is necessary when the requested addresses are already in use[3]. If durable data contains direct pointers to persistent data, these pointers are invalidated when NVM memory is mapped to a different location.

There are two existing solutions to the NVM pointer problem. The simpler solution, adopted by most research prototypes, is to abort recovery if the remapped address differs from the original one. In the usual case, when remapping works, a programmer can use native pointers and the system avoids the necessity to update pointers. However, this approach is impractical for a real system, as remapping failures means that the program will no longer execute and the persistent data is effectively lost. Durable data is not really durable if it cannot survive an OS update.

Industry solutions, such as Oracle's NVM Direct [16], use a different approach, self-relative pointers, to resolve this problem. A direct pointer to an NVM location is replaced by its offset relative to the memory location that contains the pointer itself. When one of these "pointers" is dereferenced, the location's memory address is added to its contents to produce a direct pointer for a load or store instruction. Intel's NVML system [17] uses a different approach called extents (contiguous regions of durable memory). Each NVM pointer consists of an extent identifier and a relative offset in the extent, in effect a software-implemented segmented-memory system [8].

There are two drawbacks to these approaches. The first is performance and space overhead. Second, without language and compiler support, each durable pointer dereference must be annotated. These annotations introduce syntactic clutter, reduce programmer productivity, and cause errors as programmers must be aware of the two different pointer types (native and offset) and use each appropriately. A recent port of memcached [36] also reported debugging challenges with the offset-based pointers.

Code Relocation. Code segments can also be loaded at a different virtual address than the prior execution when libraries are loaded at different addresses, code is reorganized by address-space layout randomization (ASLR) [40], or a program is modified by a developer. Prohibiting these possibilities would compromise security and make durable storage significantly less practical.

Direct pointers to methods and functions are common in data structures but forbidden in existing NVM-specific durable storage systems. The most prevalent example of code pointers is the Virtual Table Pointer (VTP) used by C++ to implement virtual methods. These pointers are part of an object and elaborate its behavior in object-oriented designs. Non-object-oriented languages (e.g., C), also use function pointers for this and other abstraction purposes. Both the VTP and the function pointers are established in the execution in which an object is initialized. Attempts to invoke stale virtual methods or function pointers during re-execution might result in undefined behavior or corrupt data.

The read-only data segment of an application can also be mapped to a different address during re-execution and will need to be updated. These pointers occur, for example, if a pointer references a string literal (i.e., p = "Hello world").

[3]The MAP_FIXED flag specifies that the OS must not map the memory to a different virtual address than the one specified. However, if the requested address cannot be satisfied by the OS, the mapping immediately fails.

Listing 1. Code Annotations (based on the Echo key-value store [7])

```
struct kp_vt_struct{
    kp_kvstore *parent; //back−pointer to parent kvstore
    transient pthread_mutex_t *lock; //lock for this version table
    ...
    reconstructor(kp_vt_struct* o){
        assert(kp_mutex_create("(*new_vt)−>lock", &(o−>lock))==0);
    }
}
void main(){
    kp_vt_struct *new_vt = pnew kp_vt_struct;
    ...
    pdelete new_vt;
}
```

We are not aware of an NVM system that explicitly supports (or, for that matter, forbids) durable objects with virtual methods and function pointers.[4] The assumption in these systems is that data are stored in C structures, which do not require virtual-function tables and do not contain pointers to functions or read-only data, or that function locations and read-only data do not change position between executions.

2.3 Upgrades and Object Migration

As persistent objects are expected to live for a long time, an application may evolve while its objects remain alive. In many cases, implementing new capabilities or fixing bugs requires adding new fields to an object, making it too large for its existing allocated space. In this case, the system must migrate the upgraded object to another (sufficiently large) memory location. Then, every pointer to the object must be redirected to its new location. So, upgrading a single object might require iterating over all live objects in the heap.

Fragmentation. Compacting the heap reduces fragmentation by moving objects to a continuous region of memory. Although de-fragmentation is beneficial for standard applications (as demonstrated by managed language garbage collection), it is even more important for persistent data. NVM might contain many persistent heaps, either because many applications each create a heap or because an application creates many persistent heaps (e.g., each representing a state, or a "document", of that application). Each persistent heap can be fragmented. When fragmentation is aggregated across many heaps, this leads to a significantly inefficient use of space.

3 PERSISTENT PROGRAMMING EXTENSION

The `NVMReconstruction` C++ extension consists of four new keywords: `transient`, `pnew`, `pdelete`, and `reconstructor`. Listing 1 presents a code example.

The `transient` keyword annotates fields of a class (or struct). It implies that after terminating the application (either gracefully or after a crash) and re-executing, instances of the field in NVM will be initialized to a zero (null) value, not the value from the prior execution. Java uses this keyword for a similar purpose in serialization: a transient field is not serialized with an object and,

[4]Some object-oriented DBs [24] support virtual methods, but they differ from NVM because all data access occurs through a high-level, software-mediated interface.

after de-serialization, the transient field is null. Our definition directly follows the Java definition. If an instance of a class resides in NVM and a field is not marked `transient`, it is the responsibility of the programmer to ensure that the field's content has meaning after restarting the application.

The `reconstructor` keyword annotates a method that is invoked on each instance of a class that resides in NVM and survives a crash. The `reconstructor` method can be used to initialize transient fields to non-zero values. In a typical case, we expect the reconstructor code to follow the behavior of the object's `constructor` by reinitializing the transient fields, e.g. by creating a lock or opening a file. The reconstructor method can also modify non-transient fields. However, the reconstructor function can access only *the* object being reconstructed and DRAM-allocated objects. It must not access any other persistent object, which might still be invalid (not yet reconstructed).[5] We chose to invoke the `reconstructor` method after all transient fields of an object are set to null values. This reduces the risk of mistakenly using stale transient values during reconstruction.

Finally, the `pnew` keyword is equivalent to the `new` keyword in C++, except that the object is allocated in NVM instead of transient DRAM. It is possible to allocate instances of the same class, both in the transient heap (using `new`) and in the NVM heap (using `pnew`). If an instance is allocated via `new`, it is not durable and all annotations can be ignored. An object allocated via `pnew` must be deleted via `pdelete`. An alternative would have been to use `delete` for both types of objects and to detect at runtime where an object resides. This is unattractive as it introduces overhead at each invocation of `delete`, even if a application does not use NVM. Consequently, `delete` of a persistent object has undefined behavior. In contrast, `pdelete` of a transient object causes a runtime failure, as we assume that the persistent allocator can afford to check its arguments.

4 IMPLEMENTATION OF `NVMRECONSTRUCTION`

We have implemented `NVMReconstruction` as a Clang/LLVM extension and a runtime library. `NVMReconstruction` uses a runtime type system to safely recover objects for re-execution (reconstruction) and provides an efficient method for resolving fragmentation and for supporting upgrades (compaction).

4.1 Runtime Type System

Identifying objects in the persistent heap requires runtime type information. For each object *instance*, the system has to determine — for each of its field — whether it is a transient value, a non-transient (persistent) pointer, or a literal. A transient field needs to be zeroed, a non-transient pointer needs to be fixed (during reconstruction) or forwarded (if the pointed object is migrated), and a literal field remains unmodified.

Similarly to managed languages, the `NVMReconstruction` implementation uses a header in each object (8 bytes before its first byte) to store an object's type. The Clang/LLVM extension analyzes the code and produces a unique ID for each class type. Allocations via `pnew` allocate memory for an object and initialize the header to the type ID. In addition, the compiler emits a global structure. For each object type, this global structure records its type ID, the offsets of persistent pointers, a function for zeroing transient fields and for invoking the (user-defined) reconstructor, and the virtual table pointer for the class (if applicable). The compiler also emits type information for every function that might be used as the target of a function pointer (i.e., each function whose address is taken).

Although this provides necessary type information, it is insufficient to upgrade persistent objects. If an application is changed and compiled, the `NVMReconstruction` Clang/LLVM extension captures the current (upgraded) application semantics. We do not assume that the old version of

[5]Static analysis to warn of incorrect access to persistent objects is still work in progress.

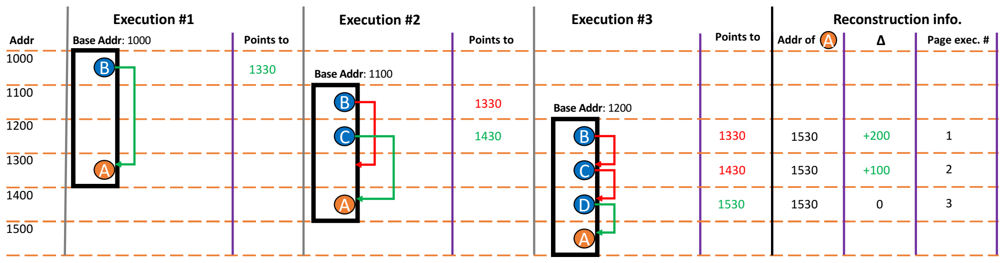


Fig. 1. Illustrating different virtual addresses for the same object.

the application is available to the compiler. This creates a mismatch: the persistent objects that need to be traversed might have been created by an older version of the application, whereas the Clang/LLVM-produced information corresponds to the latest version of the application.

Furthermore, even without any change to the application, the Clang/LLVM type information can be insufficient for interpreting pointers. As mentioned in Section 2, when an application restarts, the OS can map the persistent heap or the code region to a different virtual address. In such a case, a pointer needs to be interpreted according to the mapped address of the *prior execution*. Lazy reconstruction complicates this behavior even further. When an application stops, there is no guarantee that all blocks in NVM have been reconstructed and that, subsequently, all addresses can be interpreted according to the current address space. Hence, it is possible that different NVM objects need to be interpreted with respect to different address spaces.

Consider, for example, the scenario illustrated in Figure 1. It represents 3 different executions, each time the NVM heap is mapped to a different address. During the third execution, both Ⓑ, Ⓒ and Ⓓ points to Ⓐ. However, each of these pointers use a different virtual address, even though they all point to the same object Ⓐ.

To correctly interpret a persistent object created in an earlier execution, the `NVMReconstruction` runtime records information about the application in a dedicated *meta NVM heap*. Each execution is assigned a unique *execution number*. For each execution, `NVMReconstruction` stores the type information emitted by Clang/LLVM, the mapped addresses of the persistent heap, and the mapped addresses of the read-only data segments. The persistent heap is partitioned into *blocks* or *pages* that are multiples of virtual memory page size (typically 4K). For each block, `NVMReconstruction` stores the execution number by which the objects in the block should be interpreted (last column of Figure 1). The runtime maintains a mapping from execution number to a `executionInfo` that holds the runtime information for the corresponding execution. The runtime also collects the execution numbers of all blocks in the persistent heap and prunes `executionInfos` that do not correspond to any block.

4.2 Reconstruction

During re-execution, `NVMReconstruction` reconstructs the NVM heap by iterating over all live objects in the persistent heap and by *reconstructing* each object field-by-field:

Clear Transient Fields (Section 2.1) Each transient field is zeroed. This prevents the program from mistakenly using a stale transient field.

Pointer Relocation (Section 2.2) Each pointer into an NVM region is corrected to the current location of the target object. Virtual table pointers, function pointers, and pointers to read-only data are update to the correct location.

Customized Reconstructor Function Finally, a type-specific reconstruction function is invoked on the object. This enables a programmer to specify more complex recovery patterns, such as reinitializing transient fields.

Reconstruction works on a memory block. For each block with execution number e_{old} (earlier than e_{cur}), the system consults the execution information between e_{old} and e_{cur} stored in the meta NVM heap. This provides the necessary basis to interpret pointers and modify them so they can be dereferenced in the current execution. The system iterates over all live objects in the block and reconstructs each one, as described above.

To reconstruct a block, three challenges must be handled correctly and efficiently: recovery during reconstruction, fast application restart, and preventing threads from accessing a stale block.

The first challenge is that an application can crash while reconstructing a block. Reconstructing an object more than once can have a disastrous effect, because pointers are updated by adding the difference between the old mapped address and the new address adding the difference twice is wrong. To provide recoverability, we use undo logging at a block granularity. Before reconstructing a block, the entire block is copied to the meta NVM heap. If a failure occurs before reconstruction finishes, the block is reverted to its initial state before (another) reconstruction attempt. Once a block is processed, the system flushes its updated content to NVM and sets its execution number to the current execution number, thus signifying that its contents are up-to-date. Finally, it deletes the undo log.

The second challenge is fast recovery. To ensure that an application never accesses stale data, it is crucial that reconstruction runs before the application's first access to an object. However, NVM's high density and low power consumption makes possible extremely large memories, so iterating over the entire NVM heap could require hours for applications with large data sets, thus making recovery extremely expensive [23, 36].

We implemented `NVMReconstruction` recovery in a lazy manner, which allows an application to start execution immediately and its recovery to run concurrently. If the application accesses an object before it is reconstructed, then the application must be delayed until the object is reconstructed. Detecting these accesses employs a memory-protection mechanism to trap the first access to a block of memory (one or more contiguous pages), so that all live objects in the block can be reconstructed before the application resumes. Our algorithm is summarized in Listing 2. The block size is a parameter of the system, but it must be a multiple of the virtual memory page size and no object can cross a block boundary. We intend to support objects larger than a block in future work.

The third challenge is to grant the reconstructor thread read-and-write access to a block, without allowing access by other application threads. We use a well-known technique that maps the persistent heap twice with different memory protections. The first mapping is the standard persistent heap used by the application. A block starts as inaccessible in this mapping and becomes accessible when reconstructed. The second mapping is used only by reconstruction and allows full read-and-write access to the blocks.

4.3 Compaction

The reconstruction algorithm above works when object migration is not needed. However, sometimes object migration is necessary, either due to a change that enlarges an existing object or due to a serious memory fragmentation in the persistent heap. In these cases, `NVMReconstruction` compaction is similar to a compacting garbage collector.

But, unlike reconstruction, compaction runs fully offline and is not interleaved with the execution of the application. Offline compaction is acceptable because we expect it to be invoked infrequently

Listing 2. NVMReconstruction

```
LazyReconstructHeap() {
    registerHandler(memoryTrap, memoryTrapHandler); // Run before application is started
    mprotect(NVM, NO_READ|NO_WRITE); // Trap at first access to NVM blocks
    in background do: // Reconstruct concurrently with application
        foreach block b in persistentAllocator.getBlocks():
            ReconstructBlock(b);
}
memoryTrapHandler(blockAddress) {
    ReconstructBlock(blockAddress);
}
ReconstructBlock(blockAddress) {
    Block &block = getBlockFromAddress(blockAddress);
    block.lock();
    if (block.is_processed())
        {block.unlock(); return; }
    backup(block);
    PtrFixer *fixer = getFixerForExecution(block.getExecutionNumber());
    munprotect(block, READ|WRITE, thisThread); // Unprotect block only for this function
    foreach object O in persistentAllocator.getLiveObjectsInBlock(block):
        ReconstructPersistentObj(fixer,O);
    flush(block);
    block.setExecutionNumber(currExecutionNumber);
    munprotect(block, READ|WRITE); // Unprotect block for everyone
}
ReconstructPersistentObj(fixer,O) {
    Type *ty = getType(O−>header);
    foreach field p in ty−>getPointerFields()
        O−>p = fixer.computeNewMapping(O−>p);
    // zero transient fields and call user−defined reconstructor
    ty−>zeroAndReconstruct(O);
}
```

and its cost is significantly higher than reconstruction. Running offline greatly reduces the complexity of compaction, as the pointers that are forwarded all reside in memory, not in registers and stacks in a running process. Thus, our compaction algorithm does not place restriction on how machine code is generated and how it manipulates pointers. Only the *persistent heap pointers*, pointers from a persistent object to another persistent object, must be in a standard format (e.g., XORing two pointers and storing the result in a register is permitted, but storing the result in a heap object is not).

Although compaction runs offline, the heap's state can still reflect many different executions, each with a different execution number. Compaction starts by reconstructing the entire heap to a consistent (the current) execution, in which all pointers are valid for the current mapped address.

The compaction algorithm works in four stages. First, it gathers the *roots* of the NVM persistent heap. These are pointers to persistent objects that are located at a known position. These roots can be accessed, even if the application does not have a pointer to a persistent object. After a crash, only objects reachable from a persistent root can be accessed by the application when it restarts. Second, the algorithm computes the transitive closure of the roots, marking (in a side markbit table) all live objects reachable from these roots. It also creates a new — empty — persistent region, denoted *to-space*. Third, the algorithm computes a forwarding pointer for each live object. If an object is upgraded, the forwarding pointer is computed from the new size of the object. Fourth, each object is copied to its new location in to-space, during which each pointer field is modified according to the forwarding pointer of the target object. The from-space persistent region is deleted, making the to-space the valid copy of the persistent heap.

The compaction algorithm does not modify the from-space. Hence, supporting fault tolerance is trivial. If the system crashes before the upgrade finishes, the (incomplete) to-space is discarded and the process begins from scratch.

Compacting is invoked by the application with the `NVM_Compact` API. `NVM_Compact` must not be invoked when the NVM region is opened for access by the application. It can be used either before opening the NVM region (when de-fragmentation is necessary) or after closing it (to reduce NVM consumption).

5 DISCUSSION

5.1 Atomicity-Enforcement Component

`NVMReconstruction` is orthogonal to the choice of the failure-atomicity enforcement system (e.g., transactional system) that *recovers* the heap state to a consistent point after a crash. However, these two subsystems interact. A failure-atomicity enforcement system typically relies on a persistent log for recovery. These logs contain pointers to objects being modified. Recovery is independent if these pointers are implemented as offsets. However, if these pointers are direct addresses, they need to be reconstructed *before* recovery to correctly apply the log during recovery. Reconstruction and recovery can run in either order. `NVMReconstruction` can reconstruct the pointers in the persistent log records before recovery is invoked. Or, reconstruction can be integrated into recovery so that log records are updated and recovery run before `NVMReconstruction`'s reconstruction phase.

5.2 Persistent Allocator

Instead of incorporating an allocator into `NVMReconstruction`, we consider the persistent allocator as an orthogonal component. Persistent allocators are often integrated with an atomicity enforcement system, and this separation permits use of novel memory allocators without modifying `NVMReconstruction`. `NVMReconstruction` requires the persistent allocator to iterate over the set of blocks in the NVM heap and, for each block, to iterate over its live objects. `NVMReconstruction` also requires the persistent allocator to provide an API to enumerate the roots of the NVM region. For example, Atlas's allocator [10] provides the `NVM_SetRegionRoot` and `NVM_GetRegionRoot` methods for setting and reading the root of an NVM region.

Blocks must be a multiple of the virtual memory page size (typically 4K) and objects are not permitted to cross a block boundary. In general, it is easy to add support to a persistent allocator to iterate over live objects. However, partitioning the heap into blocks never crossed by an object is not always easy. If a system cannot implement this capability, it is still possible to use the non-lazy algorithm by treating the entire heap as a single block.

Another restriction is that persistent memory allocation *must not* be invoked directly by an application. Since user code does not properly initialize the header of an object, it will be missing the runtime type information necessary to interpret the object. `pnew` must be used exclusively to create persistent objects, as it ensures that a correct type ID is stored in the object's header.

5.3 Limitations

`NVMReconstruction` places some restrictions on an application. If the application uses integers derived from pointers, e.g., hash values, these values are not automatically updated when addresses in the NVM change. In this case, a programmer must provide a `reconstructor` method to recalculate these hash values. Similarly, pointers coerced to integer values or stored in a scalar-typed buffer are not updated by the system. Again, the programmer can write a `reconstructor` method to fix these pointers.

`NVMReconstruction` was designed to use the NVM memory as a persistent heap owned by a single process and opened for read/write access. Sharing the NVM region between multiple processes is not supported. This poses challenges if processes map the NVM region to different virtual addresses, in which case indirection is necessary. Opening an NVM region for read-only access could be supported, but `NVMReconstruction` still requires its additional mapping for reconstruction that allows both reading and writing.

Cross-region pointers are supported only if the two regions are loaded simultaneously. In such a case, the regions are treated as a single heap with non-continuously virtual mapping.

`NVMReconstruction` does not support union types that overlay two (or more) fields in the same memory location. The problem is that the system cannot know how to interpret a field during reconstruction. Unions are not supported in managed languages with garbage collectors, such as Java and C#, for a similar reason.

Our current implementation supports modifying a class only by adding fields at its end. Reordering fields is not yet supported. Another limitation is that the name of a modified class must remain unchanged so that the system can identify object instances to upgrade. Similarly, the name of a function pointed to by a persistent object must not be modified. Allowing renaming is future work.

5.4 Language Choice

We built `NVMReconstruction` as a C++ language extension, a natural choice since most high-performance systems are written in unmanaged languages, and most research on NVM programming uses them as well. Another option would have been to use a managed language such as Java or C#. This would have simplified aspects, as these languages already have a runtime type system and a garbage collector. However, we would have needed to implement reconstruction in a large and complex runtime system. We leave that effort to braver individuals and are confident that the ideas from `NVMReconstruction` will carry over to implement reconstruction for persistent managed objects.

6 DEMONSTRATION AND MEASUREMENT

`NVMReconstruction` trades performance — the time to reconstruct all objects in the NVM heap — for programmability. Thus, we evaluate `NVMReconstruction` along three dimensions. First, what is the effect of `NVMReconstruction` on an application's performance? What is the overhead of reconstruction as well as the cost of compacting the heap? Second, how much effort is needed to employ the `NVMReconstruction` annotations? Third, can durable data survive OS upgrades and other modifications to the system? These three questions are considered below.

Except when specified otherwise (Section 6.3), all experiments were executed on a machine containing an Intel i7−6700 CPU @ 3.40GHz with 2 Kingston 8GB DDR4 DRAM @ 2133 MHz running Linux Ubuntu 16.04. `NVMReconstruction` was incorporated into Clang version 5.0.0, which was used to compile all code.

6.1 ReAtlas

Atlas [10] is a system designed to simplify or eliminate the task of identifying persistent transactions by using existing synchronization primitives in a multi-threaded application to identify the points in execution at which the program's state is consistent (i.e., the boundaries of persistent transactions). We combine Atlas and `NVMReconstruction` as the systems are complementary and focus on different aspects of NVM programming. NVMReconstruction, however, could be have been equally well integrated with other NVM transactional systems [12, 27, 34, 37, 46]. Atlas has several limitations that NVMReconstruction corrects. It did not permit NVM to be mapped to different addresses across executions. If the operating system was unable to satisfy the requested address, Atlas fails immediately. Atlas also does not support persistent objects that contain virtual methods, function pointers, or pointers to read-only data.

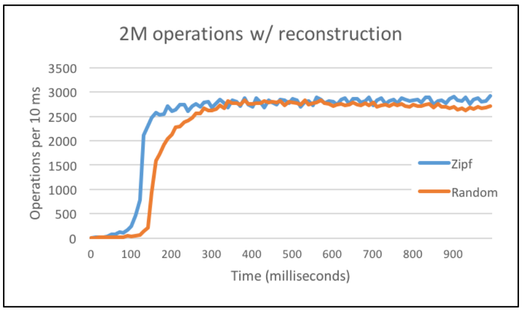
The integrated system is called ReAtlas. It extends Atlas and supports changing mmap addresses and persistent objects that contain virtual methods, function pointers, and pointers to read-only data. NVM is emulated using the /dev/shm directory in Linux, which is backed by DRAM. Atlas uses an undo log to ensure failure-safety, with clflushes used to ensure write ordering to NVM. The key difference of ReAtlas, as compared to the original Atlas, is the reconstruction, which we report below. After reconstruction completes, the overhead of ReAtlas, as compared to Atlas, was less than 0.9%. We also report compaction time, an optional feature of ReAtlas.

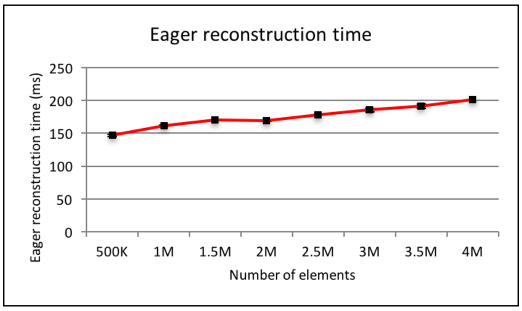
ReAtlas affects performance because of the page faults that occur when objects are reconstructed. Initially, every NVM memory access triggers a page fault and reconstruction, leading to a “fault storm” and poor performance. After the initial working set is reconstructed, reconstruction runs mostly in the background and has little effect on performance.

To measure these costs, we conducted the following experiment. First, we built a simple key-value store based on a hash map and ran it in ReAtlas. During the initial execution, a key-value store populated with 500K elements was put in a 1GB NVM heap. Each element was a mapping from an 8-byte key to a data buffer of 1000 bytes. After populating the hash map, 1M operations were executed. The operation distribution was 50% reads and 50% puts, similar to YCSB workload A (write heavy). Then, the application terminated and re-executed. Note that no reconstruction was needed during the initial execution and no NVMReconstruction performance penalty was observed.

During re-execution, we used the same operation distribution (50% reads, 50% puts) and measured the number of operations executed in each 10ms time interval. Recall that reconstruction is frequent at the beginning of the execution, during the fault storm, and fades as execution continues. We measured two configurations: one in which keys are chosen uniformly at random and the other in which keys are picked using a Zipfian distribution with skew 1. The experiment was executed 10 times and we report the average throughput for each 10ms time interval. Results are depicted in Figure 2. In both configurations, we observe the fault storm, but its duration differs. With the Zipfian distribution, the system executes 4–10 operations during the first 10 milliseconds. After 80 milliseconds, throughput is above 120 operations per 10 milliseconds. Finally, after 140ms, throughput reaches 90% of peak performance. With the random distribution, the system also executes 4–10 operations during the first 10 milliseconds, but reconstruction takes longer. It takes 140ms to reach a throughput of 120 operations per 10 milliseconds and 260ms to reach 90% of peak performance. The main difference between the two is the working set size. The working set with the Zipfian distribution is small, so popular keys are reconstructed early and cease to affect performance. By contrast, the working set for the random distribution is larger, so reconstruction affects performance for a longer time.

The principal cost of reconstruction is backing up and flushing pages that are reconstructed. The actual reconstruction effort per object is small. To measure these contributions, we slightly modified the hash map experiment. We fixed the amount of NVM memory at 0.5GB to keep the number of blocks constant. As we increased the number of elements in the hash map, from 500K to 4M, we proportionally decreased the size of the data buffer in each element to keep the total memory usage constant. To avoid inference with the application workload, we measured reconstruction time using eager reconstruction, which ran before the application started. Results are depicted in Figure 2. Comparing the two extreme points (500K and 4M elements), we see that the number of





(a) (b)

Fig. 2. Measuring reconstruction. (a) Throughput of the key-value store in 10ms time intervals (higher is better). Initially, most accesses cause reconstruction, but the overhead quickly decreases once the popular elements are reconstructed. Zipfian distribution converges faster since its working set is smaller than randomly distributed keys. (b) Time to finish eager reconstruction (lower is better). x-axis reports the number of elements in the hash map. The size of each element is decreased so that the total amount of memory used remains fixed. The y-axis reports eager reconstruction time. The performance change is not as large as the increase in the number of objects since most of cost is attributable to the number of pages of memory accessed.

reconstructed objects increased by 8x, whereas reconstruction time increased only by 37%. The overhead of reconstruction comes from backing up data before reconstructing a block, with only a slight cost to process each object on a page.

