Cache-Line Transactions: Building Blocks for Persistent Kernel Data Structures Enabled by AspectC++

Marcel Köppen
Osnabrück University
Germany
Marcel.Koeppen@uos.de

Jana Traue
BTU Cottbus
Germany
Jana.Traue@b-tu.de

Christoph Borchert
Osnabrück University
Germany
Christoph.Borchert@uos.de

Jörg Nolte
BTU Cottbus
Germany
Joerg.Nolte@b-tu.de

Olaf Spinczyk
Osnabrück University
Germany
Olaf.Spinczyk@uos.de

Abstract

With the availability of systems that contain large amounts of byte-addressable non-volatile memory (NVRAM), there is a growing need for data structures that can be mapped into a process's address space and be used without data (de-)serialization. While NVRAM is able to retain memory contents during system failure and power loss, data consistency has to be preserved by using transactional operations for data manipulation.

This paper describes a lightweight and efficient transaction mechanism for small data structures in memory-mapped NVRAM. The size per data structure is limited to half a cache-line, so that the approach cannot serve as a general purpose mechanism for arbitrary applications, but could be used within an operating system as a low-level building block for more complex data structures. By using aspect-oriented programming with AspectC++, the mechanism can be used in an almost transparent manner, which helps to avoid many possible sources for bugs.

CCS Concepts • **Software and its engineering → Consistency**; *Language features*; • **Hardware** → *Non-volatile memory.*

Keywords Non-Volatile Memory, Persistent Data Structures, Aspect-Oriented Programming

PLOS '19, October 27, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-7017-2/19/10…$15.00
https://doi.org/10.1145/3365137.3365396

1 Introduction

In recent years, there has been an increasing number of research articles on the development of new memory technologies [18] and the implications of fast non-volatile byte-addressable main memory on the design and implementation of operating systems [1].

Meanwhile, NVRAM has become available in commercial products such as FRAM-based microcontrollers from Texas Instruments and 3D-XPoint/Optane DC memory in Intel's server CPU generation Cascade Lake. It is time to exploit this almost ideal memory on all layers of the software stack, especially within the operating system for its own data structures and its applications. For example, file systems and databases for NVRAM have been proposed [9, 21]. An alternative interface is to directly map NVRAM into the address space of applications [27]. A consensus on a programming model for NVRAM has not yet been found [3].

Since NVRAM modules are typically used behind a fast volatile cache, the order of write operations and the eviction/write back strategy of the cache are of utmost importance to avoid inconsistent data. Data only becomes persistent when it reaches the NVRAM module. This calls for software transactions with explicit cache flush operations during a commit. It is also important that modifications that reach the NVRAM before the commit can be rolled back. Figure 1 shows how this can be achieved with Intel's Persistent Memory Development Kit (PMDK) [12], which implements log-based transactions on data structures in persistent memory pools. In this example, each public member function of the class has transactional semantics. This interface has two problems: First, log-based transactions are a heavy-weight mechanism that comes at high cost, especially for frequently used small data structures. Second, the various library calls and wrapper templates for member variables complicate the code base and give the programmer various "opportunities" to introduce bugs. As a remedy to these problems, this paper makes two contributions:

```
class PMDKBoundedBuffer {
  static constexpr int S = 29; // buffer SIZE
  typedef pmem::obj::p<char> pc;
  pmem::obj::persistent_ptr<pc[]> buf;
  pmem::obj::p<uint8_t> in, out;
public:
  PMDKBoundedBuffer() : in(0), out(0) {
    buf=pmem::obj::make_persistent<pc[]>(S);}

  ~PMDKBoundedBuffer() {
   pmem::obj::delete_persistent<pc[]>(buf,S);}

  void addByte(char data) {
    auto pop = pmem::obj::pool_by_vptr(this);
    pmem::obj::transaction::exec_tx(pop, [&]{
      if ((in+1) % S == out) { return; }
      buf[in] = data;
      in = (in + 1) % S;
    });
  }

  char getByte() {
    char result = 0;
    auto pop = pmem::obj::pool_by_vptr(this);
    pmem::obj::transaction::exec_tx(pop, [&]{
      if (out == in) { return; }
      result = buf[out];
      out = (out + 1) % S;
    });
    return result;
} };
```

Figure 1. Implementation of a persistent bounded buffer using PMDK member wrappers and transactions.

