Understanding and Detecting Deep Memory Persistency Bugs in NVM Programs with DeepMC

Benjamin Reidys
University of Illinois, Urbana-Champaign, USA
breidys2@illinois.edu

Jian Huang
University of Illinois, Urbana-Champaign, USA
jianh@illinois.edu

Abstract

To facilitate programming with non-volatile memory (NVM), a set of memory persistency models, such as strict and epoch persistency, have been proposed. Although these models provide high-level guidance for reasoning about the data persistence, implementing them correctly is nontrivial. Our study of the well-developed NVM frameworks and libraries reveals that many of them have deep semantic bugs that are strongly relevant to the model specifications. Furthermore, it is difficult to detect them with existing testing and bug-finding tools.

To further understand these persistency bugs, we conduct a characterization study, and present a taxonomy of these persistency bugs. We find that many persistency bugs are caused by the semantic mismatches between the model specifications and their real implementation in NVM programs. To identify these deep persistency bugs, we build a toolkit named DeepMC with both static and dynamic analysis. DeepMC is driven by a set of rules based on our characterization study and persistency model specifications. Our results show that DeepMC can efficiently pinpoint various persistency bugs in a variety of NVM programming frameworks/libraries, and their example programs, including PMDK and persistent memory file system (PMFS) from Intel, the NVM-Direct library from Oracle, and Mnemosyne framework from academia.

CCS Concepts: • **Hardware** → **Memory and dense storage**; • **Software and its engineering** → *Software maintenance tools.*

Keywords: Memory Persistency, Non-Volatile Memory, Persistency Bugs, Performance Bugs

1 Introduction

New and emerging non-volatile memory (NVM) technologies, such as NVDIMM [45], PCM [30, 51], STT-RAM [23], and 3D XPoint [19], offer promising performance and capacity properties. Unlike DRAM-based systems, applications running on NVM require memory persistency to ensure crash safety [3, 15, 24, 47, 60]. This means that a set of data updates must behave in an atomic, consistent, and durable manner with respect to system failures and crashes. Ensuring memory persistency with commodity out-of-order processors and memory hierarchy, however, is challenging due to unpredictable cache evictions. Programmers have to explicitly call dedicated instructions such as `clflush` and `mfence` to enforce the ordering and durability of persistent operations. This inevitably complicates the NVM programming and introduces both correctness and performance bugs [9, 13, 35–37, 43, 53].

To guide NVM programming, prior work proposed memory persistency models to specify the ordering and durability guarantees of persist operations for ensuring crash consistency [5, 26, 37]. These models include strict persistency, epoch persistency, and strand persistency (see detailed discussion in § 2.2). Persistency models provide programmers a means to reason about the trade-off between the performance and crash-safety of NVM programs. For instance, as specified in the strict persistency model, all persistent stores have to be executed in program order. The strict model enables easy implementation, but it causes low application performance. The epoch model relaxes the persistence order with an epoch granularity for improved performance. However, it increases the complexity of NVM programming, since it requires the developer to specify the epoch boundaries and enforce the persistent ordering between the epochs.

Implementing these persistency models correctly is nontrivial as it requires developers to have a thorough understanding of subtle specifications of each model. Specifically, developers may wish to follow a specific memory persistency model to ease the reasoning for their NVM program. However, there is a semantic gap between the demand from developers (i.e., *the memory persistency model used in the NVM program*) and the real implementation (i.e., *whether the memory persistency model is implemented properly following the model specifications*). We define these bugs caused by the semantic gap between the model specifications and the model implementations as the deep persistency bugs. Our study discloses that, even though these popular NVM programming frameworks have been developed for more than five years, they are still suffering from many deep persistency bugs (see our study

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9204-4/22/04...$15.00
https://doi.org/10.1145/3503221.3508427

Table 1. Summary of detected persistency bugs in popular NVM frameworks. PMDK and NVM-Direct use strict persistency model, PMFS and Mnemosyne use epoch persistency model. DeepMC reports 50 warnings in total, among which we validate 43 persistency bugs. We show the number of validated-bugs/warnings reported by DeepMC.

	Bug Description	**PMDK**	**NVM-Direct**	**PMFS**	**Mnemosyne**
Model Viol.	Multiple writes made durable at once	-	-	1/2	-
	Unflushed write	1/2	1/1	-	1/1
	Missing persist barriers	2/2	2/2	-	-
	Missing persist barriers in nested transactions	-	-	1/1	-
	Mismatch between program semantics and model	6/7	-	-	-
Perf.	Multiple flushes to a persistent object	3/4	1/1	3/3	1/1
	Flush an unmodified object	3/3	2/3	4/5	-
	Persist the same object multiple times in a transaction	3/3	-	-	2/2
	Durable transaction without persistent writes	5/5	1/2	-	-
	Total	23/26	7/9	9/11	4/4

in § 3). As we develop more complicated NVM programs, ensuring their correctness will only become harder, which will inevitably hurt the productivity of NVM programming.

To simplify NVM programming, many frameworks/libraries, such as PMDK [49], are extensively developed. However, they provide only high-level interfaces for developers to specify the order and atomicity of persist operations. They still require the developers themselves to ensure the correctness of their persistency implementations. To make matters worse, the implementations of these frameworks/libraries are also buggy [9, 37]. Therefore, it is desirable to have NVM program analysis tools that can automatically verify whether a desired persistency model is implemented properly.

To facilitate the debugging of NVM programs, a few testing tools have been developed. However, these tools, such as Intel's pmemcheck [18], Persistency Inspector [16], and Yat [28], lack generality, since they mainly implement program checks for Intel's Persistent Memory Development Kit [49] and Persistent Memory File System (PMFS) [11]. They cannot be easily extended to other NVM frameworks. A few tools [9, 13, 35–37, 43] were developed recently to detect crash-consistency bugs and performance bugs in NVM programs. They utilized a variety of bug-detection techniques such as model checking [13], symbolic execution [43], and fuzzing [35] to develop their tools. However, most of them focused on basic programming bugs and fall short of detecting the violations of a specific memory persistency model specified by developers. In other words, none of them can explicitly indicate whether a specific memory persistency model (e.g., strict and epoch persistency model) is implemented properly.

In this paper, we aim not only to check whether the specified memory persistency model is implemented correctly (i.e., correctness bugs), but also to verify whether the persistency model is developed efficiently (i.e., performance bugs). As these bugs are strongly related to each specific persistency model, we define them as deep persistency bugs.

To achieve our goals, we first conduct a characterization study of persistency bugs to understand their causes and consequences. We carefully examined the persistency bugs collected from popular NVM programming frameworks and libraries from both industry and academia. They include PMDK [49], PMFS [11], NVM-Direct [46], and Mnemosyne [58]. These persistency bugs can be categorized into two types: *persistency model violation bugs* that affect crash consistency of NVM programs, and *performance bugs* that do not necessarily impact crash consistency but hurt program performance. We briefly summarize our findings in Table 1.

According to our study, most of these persistency bugs, such as those found in NVM programs implemented with strict and epoch persistency models, can be detected by performing static program analysis. They require only simple specifications about the intended memory persistency model. Unlike existing bug-finding tools that require code instrumentation to track persistent operations in NVM programs [28, 37], we can apply the well-defined memory persistency models to static analysis tools, and check whether the instruction order violates the specified memory persistency model.

For NVM programs that would use more complicated persistency models, such as strand persistency [12], their persistency bugs could be caused by the violations of memory dependence between strands. To detect such bugs, we utilize dynamic analysis to track the dependency of persistent operations, and check them with specifications of strand persistency model. Since NVM programs use special annotations to implement the strand persistency model, we only need to track the persistent operations inside the annotated code regions, which significantly reduces the cost of code instrumentation and checking.

With the insights discussed above, we extract a set of rules for detecting persistency bugs in NVM programs using different persistency models. We build a persistency-model aware checking toolkit with LLVM [57], named DeepMC. As our extracted rules are generic, DeepMC can leverage these rules to identify persistency bugs in various NVM programs. DeepMC is systematic, because it implements a set of checking rules for all the available persistency models. DeepMC is simple, as programmers only need to set a flag in the compiler configuration to indicate the persistency model they intend to implement. Overall, we make the following contributions in this paper.

- We conduct a thorough study on persistency bugs that violate the defined persistency model and present a taxonomy of these deep memory persistency bugs.
- We develop a checking toolkit, named DeepMC, with customized static analysis and dynamic analysis to detect persistency bugs by applying the rules extracted from our study and defined model specifications.
- We evaluate the efficiency of DeepMC with a variety of NVM programs. Experimental results demonstrate that DeepMC can detect new persistency bugs without introducing much performance overhead to NVM programs, and without requiring much effort from developers.

To evaluate the efficiency of DeepMC, we use it to detect persistency bugs in PMDK [49], PMFS [11], NVM-Direct [46], and Mnemosyne [58]. As discussed, these frameworks and libraries use different persistency models. DeepMC reported 50 warnings, among which we validate 43 persistency bugs (Table 1). Particularly, DeepMC identified all the 19 bugs covered in our study (§3) to show its completeness. Additionally, DeepMC reported 24 new bugs which are not covered in our study. Specifically, DeepMC pinpointed 8 new persistency model violation bugs and 16 new performance bugs (Table 8). We manually reproduced and validated all these 24 new bugs which have existed for 5.4 years on average.

2 Background and Motivation

In this section, we briefly present the challenges with NVM programming and the defined memory persistency models.

2.1 NVM and Its Programming Challenges

NVM is emerging as a revolutionary technology for computing systems. Intel recently released its first Optane DC persistent memory [17], showing that NVM has become realizable. A key property of NVM is its non-volatility, which enables it to be used as persistent storage. However, NVM requires memory persistency operations to ensure crash consistency, in case the system crashes and the program fails [25, 42, 48, 59].

Ensuring memory persistency with modern memory hierarchies is challenging, since there are multiple levels of volatile caches between the processor and NVM. With commodity out-of-order processors and cache hierarchy, the order in which stored values are made persistent depends on the order in which they are evicted from the cache hierarchy. To facilitate memory persistency, explicit instructions have been developed. For instance, x86-64 processors have the `clwb` instruction [20] to write back a cache line to NVM, and the store fence instruction, `sfence`, to guarantee that a `clwb` completes.

However, requiring users to deal with memory persistency complicates software development. Programmers need to specify persist ordering and persist atomicity [31, 40, 61, 65]. Using such low-level, architecture-specific primitives to develop software is challenging and error-prone, even with the help of libraries [6, 8, 49, 58]. For instance, PMDK, which was built upon these low-level primitives, still requires programmers to understand their durability and ordering guarantees. As we develop programs with NVM programming frameworks/libraries, persistency bugs can occur in both the implementations of the frameworks/libraries, and the user-space NVM programs that use these frameworks/libraries. Since the bug location (i.e., NVM frameworks/libraries or user-space programs) will not affect our bug-finding approaches, we use NVM programs to represent any program developed with a memory persistency model in this paper.

2.2 Memory Persistency Models

Inspired by memory consistency [41], a recent study [48] defined three memory persistency models: strict, epoch, and strand persistency. With these models, developers can reason about the trade-off between performance and data persistence on a multi-core machine. Since most of NVM programs today use one of these memory persistency models, we focus on ensuring their implementation correctness in this paper.

In strict persistency, all persistent stores have to be executed in the program order. It suffers from low performance, due to the imposed strict order and frequent cacheline flushes. The strict persistency model is easy for developers to implement, which has been adopted in many NVM framework and systems, such as PMDK [50] and NVM-Direct [46].

To improve persistence performance, relaxed memory persistency models can be used. Epoch persistency introduces the concept of an epoch. All persistent stores before an epoch boundary have to be persisted before any store after the boundary. It relaxes the order of persistence within an epoch to improve persist concurrency. Because of its improved performance, epoch persistency has also been adopted in NVM-based software, such as PMFS [11] and Mnemosyne [59].

Table 2. Number of persistency bugs studied in this paper.

Framework/ Library	Model Violation Bugs	Performance Bugs	Total Bugs
PMDK [49]	5	6	11
PMFS [11]	2	3	5
NVM-Direct [46]	2	1	3
Total	9	10	19

Although epoch persistency enables the concurrent persists within each epoch, it may miss the opportunity of exploring the concurrency for persists between epochs. Therefore, strand persistency was proposed. The definition of a strand is similar to that of an epoch. However, it aims to eliminate the false dependencies in epoch persistency. Therefore, when there is no dependence between strands, it can improve the parallelism of persist operations. However, strand persistency increases the difficulty of NVM programming, as it

Table 3. List of persistency bugs studied in this paper. [V]: persistency model violation bugs; [P]: performance bugs. We indicate whether a bug is in the NVM framework/libraries (LIB) or their example programs (EP) in the 4th column.

NVM Library	File	Location (#Line)	File Location	Bug Description
PMDK	btree_map.c	201	EP	[V] Modify tree node without making it durable
	rbtree_map.c	197, 231	EP	[P] Log unmodified fields of a tree node
	rbtree_map.c	379	EP	[V] Modified object not made durable
	pminvaders.c	256, 301	EP	[P] Durable transaction without persistent writes
	pminvaders.c	246, 143	EP	[P] Flush unmodified fields of an object
	obj_pmemlog.c	91	LIB	[V] Multiple epochs writing to different fields of an object
	hash_map.c	120, 264	EP	[V] Multiple epochs writing to different fields of an object
PMFS	journal.c	632	LIB	[P] Flush redundant data when committing
	symlink.c	38	LIB	[V] Missing persistent barrier
	xips.c	207, 262	LIB	[P] Flush the same buffer multiple times
	files.c	232	LIB	[P] Flush unmodified object
NVM-Direct	nvm_region.c	614, 933	LIB	[V] Missing persist barrier between epoch transactions
	nvm_heap.c	1965	LIB	[P] Redundant flushes of persistent object

usually needs developers to define the scope of strands with the knowledge of program semantics. Although strand persistency has not yet been widely used in open-sourced NVM-based systems, we believe such a model or similar ones would be promising for improved performance for NVM programs.

Although these memory persistency models provide high-level guidance for NVM programming, it is still challenging for programmers to develop bug-free NVM programs, due to their intrinsic complexities. For example, many developers understand the high-level principles of each persistency model [53], but their implementation of NVM programs may not exactly follow the specifications, resulting in incorrect program execution or suboptimal performance.

3 A Study of Persistency Bugs

3.1 Study Methodology

We manually investigated 19 persistency bugs in popular NVM frameworks/libraries, including PMDK [49], PMFS [11], and NVM-Direct [46], as shown in Table 2. We focus on them because of their extensive use in academia and industry. Similar to prior studies that may suffer from limitations of sampling [32, 34, 64], we make our best effort to collect open-sourced NVM programs. Since the development of NVM programming is still at an early stage, the number of persistency bugs we collected is limited. However, we believe they are representative and this limitation does not invalidate our study results. We encourage readers to focus on the root causes behind each individual case rather than the numbers.

We list each of the studied bugs in Table 3. We categorize these persistency bugs into two major types: persistency model violation and performance bugs. *Persistency model violation bugs* are specific to the individual persistency model that should be followed for ensuring crash consistency in an NVM program. These bugs are caused by violations of the rules for ensuring durability and ordering of persist operations in different persistency models (see § 2.2). *Performance bugs* could have a negative impact on the performance of NVM programs, due to the unnecessary persistent operations or persistence enforcement.

Note that some patterns obtained from our study could exist in the implementations of two or more persistency models. Some of our study results, such as unflushed writes and extra flushes, overlap with existing studies [9, 13, 37, 43]. However, our study mainly focuses on how a specific memory persistency model is not implemented properly in practice.

3.2 Persistency Model Violation Bugs

According to our study, persistency model violation bugs constitute nearly 47% of the examined persistency bugs. We describe the major causes of these bugs in Table 4, and discuss the real-world examples of these bugs.

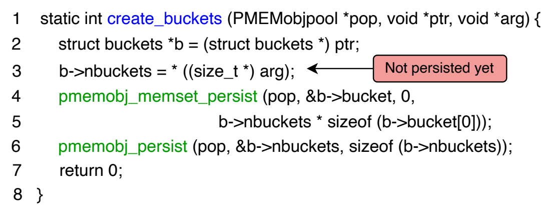

```
1  static int create_buckets (PMEMobjpool *pop, void *ptr, void *arg) {
2      struct buckets *b = (struct buckets *) ptr;
3      b->nbuckets = * ((size_t *) arg);
4      pmemobj_memset_persist (pop, &b->bucket, 0,
5                                  b->nbuckets * sizeof (b->bucket[0]));
6      pmemobj_persist (pop, &b->nbuckets, sizeof (b->nbuckets));
7      return 0;
8  }
```

Figure 1. Semantic gap between NVM program and its implementation in a PMDK example program *hashmap*.

Mismatch between program semantics and real implementation of persistent operations. These bugs are mainly caused by the gap between program semantics and real implementation of persistent operations. Take a *hashmap* implementation with PMDK as an example (see Figure 1). The hashmap buckets are created, initialized and persisted through PMDK's variant of memcpy named `pmemobj_memset_persist` in line 4. The number of buckets (`nbuckets`) is initialized in Line 3 but is not persisted until Line 6 with `pmemobj_persist`, after all the buckets are initialized. In this case, if a crash happens after Line 5, the write to the variable

Table 4. Summary of crash-consistency bugs in different memory persistency model.

Model	Persistency Model Violation	Checking Rules
Strict	Unflushed/unlogged write	An operation W writing to addr A_1, should be followed by a flush F at addr A_2, where $A_1 = A_2$.
	Multiple writes made durable at once	A persist barrier P should be preceded by only one write W.
Epoch	Missing persist barriers between epochs	For any consecutive disjoint epochs E_1 and E_2, there should be a persist barrier P at the end E_1.
	Missing persist barriers in nested transactions	For any epoch E_1 inside of epoch E_2, there should be a persist barrier P at the end E_1.
	Unflushed/unlogged write	A W writing to addr A_1, should be followed by a flush F at addr A_2, where $A_1 \cap A_2 = A_1$.
	Mismatch between program semantics and real implementation of persistent operations	For any consecutive epochs E_1 and E_2 writing to addresses A_1 and A_2 respectively, where $A_1 \in O_1$ and $A_2 \in O_2$, then $O_1 \neq O_2$.
Strand	Having data dependencies between strands	For any concurrent strands S_1 and S_2, operating on addrs A_1 and A_2 respectively, $A_1 \cap A_2 = \emptyset$.

nbuckets in Line 3 is not persisted. This will cause data inconsistency. Developers of this NVM program did not exactly follow the specifications of strict persistency.

Similarly, developers may think that they have developed an NVM program following strict persistency. However, they enforce the durability of multiple writes at once. This may execute without causing any program incorrectness, but it does not reflect strict persistency, which could generate unpredictable or misleading performance behaviors.

Similar bugs can happen in NVM programs using epoch or strand persistency. For example, the strands, which will execute concurrently according to the definition of strand persistency, may have data dependencies [12]. To identify such bugs at the software programming level, we need to exploit dynamic analysis to check the data dependencies at runtime.

```
1  static struct tree_map_node *
2  btree_map_create_split_node (struct tree_map_node *node,
3                               struct tree_map_node _item *m) {
4    .........
5    // Modifying item without logging it.
6    node->items[c - 1] = EMPTY_ITEM;
7    .........
8    return 0;
9  } // This function is executed in a transaction.
```

Figure 2. Unflushed/unlogged write in a transaction of a PMDK example program *btree_map*.

Unflushed/unlogged writes. Writes to NVM are expected to be durable through cacheline flush operations. To simplify the NVM programming, existing NVM frameworks and libraries usually provide a high-level transactional interface to programmers for persisting data structures in NVM. Take PMDK for example, programmers can use a function called TX_ADD() to create a copy of persistent objects (i.e., the element or memory region allocated from NVM) before updating them. Then, we can roll back to their old versions upon system crashes or program failures. This function ensures data durability of persistent writes by placing cacheline flush operations at the end of the transaction. Failing to flush these writes can cause data inconsistency or data loss.

We demonstrate an example in Figure 2. The btree_map_create_split_node function in btree_map.c from PMDK is invoked from a transactional interface. It splits a given B-tree node in NVM and all changes to the tree should be persisted at the end of the transaction. As shown in Figure 2, the items field of a given node is modified in Line 6. However, the node is not logged into the transaction with TX_ADD(), therefore, the update in Line 6 is not guaranteed to be durable. Although existing NVM frameworks usually provide APIs for developers to specify persistent data structures and their operations, they rely on programmers to ensure the correctness of the implemented persistency model. With the knowledge of persistent data structures specified by programmers, we can pinpoint these bugs with program analysis techniques.

```
1   nvm_desc nvm_create_region (nvm_desc desc, const char* pathname,
2   const char *regionname, void *attach, size_t vspace, size_t pspace, mode_t mode) {
3     .........
4     nvm_flush (region, sizeof (*region));
5     // Missing persist barrier.
6     nvm_app_data *ad = nvm_get_app_data ();
7     nvm_txbegin (desc);
8     .........
9     nvm_txend ();
10    return desc;
11  }
```

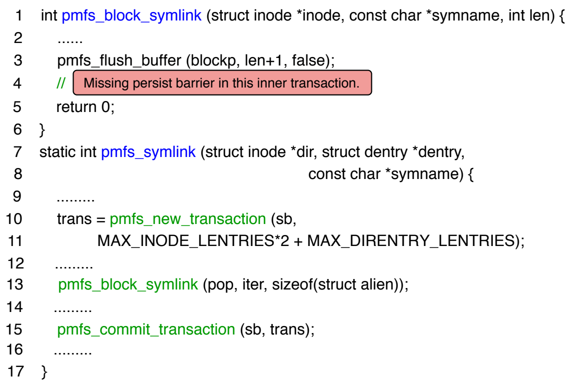
Figure 3. Missing persist barrier after the cacheline flush in the NVM-Direct framework.

Missing persist barrier. Persist barriers provide the ordering guarantees for persist operations to NVM. They ensure that all previous flushes have completed before the next persistent operation is issued. The barrier is necessary, since all persistent operations in one transaction should precede the memory operations in the next transaction. Without it, writes from one transaction could be interleaved with another, which violates the ACID (atomicity, consistency, isolation, and durability) guarantees of transactions. We show an example of a missing persist barrier between transactions in nvm_region.c from NVM-Direct that uses strict persistency in Figure 3. After a region in NVM was created, initialized, and flushed with nvm_flush (Line 4), a persist barrier should be present before another transaction begins in Line 7. Due to the missing persist barrier following the cacheline flush in Line 4, the transactions before and after the flush cannot be guaranteed to be executed in the program order.

According to our study, the nested transactions are inevitably used in systems software. In nested transactions, the inner transactions are persisted before the outer transactions. Missing persist barriers at the end of the inner transaction can lead to crash consistency issues. We show an example

Table 5. Performance bugs in NVM programs.

Performance Bugs	Checking Rules
Writing back unmodified data	For operation F flushing addr A_1, there should be a preceding operation W writing to addr A_2 and $A_1 = A_2$.
Redundant write-backs of modified data	For any two operations F_1 and F_2 in a transaction flushing addresses A_1 and A_2 respectively, $A_1 \cap A_2 = \emptyset$.
Durable transaction without persistent writes	Every durable transaction should contain at least one persistent write to NVM.



```
1   int pmfs_block_symlink (struct inode *inode, const char *symname, int len) {
2     ......
3     pmfs_flush_buffer (blockp, len+1, false);
4     // Missing persist barrier in this inner transaction.
5     return 0;
6   }
7   static int pmfs_symlink (struct inode *dir, struct dentry *dentry,
8                                          const char *symname) {
9     .........
10    trans = pmfs_new_transaction (sb,
11          MAX_INODE_LENTRIES*2 + MAX_DIRENTRY_LENTRIES);
12    .........
13    pmfs_block_symlink (pop, iter, sizeof(struct alien));
14    .........
15    pmfs_commit_transaction (sb, trans);
16    .........
17  }
```

Figure 4. Missing persist barrier in nested transactions in the PMFS library.

in Figure 4. Persist operations in `pmfs_block_symlink` (in `symlink.c`) constitute an inner transaction invoked from an outer transaction in `pmfs_symlink` (in `namei.c`). The writes to NVM in `pmfs_block_symlink` must be persisted before it returns back to the outer transaction (Line 13).

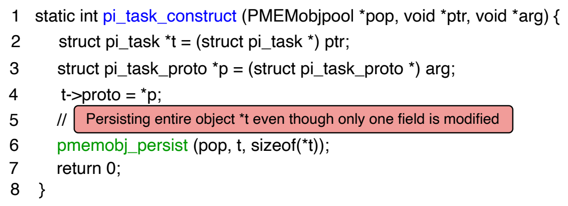
A missing persist barrier could happen in any persistency model, however, the location of enforcing persist barriers is different. For instance, the strict persistency model requires a persist barrier after each persist, and the epoch persistency model requires persist barriers at epoch boundaries.

Checking rules for persistency model violation bugs. We analyze the model violation bugs in our study and formalize them into checking rules summarized in Table 4. Our study assists in generating the types of model violations, while the model specification dictates the associated checking rules. For example, strict persistency requires checking for unflushed writes between consecutive persistent writes, while epoch persistency relaxes the corresponding checking rules.

3.3 Performance Bugs in NVM Programs

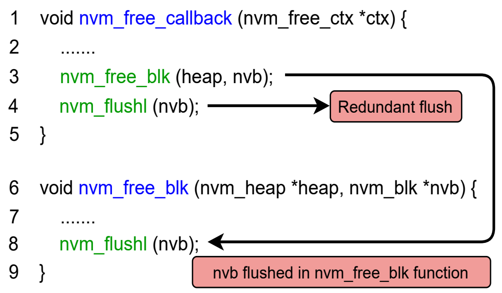
According to our study, performance bugs occupy 53% of all the bugs we studied. We present them in Table 5.

Writing back unmodified data. Unnecessary cacheline flushes do not violate crash consistency, but could negatively impact the program performance and increase the write traffic to NVM. As shown in Figure 5, only one field of the object of type `struct pi_task_proto` is modified (Line 4), but the entire object is persisted (Line 6) using `pmemobj_persist` interface of PMDK. We can detect this type of bug by checking that only the corresponding addresses of modified data are flushed.



```
1  static int pi_task_construct (PMEMobjpool *pop, void *ptr, void *arg) {
2    struct pi_task *t = (struct pi_task *) ptr;
3    struct pi_task_proto *p = (struct pi_task_proto *) arg;
4    t->proto = *p;
5    // Persisting entire object *t even though only one field is modified
6    pmemobj_persist (pop, t, sizeof(*t));
7    return 0;
8  }
```

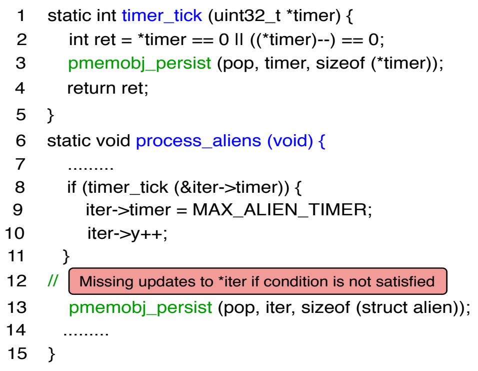
Figure 5. Flushing unmodified data in the PMDK library.



```
1  void nvm_free_callback (nvm_free_ctx *ctx) {
2    .......
3    nvm_free_blk (heap, nvb);
4    nvm_flushl (nvb);
5  }

6  void nvm_free_blk (nvm_heap *heap, nvm_blk *nvb) {
7    .......
8    nvm_flushl (nvb);
9  }
```

Figure 6. Redundant cacheline flushes in the NVM-Direct framework.



```
1   static int timer_tick (uint32_t *timer) {
2     int ret = *timer == 0 || ((*timer)--) == 0;
3     pmemobj_persist (pop, timer, sizeof (*timer));
4     return ret;
5   }
6   static void process_aliens (void) {
7     .........
8     if (timer_tick (&iter->timer)) {
9       iter->timer = MAX_ALIEN_TIMER;
10      iter->y++;
11    }
12    // Missing updates to *iter if condition is not satisfied
13    pmemobj_persist (pop, iter, sizeof (struct alien));
14    .........
15  }
```

Figure 7. A durable transaction without persistent writes in the PMDK example program *pminvaders*.

Redundant writebacks of modified data. Once an NVM object is modified, it needs to be flushed only once for durability. An example is shown in Figure 6. The function `nvm_free_callback` invokes `nvm_free_blk` which flushes an object with `nvm_flush1`, but the same object will be flushed again. An additional writeback can introduce extra latency by 2–4× [11, 21] as well as increase the write traffic to NVM. To detect this type of bug, we can statically examine the redundant write-back operations against the persistent objects that have been written back to NVM.

Durable transaction without updates. As for transactions used in NVM programs, if they do not have persistent

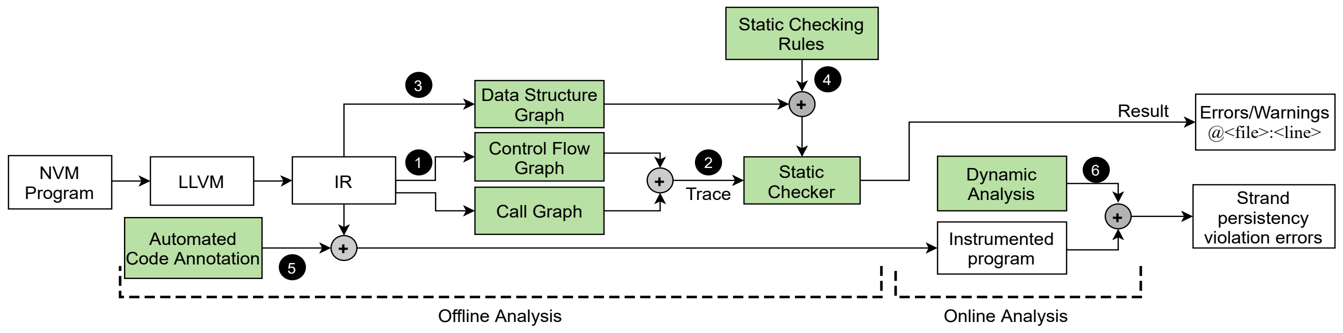


Figure 8. System overview of DeepMC.

writes, the persistent operations that provide durability and ordering guarantees at the end of the transactions become unnecessary. Figure 7 shows an example of such a performance bug in `pminvaders.c` using PMDK. In this example, function `timer_tick` persists a timer at line 3, and an object pointed by `iter` is persisted at line 13 with `pmemobj_persist`. If the condition at line 8 is not true, the object `iter` will not be updated, and it is unnecessarily persisted. This transaction (line 13) is not necessary because no corresponding data is modified. We can pinpoint this type of performance bug by statically checking the presence of persistent writes to NVM, and analyzing their control flow graphs across transactions.

All these types of performance bugs could exist in the implementations of any persistency models. They will make the performance behavior of NVM programs unpredictable. For example, a NVM program with redundant writebacks in its implementation of epoch persistency model can perform worse than the strict persistency. Fortunately, according to our characterization study of persistency bugs, we find that most of them can be identified with program analysis techniques by following the defined persistency model specifications.

Checking rules for performance bugs. Our study helps DeepMC identify the common types of performance bugs. We summarize their corresponding checking rules in Table 5. Since these bugs manifest across persistency models, we do not use separate performance checking rules for different memory persistency models. By checking with persistency model specifications, these performance bugs will not cause program incorrectness, but they introduce unnecessary persistent operations, which affects program performance.

4 Design and Implementation

Based on our study in § 3, we develop a toolkit, DeepMC, which uses both static and dynamic analysis techniques to pinpoint deep persistency bugs. DeepMC only requires its users (i.e., program developers and testers) to specify the memory persistency model they intend to use in an NVM program, using a compile-time flag. DeepMC uses the persistency model specifications as the checking rules (Table 4 and Table 5). We believe DeepMC is general, its approach would still work by extending the checking rules based on the model specifications, even though a new persistency model would be created.

4.1 System Overview

To accurately detect deep persistency bugs, DeepMC uses both offline (static) and online (dynamic) analysis. We show the system workflow of DeepMC in Figure 8.

During the offline analysis, we use the LLVM intermediate representation (IR) of an NVM program to generate the Control Flow Graphs (CFGs) and Call Graphs (CGs) (step ❶). We use CFGs to obtain the traces within each function of the NVM program and utilize the CG to collect traces across the functions. The collected traces are ingested into a static checker for further analysis (step ❷, see details in § 4.3). In step ❸, a data structure graph (DSG) is extracted from LLVM to perform the necessary alias analysis. In step ❹, the static checker applies the static checking rules as described in § 3 to check the collected traces and identify violations of the memory persistency models (in particular the strict and epoch persistency models, and performance bugs). The static checker will report the identified persistency bugs at compilation stage.

In order to perform online analysis, DeepMC uses an instrumenter to insert function calls into the IR, such that it can invoke a runtime library during the execution of the instrumented program (step ❺). Note that NVM programs usually use pre-defined annotations to specify the epoch or strand boundaries, therefore, the instrumentation can be easily fulfilled by tracking the annotated code regions. DeepMC detects violations against the specifications of epoch and strand persistency by using the instrumented program to invoke the runtime library. It maintains the metadata for different epochs or strands that executed concurrently, and performs necessary runtime checks (step ❻), such as whether there is data dependence between these epochs or strands. DeepMC will report `WARNING` for both persistency model violations and performance bugs. DeepMC uses an interface to track every function that performs persistent operations for both offline and online analysis. Since the user knows which annotations they are using, this requires very few lines of code. DeepMC uses this information to know when persistent objects are allocated and flushed, and when memory fences occur.

Note that persistency bugs can occur in both the implementation of the NVM framework/library itself, and the programs that use the framework/library. DeepMC's construction allows it to detect persistency bugs in both situations. Programmers can immediately identify the difference from where the bug is reported. We will discuss each critical component of DeepMC in the following sections.

4.2 Data Structure Analysis

Distinguishing and tracking persistent operations against different persistent objects or data structures in the collected trace is crucial for pinpointing persistency bugs. It requires a context-sensitive and field-sensitive abstraction of persistent objects. To achieve this, DeepMC uses Data Structure Analysis (DSA) [29] to track persistent instances of data structures in an NVM program, and generates a DSG (step ❸ in Figure 8) to track the persistent operations against these instances. DSA and the corresponding DSG were originally proposed in [29] as an alias analysis, and have been incorporated into LLVM [38]. The DSA is context-sensitive and field-sensitive, which meets our requirements. Since the original DSA does not account for persistent data structures, we extend it to track objects in persistent memory.

The DSG contains information about each persistent object and the objects it points to. Each DSG node represents a single persistent object. Directed edges are added between two nodes if one points to another. The DSG is field-sensitive, because it tracks points-to information for each field of each object. DSG also has nodes for function calls that contain the arguments and return value. These nodes are used to merge callee and caller information into each function's local graph. We remove nodes representing objects that are not allocated from persistent memory during the DSA procedure.

The DSG is generated by DSA in three phases. In the first phase (*Local Analysis*), DeepMC inspects the IR of an NVM program to generate a local DSG for each function which captures all its memory dependences. When constructing the local DSG, new nodes are created at the calls to malloc-like functions. We track malloc-like functions, because they are where persistent objects are allocated. DeepMC will track external functions only if they have persistent annotations.

In the second phase (*Bottom-Up Analysis*), the call graph of the program is traversed in post-order (i.e., visiting callees before callers) to build a graph that summarizes the effects of calling that function. For each function call identified in the first phase, we update its local DSG nodes for each argument passed to the function by incorporating the knowledge of how the function call will modify the corresponding data structures. For example, when an object is passed to a function which will modify one of the fields of the object, we will update the field in the corresponding node of the local DSG of the function. If a function has a return value, we also update the corresponding node for the return value of the function. The graph built with bottom-up analysis includes the alias and mod/ref information that indicates whether the referred objects have been updated or read. This information is incorporated into the *local DSGs* of caller functions and used by the static checker when applying checking rules.

In the third phase (*Top-Down Analysis*), the arguments for the callees are incorporated into the graph generated in the second phase. At the end of this phase, each local DSG can determine whether each object was allocated from persistent memory and which other objects it points to. Because DSG keeps track of all memory operations in an NVM program, it is also used for memory dependence analysis. This helps DeepMC analyze the memory persistency for NVM programs, as the persistency is determined by the order of memory updates and whether those updates are persisted at the points specified by a persistency model.

DSA is not only context-sensitive but also field-sensitive, which makes it more precise than traditional alias analyses such as Andersens's or Steensgaard's algorithms [1, 29, 56]. This precision enables DeepMC to analyze memory objects at much finer granularity and further avoid false negatives. The final product allows the static checker to verify each function, all persistent memory objects, and their dependencies. Specifically, the static checker can use this information to establish the history of persistent operations and apply the checking rules against them to identify potential persistency bugs.

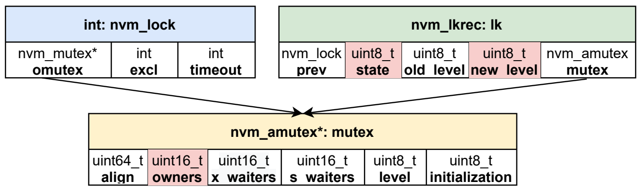
```
1   int nvm_lock (nvm_mutex *omutex, int excl, int timeout) {
2       nvm_amutex *mutex = (nvm_amutex*)omutex;
3       nvm_lkrec *lk = nvm_add_lock_op(tx,td,mutex,st);
          ...
4       lk->state = nvm_lock_acquire_s;
5       nvm_persist1(&lk->state);
6       mutex->owners--;
7       nvm_persist1(&mutex->owners);
8       if (mutex->level > lk->new_level)
9           lk->new_level = mutex->level;
10      lk->state = nvm_lock_held_s;
11      nvm_persist1(&lk->state);
12  }
```

Figure 9. An example from the NVM-Direct framework.

To illustrate the procedure of using DSG to pinpoint persistency bugs, we use an NVM program as an example (see Figure 9), and show its DSG in Figure 10. Figure 9 shows the implementation of the *nvm_lock* function in the NVM-Direct framework. It has a persistency bug of missing the persistent operation for *new_level* (Line 9).

As shown in Figure 10, the DSG consists of three nodes. Two nodes are for persistent objects `mutex` and `lk`, and one node for the initial function call `nvm_lock`. Each object node consists of its fields and the corresponding type information, and each function call node stores the arguments of the function.

During the first phase, the local DSG of the *nvm_lock* function consists of all the function parameters (`omutex`, `excl`, and `timeout`), all the newly created variables (`mutex` and `lk`), and



Persistent Operations

OP	Write	Flush	Fence	Write	Flush	Fence	Write	Write	Flush	Fence
LINE	4	5	5	6	7	7	9	10	11	11
LOC	state	state	state	owners	owners	owners	new_level	state	state	state

Figure 10. The DSG created for the *nvm_lock* function in the NVM-Direct framework.

the unresolved function calls (`nvm_add_lock_op` and `nvm_-persist1`). The first phase also establishes that `omutex` and `mutex` refer to the same object. During the second phase, we resolve the function `nvm_add_lock_op` (see Figure 9), and obtain the knowledge that `lk` is allocated from persistent memory. We also resolve the function `nvm_persist1`, and update the DSG with the information that the corresponding addresses are flushed. But, this does not generate additional nodes, since the function *nvm_persist1* does not have a return value. In the third phase, the caller information indicates that `mutex` is allocated from the persistent memory. At the same time, we can safely remove `excl` and `timeout` from the DSG, because they are not persistent objects. We track the operations against persistent objects (see the bottom of Figure 10). By applying the checking rules as discussed in Table 4, we can identify the persistency bugs in Figure 9.

Since the `if` has no function calls or pointer operations, it does not affect the resulting DSG. The final output shows the relationship between all objects that are stored in persistent memory (see Figure 10) and can assist in developing our checking rules.

4.3 Static Checker with Offline Analysis

To detect persistency bugs in NVM programs, the static checker of DeepMC applies the checking rules (see Table 4 and Table 5) to the collected traces with the CFG and CG. Each trace includes a sequence of instructions that contain persistent writes and ends with a persistent barrier.

Trace collection. We divide the trace collection procedure into two phases. In the first phase, DeepMC traverses the CFG generated in step ❷ of a function using depth-first search algorithm. It scans and tracks the instructions, writes to NVM, and call/return instructions in a set until a persist barrier is encountered. For a write, DeepMC uses the DSG to determine whether the modified object is in NVM. If it is, it will be inserted into the set. To avoid path explosion, DeepMC has priority to explore the paths involving persistent operations, and explores only a small number of paths for loop iterations (10 by default). Similarly, we limit recursion (5 by default). Note that all the traces are collected in the program order.

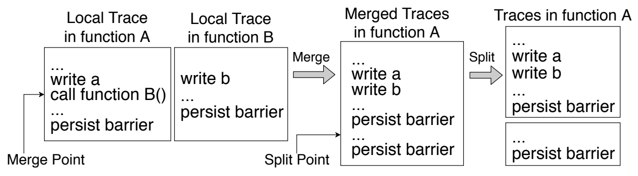

Local Trace in function A
... write a call function B() ... persist barrier
Merge Point
Local Trace in function B
write b ... persist barrier
Merge
Split Point
Merged Traces in function A
... write a write b ... persist barrier ... persist barrier
Split
Traces in function A
... write a write b ... persist barrier
... persist barrier

Figure 11. Interprocedural operations on traces.

In the second phase, DeepMC traverses the CG of the NVM program (generated in step ❶ in post-order). It will merge the traces in callee functions into the traces of the call sites, as shown in Figure 11, such that we will have entire traces of the NVM program and enable the inter-procedure analysis. DeepMC maintains metadata associated with each trace entry. It includes the line numbers of the operations in a trace.

Unlike symbolic execution, DeepMC's trace collection procedure does not track the entire state of persistent memory regions. Instead, DeepMC only tracks the persistent operations based on the DSG. The DSG provides two major benefits. First, by tracking the persistent objects and their memory dependencies, the DSG limits traces to only operations involving persistent memory. Second, the tracking of mod/ref information for persistent objects in the DSG allows DeepMC to prioritize paths with persistent operations.

DeepMC passes the traces to the static checker (step ❷ in Figure 8). After the static analysis, DeepMC will create a detailed report of warnings, which shows the line numbers of the bugs. The developers can use this detailed information to fix them. Automated bug fixing is out of the scope of this work, but we wish to explore it as future work.We will discuss the detailed procedure of using static analysis techniques to identify persistency bugs as follows.

Static checking. After trace collection, the static checker of DeepMC scans through all the collected traces from an NVM program, and applies static checking rules to each trace to detect the potential persistency bugs. We describe how the static checking rules are applied to any given trace as follows.

- **Checking whether a write should be written back or not.** In order to ensure that a given write W in a trace writing to address A_1, our static checker sequentially iterates over the trace in program order to check a cacheline flush F to A_1 by querying the DSG. It also checks whether W occurs before F in the trace.
- **Checking for missing persist barrier.** For NVM programs using the strict persistency model, our static checker will check whether there is a persist barrier after each persistent write. As for NVM programs using epoch or strand persistency models, the static checker will check whether there is a persist barrier between epochs or strands.
- **Checking for unnecessary write-back.** For every operation F flushing address A_1, our static checker will check whether there is a preceding operation W writing to the same address. To achieve this, for any given write-back

operation F, the static checker scans the trace in program order, and uses the DSG to check the address and whether W occurs before F in the trace.

- **Checking for consecutive epochs or strands.** In order to ensure that two consecutive epochs or strands E_1 and E_2 writing to addresses A_1 and A_2, where $A_1 \in O_1$ and $A_2 \in O_2$ (O_1 and O_2 are persistent objects), our static checker will use the DSG to perform field-sensitive memory disambiguation to check whether $O_1 \neq O_2$ (i.e., the epochs or strands should write to different persistent objects).
- **Checking for redundant persistent operations.** To check redundancies for an operation OP_1 of a type T (write or write-back), the static checker traverses a given trace to look for an operation OP_2 of type T and checks whether the two operations operate on overlapped memory addresses. If yes, it means redundant persistent operations happen, which could hurt the performance of NVM programs.
- **Checking for durable transactions.** If no persistent writes are identified in a transaction, it indicates that the transaction is not implemented correctly, or the transaction is unnecessary, which would affect the program performance. DeepMC will report a warning for the potential bugs.

To facilitate our discussion, we use the example given in Figure 9 and 10 to show how we apply the checking rules to pinpoint its persistency bug. As NVM-Direct follows strict persistency model in its implementation, DeepMC applies the corresponding checking rules in the bug-finding procedure. Its static checker uses the DSG to track all persistent operations in a function (as shown in Figure 10). DeepMC stores all persistent writes, flushes, and fences. In this example, fences occur at the Line 5, 7, and 11 (i.e., `nvm_persist1()` flushes the data and issues a fence). When DeepMC encounters a fence, it checks the corresponding flushes and writes. For the first fence (Line 5), we have one logged write and a flush to the same location, which adheres to the checking rules for strict persistency. However, when encountering the fence at Line 11, DeepMC observes writes to `lk->new_level` and `lk->state` with only a single flush of `lk->state`. This indicates an unflushed write. DeepMC will report this bug.

4.4 Dynamic Checker with Online Analysis

For NVM programs using more relaxed models like epoch and strand persistency, data dependencies between epochs or strands (see Table 4) could also cause bugs. To address this, we use dynamic analysis to detect model violations at runtime.

As discussed in §2.2, strand persistency allows different strands (similar to epochs) to execute concurrently, as long as they do not have write after write (WAW) and read after write (RAW) dependencies. Although strand persistency has not been explicitly implemented in many NVM frameworks today, it offers guidance for facilitating the development of highly concurrent NVM programs, such as high-throughput transactional databases and key-value stores [25]. If there is WAW or RAW dependency between strands, they should be placed in the same strand and a barrier is used to enforce the order. To detect data races between concurrent strands, we develop a lightweight dynamic analysis approach by only tracking and analyzing the essential writes to NVM that could potentially cause persistency bugs. The dynamic checker of DeepMC has two major parts: the program instrumenter and dynamic analysis.

Program instrumenter. To facilitate dynamic analysis, DeepMC enables automatic code annotation and injects calls to the IR of NVM programs at the compilation stage (step ❺ in Figure 8). Unlike existing dynamic analysis tools [54] that annotate all memory accesses in a program, DeepMC annotates only the essential memory accesses for persistency. First, DeepMC introduces DSA logic to the instrumenter. By enabling DSA capabilities in the instrumenter, DeepMC can avoid unnecessary instrumentation of objects that do not reside in the NVM. Second, DeepMC only instruments write operations to the NVM in programmer-specified code regions.

NVM programs usually use pre-defined annotations to specify the epoch and strand boundaries. DeepMC does not require its end users to add annotations, instead, it utilizes the annotations that have been used in the NVM programs to identify the boundaries between epochs or strands.

Dynamic analysis. For NVM programs using the strand persistency model or a strand-like persistency model, we rely on dynamic analysis techniques to identify the data dependency. Specifically, an instrumented NVM program will invoke DeepMC's runtime analysis library to detect violations against the strand or strand-like persistency model (step ❻ in Figure 8). Unlike existing analysis tools used for detecting data races in multithreaded programs, DeepMC reduces the performance and storage overhead by only tracking the writes modifying the same or overlapped persistent memory regions. Within these regions, it uses happens-before race detection to identify WAW and RAW data races. DeepMC maps the NVM program's persistent address space to a shadow segment. The shadow segment is responsible for tracking the history of reads and writes issued by a set of strands (or threads) to each persistent memory address. In order to check races between threads for a given segment, the runtime iterates the shadow segment to check whether different threads access the same address. If DeepMC finds such threads, it will generate an error report. Otherwise, the NVM program will continue the execution normally. This allows DeepMC to generate elaborate error reports regarding WAW and RAW races between strands, without introducing much performance overhead to NVM programs.

4.5 DeepMC Implementation

DeepMC has both static analysis and dynamic analysis to pinpoint the persistency bugs in NVM programs. As for the static analysis, we implement it based on LLVM/Clang. DeepMC

Table 6. Benchmarks used in our paper.

Application	Library	Benchmark
Memcached	Mnemosyne	memslap (1M transactions, 4 clients)
Redis	PMDK	redis-benchmarks (1M transactions, 50 clients)
NStore	Low-level implts	YCSB (1M transactions, 4 clients)

uses LLVM/Clang to generate control flow graphs, call graphs, and data structure graphs. These graphs are explored by the static checker for bug detection. DeepMC applies the rules defined in §4.3 and specifications of persistency models [47] to check NVM programs. For the dynamic analysis, we use the data race detection of Google's ThreadSanitizer to detect data races in NVM programs, but customize the tracking of memory accesses in ThreadSanitizer with shadow segments (implemented with 458 lines of code). We also enable ThreadSanitizer to access the DSA/DSG that used for tracking operations against persistent objects within annotated regions. As we run DeepMC, any checking that violates the rules will be reported as warnings.

Note that DeepMC only requires users to specify the implemented model with `-strict`, `-epoch` or `-strand` flag at compilation. which makes it easy to use. DeepMC currently does not support the scenario that part of a program uses one model and other parts of the program use another model.

5 Evaluation

Our evaluation shows: (1) the efficiency of DeepMC in detecting new persistency bugs in different NVM programs using various NVM frameworks and libraries (§ 5.1); (2) its performance overhead introduced by the program analysis (§ 5.2); (3) its completeness in executing the static checking rules (§ 5.3); (4) the reasons for DeepMC to produce false warnings (§ 5.4).

Experimental setup. We apply DeepMC to 16 NVM programs built upon PMDK (102K LoCs), PMFS (8.3K LoCs), NVM-Direct (37K LoCs), and Mnemosyne (11K LoCs). We use the studied NVM programs to evaluate the completeness of DeepMC. We also use real applications to evaluate the performance overhead of DeepMC. These applications include popular key-value stores Memcached [39] and Redis [52], and transactional database NStore [44]. We list their benchmarks in Table 6. We check the source code of these NVM programs to understand their semantics and specify the intended persistency model with a simple flag at their compilation. The experiments are conducted on a real system as shown in Table 7.

5.1 New Persistency Bugs

We use DeepMC to check persistency bugs in 8 NVM programs. It reports 50 warnings. We manually examine all of them and confirm 43 persistency bugs (see Table 1). Among these bugs, we identify 24 new bugs, 18 of them were discovered by the static checker and 6 were discovered dynamically. To further confirm these bugs, we use two approaches: (1) check whether the bug has been fixed in a newer version of the NVM framework/libraries or programs; and (2) report them to the open-source community. For the 24 new bugs we pinpointed, 18 of them are confirmed with these approaches. We summarize all the new persistency bugs in Table 8. These bugs have existed for 5.4 years on average. DeepMC identifies persistency bugs in both NVM frameworks/libraries and their example programs. DeepMC also pinpointed all the 19 bugs covered in our study, which shows its completeness (see §5.3).

Table 7. System configuration.

Processor	Intel Xeon(R), 3,3 GHz, 8 cores, 16 threads, 8MB L3
Memory	16GB main memory
OS	Ubuntu 18.04, Linux kernel 5.0.0-36-generic
Compiler	Clang/Clang++ 7.0.0, O3 optimization

The persistency bugs identified by DeepMC can cause severe consequences. For example, the model-violation bugs can cause program incorrectness, and the performance bugs can slow down the application performance. Specifically, DeepMC finds 8 persistency model violation bugs. For example, in the *hashmap* program using PMDK, programmers expect that the bucket initialization (i.e., initializing the number of buckets and each bucket item) should be fulfilled and persisted atomically. However, they are implemented in separate transactions. DeepMC also identifies 16 performance bugs, many of them are caused by writing back unmodified objects during transactions. For instance, when PMFS fails to recover a superblock, it repairs this issue with a redundant copy, and flushes it for durability. However, PMFS writes back the superblock even though the recovery is successful, resulting in unnecessary write-backs. For these identified performance bugs, we manually fix them and see application performance improvement by up to 43%.

Note that 31% of performance bugs are related to the case of flushing an entire object when only a single field is modified. With the field-sensitive analysis in DSA, we can avoid the false negatives in identifying performance bugs, which demonstrates the necessity of the field-sensitive analysis.

As discussed in §2.2, the strand persistency is similar to the epoch persistency, it is further relaxed by allowing multiple strands to execute concurrently as long as these strands do not have data dependencies. As the strand persistency requires that programmers have the knowledge of the data dependencies of their programs, we realize that strand persistency model is not used in any open-sourced NVM programs. Therefore, we do not report any bugs related to strand persistency in our evaluation.

5.2 Performance Overhead

We evaluate the performance overhead of DeepMC with real-world applications, as described in Table 6.

Offline analysis. For the performance overhead introduced by the offline analysis, we compare DeepMC with the baseline that does not apply any static analysis. We collect the performance numbers when compiling the benchmarks listed

Table 8. New persistency bugs detected by DeepMC in NVM programs using PMDK (Strict), PMFS (Epoch), NVM-Direct (Strict), and Mnemosyne (Epoch). DeepMC identified 8 new persistency model violation bugs, and 16 new performance bugs. We indicate whether a bug is in the NVM frameworks/libraries (LIB) or their example programs (EP) in the 5th column.

Library	File	Line	Bug Description	Location	Consequences	Years
PMDK v1.2	btree_map.c	365, 465	Flushing unmodified fields of tree node	EP	Perf. Overhead	4.4
	rbtree_map.c	259	Flushing unmodified fields of tree node	EP	Perf. Overhead	4.4
	pminvaders.c	249, 266, 351	Durable transaction without persistent writes	EP	Perf. Overhead	4.4
	hashmap_atomic.c	120, 264, 285, 496	Multiple epochs write to different fields of an object	EP	Model Violation	4.4
	obj_pmemlog_simple.c	207, 252	Multiple epochs write to different fields of an object	LIB	Model Violation	4.4
PMFS	super.c	542, 543, 579, 584	Flushing unmodified fields of an object	LIB	Perf. Overhead	3.2
NVM-Direct v0.3	nvm_locks.c	905	Durable transaction without persistent writes	LIB	Perf. Overhead	5.3
	nvm_locks.c	1411	Flushing unmodified fields of an object	LIB	Perf. Overhead	5.3
	nvm_locks.c	932	Missing flush	LIB	Model Violation	5.3
	nvm_heap.c	1675	Flushing unmodified fields of an object	LIB	Perf. Overhead	5.3
Mnemosyne	phlog_base.c	132	Unflushed write	LIB	Model Violation	10.0
	chhash.c	185, 270	Multiple writes to the same object in a transaction	LIB	Perf. Overhead	10.0
	CHash.c	150	Multiple flushes to a persistent object	LIB	Perf. Overhead	10.0

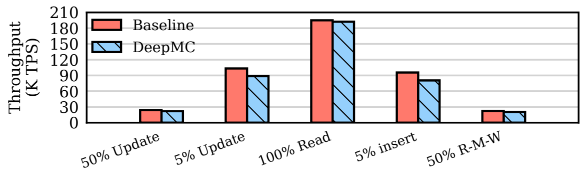
Table 9. Execution time of compiling real applications.

Benchmark	Baseline (secs)	Compilation with DeepMC (secs)
Memcached	8.5	11.9
Redis	54.9	62.4
NStore	31.9	35.6

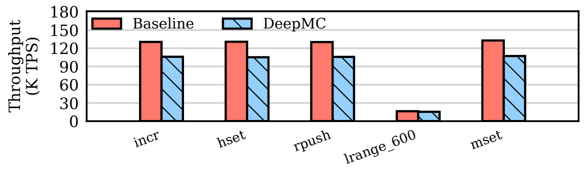
in Table 6 with and without DeepMC. The reported numbers include the entire static checking procedure as shown in Figure 8. We show the compilation time of the real-world applications in Table 9. DeepMC takes 3.4–7.5 seconds more to finish the compilation, compared with the baseline, which is acceptable in practice.

Online analysis. We apply DeepMC's dynamic analysis to real-world key-value stores that include Memcached, Redis, and NStore [39, 52], and present its impact on the runtime performance (transactions per second) in Figure 12. For Memcached, we run different benchmarks, including (1) 50% update, 50% read; (2) 5% update, 95% read; (3) 100% read; (4) 5% insert, 95% read; and (5) 50% read-modify-write, 50% read. For Redis, we run its default benchmarks [2]. For NStore, we run Yahoo Cloud Service Benchmark (YCSB) which includes a variety of Internet benchmarks [7]. Compared with the baseline that does not have dynamic analysis, DeepMC decreases the throughput by 1.7%-14.2% for Memcached, 2.5%-16.1% for Redis, 3.12%-15.7% for NStore, respectively. For different workloads, we observe various performance overheads. This is because these workloads have different persistent write/read ratios, and DeepMC will track the persistent write/read operations for program analysis.

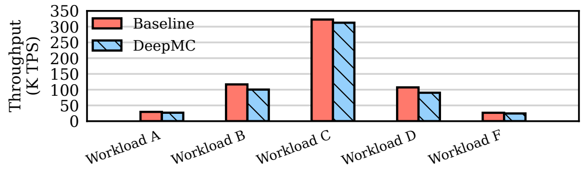
Scalability. As DeepMC only tracks persistent memory regions, it scales with the amount of persistent memory regions instead of total memory. By avoiding tracking all memory regions, we make DeepMC scalable. As we track more persistent memory regions, the checking overhead would increase. However, the overhead of DeepMC is small in general, as we apply DeepMC to different NVM frameworks that have various codebase sizes (see Table 6).



(a) Memcached



(b) Redis



(c) NStore

Figure 12. The performance impact of DeepMC on real-world applications using NVM.

Programmer's effort for use. DeepMC is easy to use. The usefulness of a tool highly depends on how programmer-friendly it is. Take the PMTest [37] for example, it requires programmers to manually annotate the transactions in NVM programs. However, inserting these checkers in programs is error-prone and requires substantial effort from programmers. DeepMC only requires users to specify the persistency model implemented in programs with a single flag (i.e., `-strict`, `-epoch`, or `-strand`) at compilation stage.

5.3 Completeness

To evaluate the completeness of DeepMC, we use the 19 persistency bugs covered in our study. DeepMC can identify all of them (see Table 3), since it runs the checking in a conservative manner. These bugs occurred due to the violations of strict or epoch persistency models, and redundant persistent writeback operations. This demonstrates that DeepMC can identify the potential persistency bugs in a conservative way.

5.4 False Positives

As discussed in § 5.1, DeepMC reports nearly 50 warnings. We carefully validate all the reported bugs by manually examining the source code. We find that 14% of persistency bugs reported by DeepMC are false positives. We investigate them and present the major reasons as follows.

One of the major reasons is the conservative static analysis in DeepMC. DeepMC uses DSA and symbolic analysis for memory disambiguation and memory dependency analysis among instructions in the collected traces. However, due to the lack of dynamic contextual information, DSA and symbolic analysis fail to resolve the memory dependences statically, DeepMC resorts to reporting such cases as bugs.

Another reason is that DeepMC performs the program analysis based on the rules extracted from our characterization study of persistency bugs, and the defined specifications of persistency models, it may not account for certain scenarios where programmers might implement the persistency model in a way according to their own intentions. This requires DeepMC to be aware of program semantics.

To further reduce false positives, we could maintain a database of user-specified rules to filter out some warnings. The database can be updated with the learned experiences of previously validated false positives. Since the current effort required for verifying a reported warning is low, we wish to develop the database idea as future work.

6 Related Work

Persistency model implementation. Prior works have provided various frameworks to allow programmers to manage persistent data with a variety of memory persistency models. User-space libraries, such as NV-heaps [6], NVL-C [8], Mnemosyne [58], PMDK [49], and REWIND [4], provide a transactional interface to support persistence for in-memory data objects. Recent works, such as HOPS [14] and DSO [26], provide high-level ISA primitives for applications to express durability and ordering constraints. Systems software, such as PMFS [11], Strata [27], NOVA [62], NOVA-Fortis [63], SplitFS [22], and ZoFS [10], are filesystem implementations for managing persistent data. Moreover, researchers also extended managed runtime systems to facilitate the NVM programming and persistent object management [33, 55]. However, they do not verify the correctness of their implementations. Our study on these popular NVM frameworks/libraries discloses that many of them suffer from persistency bugs, which can cause data inconsistency and performance issues. Unlike prior studies [9, 43], our study focuses on how these persistency bugs violate each individual persistency model. Specifically, the model-violation bugs identified by DeepMC cannot be detected by existing tools such as AGAMOTTO [43]

NVM program testing and bug detection. Recent studies developed tools to test and debug NVM-based programs. Yat [28] was designed for testing PMFS [11] using an exhaustive testing method, which makes it extremely slow for practical use. Pmemcheck [18] and Persistence Inspector [16] are designed by Intel for testing NVM programs developed with the PMDK library. However, most of them cause huge runtime slowdowns, and require built-in checkers for PMDK operations. They cannot be easily extended for NVM programs not using PMDK. PMTest [37] and XFDetector [36] are efficient tools that can verify crash consistency in NVM programs. However, they require significant effort from developers. Unlike them, DeepMC requires minimal effort from developers or testers. Most recently, a few tools [9, 13, 35, 43] utilized different techniques such as model checking [13] and symbolic execution [43], fuzzing [35] to detect persistency bugs. However, they focused on basic programming bugs, and none of them can detect the implementation violations of a memory persistency model specified by developers. In addition, each of them has limitations. For instance, model checking cannot identify the data dependency at runtime, symbolic execution suffers from state explosion and cannot handle memory aliasing in NVM programs, and fuzzing technique is mainly used to generate test programs for testing NVM programs.

7 Conclusion

We conduct a thorough study on memory persistency bugs with a focus on investigating how they violate the specifications of a specific memory persistency model. Based on this study, we build a checking toolkit named DeepMC, which uses both static and dynamic analysis techniques to detect persistency bugs with minimal cost. Our evaluation shows that DeepMC can detect new persistency bugs.

Acknowledgements

We thank the anonymous reviewers and shepherd for their helpful comments and feedback. This work was partially supported by the NSF grant CCF-2107470.

References

[1] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph.D. Dissertation. University of Cophenhagen.

[2] Redis Benchmark. 2020. Redis Benchmark. https://redis.io/topics/benchmarks.

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

[4] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. 2015. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment* (2015).

[5] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2013. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM.

[6] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. Newport Beach, California, USA.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. Indianapolis, IN.

[8] Denny, Joel E and Lee, Seyong and Vetter, Jeffrey S. 2016. NVL-C: Static analysis techniques for efficient, correct programming of non-volatile main memory systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*.

[9] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. Fast, Flexible and Comprehensive Bug Detection for Persistent Memory Programs. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. Virtual Event.

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

[11] 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*. ACM.

[12] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2020. Relaxed Persist Ordering Using Strand Persistency. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*.

[13] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2021. Jaaru: Efficiently Model Checking Persistent Memory Programs. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. Virtual Event.

[14] Swapnil Haria, Sanketh Nalli, MM Swift, MD Hill, H Volos, and K Keeton. 2017. Hands-off persistence system (HOPS). In *Nonvolatile Memories Workshop*.

[15] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2015. NVRAM-award Logging in Transaction Systems. In *Proceedings of the 41th International Conference on Very Large Data Bases (VLDB'15)*. Kohala Coast, HI.

[16] Intel. [n.d.]. How to detect persistent memory programming errors using Intel Inspector. https://software.intel.com/en-us/articles/detect-persistent-memory-programming-errors-with-intel-inspector-persistence-inspector.

[17] Intel. 2018. Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[18] Intel. 2019. Discover Persistent Memory Programming Errors with Pmemcheck. https://software.intel.com/en-us/articles/discover-persistent-memory-programming-errors-with-pmemcheck.

[19] Intel 2019. Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/optane-for-data-centers.html.

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

[21] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv:1903.05714 [cs.DC]

[22] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. Huntsville, Ontario, Canada.

[23] Takayuki Kawahara. 2011. Scalable Spin-Transfer Torque RAM Technology for Normally-Off Computing. *IEEE Design & Test of Computers* 28, 1 (2011), 52–63.

[24] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-Level Persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. Toronto, ON, Canada.

[25] Aasheesh Kolli, Steven Pelley, Ali G. 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, Atlanta, GA, USA, April 2-6, 2016*. 399–411.

[26] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M Chen, and Thomas F Wenisch. 2016. Delegated persist ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press.

[27] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM.

[28] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A validation framework for persistent memory software. In *2014 USENIX Annual Technical Conference (USENIX ATC'14)*. 433–438.

[29] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. San Diego, California.

[30] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *36th International Symposium on Computer Architecture (ISCA'09)*. Austin, TX.

[31] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. *ACM SIGARCH Computer Architecture News* (2009).

[32] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. Atlanta, GA.

[33] Daixuan Li, Benjamin Reidys, Jinghan Sun, Thomas Shull, Josep Torrellas, and Jian Huang. 2021. UniHeap: Managing Persistent Objects across Managed Runtimes for Non-Volatile Memory. In *Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR'21)*. Haifa, Israel.

[34] Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi S. Gunawi, and Shan Lu. 2019. DFix: Automatically Fixing Timing Bugs in Distributed Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)* (Phoenix, AZ, USA).

[35] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. 2021. PMFuzz: Test Case Generation for Persistent Memory Programs. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. Virtual Event.

[36] Sihang Liu, Korakit Seemakhput, Yizhou Wei, Thomas F. Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-Failure Bug Detection in Persistent Memory Programs. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. Lausanne, Switzerland.

[37] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. Pmtest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. ACM.

[38] LLVM DSA 2021. LLVM DSA. https://llvm.org/docs/AliasAnalysis.html#the-ds-aa-pass.

[39] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/hotstorage17/program/presentation/marathe

[40] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*. ACM.

[41] David Mosberger. 1993. Memory Consistency Models. *ACM SIGOPS Operating Systems Review* (1993).

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

[43] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. AGAMOTTO: How Persistent is your Persistent Memory Application?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*.

[44] nstore [n.d.]. An Efficient Database System Designed for Non-Volatile Memory (NVM). https://github.com/snalli/nstore.

[45] NVDIMM. 2020. Non-Volatile Dual In-line Memory Module (NVDIMM). https://en.wikipedia.org/wiki/NVDIMM.

[46] Oracle. 2015. NVM Direct. https://github.com/oracle/nvm-direct.

[47] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*.

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

[49] pmemio 2017. Persistent Memory Programming. https://pmem.io/.

[50] pmem:PMDK [n.d.]. Persistent Memory Development Kit. http://pmem.io/pmdk/

[51] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*. 24–33.

[52] Redis. 2020. Redis Key-Value Store. https://redis.io/.

[53] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. 2017. Programming for Non-Volatile Main Memory Is Hard. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys'17)*. Mumbai, India.

[54] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*. ACM.

[55] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. AutoPersist: An Easy-to-Use Java NVM Framework Based on Reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. Phoenix, AZ, USA.

[56] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM.

[57] LLVM team. [n.d.]. The LLVM Compiler Infrastructure. https://llvm.org.

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

[59] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[60] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Espresso: Brewing Java For More Non-Volatility with Non-volatile Memory. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*.

[61] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE.

[62] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST'16)*.

[63] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM.

[64] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11)*. Cascais, Portugal.

[65] Yiying Zhang and Steven Swanson. 2015. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE.