Storage Networking Industry Association
Technical White Paper

Persistent Memory Atomics and Transactions

Version 1 Revision 1

January 10, 2017

ABSTRACT: This white paper describes considerations for supporting atomics and transactions using extensions to the NVM Programming Model Specification.

Revision History

Revision	Date	Sections	Originator:	Comments
0.17	7/15/16	All	D. Voigt	Initial Publication
1.0	10/13/16		SNIA TC	Approved for publication by Technical Council
1.0 R1	1/10/17		D. Voigt	Editorial corrections throughout

Table of Contents

Scope

This whitepaper describes considerations in developing libraries implementing atomic updates and transactions within the context of byte-addressable persistent memory (PM). PM is accessed (like volatile memory) using processor load and store instructions, but it retains its contents across power loss (like storage). The problem this paper addresses is the ability to store and access program data structures in persistent memory in a way which is robust against power or system failures. While there are several solutions to the problem, this paper focuses on transactions and atomic operations implemented in libraries.

This paper builds on SNIA's *NVM Programming Model* specification [see NPM], which defines behavior for PM aware file systems, but does not address atomic operations or transactions of arbitrary sizes. Such capabilities have been addressed to varying levels in published academic and research efforts and this paper describes how they apply to PM. Strategies for defining data structures supporting transactions in persistent memory are also considered.

Processor Instruction Set Architectures define operations where multi-thread atomicity is guaranteed. For example, on x86 architectures, MOV instructions with naturally aligned operands of at most 64 bits qualify. Failure atomicity for persistent memory is accomplished using CPU flush and fence instructions.

There are several ways the limited failure atomicity provided in hardware can be generalized by software, including:

- Software may implement an **abstraction of hardware failure atomicity** that allows software to use hardware independent APIs. These use architecture-specific operations where appropriate or software alternatives. The software alternatives may utilize locking to assure multi-thread atomicity.
- **Software failure atomic operations** provide failure atomicity for a specific type of data structure (for example, updating a linked list).
- A **transaction manager may** provide failure atomicity for a list of updates to members of one or more types of data structures.

This whitepaper does not prescribe a specific implementation but discusses considerations for developing PM-aware libraries implementing failure atomic data structure updates. The discussions apply to libraries implementing combinations of these three software generalizations.

1 Introduction

1.1 Assumptions

This paper presents concepts that apply to different programming languages. Libraries that implement the behavior described here are often written in C, so C is used in examples. Similarly, file systems features (such as memory mapped files) are available in many operating systems, but the examples here use POSIX APIs.

The SNIA programming model (see 1.2 Relationship with the SNIA NVM Programming Model) uses files to name ranges of PM. The application maps these ranges to associate physical PM with process virtual memory. The assumption is that PM files hold a large amount of data, possibly all of an application's data (or some set or related data structures). Typically there would not be a separate PM file for each variable.

All discussions assume there are ways of sharing memory across different threads of execution with the ability to share address space and virtual to physical address space mapping.

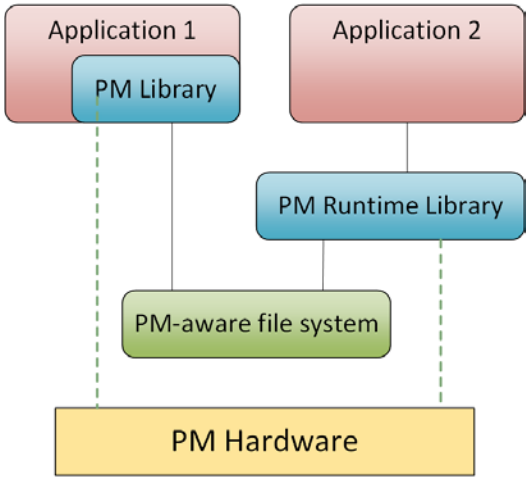
This paper does not address behavior across threads in different processes.

System architectures may include other ways of accessing PM, including DMA.

1.2 Relationship with the SNIA NVM Programming Model

The SNIA *NVM Programming Model* [see NPM] defines the behavior user-space software uses to access PM. A typical pattern for applications using this model is:

- open a file on a PM-aware file system
- memory map the file into process virtual memory. Unlike a legacy block-device oriented file system, a PM memory maped file enables direct access to PM without paging
- use load/store operations to access PM
- flush data from CPU caches to PM



The libraries described in this paper act as a layer between applications and the file system and PM hardware. The libraries provide failure atomicity and transaction support, implemented similarly to existing transaction manager implementations, but optimized around unique characteristics of PM. Possible deployments are depicted in the above figure. The dashed lines depict load/store operations; the solid lines depict API calls. Application 1 links in a PM-aware library, but uses a compiler without PM support. Application 2 uses a PM-aware compiler and run-time library.

The behavior described here may apply to kernel code, but the focus of this paper is user space, non-kernel software.

In general, PM-aware libraries may be accessed by an end-user application or a middleware component such as a transaction manager or data-base middleware. In this paper, *PM library consumer* may refer to any software that uses PM libraries' APIs.

1.3 PM and Crash Consistency

Crash consistency is a common recovery model for persisting data in today's storage devices such as disk drives. In disk drives, the order of completion of outstanding writes is indeterminate. In addition, a write may not be completed if power fails. Since disk drives offer weak ordering guarantees, applications must be prepared to recover from any state of the writes that are in flight when a failure occurs. This brings us to the concept of crash consistency, in which the state of an application's data after a failure need only match the indeterminate write order guarantee of a group of disk drives.

More formally, an application's data state is considered crash consistent if it could have resulted from power loss of a group of direct attached disks given the sequence of write commands and completions leading up to the failure. This means that there is a rolling window of outstanding write requests whose order is uncertain. Applications must be able to recover from any order of those requests and must account for storage system atomicity nuances in the process. For an application, recovery from a crash consistent image is the same as a cold restart after a system crash.

Now consider the map-and-sync methodology described in the NVM.PM.FILE mode of the NVM Programming Model NPM. Sync has a very specific meaning. The only guarantee that sync makes is that all stores in the address range of the sync that occurred before the sync are in persistent memory when the sync completes. Sync does not otherwise restrict the order in which data reaches persistent memory. For example, if cache lines 1 through 5 were written in order by the application before a sync, cache line 5 might have reached persistent memory first, possibly before the sync even started. This flexibility enables CPUs to optimize write ordering for cache performance.

This uncertainty gives rise to a lowest common denominator for NVM.PM.FILE recovery that is highly analogous to that which exists for disk drives. Specifically, the application is uncertain as to which of the store instructions between two sync actions will appear in persistent memory after a failure that occurs before completion of the second sync action. If the actions and attributes of the NVM Programming Model are all that is available, then the application must execute additional sync actions whenever the order of stores to persistent memory matters.

More formally, a persistent memory range is crash consistent if its contents at the start of recovery could have resulted from the pattern of stores and syncs executed on the processor(s) with data in flight to the persistent memory prior to failure. In both disk drives and persistent memory, some aspect of failure atomicity is built into the crash consistency assertion. The NVM programming model describes atomicity for both disk drives and persistent memory.

Crash consistency is a complex approach to recovery from an application standpoint. It also forces considerable overhead to precisely communicate every sync action to persistent memory. This further illustrates the motivation for some notion of consistency points such as persistent memory transactions.

1.4 Requirements

Databases and transaction processing systems have been providing atomic operations and transactions with persistent storage for decades. Such systems have provided atomicity, consistency, isolation, and durability (ACID) properties through a transactional interface. Such an interface provides the programmer with a mechanism for the reliable transition of data from one consistent state to another. Implementations have utilized various logging techniques (e.g. write ahead logging) to provide such behavior with block devices, such as HDDs and SSDs, as the target for the atomicity and durability of transactions. This paper explores techniques for providing transactions and atomic operations on PM; some are unique to PM and some are closely related to techniques used with block devices.

In developing extensions to the existing *NVM programming model* for atomics and transactions, the following goals for such capability were identified:

1. *Provide for atomic updates to PM that support durability in the event of system or power failures* –The techniques described in this paper help software achieve consistency for data residing in PM.
2. *Provide for failure atomic updates on large address ranges or groups of ranges* – The proposed techniques allow implementations to achieve atomic updates for multiple ranges larger than the granules supported by single hardware instructions.
3. *Work with existing compilers* – This document does not elaborate on approaches that would require core language extensions or modifications. Standard language support for PM is a goal, but may take a long time to be defined, implemented and adopted by applications.
4. *Work with existing processors* - Any approach to atomicity and transactions shall rely on existing or near term processor capabilities, but also consider emerging PM-related hardware features.
5. *Ability to avoid unnecessary serialization or redundant instrumentation* – An application developer may know that in certain cases, PM data can be safely accessed without locks. Similarly there may be cases where durability goals can be achieved while selectively bypassing read/write fences that enforce ordering. Implementations that prevent developer control of instrumentation or serialization related to locks or fences may prevent optimal performance.

To address these requirements, this paper proposes behavior for libraries supporting failure atomicity and transactions for families of PM-aware data structures. Library consumers control the way data is stored in PM: the consumer may opt for fixed-sized records, variable-length key/value strings, or any other layout. *PM-awa*re refers to mechanisms that address unique characteristics of persistent memory such as avoidance of persistent memory leaks. The libraries may support transactions that include a mix of different types of data structures.

1.5 Key concepts for PM programming

1.5.1 Two senses of atomicity

There are two aspects of atomicity related to the proposed libraries: *failure atomicity* and *multi-thread atomicity.* Both aspects guarantee that all or none of the updates are completed. For both aspects, techniques used in existing transaction managers apply. For PM, failure atomicity requires PM-aware libraries be sensitive to behavior related to

CPU caches. This paper includes extensive discussion of the relationship between CPU caches and failure atomicity.

This paper does not address multi-thread atomicity. Implementations of PM libraries may choose to give guarantees of behavior across threads or allow consumers to control this behavior, but PM does not introduce new behavior. Once again, techniques used in existing transaction managers apply.

1.5.2 Unique concerns for PM

Many practices that are used to implement atomicity with volatile memory and disk drives also apply to persistent memory atomicity. For example,

- running recovery logic after a restart that rolls back incomplete atomic updates
- use of a log to record pending persistent updates (used by recovery logic),
- techniques that manage concurrent updates

PM introduces some new considerations for implementations. In today's systems when an application restarts, its allocated memory is implicitly freed and made available to other applications. Allocation of PM, on the other hand, survives application restarts. An allocation of persistent memory is typically followed by steps to reference the allocation from a data structure such as a list or tree. A failure may cause newly allocated persistent memory to remain unreferenced, potentially creating a persistent memory leak. One way that PM-aware software can protect from leaks is to use recovery logic, possibly assisted by a log, to keep track of allocation as well as updates.

PM-aware software may also wish to consider memory-related granularities (cache-line, atomic write, hardware, and kernel copy sizes) to minimize inefficiencies due to read/modify/write sequences.

1.6 Related Work

Related academic and research efforts were investigated, which will be discussed below. This investigation was used to identify common characteristics, generalize, and form a transaction approach for PM. Some approaches were avoided because they rely on processor and/or language extensions that are not aligned with the requirements stated in section 1.4. For example **BPFS** [see Condit] provides for fine-grained atomic updates to persistent memory as a part of their file system. However it relies on core processor changes (e.g. cache controller, memory controller). **NV Heaps** [see Coburn] was also investigated for its flexible ACID transactions but it has the same processor enhancement requirements as BPFS. Additionally **Kiln** [see Zhao] was considered for its ability to perform in-place updates without copy-on-write (COW) or logging however it requires cache controller and ISA extensions. **Transactional Memory** [see Herlihy] defines behavior similar to the goals of this paper, but requires extensions to processor or cache hardware. **Mnemosyne** [see Volos] provides persistence primitives and a durable memory transaction mechanism that enables consistent updates of arbitrary data structures. However the model is supported through language extensions.

Software Transactional Memory (STM) [see Shavit] adapts Transactional Memory to use software rather than hardware extensions. The goal for transactional memory is minimizing the need for programmers to explicitly manage locks. But existing STM implementations have a few drawbacks. The first is that STM proposals typically define programming language extensions. The second is that proposed STM implementations add instrumentation code to monitor memory or lock usage which usually prevents linking with existing binaries. Further, the injected instrumentation often creates redundant read or write barriers, causing STM implementations to perform significantly slower than implementations with manually managed locks [see Cascaval and Yoo].

In **Atlas** [see Chakrabarti] all ACID properties are supported by ensuring that critical sections, which are already used to demarcate regions in which data structures are inconsistent, are committed atomically to PM. Isolation is provided using existing lock structures and consistency is supported using existing application techniques. They key component added in Atlas is failure atomicity (durability) for memory areas that are defined as persistent (e.g. persistent regions).

The **C11 and C++11 standard** [see C11] does not address persistent memory or failure atomicity.

In summary, many of the techniques used by transaction manager implementations targeting block devices also apply to PM. PM-specific considerations relate to recovery. For example, rolling back a transaction may require un-allocating PM. Research discusses benefits (by reducing coder errors) of language extensions and software transactional memory, but in practice, the associated performance costs have been high [see Cascaval].

2 Considerations for Developing PM-aware Libraries

2.1 Hardware Considerations

The term “aligned operations on fundamental data types” refers to the set of operations that enable optimal CPU/memory performance. Aligned operations on fundamental data types are usually the same operations that under normal operation become visible to other threads/data producers atomically. In other words, aligned operations on fundamental data types can be made atomic without a lock. For a library to update a larger range atomically, it needs to lock out other threads, invoke a set of store operations no larger than fundamental data sizes, invoke a fence operation, then unlock.

As an example, consider a store of nine-bytes of data

```
  char data[16] = "xxxxxxxxx"; // allocate 16 bytes
  memcpy(data, "123456789", 9); // change 9
```

On an architecture with a 64-bit fundamental data type size, the 9 byte memcpy operation may cause the compiler to generate two 8 byte atomic store instructions, not a single atomic operation. Because this straddles the fundamental data type size, cache pressure may cause the two stores to be flushed to persistence in an arbitrary order or not at all. A failure during the flush may cause undefined contents after a restart. There are multiple scenarios where a failure while the memcpy is being executed may cause a torn write. The recipe described in the previous paragraph (lock, multiple stores of fundamental data type sizes, flush, fence, unlock) will provide multi-thread atomicity, but not failure atomicity for PM.

A PM-aware library may wish to provide an abstraction of hardware failure atomicity. The library could provide a method to flush CPU cache to PM that would use the most optimal CPU instructions available. For example on Intel products this may involve CLFLUSHOPT and CLWB on newer platforms, falling back to CLFLUSH on older platforms.

Not all CPU instructions are optimized for PM. For example, Intel CMPXCHG (Compare and Exchange) provides an atomic compare and memory update, but does not flush to PM. Adapting lockless algorithms using CMPXCHG may require introducing a lock to provide atomicity across the CMPXCHG and CLFLUSH.

2.2 Intrinsically atomic PM data structures

In some cases simple PM data structures can be implemented using a single store to a fundamental data type to provide failure atomicity. This approach is limited to situations that meet all of the following requirements.

- Only one location containing valid data in the data structure is updated in place.
- Space allocation and management is self-contained within the PM data structure implementation.
- The application does not require failure atomicity from multiple data structures at the same time.

Several examples of intrinsically atomic data structures are described in section 3. Any use case that does not meet all of the above requirements requires a transaction.

2.3 Identifying code locations with failure atomic considerations

The places where locks protect data structures for multi-thread safety are often the places where you need to consider failure atomicity.

2.4 Visibility and Isolation

Transaction managers allow applications to specify a list of updates to multiple nodes in multiple data structures, and assure the entire list of updates is performed atomically. A common technique used in transaction managers is a *transaction log* (or *journal*) where information about transactional updates is stored. After a failure, the transaction manager "reviews the database logs for uncommitted transactions and rolls back the changes made by these transactions" (from Wikipedia "Transaction log"). There are numerous approaches to transaction logs. One of the differentiating characteristics is whether the implementation stores copies of pre-transaction or intended values in the log. **Before image** (undo) logging copies the pre-transaction data to the log, then updates the PM data in place. Recovery code removes the effect of any partially committed transactions. **After image** (redo) logging writes intended updates to data in the Log, then updates the PM data in place after the transaction commits. Recovery consists of replaying all committed transactions in the intent log. *Free Transactions with Rio Vista* (see Lowell) describes a transaction manager using a before image log. The ARIES paper (see C. Mohan) describes a transaction manager using an after image log.

The choice of before or after image logging influences the visibility of data in partially completed transactions. Consider software accessing data in a before image logging implementation. A load instruction of that data address exposes updates made previously in the transaction. But a load instruction in a redo logging implementation will access the pre-transaction version. PM libraries with transaction logs should document how their implementation affects visibility. Locks with memory barriers may be required to achieve the desired behavior.

2.5 Persistent Memory Allocator

Legacy applications typically use data structures such as lists or trees to allocate memory dynamically. Allocations of volatile memory come from a single anonymous heap. On the other hand, the SNIA *NVM Programming Model* allows PM to be assigned to files, which can be viewed as PM heaps. In order to enable the use of data structures in PM, a library may wish to provide a PM allocator. An application would use this allocator similarly to malloc() to allocate PM for list or tree nodes. If a library implements an allocator for PM, it should provide a mechanism to enable detection of allocations that are not being used to account for scenarios such as the following.

A library implementing a transaction manager is probably using some sort of transaction log (e.g., undo log, redo log), or some other mechanism that tracks updates relative to each transaction. Recovery logic runs at application startup to find incomplete transactions and perform tasks needed to regain consistency. This same log and recovery logic can be enhanced to track allocations. One approach is to track the state of an allocated range, differentiating between reserved and allocated states. If the range has a state indicating it’s reserved, but not allocated; then the recovery logic frees that range.

2.6 Persistent Memory Transaction Manager

When failure atomicity must span multiple data structures such as records in a database or key value store, a PM transaction manager is required. In this paper, the following commands represent APIs commonly included in Transactional APIs:

- begin_transaction: start a transaction
- end_transaction: end a transaction and commit updates since its corresponding begin_transaction
- set_range: informs the library which ranges of PM must be updated transactionally

A transaction manager may provide the capability to atomically update data in multiple data structures (for example, add an item to a shopping cart data structure, while removing it from an inventory data structure).

Psuedo-code for a use case involving a PM transaction manager appears in section 3.3.

2.7 Recovery

Recovery refers to logic run at application startup, or when an error is detected, to return data to a consistent state. Recovery logic looks at data that has recently changed, often prior to a restart, to detect cases where atomic updates failed or were incomplete. Consistency is defined by the application and varies with the type of atomicity provided and implementation choices. For example, recovery of an atomic log append may be restricted to truncating the log to exclude partial operations. Recovery for a transaction manager with a redo log may be able to complete in-flight transactions after a restart.

3 Use Cases

This section presents a few use cases where an application uses capabilities of PM-aware libraries. The use cases show applicability of ideas introduced in section 2 Considerations for Developing PM-aware Libraries. The simpler use cases include both pseudo-code and C language code examples. As the use cases get more complex, example code gets lengthier and is omitted from more advanced use cases.

3.1 Add a node to a linked list (atomic operation)

In this use case, an application allocates memory for a list node, populates the data, and then inserts the node in the list. To achieve failure atomicity, partially completed effects need to be reverted after a failure. Three variations are included, showing how a basic implementation is expanded to provide multi-thread and failure atomicity support.

Assumptions:

Each of the three variations share the following data structures.

```
/* the node struct represents a list element */
struct node {
    struct node *nextp;
    int data;
};
/* a single root struct that points to the list head */
struct root {
    struct node *headp;
    /* the second and third variations include a list lock */
};
```

The addnode() function takes two arguments: a pointer to the list’s root node and the value to be set in the new node.

The linked list API is assumed to support one specific node type. In practice, a general-purpose linked list API is desirable. Existing techniques for implementing APIs supporting application-defined types can be used with PM; but for simplicity, this use case assumes the application and library share the definition of the node struct.

Basic version of addnode()

The first version of addnode() uses volatile memory and does not consider multi-threading.

1. allocate a new *node* struct
2. assign the call's data value to newnode->data
3. assign newnode's next pointer to the previous list head (root's headp)
4. assign root's head pointer to the new node

```
void
addnode(struct root *rootp, int data)
{
    struct node *newnodep;

    if ((newnodep = calloc(1,
        sizeof(struct node))) == NULL)
      fatal("out of memory");

    newnodep->data = data;
    newnodep->nextp = rootp->headp;
    rootp->headp = newnodep;
}
```

Add Multi-thread Safety

The second version adds locking to support multi-threading. This is accomplished with a single pthread_mutex which protects the entire list. Three changes are made to the basic example:

1. listlock is added to the definition of *node*
2. the list is locked before updating the *root* and *node* objects
3. the list is unlocked after updating the *root* and *node* objects

```
void
addnode(struct root *rootp, int data)
{
    struct node *newnodep;

    if ((newnodep = calloc(1,
        sizeof(struct node))) == NULL)
      fatal("out of memory");

    newnodep->data = data;

    /* lock the critical section */
    pthread_mutex_lock(
       &rootp->listlock);
    newnode->nextp = rootp->headp;
    rootp->headp = newnodep;
    pthread_mutex_unlock (
       &rootp->listlock);
}
```

Persistent Memory Version

This version of addnode() uses persistent memory in a way that provides intrinsic failure atomicity. Two simple PM operations are used in the example:

- The library implements a PM allocator. For the purpose of this use case, this allocator provides an alternative (pm_calloc) to standard calloc() to allocate and zero a range of memory from a PM file.
- This version introduces a flush operation (pm_flush). When software gets a response from a flush operation, it knows that the specified range of addresses have been flushed to persistent memory. A systematic approach to storing and flushing data allows software to determine which data was stored successfully prior to a failure.

In the pseudo code below, the critical section includes flushing the newnode instance and exploiting the machine architecture's atomicity when doing aligned stores to 8-byte fundamental data types.

The steps for the persistent version:

1. allocate a node from PM, and save its address
2. lock out other threads from updating the linked list
3. assign the new node's next pointer to the root pointer's next (i.e., the head of the list)
4. assign the value (passed in by the caller) to the new node's *value*
5. flush the new node's memory range from CPU cache
6. assign the root pointer's head to the new node, implemented as an aligned operation on a fundamental data type – treated as a failure - atomic store
7. flush the root pointer's head from CPU cache
8. unlock the list

```
void
addnode(struct root *rootp, int data)
{
    struct node *newnodep;

    if ((newnodep = pm_calloc(1,
        sizeof(struct node))) == NULL)
      fatal("out of memory");

    newnodep->data = data;

    /* lock the critical section */
    pthread_mutex_lock(
       &rootp->listlock);
    newnodep->nextp = rootp->headp;
    pm_flush(newnode,
        sizeof(struct node));
    rootp->headp = newnodep;
    pm_flush(&(rootp->headp),
        sizeof(rootp->headp));
    pthread_mutex_unlock (
       &rootp->listlock);
}
```

Recovery code - Step 6 uses an atomic store capability provided by the machine architecture to store and flush a pointer. This enables a simple start-up recovery implementation with assistance from the pm_calloc implementation. The linked list recovery code must examine the saved address of allocated node structures recorded by pm_calloc, and free any that are not part of the list. These occur as partial effects due to failures while nodes were being added.

3.2 Append to a file atomically (atomic operation)

Append logging is used when an application appends data to a log file to allow recovery to a consistent state after a failure. This log file could be used to allow restoring a database to a consistent state when in-place data updates are interrupted. This example does not use transactions; it's a restricted use case where intrinsic failure atomicity can be provided without transactions.

Ignoring details of the API, the steps are

1. The application calls an API to append referenced data to the log
2. The library implementing the API
   a. locks other threads from updating the data being logged and from updating the append log file
   b. appends the data to the log file
   c. updates and flushes the file's end pointer
   d. unlocks the lock from step a

3. When the application restarts, it calls the library's recovery logic which uses the log to determine whether interrupted append operations can be completed. The library

undoes any partial effects related to append operations that can't be completed. For example, allocated blocks that are not in any file may need to be recovered. Since this is a common practice to avoid loss of free space in file systems, the append logging library may not need any recovery logic of its own.

There are several reasons why this use case is simpler than the insert-a-node use case.

- Appends don't involve in-place updates
- The library uses a single write/store command to update the file.

3.3 Transactional Multi-record update

This use case illustrates the swapping of data values in 2 records such as might reside in a database or key value store. Consumers of a transaction manager may need to consider the library's approach to visibility (see 2.4 Visibility and Isolation). This is shown in pseudo-code for before and after image transaction managers.

Before image log approach

The application code

1. begin_transaction
    a. the library starts a transaction, initializes a transaction log record
2. Lock out other threads
3. set_range record1
    a. the library keeps track of this range; copies this range into the transaction log at appropriate time
4. set_range record2
    a. the library keeps track of this range; copies this range into the transaction log at appropriate time
5. Copy record1 to a temp buffer
6. Copy record 2 to record1
7. Copy temp buffer to record2
8. Release lock
9. end_transaction

The library commits the updates or reverts to pre- begin_transaction state on failures

The open source NVML (http://pmem.io/nvml/) libpmemobj library provides a before image PM transaction manager that maps closely to the pseudo-code above.

After image log approach

1. begin_transaction
    a. the library starts a transaction, initializes a transaction log record
2. Lock out other threads from writing and reading records 1 and 2
3. Copy record1 into the log as the intended value of record 2
4. Copy record 2 into the log as the intended value of record 1
5. Release lock
6. end_transaction

4 Conclusions

The aspects of atomicity that are specific to PM are those related to failure atomicity. The proposed approach to PM atomicity and transactions in this whitepaper was developed by leveraging related academic and research publications. Many of these publications rely on hardware capabilities that don't appear to be planned for production systems, programming language extensions that are not part of compiler implementations, or fail to address failure atomicity. This led to defining requirements (see 1.4 Requirements) that allowed implementations using the first generations of hardware and kernel support for PM.

PM introduces the opportunity to achieve very high performance failure atomicity using intrinsically atomic PM data structures. When used in isolation these data structures do not require transactions, however they are only applicable under certain conditions. Situations involving multiple PM data structures or complex space management require transactions. Ideally these transactions comprise groups of the same intrinsically atomic PM data structures that can otherwise be used in isolation.

Because many of the considerations for PM atomics and transactions also apply to volatile memory, existing transaction manager APIs provide a good starting point for developing PM libraries. Many of the PM-specific considerations can be implemented in the library.

5 Future Work

This paper discusses implementing PM libraries using standard C and existing compilers. Forward looking work continues on complier extensions for PM atomicity.

5.1 Atlas

Atlas builds on the relationship between locks and failure-atomicity (see 2.3 Identifying code locations with failure atomic considerations). Atlas identifies failure- atomic sections of code based on existing critical sections and provides a log-based implementation that can be used to recover a consistent state after a failure. This is accomplished by implementing a compilation pass that instruments synchronization operations and store operations that appear to be directed to persistent memory. This results in calls to the Atlas runtime library, where synchronization operations and stores to persistent memory are tracked in a persistent log. This log is used when the application restarts and while the application is running.

An application known to be free of data races, implemented using a PM-aware file system (as described in the SNIA NVM Programming Model NVM.PM.File mode – see NPM), can easily be adapted to achieve failure atomicity with Atlas.

Atlas has been released under the GNU Lesser Public License Version 3. It is available at https://github.com/HewlettPackard/Atlas.

5.2 Oracle NVM Direct

NVM Direct defines language extensions for PM and includes a library that provides entry points called by the extensions as well as functions that applications call directly. It is possible to use the library without the C extensions.

The API provides facilities for an application to map PM into its virtual address space and access it with loads and stores. This requires an OS file system that implements the NVM Programming Model NVM.PM.File mode (see NPM).

- The API manages region files that are formatted for use by the library.
- The API provides transactions to atomically update complex data structures in PM
- The API provides PM mutexes that can be used to coordinate access to PM data by a multi-threaded application
- The API supports multiple PM heaps for allocating application defined structs in PM

NVM Direct defines language extensions that can be implemented using a pre-compiler. These language extensions simplify writing NVM code, automate some coding to reduce bugs, and add runtime checks to catch corruption early.

6 Bibliography

C. Mohan	ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, http://www.cs.berkeley.edu/~brewer/cs262/Aries.pdf
Arpaci-Dusseau	Crash Consistency: FSCK and Journaling, http://pages.cs.wisc.edu/~remzi/OSTEP/file-journaling.pdf
Chakrabarti	D. Chakrabarti, H. Boehm and K. Bhandari, "Atlas: Leveraging Locks for Non-volatile Memory Consistency"
Condit	J. Condit, "Better I/O Through Byte-Addressable, Persistent Memory".
Cascaval	Cascaval, Blundell, et al, “Software Transactional Memory: Why is it Only a Research Toy?”
C11	The current standard for Programming Language C (C11) is ISO/IEC 9899:2011, published 2011-12-08. Technical Corrigendum 1 (ISO/IEC 9899:2011/Cor. 1:2012) was published in 2012. Published ISO and IEC standards can be purchased from a member body of ISO or IEC. The latest publically available version of the C11 standard is the document WG14 N1570, dated 2011-04-12. (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf) This is a WG14 working paper, but it reflects what was to become the standard at the time of issue.
Zhao	J. Zhao, "Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support”
Volos	H. Volos, A. J. Tack and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory".
Coburn	J. Coburn, "NV-Heaps: Making Persistent Objects Fast and Safe".
Herlihy	M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for

	Lock-Free Data Structures".
Lowell	David E. Lowell, Peter M. Chen ,"Free Transactions with Rio Vista" http://web.eecs.umich.edu/virtual/papers/lowell97.pdf
NPM	SNIA NVM Programming Model http://www.snia.org/tech_activities/standards/curr_standards/npm
NVM-Direct	https://github.com/oracle/NVM-Direct
Shavit	Nir Shavit, Dan Touitou: Software Transactional Memory.
Yoo	R. Yoo, Y Ni, et al "Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough"
Intel	Intel® Architecture Instruction Set Extensions Programming Reference, https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf

7 Appendix A – Open Source NVM Library

Many of the examples and behaviors described in this white paper have been implemented in the open source NVM Library "NVML", available at http://pmem.io/nvml/.