2018 IEEE 36th International Conference on Computer Design

Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software

Amirsaman Memaripour
Department of Computer Science and Engineering
University of California, San Diego
California, United States
amemarip@eng.ucsd.edu

Steven Swanson
Department of Computer Science and Engineering
University of California, San Diego
California, United States
swanson@cs.ucsd.edu

***Abstract*—Non-volatile main memory (NVMM) technologies, such as phase change memory and 3D XPoint, offer DRAM-like performance and byte-addressable access to persistent data. A wide range of applications (e.g., key-value stores and database systems) stand to benefit from the performance potential of these technologies. These potential benefits are greatest when applications can access memory directly via load/store instructions rather than conventional file-based interfaces. This approach presents several challenges. In particular, applications need guaranteed consistency and safety semantics to protect their data structures in the face of system failures and programming errors. Implementing data structures that meet these requirements is challenging and error-prone. Researchers have proposed several libraries and programming language extensions that simplify this task, but, to date, all the proposed solutions either require pervasive changes to existing software or rely on special hardware support. As a result, porting legacy applications to leverage NVMM is likely to be prohibitively difficult and time consuming.**

We propose Breeze, a NVMM toolchain that minimizes the changes necessary to enable legacy code to reap the benefits of directly accessing NVMM. Breeze guarantees data consistency and validity of persistent pointers regardless of failures. The toolchain transparently detects and logs writes to NVMM and provides a simple mechanism for identifying atomic sections while avoiding complications common in previous systems such as special persistent pointer types. Porting Memcached and MongoDB to use Breeze only requires changes to 5% of the source code compared to 7-14% for NVML and NVM-Direct. Breeze also provides equal or superior performance compared to NVML and NVM-Direct, outperforming them by up to 10x.

I. Introduction

Emerging non-volatile main memory (NVMM) technologies, such as Intel and Micron's 3D XPoint [31], [17], offer DRAM-like performance but with higher density and lower cost-per-bit. Applications such as databases and key-value stores could avoid serialization costs and improve response times by leveraging the performance, fine-grain access, and persistence that these memories offer.

NVMMs promise orders of magnitude better performance compared to conventional hard and solid state disks, but, unleashing their potential is challenging. To maximize the performance benefits of NVMM, applications should access them directly via load/store instructions rather than conventional file-based interfaces. This requires programmers to construct persistent data structures that are resilient in the face of system failures, avoid (persistent) memory leaks, prevent the creation of dangling pointers, and support multi-threaded operations.

Building these data structures is hard because it requires careful reasoning about the order in which updates to NVMMs will become persistent. Since caches will remain volatile and processors can reorder stores, programmers must issue barriers and/or flush cache lines to enforce the correct persist order and ensure data consistency. Applying these barriers correctly is challenging: excessive use leads to degraded performance [5], [46], but missing barriers can lead to data corruption.

Researchers have proposed several programming systems [8], [42], [6], [43] to hide this complexity from the programmer by providing library- or language-based mechanisms to manage and allocate NVMM, define persistent data structures, and specify the atomic operations that transform those structures from one consistent state to another. While these systems provide comprehensive solutions to many of the challenges that NVMM programming presents, they offer limited support for porting existing programs and data structures to make use of NVMM. Applying them to existing codes require pervasive changes and enormous programmer effort.

External libraries pose a particular problem, since, in the near term, they are unlikely to have been built with persistent memory in mind. Given these challenges, and without an alternative solution, it is likely that most legacy code will not fully benefit from the performance that NVMM offers. This will, in turn, reduce the rate of adoption of NVMM.

We propose Breeze that provides programs with transactional access to NVMM and minimizes disruptive changes to the source code. Breeze includes a user-level library and a C compiler. It guarantees consistency of persistent data in the event of failures and minimizes changes to the source code by transparently providing failure recovery, referential integrity (i.e., ensuring that references point to valid data) and garbage collection. In contrast to existing systems for low-level languages (e.g., C), Breeze does not require programmers to explicitly create log entries before touching persistent objects or use specific pointer types to reference persistent memory. Also, it can recover leaked memory regions without help from the programmer.

Breeze makes the following contributions:

2576-6996/18/$31.00 ©2018 IEEE
DOI 10.1109/ICCD.2018.00069

- Its compiler automatically generates log entries for non-volatile data updates.
- It lets programs use normal pointers to refer to persistent data rather than "fat" or "swizzled" pointers.
- It supports automatic garbage collection for persistent memory in C.
- It can generate log entries for changes that many third-party functions make to persistent memory.

We evaluate Breeze in terms of how extensively a programmer must modify a code base to make use of NVMM and the performance it offers. To quantify this, we port two applications, MongoDB [33] and Memcached [12], as well as a B+Tree and a hash table to use Breeze. We find that Breeze requires fewer, simpler changes to the source code and meets or exceeds the performance of competing systems.

The rest of this paper is organized as follows. We describe the background and motivation for this work in Section II. Section III presents the architecture of Breeze while Section IV describes its implementation in detail. Next, we evaluate ease of use and performance of Breeze in Section V. Finally, we provide a summary of this work in Section VI.

II. Background and Motivation

Several NVMM technologies are expected to appear in the market in the next few years. These technologies present significant challenges to programmers, and researchers have proposed several systems to simplify programming for NVMMs. Below, we explore these issues in more detail and place Breeze in the context of previous work.

A. NVMM Technologies

Emerging NVMM technologies (e.g., 3D XPoint) promise latency and bandwidth comparable to DRAM, while offering persistence and higher density [31], [17], [3], [32]. This enables computer architects to directly connect NVMM to the processor's memory bus and allow applications to access persistent data using load/store instructions. Incorporating NVMM into a system requires architectural support to allow programmers to reason about how and when updates occur to NVMM [47], [19]. Currently, processors provide support for 64-bit atomic writes [16], and provide instructions to force write backs from volatile caches to NVMM. Together, these mechanisms are sufficient to build complex, strongly-consistent data structures that can survive power failures.

B. Programming with NVMM

While atomic writes and cache control instructions are sufficient, in principle, to unlock NVMM's benefits, building complex, fault-tolerant, and highly concurrent data structures using those primitives is very challenging. In particular, programmers must address all the challenges that volatile data structures present, such as memory management and locking. Both of these areas are well-known sources of bugs and the resulting inconsistencies in the data structures will be permanent with NVMM. Programmers must also reason carefully about the order in which updates occur, become visible to other threads, and reach NVMM. In this respect, building persistent data structures resembles building lock-free data structures, a notoriously tricky, subtle, and error-prone discipline.

NVMM also introduces new classes of bugs that are at least as pernicious as memory management and locking errors. For instance, pointers from a non-volatile data structure to volatile memory are inherently unsafe, as are pointers between two independent NVMM regions (e.g., two mmap'd files) [8].

Researchers have proposed several programming systems that address these challenges and make it easier to construct reliable, persistent data structures. These include NV-Heaps [8], Mnemosyne [42], and NVM-Direct [6], and they each provide a similar set of core facilities. First, they provide memory allocation and garbage collection mechanisms that are robust in the face of system failures. These eliminate memory leaks and dangling pointers. Second, NV-Heaps and NVM-Direct provide protections against creating unsafe pointers from non-volatile memory to volatile memory and between independent regions of non-volatile memory. Third, they provide atomic sections that let the programmer specify which operations move a persistent data structure from one consistent state to another, providing a more flexible atomicity primitive than the 64-bit stores that hardware provides natively.

All these systems also place strong constraints on how programmers write code. For instance, NV-heaps is a C++ library and requires that all objects inherit from a persistent object base class, and NVM-Direct adds syntax to C in order to distinguish between volatile and non-volatile pointers. These constraints are tolerable for writing new code, but it makes adapting existing code to use NVMM very labor intensive. This is unfortunate, because there are many legacy applications that could benefit from NVMM.

Breeze provides a similar set of NVMM programming facilities, but it also minimizes changes that programmers must make to enable existing programs to use NVMM. It addresses challenges such as failure consistency, referential integrity and garbage collection and paves the way for existing applications to exploit NVMM without requiring invasive changes.

C. Previous Work

Designing NVMM-optimized systems is one way to address the consistency and safety challenges. Examples are NVMM-optimized logging schemes [15], [20], [45], [21], data structures (e.g., B-Trees and RB-Trees) [46], [44], [25], and database transactional protocols such as OLTP [38], [21], [41]. These systems also serve as building blocks for larger systems like TANGO [2] that uses shared logs to build distributed data structures. In contrast to Breeze, these systems do not provide a general approach for optimizing existing software and assume applications are created from scratch.

Another direction to approach failure consistency is adding persistence to the memory controller and processor caches. Whole system persistence [35] and JUSTDO Logging [18] are examples of exploiting persistent caches to recover from failures. Additional support from the software is still necessary to prevent potential unrecoverable conditions such as

deadlocks [1]. Klin [48], WrAP [10], NVM Duet [28] and ThyNVM [39] combine hardware and software techniques to offer atomic writes to NVMM. In contrast, Breeze only relies on existing hardware support in commodity processors.

Other efforts either focus on providing transactional semantics and safety features through programming support [8], [42], [6], [43], [27], [7], [14] or improving the performance of such transactional systems [23], [30]. NV-Heaps [8] and Mnemosyne [42] are among the first to offer such semantics at user-level. Unlike Breeze, Mnemosyne does not offer garbage collection and sacrifices flexibility to provide pointer safety by not allowing persistent memory regions to be mapped into a different virtual address space after creation. NV-Heaps provides atomicity and safety guarantees through C++ classes, however, it requires disruptive changes to existing code.

NVM-Direct [6] provides similar semantics as NV-Heaps by introducing new programming keywords. In contrast to Breeze, NVM-Direct introduces new syntax and requires programmers to use specific keywords to define persistent pointers and perform atomic updates. Thus, it requires far more changes to the source code of legacy applications.

Intel's NVML [43] library provides a framework for building NVMM-optimized applications and data structures. The library provides a set of primitives to create transactions and manage persistent regions. In contrast to Breeze, it does not guarantee referential integrity of persistent data and requires disruptive changes to existing programs since programmers are required to use special pointer types and create log entries.

Other proposals such as Atlas [7], NVthreads [14] and Atomic-msync [37] adopt existing programming constructs to help support NVMM programming. However, they either are only applicable to a particular class of applications or impose high performance overhead. There are also other proposals such as Rio Vista [29] and RVM [40] that offer transactional semantics and failure recovery for byte-addressable storage, however, they do not offer features such as referential integrity.

III. Breeze Design Overview

Breeze provides existing programs with direct access to NVMM (Figure 1) and maintains consistency of persistent data while minimizing changes to the source code. It exposes a small set of interfaces which programmers employ to create/open NVMM-resident memory-mapped files, define persistent data structures, specify persistent pointers, and transactionally create and modify instances of those structures.

Applying Breeze to existing code comprises four steps (Figure 2). Programmers use the C macros from Table I to label persistent structures and pointers. Next, they define transaction boundaries and replace volatile memory management function calls with persistent counterparts. Then, Breeze's compiler uses this information to generate code to log updates to NVMM and recover from failures. Finally, programmers link the object files to Breeze's user-level NVMM library.