- A lightweight transaction mechanism for small data objects, which exploits the memory ordering of stores to a single cache-line.
- A convenient programming interface based on AspectC++, which is a general purpose aspect-oriented programming language extension for C++ [25].

The remainder of this paper is organized as follows: After discussing related work in Section 2, Section 3 introduces our lightweight transaction mechanism for NVRAM. The programming interface is described in Section 4. Finally, we quantitatively compare our approach with a PDMK-based implementation to illustrate its superior performance in Section 5. The paper ends with a general discussion of possible use cases and other conclusions in Section 6.

2 Related Work

Commercial products with non-volatile byte-addressable memory like Intel's Optane DIMMs hit the market only recently. In the meantime, research on system integration of NVRAM had to speculate on the underlying technology and its properties like latency and durability. One of the central questions was whether NVRAM should be considered as slow main memory or as fast storage. These different views lead to different approaches of how to integrate NVRAM into systems.

```
class [[NVM::transactional]] alignas(64)
BoundedBuffer {
  static constexpr int S = 29; // buffer SIZE
  char buf[S];
  uint8_t in, out;
public:
  BoundedBuffer() : in(0), out(0) {}

  void addByte(char data) {
    if ((in+1) % S == out) { return; }
    buf[in] = data;
    in = (in + 1) % S;
  }

  char getByte() {
    if (out == in) { return 0; }
    char result = buf[out];
    out = (out + 1) % S;
    return result;
} };
```

Figure 2. Bounded buffer implementation augmented with cache-line transactions using AspectC++.

When NVRAM is seen as a fast, byte-addressable storage device, it can be used with a traditional interface for storage: file systems. Byte-wise modifications of file contents have long been possible with memory mapped files, but file system metadata was optimized for block-based access. File systems for NVRAM, like PMFS [10], BPFS [9], and SCMFS [28], focus on efficient metadata management for byte addressable storage.

Databases are another interface for storage. In contrast to file systems, they guarantee consistency not only for metadata but also for the data itself. Modern in-memory databases, like Sofort [21], hold most information in main memory and storage is only needed for durability. Such an architecture can benefit enormously when disks are replaced by NVRAM, as analyzed by Bailey et. al. for a minimalistic in-memory database: the Echo key-value store [2]

The other approach, i.e. treating NVRAM like DRAM, discusses the mapping of persistent memory to certain ranges in applications' address spaces. Changes to programming languages and run-time systems have been discussed for the persistent memory frameworks Mnemosyne [27] and NV-Heaps [8].

All these approaches for NVRAM integration have in common that they need to guarantee consistency of persistent information, at least for metadata. NVRAM brings a challenge which storage had hardly faced before: ordering. Reads and writes to traditional storage devices have been completely controlled by software. A system with NVRAM is likely to use volatile buffers, such as caches, which may cause the order of writes at the memory controller to differ from program order. Moreover, these buffers lose their information when the power runs out. The consequences can be harmful for consistency, e.g. when a log entry is marked as valid before its creation is complete and the power runs out.

A reactive way to address volatile buffers is to flush their contents to a persistent memory region when the power runs out. One implementation of such a Timely Sufficient Persistence model [20] is Whole-System Persistence [19]. This idea has the benefit of imposing no runtime overhead, but it relies completely on the correctness of the failure handling. If the system does not have sufficient capacity to store all data, information is lost.

An alternative is to pro-actively care about durability and ordering of data. A system which writes all information to NVRAM as defined by program order adheres to the model of *Strict Persistency* [23]. Its effects can be projected by imagining that caches are completely disabled or set to write-through. The performance degradation for both modes will be immense. Inspired by research in the field of parallel programming with shared memory, relaxed persistency models like *Buffered Strict Persistency* [23], *Epoch Persistency* [23], and *Speculative Persistency* [17] which allow for relaxed ordering of writes have been proposed. But so far, none of the ideas has found its way into processors, so that NVRAM users will have to rely on memory barriers and cache-line flushes.

In the field of databases, transactional semantics are typically established using undo logging, redo logging, or multiversioning. In the context of NVRAM, undo logs are used in PMFS [10], Atlas [5], and by Kolli et al. [13]. Because the costs of preserving consistency with NVRAM will be dominated by cache-line flushes [15, 17, 28], their usage should be kept at a minimum. Lu et al. [16] proposed an undo log optimization called *Eager Commit* to minimize flushes. Redo logging is employed by Mnemosyne [27]. Because there is no obvious performance benefit to undo or redo logging, the persistent programming library REWIND [6] lets its users select the fitting scheme.

Both undo and redo logging write frequently to the memory location hosting the log. This does not fit memory technologies which suffer from wearing effects. One alternative is to use multiversioning and another option is to use hand crafted data structures which are optimized for NVRAM. Since trees are a central data structure for file systems and key value stores, multiversioning trees [9, 26] as well as specially crafted tree variants [7, 14, 22, 29] for NVRAM have been designed.

Automatic application of transactions to annotated Java code has been described for AspectJ [24]. In contrast to our contribution, this solution cannot transparently change the object layout.

3 NVRAM Cache-Line Transactions

The x86 memory-ordering model guarantees that an older store will not pass a younger store to the same cache-line. This holds true when the transfer of a cache-line to the NVDIMM is terminated by power failure during a cache-line flush or when a cache-line is evicted from the cache. We use these properties to build a transaction mechanism on a single cache-line.

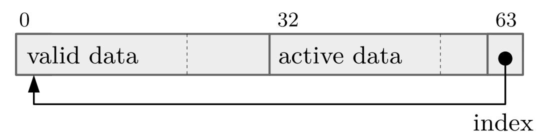

Figure 3. Cache-line memory layout

A typical cache-line has a size of 64 bytes (see Figure 3). We use this memory as a kind of double buffer, i.e. one half of the memory contains the last consistent version of the transactional data structure, while the other half contains a working copy. Furthermore, we reserve one byte of the cache-line to indicate which part of the double buffer is currently considered valid.

On the start of a transaction, the working copy is created from the valid part of the cache-line. During the transaction, write operations are carried out on this active part. The commit operation flips the index, so that the modified version becomes the new consistent version. A compiler memory barrier guarantees that the flipping is the last write operation of the transaction. After that, the cache-line is flushed into NVRAM using a `clwb` instruction followed by an `sfence`.

It is important to note that this mechanism is correct and very efficient. Correctness is achieved by the ordering guarantees for stores to the same cache-line. For example, if a transactional data structure is evicted from the cache during a transaction, the state becomes persistent, but the indicator shows that the old version is the valid one. In case of a shutdown or crash in this state, the next transaction will start by copying the valid version into the other half, which is effectively a rollback. Efficiency is achieved by avoiding complex log data structures and recovery after restarts.

The bounded buffer presented in Figure 1 is simple enough to fit into half a cache-line. Within an imaginary operating system with persistent server processes, the buffer could be used as an input queue for messages. In case of an unexpected power loss, the persistent queue would always remain in a consistent state.

4 Implementation using AspectC++

With the help of the AspectC++ language, a generic implementation of cache-line transactions can be applied transparently to any sufficiently small data structure. As shown in Figure 2, the user only needs to annotate a data structure with the attribute `[[NVM::transactional]] alignas(64)`. The actual functionality of cache-line transactions is implemented by a separate *aspect*, which gets applied to the annotated data structures by the AspectC++ compiler.

4.1 Fitting the Data Layout

Cache-line transactions as outlined in Section 3 exploit a tailored layout of the data structure. Thus, the generic implementation using AspectC++ shall adapt the layout of any annotated data structure to fit the cache-line transaction scheme. In particular, the data structures need to be extended by a copy of the original data members, aligned at the second half of one cache-line.

Figure 4 shows the simplified source code. The keyword `aspect` in the first line declares a module similar to a C++ class that contains a piece of `advice` in line 2. It introduces additional data members into any data structure that is annotated with the attribute `[[NVM::transactional]]`. The remaining lines of code refer to the introduced members.