Lazy reconstruction enables an application to start providing functionality almost immediately, and with this small benchmark, it reaches full throughput in less than 300ms. The reconstruction penalty and interval obviously depends on the working set size and secondarily on the number of objects.

Next, we measure the effect of compaction. Recall that compaction is an optional feature of `NVM-Reconstruction`, which is explicitly activated before accessing the key-value store. On a workload with 50% reads, 50% puts, and 500K elements, each of size 1000 bytes, compaction requires 1.5 seconds, during which no key-value operations are executed. We also measured how the number of items in the heap affects performance. We used the same technique as before: fix the NVM region size to 0.5 GB and vary the number of elements in the hash table. The results are depicted in Figure 3. Unlike reconstruction time, compaction is strongly dependent on the number of elements in the heap. Tracing the heap and allocating objects in the to-space dominates the cost of compaction. Both have a cost proportional to the number of elements in the hash map.

Compaction, however, can improve the overall performance of the system by de-fragmenting the NVM heap. To show this effect, we conducted the following experiment. The operation distribution is 50% reads, 25% puts, and 25% removes. The heap was 65%–70% occupied. Each element is a mapping from an 8-byte key to a data buffer of length k, where k is picked uniformly at random in the range 8–2000 bytes. Due to this randomness, a newly created element is unlikely to fit exactly in the space freed by previous remove operations. Initially, the allocator satisfies most allocations from the current memory arena, which is relatively inexpensive. However, as the heap become fragmented, allocation requires a more expensive traversal of the heap, leading to lower performance.

Initially, the NVM heap was allocated and fragmented by running 4M operations. Then the application was terminated. We then ran 2M additional operations in two configurations. The

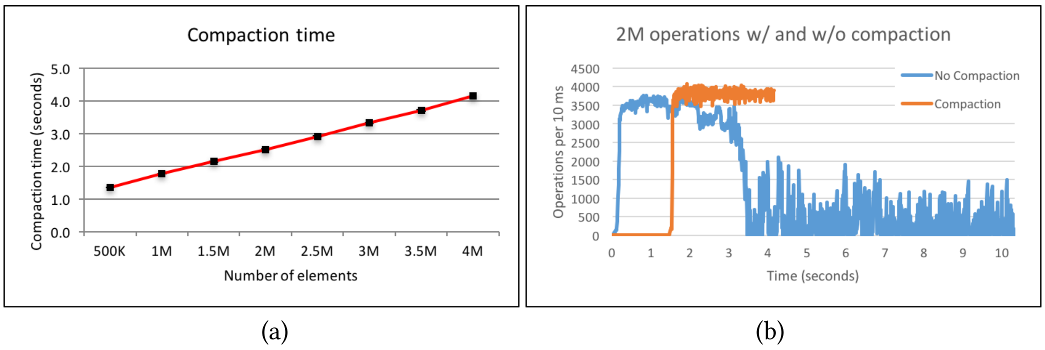


Fig. 3. Measuring compaction time. (a) Time to compact the entire heap (lower is better). x-axis reports the number of elements in the hash map. The size of each element is decreased so that the total amount of memory used remains fixed. The y-axis reports compaction time. Compaction time is clearly dependent on the number of elements in the heap. (b) Throughput of a key-value store with and without compaction (higher is better). With compaction, initial throughput is zero, but it quick catches up as it does not suffer from memory fragmentation that lowers performance.

first executed a compaction before starting execution, while the second immediately started execution (and reconstruction). We executed the experiment 10 times and observed similar trends, but we report one run of the experiments in Figure 3 to avoid "smoothing out" the results (with fragmentation, results are highly variant).

For the first 1.5 seconds, the no-compaction version runs faster, as the other version is not yet running the application. Even without compaction, throughput is initially high as the allocator manages to recycle memory locally, avoiding expensive searches for free space. After the first 1.5 seconds, the situation reverses. With compaction, the heap is defragmented and operations run at peak throughput. But without compaction, the allocator is often unable to locally recycle a node and instead reverts to an expensive search for a free chunk. Performance is more than 4x lower than peak throughput and the variance is high. Overall, 2M operations finished in less than 5 seconds with compaction, but required more than 10 seconds without compaction. In this case, compaction clearly pays off. This suggests that if a heap becomes highly fragmented, compaction is extremely useful and eventually improves overall performance.

We note that the experiments above used the Atlas memory allocator and could perform differently on other memory allocators. However, there exist programs that create high fragmentation for any possible allocator [15, 42]. Another motivation for compaction is the ability to upgrade an application and increase the size of its existing objects. This capability does not improve performance, but it does greatly improves the practicality of storing durable objects in NVM.

6.2 Echo Key-Value Store

Echo [7] is a key-value store designed specifically for NVM. It is part of the Whisper benchmark suite [39] and performs well. It has been shown to significantly outperform Google's LevelDB [21] (when using ramdisk for storage) and to perform similarly to the Masstree transient key-value store. However, its authors did not implement recovery code for data residing in NVM. Without recovery code, an Echo DB is effectively transient as durable data cannot be retrieved after a crash. The benchmark always allocates a new instance of the key-value store and measures its performance. Thus, this program offers an opportunity to measure the complexity of adding recovery to an application designed for NVM.

Table 1. Number of line changes to use NVMReconstruction in Echo.

pnew	pdelete	realloc extras	transient	reconstructor	**Total**
38	68	25	19	64	214

There are several challenges in adding recovery to Echo. NVMReconstruction easily solves these problems. Echo frequently uses mutexes to synchronize access to an object. These mutexes are allocated in DRAM and are lost during a crash, but they are pointed to by objects in NVM. Echo does not contain any code for checking the validity of these pointers or for initializing them during recovery. It simply assumes that after initializing an object, all its fields remain valid, even across crashes. To solve this problem, we annotated these mutex pointers as transient and wrote a reconstructor method that initialized a mutex pointer with a freshly allocated mutex. Without NVMReconstruction, initializing these mutexes would require thorough understanding of the Echo system, either to iterate over all persistent objects or to capture the first access to an object. Adding transient annotations and reconstructors required less than a day, even though we possessed no prior knowledge of the system.

Echo also uses function pointers to customize the behavior of the internal hash map. If these functions are mapped to a different address after recovery, the existing Echo system is likely to crash when it uses a function. NVMReconstruction solves this problem by correctly remapping function pointers on re-execution.

An additional issue arose with NVM pointers. The memory allocator of Echo is designed to use offset-based pointers, so allocation runs correctly when the heap is mapped at a different address than the previous execution. However, the key-value store implementation used native pointers. It exhibits undefined behavior if the kernel is unable to map the heap file to the same virtual address as the previous execution.[6] The NVMReconstruction neatly solves this problem by updating pointers.

Echo uses C-style allocations for persistent objects. To store runtime information for recovery, we modified Echo to use the allocation/deallocation keywords (e.g., pnew/pdelete) instead of the existing C malloc functions for objects that can be persistent. For re-allocations, we created a new storage space, copied the object, and deleted the old storage.

These changes affect 214 lines (SLOC) of Echo's 22503 SLOC of code. The breakdown of these modifications is shown in Table 1. Overall, the Echo system defines 34 types (structs), 17 of which can be allocated in the persistent heap. Out of these 17 types, we had to modify 7 types to define a special treatment on recovery: either some of the fields were transient or we had to define a reconstructor. There was no need to modify the other 10 types, even though they contained pointer fields that need to be reconstructed, as these are automatically handled by our Clang/LLVM extension.

Overall, Echo demonstrates that implementing recovery code without NVMReconstruction is difficult. By contrast, when NVMReconstruction was used, writing the recovery code was easy, requiring little understanding of the Echo internals. Integrating NVMReconstruction into the Echo system and NVM allocate was also simple, again showing that NVMReconstruction's interface is easy to integrate into an existing NVM system.

We also measured the performance of the Echo key-value system. We inserted a software-induced crash into Echo's test run, which simulated a power failure. To avoid affecting performance measurements, the crash occurred during the population stage of the test. After the failure, Echo

[6]In our experiments, the kernel was always able to supply the requested virtual address, so we did not observe this behavior. Of course, as discussed above, the OS need not behave this way.

was ran again from the failed location, resuming the test. We find that the overall cost of `NVMReconstruction` reconstruction is 0.35 seconds for a 1GB size heap. This accounts for 3% of the total time in the Echo test, which runs for 10 seconds. When compaction is applied, the total initialization overhead increases to 1.73 seconds.

6.3 Portability Test

Durable data is expected to exist for a long time. It must be able to survive the evolution of the computation environment, such as OS updates. To test the `NVMReconstruction` system, we wrote a simple polymorphism test and executed it twice, once allocating objects in a heap and another time using these objects. We attempted to modify the execution environment as much as possible between the two executions. First, we ran the first execution on a server running Ubuntu 16.04 and the re-execution on a Mac laptop running OS X EI Capitan 10.11.6. Between the two executions, the heap file was copied to the other machine. Second, we used -O3 optimization flag for the initial execution and -O0 for the re-execution. Third, we modified the code between the two executions by adding a boolean field to each class type. Fourth, we modified the mapped address of the heap file. For the initial execution, the heap file was mapped to address 2^{40}, whereas for the second execution it was mapped to address $3 \cdot 2^{40}$. Finally, we ran the initial execution under gdb and the re-execution natively. All these changes ensured that the execution environment differed greatly between executions. The only aspect we explicitly did not change was the name of the classes.

The test consisted of two stages. The first stage allocates various objects and executes only during the initial execution. The second stage executes during both invocations and prints the objects. We ensured that the code for printing objects had no static information about the type of the objects. The test defines three persistent types. The first is a `printableElement`, consisting of a pointer to `printableElement` and an abstract virtual print method. Using the pointers, it is possible to form a linked list of `printableElements`. The second type is a templated `printableInt` that derives from `printableElement`; the print method prints the templated integer argument. We verified (via the `sizeof` operator) that the templated argument was a compile-only variable and was not stored in the object. The third type is `printableHello` that derives from `printableElement`. The print method simply prints “hello world”.

The first stage of the test application, executed only during the initial execution, allocates objects in a random order and creates a linked list of them. It also sets the root pointer to the first element in the list. The second stage, executed during initial execution and re-execution, iterates over the linked list and invokes the virtual print method. The code that iterates over the linked list has no access to the definition of `printableInt` and `printableHello`; only `printableElement` pointers are used. Despite the many environmental differences between the initial execution and re-execution, both executions produce identical output. This demonstrates that `NVMReconstruction` is able to endure drastic changes in the execution environment and to preserve the identity of objects.

7 RELATED WORK

The idea of using C++ to access durable storage dates back to the days of Object-Oriented Database Management Systems (OODBMS) [3, 5, 6, 24, 25, 29, 31, 45]. As these systems used a hard-disk as the underlying durable media, entire pages were written to the disk. By contrast, NVM enables a fine-grained persistence, in which the cost of persistence is a cacheline flush. This motivates intermixing transient and persistent fields and so requires new reconstruction mechanisms. In addition, OODBMS are closed worlds, which do not allow pointers to code or to data in a process to be stored in a DB.

The ObjectStore OODBMS [31] enabled programmers to store arbitrary C++ objects durably. It relied on memory-protection operations to trap accesses to durable objects. At the beginning of a transaction, durable memory was protected so that accesses triggered a memory trap. Traps are expensive and the resulting overheads were unacceptable for small, fine-grained transactions. `NVMReconstruction` also uses traps, but they occur only on the first access to a group of objects and so have low overhead. Another OODBMS based on C++ is ONTOS [2]. We are not aware of papers describing the details of this system.

Pointer swizzling [38, 47] is a technique for storing wide (64-bit) pointers in an OODBMS and converting them into 32-bit virtual addresses. Pointer swizzling can be applied during page-fault time [44, 48], significantly reducing the overhead for applications with good locality. When data is written back to a disk, 32-bits virtual address pointers are un-swizzled into the wide 64-bits format. Our solution for correcting mmap'ed addresses is similar in respects to pointer swizzling but only uses a pointer single representation.

Most systems for NVM programming focus on implementing a *transactional protocol* that can recover a consistent snapshot of the NVM heap when an application crashes. Mnemosyne [46] is the first system that supports durable transactions, based on TinySTM [19]. DudeTM [34] improves the efficiency of transactional support by decoupling transactional concurrency control from durability, which is performed in the background. LSNVMM [27] improves the efficiency of transactional support by avoiding replicating data in a log. The system stores a single copy of data in a log and is able to reference it efficiently. Kamino-TX [37] improves the efficiency of transactional support by using a redo log, while avoiding the need for special lookup. The system stores a DRAM copy of data for reading and asynchronously updates the durable copy. NVThread [12] improves the efficiency of transactional support by spawning threads as separate processes and using virtual memory mapping to create a thread-local view of the memory. Kolli et al. [30] improves the performance of durable transactions by pipelining multiple stages in order to reduce the number of flushes. Cohen et al. [14] reduces the number of flushes required for a durable log by carefully utilizing the NVM memory consistency model.

The transactional protocol is mostly orthogonal to this paper. As shown in Section 2, even when the application terminates gracefully (i.e., not in the middle of a transaction), the meaning of an object might change between different executions.

Durable data structures that do not rely on transactional systems have been considered by many authors [7, 28, 43, 50, 51]. These efforts share the goal of transactional systems of avoiding exposing the intermediate state of NVM (which is inconsistent) and are also orthogonal to this paper.

Other work focuses on reducing the programming effort required to use NVM. Atlas [10] simplifies identifying durable transactions. It is based on the observation is that for multi-threaded programs, if no locks are held, then memory is in a consistent state. Thus, locks can define a *durable transaction*. Combining `NVMReconstruction` with Atlas results in a more capable and resilient system (Section 6) that enables a programmer to persist objects with virtual methods and enables the system to operate when the kernel is unable to `mmap` NVM heap file into the requested address.

Makalu [9] is a conservative garbage collection (GC) for NVM. As it is not integrated with a runtime type system, it treats every field that could be a pointer as a possible pointer. A conservative GC can prevent most persistent memory leakages (even during crashes). It also simplifies memory allocation, because a programmer need not ensure that memory is not lost during program failure. However, Makalu cannot migrate objects or fixed pointers because it cannot accurately identify pointer fields. Makalu could be profitably combined with `NVMReconstruction` to obtain these benefits.

Coburn et al. [13] propose a restrictive programming model for NVM to enforce "good" programming practices. They forbid pointers from NVM regions to DRAM or to other NVM regions. They also propose wide pointers, consisting of a region ID and offset, to enable an NVM region to be mapped to a different address. Finally, they suggested *generational locks* that are implicitly released after a restart. These restrictions may involve extensive changes to many applications, unlike `NVMReconstruction`, which cleanly isolates recovery and allows most of an application to use NVM-like conventional memory.

NVMDirect [16] is a C language extension for NVM. Pointers to NVM are implemented as self-relative offsets. It defines a syntax extension to help a programmer avoid confusing standard and NVM pointers. NVML [17] is another system designed for NVM. It uses a macro-based solution to implement NVM pointers as pool ID and offset pairs. `NVMReconstruction` uses more efficient direct memory pointers and only requires a programming to annotate transient fields in NVM-resident objects. SAVI [18] is a system that enables the sharing of objects with virtual functions across different processes. Such objects cannot be passed directly to other processes as virtual table entries can differ between the processes. The authors consider two techniques to handle this problem: duplicating virtual table entries across processes and hash-based lookups in a global virtual table. `NVMReconstruction` also tackles the problem of sharing objects with virtual functions, but across reboots instead of across processes.

8 CONCLUSION

This paper presents `NVMReconstruction`, a programming language extension to C++ and a runtime system that supports the recovery of persistent objects and the re-execution of applications that use non-volatile memory (NVM). A simple language extension ensures that durable objects stored in NVM can be properly updated and reinitialized when an applications restarts. This extension and runtime library reestablishes the consistency of the persistent data structures in the environment when resuming execution, which is orthogonal to the intensively studied atomicity constructs used to ensure the consistency of NVM during execution. `NVMReconstruction` supports NVM objects with transient fields, even if these fields require non-trivial reconstruction. It also supports, even across program restarts, direct pointers to persistent objects, virtual tables, functions, and read-only data. The system incurs negligible overhead after reconstruction, which itself is relatively inexpensive and can be performed lazily to overlap reconstruction with computation. `NVMReconst-ruction` also provides compaction to support infrequent application upgrades that modify object formats and to reduce NVM fragmentation.

ACKNOWLEDGMENTS

We would like to thank Virendra Marathe for many helpful discussions and for comments on an earlier version of the paper.

REFERENCES

[1] 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

[2] Tim Andrews. 1992. The ONTOS object database. In *Data Manag. 91 - DM91*. Ashgate Publishing. http://dl.acm.org/citation.cfm?id=144139

[3] Timothy Andrews and Craig Harris. 1987. Combining language and database advances in an object-oriented development environment. In *Object Oriented Program. Syst. Lang. Appl. - OOPSLA '87*. ACM, 430–440. https://doi.org/10.1145/38765.38847

[4] Joy Arulraj and Andrew Pavlo. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *2015 Int. Conf. Manag. Data*. 707–722. https://doi.org/10.1145/2723372.2749441

[5] Malcolm Atkinson and Ronald Morrison. 1995. Orthogonally Persistent Object Systems. *The VLDB Journal* 4, 3 (July 1995), 319–402. http://dl.acm.org/citation.cfm?id=615224.615226

[6] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. 1983. An Approach to Persistent Programming. *Comput. J.* 26, 4 (1983), 360–365. https://doi.org/10.1093/comjnl/26.4.360

[7] Katelin A. Bailey, Peter Hornyack, Luis Ceze, Steven D. Gribble, and Henry M. Levy. 2013. Exploring storage class memory with key value stores. In *1st Work. Interact. NVM/FLASH with Oper. Syst. Workload. - INFLOW '13*. ACM, 1–8.

https://doi.org/10.1145/2527792.2527799

[8] A. Bensoussan, C. T. Clingen, and R. C. Daley. 1972. The Multics virtual memory: concepts and design. *Commun. ACM* 15, 5 (may 1972), 308–318. https://doi.org/10.1145/355602.361306

[9] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: fast recoverable allocation of non-volatile memory. In *21st Object Oriented Program. Syst. Lang. Appl. - OOPSLA 20116*. ACM, 677–694. https://doi.org/10.1145/2983990.2984019

[10] Dhruva R Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *19st Conf. Object Oriented Program. Syst. Lang. Appl. - OOPSLA 2014*. ACM, 433–452. https://doi.org/10.1145/2660193.2660224 arXiv:2660224

[11] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 497–508. https://doi.org/10.14778/2735479.2735483

[12] Terry Ching, Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-threaded Applications. In *12th Eur. Conf. Comput. Syst. - EuroSys '17*. ACM, 468–482. https://doi.org/10.1145/3064176.3064204

[13] Joel Coburn, Am Caulfield, and A Akel. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *16th Archit. Support Program. Lang. Oper. Syst.- ASPLOS '11*. ACM, 105–118. https://doi.org/10.1145/1961295.1950380

[14] Nachshon Cohen, Michal Friedman, and James R Larus. 2017. Efficient Logging in Non-Volatile Memory by Exploiting Coherency Protocols. *Proc. ACM Program. Lang. - PACMPL* OOPSLA, Article 67 (Oct. 2017), 24 pages. https://doi.org/10.1145/3133891

[15] Nachshon Cohen and Erez Petrank. 2013. Limitations of partial compaction. In *34th Program. Lang. Des. Implement. - PLDI'13*, Vol. 48. ACM, 309. https://doi.org/10.1145/2499370.2491973

[16] Oracle Corporation. 2017. NVM Direct Library. http://www.oracle.com/technetwork/oracle-labs/open-nvm-download-2440119.html

[17] Krzysztof Czurylo and Andy Rudoff. 2014. NVML: NVM Library. https://github.com/pmem/nvml

[18] Izzat El Hajj, Thomas B. Jablin, Dejan Milojicic, and Wen-mei Hwu. 2017. SAVI objects: sharing and virtuality incorporated. In *Object Oriented Program. Syst. Lang. Appl. - OOPSLA'17*, Vol. 1. ACM, 1–24. https://doi.org/10.1145/3133869

[19] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *13th Princ. Pract. Parallel Program. - PPoPP '08*. ACM, 237. https://doi.org/10.1145/1345206.1345241

[20] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-free Queue for Non-volatile Memory. In *23th Princ. Pract. Parallel Program. - PPoPP '18*. ACM, 28–40. https://doi.org/10.1145/3178487.3178490

[21] Sanjay Ghemawat and Jeff Dean. 2011. LevelDB.

[22] E. R. Giles, K. Doshi, and P. Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *31st Mass Storage Systems and Technologies - MSST '15*. 1–14. https://doi.org/10.1109/MSST.2015.7208276

[23] Aakash Goel, Bhuwan Chopra, Ciprian Gerea, Dhruv Mátáni, Josh Metzler, Fahim Ul Haq, and Janet Wiener. 2014. Fast database restarts at facebook. In *Int. Conf. Manag. Data - SIGMOD '14*. ACM, 541–549. https://doi.org/10.1145/2588555.2595642

[24] Antony L. Hosking and Jiawan Chen. 1999. Mostly-copying reachability-based orthogonal persistence. In *14th Object Oriented Program. Syst. Lang. Appl. - OOPSLA '99*. ACM, 382–398. https://doi.org/10.1145/320384.320427

[25] Antony L. Hosking and J. Eliot B. Moss. 1993. Protection traps and alternatives for memory management of an object-oriented language. In *14th Symp. Oper. Syst. Princ. - SOSP '93*. ACM, 106–119. https://doi.org/10.1145/168619.168628

[26] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. 2005. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram. In *IEEE Int. Devices Meet.* IEEE, 459–462. https://doi.org/10.1109/IEDM.2005.1609379

[27] Qingda Hu, Jinglei Ren, Anirudh Badam, and Thomas Moscibroda. 2017. Log-Structured Non-Volatile Main Memory. In *Annu. Tech. Conf. - ATC 17*. USENIX. http://jinglei.ren.systems/lsnvmm{_}atc17.pdf

[28] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *30th Int. Symp. Distrib. Comput. - ISDC '16*. 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

[29] Alfons Kemper and Guido Moerkotte. 1994. *Object-oriented Database Management: Applications in Engineering and Computer Science*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[30] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *21nd Archit. Support Program. Lang. Oper. Syst. - ASPLOS '16*. ACM, 399–411. https://doi.org/10.1145/2872362.2872381

[31] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. 1991. The ObjectStore Database System. *Commun. ACM* 34, 10 (Oct. 1991), 50–63. https://doi.org/10.1145/125223.125244

[32] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *36th Int. Symp. Comput. Archit. - ISCA '09*. ACM, 2. https://doi.org/10.1145/1555754.1555758

[33] Linux. 2014. DAX: Page cache bypass for filesystems on memory storage. https://www.kernel.org/doc/Documentation/filesystems/dax.txthttps://lwn.net/Articles/618064/

[34] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *22nd Archit. Support Program. Lang. Oper. Syst. - ASPLOS '17*. ACM, 329–343. https://doi.org/10.1145/3037697.3037714

[35] Stephen Magill, Michael Hicks, Suriya Subramanian, and Kathryn S. McKinley. 2012. Automating object transformations for dynamic software updating. In *Object Oriented Program. Syst. Lang. Appl. - OOPSLA '12*. ACM Press, New York, New York, USA, 265. https://doi.org/10.1145/2384616.2384636

[36] Virendra J Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *HotStorage '17*. USENIX. https://www.usenix.org/system/files/conference/hotstorage17/hotstorage17-paper-marathe.pdf

[37] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *12th Eur. Conf. Comput. Syst. - EuroSys '17*. ACM, 499–512. https://doi.org/10.1145/3064176.3064215

[38] J. Eliot B. Moss. 1992. Working with persistent objects: to swizzle or not to swizzle. *IEEE Trans. Soft. Eng.* 18, 8 (1992), 657–673. https://doi.org/10.1109/32.153378

[39] 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 *22nd Archit. Support Program. Lang. Oper. Syst. - ASPLOS '17*. ACM, 135–148. https://doi.org/10.1145/3093315.3037730

[40] Team PaX. 2003. PaX address space layout randomization (ASLR).

[41] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *36st Int. Symp. Comput. Archit. - ISCA '09*. ACM, 24–33. https://doi.org/10.1145/1555754.1555760

[42] J. M. Robson. 1971. An Estimate of the Store Size Necessary for Dynamic Storage Allocation. *J. ACM* 18, 3 (jul 1971), 416–423. https://doi.org/10.1145/321650.321658

[43] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. 2015. NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories. In *3rd VLDB Work. In-Memory Data Mangement Anal. - IMDM '15*. ACM, 1–8. https://doi.org/10.1145/2803140.2803144

[44] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. 1993. Texas: An Efficient, Portable Persistent Store. In *Work. in Computing*. Springer, London, 11–33. https://doi.org/10.1007/978-1-4471-3209-7_2

[45] Valery Soloviev. 1992. An Overview of Three Commercial Object-oriented Database Management Systems: ONTOS, ObjectStore, and O2. *SIGMOD Rec.* 21, 1 (March 1992), 93–104. https://doi.org/10.1145/130868.130883

[46] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *16th Archit. Support Program. Lang. Oper. Syst. - ASPLOS '11*. ACM, 91–104. https://doi.org/10.1145/1950365.1950379

[47] Seth J. White and David J. DeWitt. 1992. A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies. In *18th Very Large Data Bases – VLDB*. Morgan Kaufmann Publishers Inc., 419–431. http://dl.acm.org/citation.cfm?id=645918.672341

[48] P.R. Wilson and S.V. Kakkad. 1992. Pointer swizzling at page fault time: efficiently and compatibly supporting huge address spaces on standard hardware. In *2nd Work. Object Orientat. Oper. Syst.* IEEE Comput. Soc. Press, 364–377. https://doi.org/10.1109/IWOOOS.1992.747914

[49] H. . P. Wong, H. Lee, S. Yu, Y. Chen, Y. Wu, P. Chen, B. Lee, F. T. Chen, and M. Tsai. 2012. Metal-Oxide RRAM. *Proc. IEEE* 100, 6 (June 2012), 1951–1970. https://doi.org/10.1109/JPROC.2012.2190369

[50] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th File and Storage Tech. - FAST '15*. USENIX, 167–181. https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang

[51] Jie Zhou, Yanyan Shen, Sumin Li, and Linpeng Huang. 2016. Revisiting Hash Table Design for Phase Change Memory. In *3rd IEEE/ACM Big Data Comput. Appl. Technol. - BDCAT '16*. ACM, 227–236. https://doi.org/10.1145/3006299.3006318