At step 3, Breeze's compiler exploits the definition of persistent structures and pointers to generate a set of C functions that will garbage collect and maintain referential integrity of

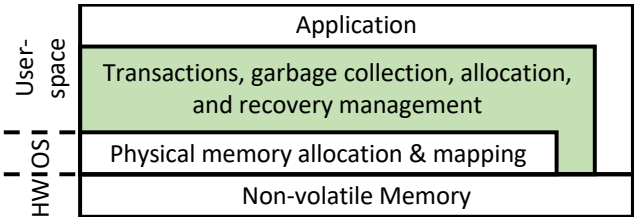

Fig. 1: The organization of Breeze allows programs to bypass the operating system and directly read/write persistent data. Breeze, the green area, is responsible for transaction and recovery management, garbage collection and allocation.

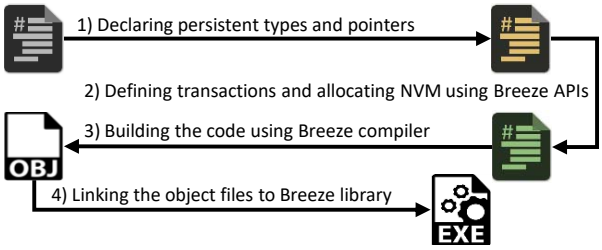

Fig. 2: Four steps of applying Breeze to legacy software.

persistent objects at run-time. The compiler also creates undo-log entries for each write to NVMM in order to maintain consistency of persistent data throughout recoverable failures (e.g., power outages).

In contrast to existing NVMM programming systems, Breeze automates logging and garbage collection without requiring use of managed pointers or special support from the hardware [43], [6], [42], [8]. The rest of this section explains Breeze's design by providing an example of adding persistence to the linked-list from Figure 3 using Breeze.

A. Declaring Persistent Types and Pointers

The first step in applying Breeze to legacy code is annotating the data structures that will be persistent using the primitives in Table I. Breeze's compiler uses these annotations to generate C functions that maintain reference count and referential integrity of persistent objects at run-time. It also exploits this information to prevent the creation of unsafe pointers from non-volatile to volatile memory and between non-volatile memory pools (i.e., contiguous regions of NVMM that reside within memory-mapped files). Figure 4 shows how we use these primitives for the linked-list example.

The changes required to the code are modest. Breeze only requires annotating data structure declarations, which comprise a small portion of the source code. In contrast to existing systems, Breeze does not require programmers to use special pointer types, inherit from a particular base class, or extend existing structures with new fields.

B. Atomic Sections

Specifying actions that take a persistent data structure from one consistent state to another is a central requirement for a NVMM programming system. This information, allows the system to restore the data structure to a consistent state in

NVM_STRUCT	This is an alternative to `struct` keyword in order to declare persistent structures.
NVM_PTR (`type`)	Declares a persistent pointer of type `type` in a persistent structure.
NVM_TYPE_ID (`type`)	Returns type identifier for `type`, a positive integer that is used for allocation purposes.

TABLE I: Macros to declare persistent data types and pointers.

```
typedef struct Node {
  struct Node *next;
  char[32] data;
} Node;
void main() {
  Node *head = NULL;
  for (int i = 0; i < 10; i++) {
    Node *t = malloc(sizeof(Node));
    if (t == NULL) break;
    t->next = head;
    strcpy(t->data, "Hello world");
    head = t;
  }
}
```

Fig. 3: This code creates a simple, volatile linked-list with 10 elements by adding new elements to the head of the list.

case of a failure. Breeze facilitates implementing ACID [13] transactions by providing atomicity, consistency and durability for all transactional updates to NVMM. It automates logging and allows performing updates on the main version of persistent objects. Programmers are responsible for isolation, but, in most cases, legacy software already includes concurrency control and Breeze allows using the same mechanism.

Breeze relies on programmers to declare transaction boundaries, and then it automatically generates undo-log entries for resulting atomic sections. Breeze's transactional interfaces (Figure 5) include `nvm_tx_begin()`, `nvm_tx_commit()` and `nvm_tx_abort()`. In contrast to other systems (e.g., Intel's NVML [43]), Breeze does not require the creation of log entries prior to updating NVMM. Instead, the compiler generates necessary code to create undo-logs before every write to NVMM.

C. Allocation

Breeze provides programmers with primitives to create non-volatile pools and allocate persistent objects. Programmers use `nvm_pool_create()` and `nvm_pool_open()` to create and map a file containing a newly-initialized pool or map an existing pool into applications' address space, respectively. In case of failures, `nvm_pool_open()` also performs necessary recovery operations on the pool to ensure its consistency. Pools are self-contained and include all the information necessary for failure recovery.

Each pool has a *root pointer* that provides access to all the live objects in the pool. Persistent objects that are not reachable from the *root pointer* are considered dead and are candidates for garbage collection. Programmers can modify the root pointer by calling `nvm_pool_set_root()`.

Programmers allocate space for persistent objects within a pool using `nvm_alloc()`. Breeze provides transactional

```
// declare persistent structure
typedef NVM_STRUCT Node {
  // declare persistent pointer
  NVM_PTR(struct Node) next;
  char[32] data;
} Node;
```

Fig. 4: Declaring persistent types and pointers using Breeze primitives. The highlights are lines that needed to be changed.

```
void main() {
  size_t pool_size = (off_t)1 << 20;
  NVM_POOL *pop = nvm_pool_create("/nvm/pool
    ", pool_size);
  Node *head = NULL;
  for (int i = 0; i < 10; i++) {
    nvm_tx_begin(pop); //begin transaction
    // allocate a new persistent object
    Node *t = nvm_alloc(pop, NVM_TYPE_ID(
    Node), sizeof(Node));
    if (t == NULL) {
      nvm_tx_abort(pop); //abort transaction
      break;
    }
    t->next = head;
    strcpy(t->data, "Hello world");
    head = t;
    // update the root pointer
    nvm_pool_set_root(pop, head);
    nvm_tx_commit(pop); //commit transaction
  }
  nvm_pool_close(pop);
}
```

Fig. 5: Using Breeze for transactional list operations.

allocation and ensures allocated objects are reclaimed if the corresponding transaction aborts.

Highlighted statements of Figure 5 shows how to use Breeze for the linked-list from Figure 3 to create persistent pools, define atomic sections and transactionally allocate persistent objects.

When a program is finished with a pool, it can close the pool with `nvm_pool_close()`. This leaves the pool in a consistent state and obviates recovery the next time a program opens it. Breeze closes all pools when the program exits.

D. Breeze's Compiler

Breeze's compiler automatically inserts logging code and generates recovery code for data structures. The compiler inserts logging code before each write to NVMM, which is a function call to the undo-logging function implemented in Breeze's user-level library.

Library functions could also write to NVMM. Breeze allows programmers to link the code to any pre-compiled library. The compiler inserts logging code before calling library functions to create undo-log entries for NVMM regions that the library function modifies. We describe how Breeze provides atomicity for library functions in Section IV-C.

In addition to automating undo-log creation, the compiler uses programmers annotations from Table I to generate code that maintains referential integrity of persistent objects,

garbage collects unreachable regions and recovers from failures. Sections IV-E and IV-F discuss the role of Breeze's compiler in maintaining referential integrity and performing garbage collection in more depth.

IV. Implementation

We implemented Breeze as a C library and a C compiler, which extends the LLVM compiler infrastructure [24], under Linux. Breeze does not require special hardware support. Below we describe how Breeze lays out data in persistent pools, handles atomicity and data allocation, maintains referential integrity, and recovers from failures.

A. Storage Layout

Breeze organizes persistent objects into continuous regions of NVMM called memory pools. We utilize filesystem's naming mechanism to find memory pools after program restarts and `mmap()` them to the program's address space. Breeze requires the filesystem to provide direct access to NVMM pages (i.e., DAX) of memory-mapped files [26].

Memory pools are divided into four segments. The first segment is the header and contains metadata about the pool (e.g., pool size) as well as the offset of the root object. Next is the data segment that contains programs' data and Breeze's allocator is responsible for managing it. The user-level library utilizes the next segments to store garbage collection data and the transaction log, discussed in Sections IV-F and IV-C respectively. In contrast to other NVMM programming systems (e.g., NVML and NVM-Direct), Breeze requires less space for its metadata since it neither persists allocation tables nor stores additional information for persistent pointers [43], [6].

B. Memory Allocation and Management

Breeze implements a Hoard like allocator (uses per-thread allocation tables to minimize false sharing and cache contention), provides transactional allocation semantics, and avoids memory leaks through reference counting [22], [4]. Breeze uses a two phase protocol for allocating/freeing space. At phase one, it creates undo-log entries in the pool's transaction log for allocate/free requests. At phase two, the library uses the undo-log entries to undo allocate/free requests if the transaction aborts. The library discards the undo-log entries on transaction commit.

To achieve faster allocation, Breeze does not persist allocation tables. Instead, it scans the pool during startup to find free regions and rebuild allocation tables.

C. Atomicity

Breeze's compiler creates undo-logs before programs modify persistent memory regions. In contrast to existing NVMM systems that aim for easy to use atomicity, Breeze's approach is applicable to all types of applications and does not require hardware support, switching between the user and kernel mode, or coarse-grain logging [14], [18], [48].

First, Breeze's compiler uses LLVM's front-end to generate LLVM IR code. Then, it identifies every instruction that could update a region of byte-addressable memory, either directly (e.g., `store`) or indirectly (e.g., `function call`). If the compiler has enough information about the instruction to decide whether it writes to volatile or persistent memory, it ignores the instruction or generates undo-logging code before it, respectively. The undo-logging code calls to Breeze's undo-log function and provides it with the address and size of the write. This function allocates space for the log entry, copies data from the memory region to the allocated space, ensures its persistence and updates the transaction metadata.

```
// Original code (no logging)
memcpy(dst, src, size);
A[i] = B[i];

// Breeze's compiler output
if (isNVMM(dst)) log(dst, size);
memcpy(dst, src, size);
if (isNVMM(&A[i])) log(&A[i], sizeof(A[i]));
A[i] = B[i];
```

Fig. 6: Breeze's compiler injects boundary checks before memory writes the durability of which is unknown. It does not add boundary checks before writes to volatile heap/stack resident regions. At runtime, if the instruction writes to a NVMM region, the undo-log function is invoked.

If the persistence of a memory region is unknown at compile time (e.g., function arguments), the compiler inserts a call to Breeze's undo-logging function before the write instruction, however, a boundary check precedes the function call to avoid invoking the undo-log function for volatile memory regions (see Figure 6). Breeze reserves a contiguous region of virtual address space for persistent memory pools at program startup and uses the lower-bound and upper-bound of this region to perform boundary checking. To minimize the frequency of such boundary checks, we have also implemented a few optimizations that are briefly explained in Section IV-G.