First, the type definition in line 3 declares the type `Copy` that clones each existing data member of an annotated class. To this end, AspectC++ provides the keyword `JoinPoint` as an interface to its compile-time introspection API. We use the template metaprogram `MemberIterator<>` from the *JoinPoint Template Library* (`JPTL`) [4] to generate the type `Copy` based on information on the number and types of existing data members prior to the piece of advice.

The resulting type is introduced as an additional data member in line 5, wrapped by the tailored cache-line alignment in lines 4 and 6. Finally, line 7 introduces the `_index` byte to select the valid half of the cache-line.

The aspect in Figure 4 introduces four member functions into the annotated data structures. These functions include procedures to commit a cache-line transaction (lines 9–14), start a new transaction log (line 15), and access the last valid version (line 17) and the active (line 19) half of a cache-line.

4.2 Transactions at Runtime

Once the layout of the annotated data structures is adapted, the cache-line transactions can be carried out at runtime. We assume that a proper object-oriented design is implemented, so that an object is in a valid state after the execution of a public method, and that non-const, i.e. potentially modifying, methods have to be made transactional.

Figure 5 shows the source code of another aspect that implements the transactional behavior. That aspect contains three `pointcut` definitions in lines 2–4. Such pointcuts are expressions that refer to entities of a C/C++ program, such as member functions, data members, and annotations. For example, the pointcut expression in line 2 refers to all data members of classes annotated with the attribute `[[NVM::transactional]]`, but excludes data members named `"_index"` and all static data members. The expressions `"%"` and `"..."` are wildcard symbols that match one identifier or a series thereof, respectively. Thus, the pointcut expression `tx_method()` in line 3 describes the member functions `commit()` and `log()`, and the pointcut expression `transaction()` in line 4 refers to all member functions of annotated classes except the aforementioned two functions and those functions declared as `const`.

Based on these reusable pointcut definitions, the aspect specifies four pieces of advice. First, the advice in line 6 intercepts any function call to a member function of annotated data structures as specified by the pointcut expression `transaction()`. On such a function call, a new cache-line transaction is started by invoking `log()` on the target object, provided by `tjp->target()`, which is part of the AspectC++ join-point API. The intercepted function call is resumed by `tjp->proceed()`. After execution of the member function, the data structure is considered as consistent, so that the cache-line transaction is finally committed in line 9.

The remaining three pieces of advice capture any read and write access to data members of annotated data structures. In short, the `set` advice in line 12f redirects any write access to the active data copy, which remains invalid until the complete transaction is committed. The AspectC++ join-point API provides the necessary pointers for the access redirection, that is, `tjp->entity()` for the accessed data member and `tjp->arg<0>()` for the new value to be written.

Likewise, the `get` advice in line 15f redirects any read access to the active data copy if the access occurs within a transaction, specified literally by `within(transaction())`. The other way around, if a data member is read *not* within a transaction, for example, by a function declared as `const`, the advice in line 18f returns the last valid version.

In summary, the aspects in Figure 4 and Figure 5 implement the cache-line transaction scheme in a generic and transparent way. In other words, both aspect modules can be applied automatically to various data structures. The AspectC++ programming language enables reuse of the shown implementation, so that the user only needs to add the one-line annotation `[[NVM::transactional]]` to augment any data structure with cache-line transactions.

5 Evaluation

To evaluate our transaction mechanism, we implemented a bounded buffer as shown in Figure 2 with a capacity of 29 bytes, so that it fits onto a single cache-line of current x86 processors when we apply the cache-line transaction (CLTX) data layout. We compare CLTX to a buffer without transactions and to a buffer using the transaction mechanism with undo log provided by the PMDK like shown in Figure 1.

5.1 System setup

All measurements were performed on a Dell PowerEdge R740 system equipped with two Intel Xeon Gold 5218 CPUs running at 2.3 GHz, 12x 32 GB DDR4-DIMMs with 2666 MT/s,

```
aspect CacheLineTransactionsIntroduction {
  advice NVM::transactional() : slice class { // introduce new members into the target class
    typedef JPTL::MemberIterator<JoinPoint, MemberCopy>::EXEC::Copy Copy;
    unsigned char _padding1[32 - sizeof(Copy)];
    Copy _copy; // allocates a copy of each data member of the target class
    unsigned char _padding2[32 - sizeof(Copy) - sizeof(unsigned char)];
    unsigned char _index = 0; // indicates which part is active and which the last version
  public:
    void commit() {
      cltx_barrier(); // use a compiler memory barrier to prevent instruction reordering
      _index ^= 32; // invert the index bit to swap active data and valid version
      cltx_flush(this); // explicitly flush the associated cache line
      cltx_sfence(); // execute an SFENCE instruction to make sure
    }                // the flush is done before proceeding
    void log() { memcpy(getActive(this), getValid(this), sizeof(Copy));}

    template <class T> T *getValid(T *member) const { return (T*)(((char*)member) + _index);}

    template <class T> T *getActive(T *member) const { return (T*)(((char*)member) + (_index ^ 32));}
};};
```

Figure 4. Generic introduction of data members implemented in the AspectC++ programming language.

```
aspect CacheLineTransactionsRuntime {
  pointcut tx_member() = NVM::transactional() && !"% ...::_index" && !"static % ...::%";
  pointcut tx_method() = "void ...::commit()" || "void ...::log(...)";
  pointcut transaction() = NVM::transactional() && !tx_method() && !"% ...::%(...) const";

  advice call(transaction()) && !within(transaction()) : around() {
    tjp->target()->log(); // start a new transaction log on the target object
    tjp->proceed(); // continue the execution of the transactional member function
    tjp->target()->commit(); // commit the transaction log and flush the cache line
  }

  advice set(tx_member()) : around() {                       // capture write access and
    *tjp->target()->getActive(tjp->entity()) = *tjp->arg<0>();} // write to active data

  advice get(tx_member()) && within(transaction()) : around() { // capture read access
    *tjp->result() = *tjp->target()->getActive(tjp->entity());} // and read active data

  advice get(tx_member()) && !within(transaction()) : around() { // capture read access
    *tjp->result() = *tjp->target()->getValid(tjp->entity());}   // and read valid version
};
```

Figure 5. Generic advice at runtime for implementing cache-line transactions in the AspectC++ programming language.

and 12x 128 GB Optane DCPMM. The NVRAM was configured in App-Direct mode. On one socket the DIMMs were configured as one interleaved set, on the other socket non-interleaved.

We used Debian Bullseye with kernel 5.2.9-2 as operating system, PMDK version 1.6.1 compiled from source code, and libpmemobj++ version 1.7. As compiler we used GCC 8.3.0 with flags `-O3 -DNDEBUG`. For the benchmarks, we used one namespace on the interleaved region and one on a non-interleaved region, each configured in fsdax mode, formatted with EXT4, and mounted with `-o dax`.

5.2 Benchmarks

We examined three scenarios using the bounded buffer:

1. filling the buffer byte-wise with one transaction per `addByte` operation,
2. filling the buffer byte-wise and then draining the buffer byte-wise with one transaction per `add`/`getByte`,
3. filling the whole buffer from an array in one transaction and then draining the whole buffer into another array in one transaction.

The scenarios were implemented using the Google Benchmark framework [11] that runs each scenario multiple times

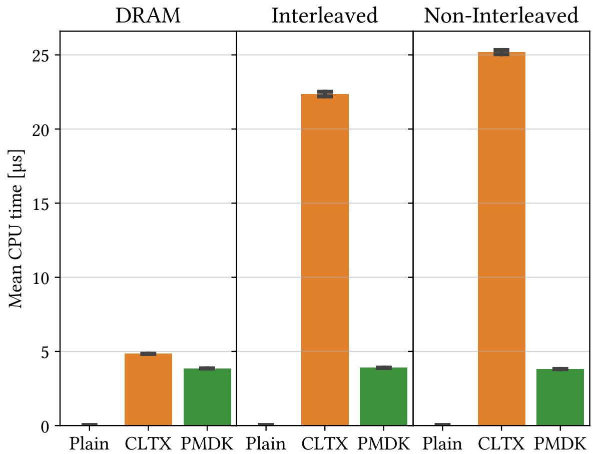

Figure 6. Scenario 1: Add 29 bytes to the buffer in 29 single transactions.

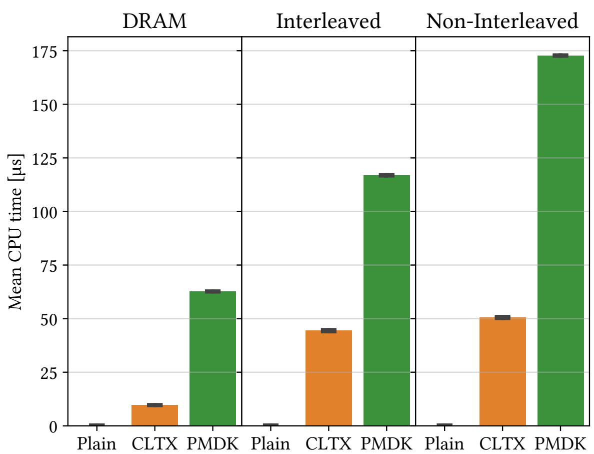

Figure 7. Scenario 2: Add 29 bytes to the buffer, then get 29 bytes in 58 transactions total.

to ensure stable measurement results. Each benchmark has been repeated 100 times for the three buffer implementations in single-threaded mode on DRAM, interleaved NVRAM, and non-interleaved NVRAM. The memory pools for all benchmarks were managed using the PMDK.

5.3 Results

The benchmark results are presented in figures 6, 7, and 8. Compared to the plain bounded buffer, the transactional implementations add a significant overhead to the operations. Without transactions, the type of memory has no influence on the measured CPU time.

In the first scenario, the PMDK implementation performs slightly better on DRAM than cache-line transactions and takes about the same time on all memory configurations. Compared to DRAM, the cache-line transactions perform 4.6 times slower on interleaved NVRAM and 5.2 times slower on non-interleaved NVRAM.

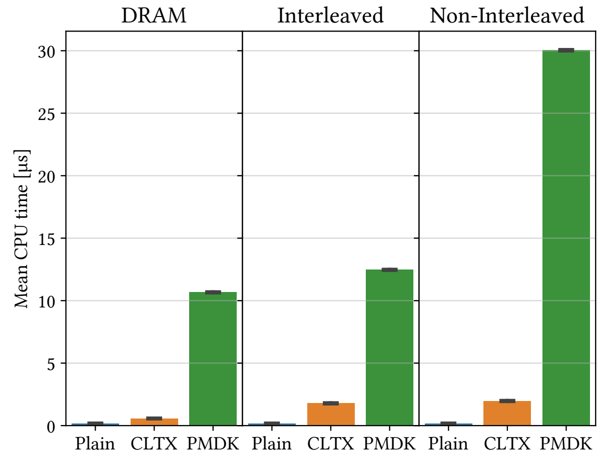

Figure 8. Scenario 3: Fill the buffer in one transaction, then drain the buffer in another transaction.

In the second scenario, the PMDK implementation is 2.6 times slower than cache-line transactions on interleaved NVRAM and 3.4 times slower on non-interleaved NVRAM. For cache-line transactions, we observe the same slowdown as in the first scenario on NVRAM compared to DRAM.

In the third scenario, cache-line transactions outperform the PMDK by a factor of 7 on interleaved NVRAM and by a factor of 15.2 on non-interleaved NVRAM.

In the first scenario, the PMDK implementation seems to benefit from optimizations that we could not clearly identify. In scenarios 2 and 3 it is notably slower on non-interleaved NVRAM than on interleaved NVRAM, while cache-line transaction are only slightly slowed down. The run-time for cache-line transactions is linear in the transaction count, but unaffected by the number of writes in one transaction.

6 Conclusion

The lightweight transaction mechanism presented in this paper might come in handy in special use cases within system software where efficiency is most important and data structure complexity is limited. Depending on the scenario, our evaluation has proven a superior performance compared to state-of-the-art transactions as implemented by Intel's PMDK. We are working on an extension of the mechanism that supports data structures spanning multiple cache-lines.

Using this mechanism without the presented generic aspect would have been extremely cumbersome and error prone. We believe that this is an excellent example showing that a general purpose AOP language such as AspectC++ is sufficient to provide a convenient and safe API to programmers without the need for special purpose language/compiler extensions.

References

[1] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. 2011. Operating System Implications of Fast, Cheap, Non-volatile Memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS'13)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=1991596.1991599

[2] Katelin A. Bailey, Peter Hornyack, Luis Ceze, Steven D. Gribble, and Henry M. Levy. 2013. Exploring Storage Class Memory with Key Value Stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW '13)*. ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/2527792.2527799

[3] Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence Programming Models for Non-volatile Memory. *SIGPLAN Not.* 51, 11 (June 2016), 55–67. https://doi.org/10.1145/3241624.2926704

[4] Christoph Borchert. 2017. *Aspect-Oriented Technology for Dependable Operating Systems.* Dissertation. Technische Universität Dortmund. https://doi.org/10.17877/DE290R-17995

[5] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. *SIGPLAN Not.* 49, 10 (Oct. 2014), 433–452. https://doi.org/10.1145/2714064.2660224

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

[7] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in Non-volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797. https://doi.org/10.14778/2752939.2752947

[8] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. *SIGPLAN Not.* 46, 3 (March 2011), 105–118. https://doi.org/10.1145/1961296.1950380

[9] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[10] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/2592798.2592814

[11] Google. 2019. Google Benchmark. https://github.com/google/benchmark Accessed: 2019-08-09.

[12] Intel. 2019. Persistent Memory Development Kit. https://github.com/pmem/pmdk Accessed: 2019-08-09.

[13] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. 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)*. ACM, New York, NY, USA, 399–411. https://doi.org/10.1145/2872362.2872381

[14] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 257–270. https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon

[15] Shuo Li, Peng Wang, Nong Xiao, Guangyu Sun, and Fang Liu. 2017. SPMS: Strand based persistent memory system. *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017* (2017), 622–625.

[16] Y. Lu, J. Shu, L. Sun, and O. Mutlu. 2014. Loose-Ordering Consistency for persistent memory. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. 216–223. https://doi.org/10.1109/ICCD.2014.6974684

[17] Y. Lu, J. Shu, L. Sun, and O. Mutlu. 2017. Improving Performance and Endurance of Persistent Memory with Loose-Ordering Consistency. *IEEE Transactions on Parallel and Distributed Systems* PP, 99 (2017), 1–1. https://doi.org/10.1109/TPDS.2017.2701364

[18] Jagan Singh Meena, Simon Min Sze, Umesh Chand, and Tseung-Yuen Tseng. 2014. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters* 9, 1 (25 Sep 2014), 526. https://doi.org/10.1186/1556-276X-9-526

[19] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system Persistence. *SIGPLAN Not.* 47, 4 (March 2012), 401–410. https://doi.org/10.1145/2248487.2151018

[20] Faisal Nawab, Dhruva R Chakrabarti, Terence Kelly, and Charles B Morrey III. 2015. Procrastination Beats Prevention: Timely Sufficient Persistence for Efficient Crash Resilience.. In *Proceedings of the 18th International Conference on Extending Database Technology*. 689–694.

[21] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware (DaMoN '14)*. ACM, New York, NY, USA, Article 8, 7 pages. https://doi.org/10.1145/2619228.2619236

[22] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251

[23] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

[24] Torvald Riegel, Christof Fetzer, and Pascal Felber. 2006. Snapshot Isolation for Software Transactional Memory. In *First ACM SIGPLAN Workshop on Languages, compilers and Hardware Support for Transactional Computing*.

[25] Olaf Spinczyk and Daniel Lohmann. 2007. The Design and Implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software* 20, 7 (Oct. 2007), 636–651. https://doi.org/10.1016/j.knosys.2007.05.004

[26] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1960475.1960480

[27] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. *SIGPLAN Not.* 47, 4 (March 2011), 91–104. https://doi.org/10.1145/2248487.1950379

[28] Xiaojian Wu, Sheng Qiu, and A. L. Narasimha Reddy. 2013. SCMFS: A File System for Storage Class Memory and Its Extensions. *Trans. Storage* 9, 3, Article 7 (Aug. 2013), 23 pages. https://doi.org/10.1145/2501620.2501621

[29] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Berkeley, CA, USA, 167–181. http://dl.acm.org/citation.cfm?id=2750482.2750495