D. Atomicity for pre-compiled code

As long as programmers use Breeze's compiler to compile the code, it can provide atomicity by inserting undo-logging code before the instructions that modify NVMM. However, pre-compiled libraries can also modify persistent data and Breeze must log these changes as well. Breeze allows pre-compiled libraries to safely modify NVMM, since recompiling third-party libraries (e.g., standard library) is not always possible or desirable. As the majority of such library functions only modify memory regions referenced by pointers passed via their arguments, the compiler can insert logging code in user's code and before the function call instruction. This technique enables Breeze to provide atomicity even for programs that call pre-compiled functions. Currently, Breeze logs changes to persistent data passed as arguments, which is all we needed for our benchmark applications, but we can extend this to handle changes to global persistent variables by enabling programmers to pass the address and size of such variables to Breeze's compiler as well.

```
memcpy={nargs:3,wr_set:{[arg0,arg2]}}  foo.unsafe
strcpy={nargs:2,wr_set:{[arg0,strlen(arg1)]}}
setCacheLine={nargs:2,wr_set:{[arg0,#64]}}
```

```
myFunction1(a,b,c); // Compiled by Breeze
memcpy(a,b,d); // Unsafe (pre-compiled)
myFunction2(a,b);
strcpy(a,b);
setCacheLine(c,0);
```

```
myFunction1(a,b,c);
if (isNVMM(a)) log(a, c); // Log argument #0
memcpy(a,b,d); // Argument #0 is logged
myFunction2(a,b);
if (isNVMM(a)) log(a, strlen(b));
strcpy(a,b);
if (isNVMM(c)) log(c, 64); // Size = 64
setCacheLine(c,0);
```

Fig. 7: Breeze's compiler uses `foo.unsafe` (top) to provide atomicity for library function calls (middle). `foo.unsafe` contains one line for each pre-compiled function that comprises the name, number of arguments (`nargs`) and the write set (`wr_set`) for the function. A `wr_set` is a series of tuples that identify the address and size of memory regions that the corresponding function modifies. The programmer provides Breeze with this information.

We classify functions into atomic-safe and atomic-unsafe. Atomic-safe functions are compiled by Breeze's compiler and contain undo-logging code for instructions in their body that modify NVMM. Thus, the compiler does not need to insert any undo-logging code before calling these functions. On the other hand, atomic-unsafe functions reside in pre-compiled libraries and Breeze's compiler has no information about presence or absence of undo-logging code inside these functions. As a result, providing these functions with pointers to NVMM regions might be unsafe and the compiler needs to insert undo-logging code before calling these functions.

The programmer must use annotations to create a file that describes how atomic-unsafe functions modify memory. This file (e.g., `foo.unsafe`) includes the list of atomic-unsafe functions and provides the compiler with the address and size of NVMM regions that each function modifies. As shown in Figure 7, the compiler uses `foo.unsafe` to understand how atomic-unsafe functions modify memory based on their arguments. For example, `memcpy` accepts three arguments (`nargs`) and writes a total of `arg2` bytes (third argument) to a memory region referenced by `arg0` (first argument).

During the process of generating object files, Breeze's compiler checks every function call against the list of atomic-safe and atomic-unsafe functions. No extra work is necessary for atomic-safe functions (green bullets). In case of atomic-unsafe functions (orange bullets), the compiler precedes the call instruction with undo-logging code using the information from `foo.unsafe`. An example of the undo-log code that the compiler generates is shown in the bottom of Figure 7.

Programmers should replace function pointers to atomic-unsafe functions with pointers to atomic-safe wrapper functions. This enables Breeze's compiler to insert undo-logging code before calls to atomic-unsafe functions made through function pointers.

E. Pointer Safety

Breeze allows programmers to directly update persistent pointers. This approach has no effect on validity of persistent pointers as long as the physical to virtual mapping of non-volatile pages does not change. In contrast to Mnemosyne [42] which maintains this mapping through kernel-level support, Breeze allows this mapping to change while maintaining validity of persistent pointers. If the operating system cannot maintain the same base address for a persistent memory pool after a restart (e.g., because another pool is already mapped to that address), Breeze adjusts persistent pointers to account for the change by running Algorithm 1. This algorithm adjusts every pointer of an object inside the persistent pool by adding the offset between the old and new base addresses.

For example, assume the operating system is mapped a persistent pool at address `0xA0` when a failure occurs. After the failure, the application calls `nvm_pool_open()` to run recovery and map the pool to its address space. However, the operating system finds that `0xA0` is in use, so it has to map the pool to address `0xB0`. To keep pointers valid, Breeze transactionally adds `0x10` to all pointers in the pool.

Since failures might occur during the execution of Algorithm 1, Breeze keeps track of the persistent pool's base address for each recovery attempt as well as generation numbers for persistent objects and the persistent pool in order to tolerate such failures. Generation numbers are integer values assigned to persistent pools and objects. Breeze increments the pool's generation number for every recovery attempt. The library updates an object's generation number with the pool's generation number once it finishes recovering the object.

Algorithm 1 Fixing persistent pointers on recovery

```
1:  procedure POINTER_REWRITE(pool, object)
2:      my_base ← pool.base_addr
3:      if object.gen_num ≠ pool.gen_num then
4:          i ← object.gen_num − pool.gen_num
5:          my_base ← pool.base_addr_log[i − 1]
6:      end if
7:      disp ← pool.new_base_addr − my_base
8:      if disp ≠ 0 then
9:          tx_begin
10:         for all ptr ∈ object.pointers do
11:             ptr ← ptr + disp
12:         end for
13:         t ← pool.gen_num + pool.recovery_attempts
14:         object.gen_num ← t
15:         tx_end
16:     end if
17: end procedure
```

Consider the previous example and assume another failure occurs before Algorithm 1 finishes updating the pointers. When the application opens the pool again, the kernel happens to map it to `0xC0`. The generation numbers and history of base addresses enables Breeze to add `0x10` to pointers that were updated during the last recovery session while adding `0x20` to pointers that were not. Breeze can run the recovery algorithm in parallel to reduce recovery time, or it could perform recovery lazily when the application accesses an object for the first time after recovery. The maximum size of NVMM required to run the recovery algorithm equals the number of recovery threads times the maximum object size.

F. Garbage Collection

The garbage collection (GC) algorithm leverages the persistent pointer information to track reference count of persistent data structures. Breeze implements a two-phase GC algorithm similar to the scheme introduced by NV-Heaps [8] and uses weak pointers to avoid memory leaks due to cycles.

Phase 1 runs during transaction commit and identifies the pointers that have changed by comparing each object in the transaction's log against its updated version. It records the new value of each pointer along with its original values in a data structure called activation record. Activation records contain the transaction identifier (`tx_id`), checksum, number of pointers changed (`size`), and a list of updated pointers (`ptrs`). Breeze ensures persistence of an activation record before marking its corresponding transaction as committed. Algorithm 2 describes the first phase of Breeze's GC: creating activation records on transaction commit. We use a circular log of size 2^{16} (`log_size`) to maintain these activation records for each transaction, thus, no transaction is expected to wait for another transaction to commit. If there is no available activation record, the oldest transaction in the circular log is restarted to open up space for new transactions.

Algorithm 2 Creating a list of pointer changes

```
1:  procedure LOG_POINTER_CHANGES(tx)
2:      log ← circular_log[tx.id mod log_size]
3:      log.tx_id ← tx.id
4:      n ← 0
5:      for all obj ∈ tx.log do
6:          for all ptr ∈ obj do
7:              if ptr.old_val ≠ ptr.new_val then
8:                  log.ptrs[n].old_val ← ptr.old_val
9:                  log.ptrs[n].new_val ← ptr.new_val
10:                 n ← n + 1
11:             end if
12:         end for
13:     end for
14:     log.size ← n
15:     Update log.checksum
16:     Persist log
17: end procedure
```

```
// Original code (no logging)
for (int i=0; i<64; i++) { A[i]=B[i]; }
```

```
// Breeze's compiler output
for (int i=0; i<64; i++) {
  if (isNVMM(A[i]))
    log(&A[i], sizeof(A[i]));
  A[i]=B[i];
}
```

```
// Breeze's optimizer output
if (isNVMM(A)) log(A, sizeof(A[0]) * 64);
for (int i=0; i<64; i++) { A[i]=B[i]; }
```

Fig. 8: An example of using the optimizer to reduce the frequency of boundary checks and calls to the log function.

Phase 2 consumes the activation records inside the circular log. For each entry, it increments the reference count on the object the pointer now points to and decrements the count on the object it used to point to. A background thread iterates over the activation records and translates them into a set of redo-logs (one for each pointer) prior to updating reference counts. Redo-logs are necessary to make updates idempotent.

G. Compiler optimizations for Breeze

The goal here is to minimize the performance overhead of using Breeze's compiler to create undo-log records. A perfect optimization technique could enable the compiler to remove all boundary checks and keep the frequency of calls to the undo-log function at the minimum. Our first step in this direction is to avoid boundary check and undo-log creation for pointers that reference stack and volatile heap resident data. Considering that most of the memory accesses for a wide range of applications are for heap and stack data [34], this significantly reduces the performance overhead of Breeze by reducing the frequency of boundary checks.

We have also implemented a simple optimization for loops to reduce the frequency of function calls and boundary checks (Figure 8). The aim of this technique is to replace multiple fine-grain undo-log entries with a single coarse-grain undo-log. As a result, this technique can reduce the frequency of function calls and boundary checks by $n-1$, where n is the total iterations of the loop (64 in our example).

We are working on a few other ways to reduce the performance overhead of Breeze on NVMM applications. An example is to avoid repeating boundary checks and calls to the undo-log function for the same memory region. The compiler utilizes use-define chains to decide if inserting undo-logging code before an instruction is necessary. This is done by tracking all reachable definitions for a use (a pointer referencing a memory region that is being updated by the corresponding instruction) and making sure undo-logging code is present for every single reachable definition.

V. Evaluation

This section evaluates Breeze and compares its performance (latency and bandwidth) and ease-of-use against NVM-Direct [6] and NVML [43] (version 1.0). We use two groups

Workload	Read	Update	Insert	Read & Update
YCSB-A	50%	50%	-	-
YCSB-B	95%	5%	-	-
YCSB-D	95%	-	5%	-
YCSB-F	50%	-	-	50%

TABLE II: The operations in each YCSB workload.

of applications to benchmark Breeze. The first group includes persistent implementations of a B+Tree and a hash table using Breeze, NVM-Direct [6] and NVML [43]. The implementation of the B+Tree is similar to that of NV-Tree [46]. We use Cuckoo hashing [36] for the hash table, which adopts DJB2 and Jenkins as hash functions.

The second group are legacy applications that can benefit from NVMM. We have incorporated Breeze and NVML with Memcached [12] and MongoDB [33] for this purpose. Memcached is a general purpose memory caching system with no durability semantics. MongoDB is a persistent document store with support for atomic updates. For these experiments, MongoDB is configured to ensure persistence of each insert/update operation before acknowledging the client.

We exercise these applications with the YCSB [9] workloads described in Table II. YCSB provides a common ground to evaluate the performance of different key-value storage systems. For our experiments, we populate the storage system with 10 million key-value pairs of size 1 KB. Then, we run each of the workloads from Table II to measure the performance of our target system.

A. Test System

We have utilized Intel's Persistent Memory Emulation Platform (PMEP) [11] to emulate the latency and bandwidth of NVMM. The latency and bandwidth of emulated NVMM are 100 ns and 1/8 of DRAM, respectively, and `clwb` takes 40 ns to complete. PMEP is a dual socket platform equipped with Intel Xeon processors with 8 cores running at 2.6 GHz. The platform has a total of four DDR3 channels, where channels 2 and 3 are marked by the BIOS for emulating NVMM.

B. Ease of use

To compare Breeze's ease of use with other systems' we use the number of lines of code that must be changed in order to enable an existing application to use NVMM. The B+Tree and hash table implementations serve as a baseline to compare Breeze against NVM-Direct [6] and NVML [43]. Table III shows that Breeze requires between 55% and 82% fewer modified lines than NVM-Direct or NVML.

We have also incorporated Breeze and NVML into MongoDB [33] and Memcached [12]. Table III shows that Breeze requires between 1.2x to 2.8x fewer changes to the source code in contrast to NVML. We did not replicate these experiments for NVM-Direct, but we expect similar numbers for NVML and NVM-Direct.

C. Performance

Table IV compares the allocation overhead of Breeze against NVM-Direct [6] and NVML [43]. We report the average latency of allocating one million objects of size 1, 2, and 4 KB. In contrast to NVML and NVM-Direct, Breeze requires 1.35x and 7.2x less time for allocation, respectively.

	Breeze	**NVML**	**NVM-Direct**
B+Tree	18	101	96
Hash Table	20	45	77
MongoDB	882	1273	-
Memcached	541	1547	-

TABLE III: Measuring ease of use – the numbers measure the lines of code that needs to be changed.

System / Allocation Size	**1 KB**	**2 KB**	**4 KB**
Breeze	2.51	2.02	2.48
NVML	4.74	4.76	6.01
NVM-Direct	16.59	16.56	16.59

TABLE IV: Measuring the average allocation latency for objects of size 1, 2 and 4 KB. Numbers are in μseconds.

We also report performance measurements of our benchmark applications using YCSB workloads. We compare latency and throughput of Memcached [12] and MongoDB [33] against the Breeze-enabled and NVML-enabled versions. We only use Breeze to compile MongoDB's storage engine to avoid unnecessary overhead of boundary checks. Figure 9 summarizes the results. Both NVMM-enabled versions of MongoDB outperform its original version by avoiding system calls (e.g., `msync()`) to persist data. NVMM-enabled versions of Memcached show higher latency and lower throughput compared to the unmodified version due to the cost of providing persistence. Breeze provides superior performance for such applications compared to the NVML version because of its optimized undo-logging scheme and not persisting allocation metadata. The only exception is running YCSB-B and YCSB-D against Memcache-Breeze where the overhead of boundary checks cancels out the performance benefits of Breeze's optimized transaction implementation.

Finally, we have incorporated Breeze, NVML and NVM-Direct with the B+Tree and hash table implementations to compare the performance of Breeze with NVM-Direct and NVML. According to Figure 10, NVM-Direct shows lower performance compared to Breeze and NVML due to its inefficient implementation of logging and allocation. Breeze outperforms NVML for YCSB A and F (write-heavy) because of its optimized undo-logging scheme. Since only the last level of the B+Tree is persistent, the performance difference between Breeze and NVML is minimal for the B+Tree benchmarks. Furthermore, Breeze and NVML show similar performance for read-heavy workloads.

D. Recovery Time

To measure the overhead of rebuilding allocation tables and rewriting persistent pointers during startup, we use objects of size 1 KB and persistent pools with a total capacity of 4, 8 and 16 gigabytes. Also, we varied the number of recovery threads to measure the scalability of our scheme. Table V shows the average recovery time of Breeze when the virtual address of persistent pools does not change (case A). The numbers directly show the overhead of recreating allocation tables.

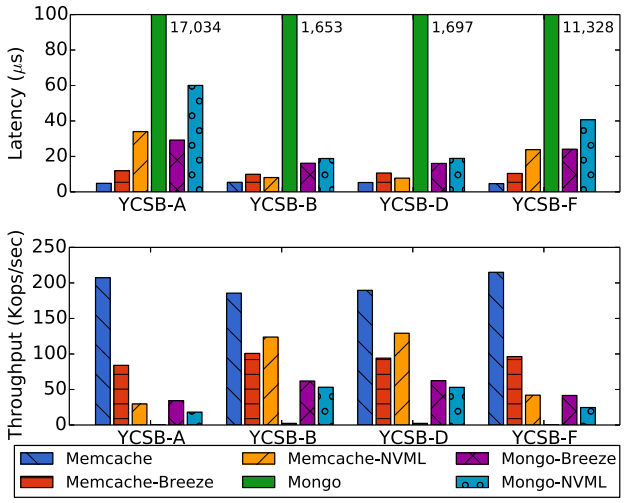

Fig. 9: Performance of unmodified and NVMM-enabled versions of MongoDB and Memcached using YCSB workloads.

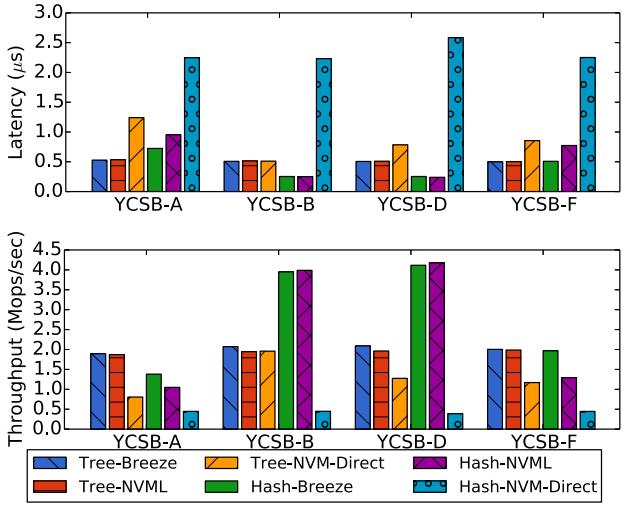

Fig. 10: Average latency and throughput of Breeze, NVM-Direct and NVML using YCSB workloads.

We also measured the overhead of rewriting persistent pointers by mapping pools into different virtual addresses during startup (case B). Table V shows the average recovery time for these experiments that indicate utilizing more threads can significantly reduce the recovery time.

VI. Conclusion

Breeze provides direct access to non-volatile memories without requiring disruptive changes to legacy software. Breeze works with commodity hardware and offers transactional semantics, referential integrity and garbage collection. The toolchain lifts the burden of logging from programmers, automatically generates recovery code, allows programmers to use normal pointers and legacy libraries to manipulate persistent data, and provides garbage collection. Our measurements show that Breeze significantly simplifies the task of modifying existing code to use NVMM while still providing better performance than other proposed systems.

Threads	A: Map to same VA			B: Map to new VA		
	4GB	8GB	16GB	4GB	8GB	16GB
1	0.62	0.86	1.72	18.95	21.25	42.20
2	0.31	0.59	1.24	10.58	21.08	41.02
4	0.22	0.35	0.70	5.00	10.16	19.94
8	0.10	0.21	0.39	3.10	6.41	12.50

TABLE V: Breeze's recovery time (seconds) for mapping the pool to the same or a new virtual address after recovery.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd for their helpful comments. We are also thankful to Subramanya R. Dulloor from Intel for his support and hardware access. Finally, we thank the members of the NVSL research group for their insightful comments.

References

[1] Anirudh Badam. How Persistent Memory will Change Software Systems. *Computer*, 46(8):45–51, August 2013.

[2] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13', pages 325–340, New York, NY, USA, 2013. ACM.

[3] Jamie Beckett. Demystifying the Memristor: Proof of Fourth Basic Circuit Element Could Transform Computing. http://www.hpl.hp.com/news/2008/apr-jun/memristor.html, April 2008.

[4] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM.

[5] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Implications of CPU Caching on Byte-Addressable Non-Volatile Memory Programming, 2012. Available at http://www.hpl.hp.com/techreports/2012/HPL-2012-236.html.

[6] Bill Bridge. NVM Support for C Applications, 2015. Available at http://www.snia.org/sites/default/files/BillBridgeNVMSummit2015Slides.pdf.

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

[8] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 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*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10', pages 143–154, New York, NY, USA, 2010. ACM.

[10] Kshitij Doshi and Peter Varman. WrAP: Managing Byte-Addressable Persistent Memory, 2012. Available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.303.5364&rep=rep1&type=pdf.

[11] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[12] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124), August 2004. Available at http://dl.acm.org/citation.cfm?id=1012889.1012894.

[13] Jim Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81', pages 144–154. VLDB Endowment, 1981.

[14] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17', pages 468–482, New York, NY, USA, 2017. ACM.

[15] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware Logging in Transaction Systems. *Proc. VLDB Endow.*, 8(4):389–400, December 2014.

[16] Intel Corporation. Intel Architecture Instruction Set Extensions Programming Reference, 2014. Availalbe at https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.

[17] Intel Technology. Non-Volatile Memory. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html.

[18] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16', pages 427–442, New York, NY, USA, 2016. ACM.

[19] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-based NIC-offloading to Accelerate Replicated Transactions in Multi-tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18', pages 297–312, New York, NY, USA, 2018. ACM.

[20] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. *SIGOPS Oper. Syst. Rev.*, 50(2):385–398, March 2016.

[21] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16', pages 385–398, New York, NY, USA, 2016. ACM.

[22] Kenneth C. Knowlton. A Fast Storage Allocator. *Commun. ACM*, 8(10):623–624, October 1965.

[23] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16', pages 399–411, New York, NY, USA, 2016. ACM.

[24] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[25] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 257–270, Santa Clara, CA, 2017. USENIX Association.

[26] Linux Kernel Organization. Direct Access for Files. Available at https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[27] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17', pages 329–343, New York, NY, USA, 2017. ACM.

[28] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. NVM Duet: Unified Working Memory and Persistent Store Architecture. *SIGPLAN Not.*, 49(4):455–470, February 2014.

[29] David E. Lowell and Peter M. Chen. Free Transactions with Rio Vista. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 92–101, New York, NY, USA, 1997. ACM.

[30] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17', pages 499–512, New York, NY, USA, 2017. ACM.

[31] Micron Technology. Breakthrough Non-Volatile Memory Technology. https://www.micron.com/about/emerging-technologies/3d-xpoint-technology.

[32] Micron Technology. NVDIMM. https://www.micron.com/products/dram-modules/nvdimm/.

[33] MongoDB, Inc. Introduction to MongoDB. Available at http://docs.mongodb.org/manual/core/introduction/.

[34] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 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*, ASPLOS '17, pages 135–148, New York, NY, USA, 2017. ACM.

[35] Dushyanth Narayanan and Orion Hodson. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 401–410, New York, NY, USA, 2012. ACM.

[36] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[37] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 225–238, New York, NY, USA, 2013. ACM.

[38] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. Storage Management in the NVRAM Era. *Proc. VLDB Endow.*, 7(2):121–132, October 2013.

[39] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 672–685, New York, NY, USA, 2015. ACM.

[40] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Trans. Comput. Syst.*, 12(1):33–57, February 1994.

[41] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. Failure-Atomic Slotted Paging for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17', pages 91–104, New York, NY, USA, 2017. ACM.

[42] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[43] Paul von Behren. NVML: Implementing Persistent Memory Applications, 2015. Available at http://www.snia.org/sites/default/files/Paul_von_behren_NVML_Implementing_Persistent_Memory.pdf.

[44] Chundong Wang, Qingsong Wei, Lingkun Wu, Sibo Wang, Cheng Chen, Xiaokui Xiao, Jun Yang, Mingdi Xue, and Yechao Yang. Persisting RB-Tree into NVM in a Consistency Perspective. *ACM Trans. Storage*, 14(1):6:1–6:27, February 2018.

[45] Tianzheng Wang and Ryan Johnson. Scalable Logging Through Emerging Non-volatile Memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, June 2014.

[46] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, Santa Clara, CA, February 2015. USENIX Association.

[47] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18, New York, NY, USA, 2015. ACM.

[48] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM.