NVMOVE: Helping Programmers Move to Byte-Based Persistence

Himanshu Chauhan [*]
UT Austin

Irina Calciu
VMware Research Group

Vijay Chidambaram
UT Austin

Eric Schkufza
VMware Research Group

Onur Mutlu
ETH Zürich

Pratap Subrahmanyam
VMware

Abstract

Programmers can utilize the upcoming non-volatile memory (NVM) technology in various ways. One appealing way is to directly store critical application data structures in NVM instead of serializing them to block-storage. Changing legacy code to achieve this, however, is laborious and prone to bugs. We present NVMOVE, a tool that simplifies this transition by analyzing the source code and automatically identifying *persistent types*, types that are serialized and persisted. Aided by this tool, programmers can modify their applications to allocate such persistent types on the non-volatile memory heap. Upon analyzing Redis, a key-value store with 122 struct types, NVMOVE identifies 25 types as persistent, with *no false negatives* and 11 false positives. We evaluate the benefits of NVMOVE by moving the identified persistent types in Redis onto a non-volatile memory heap. Redis modified in this manner offers full persistence of data, and performs within 78% of Redis with no persistence, achieving more than 2× the performance of Redis that performs logging on SSDs.

1 Introduction

Non-Volatile Memory (NVM) provides byte-addressable and low-latency access, enabled by technologies such as Phase Change Memory [12, 14, 15, 16, 31], Memristors [27] and Spin-Transfer-Torque MRAM [11]. NVM promises to provide fast, cheap, non-volatile storage at near-DRAM latencies, blurring the line between memory and storage [20]. Moreover, NVM is expected to be available commercially soon, raising the question on how to best leverage this technology. Programmers have the choice to use NVM as a drop-in replacement for traditional storage. Currently, applications persist data using a block-based interface to storage. They first serialize the in-memory data and then write the serialized form to storage; we term this *Block-based Persistence* (BLP). BLP suffers from three problems: (a) the performance overhead of serialization, (b) the additional complexity (and resulting bugs) from maintaining both in-memory and in-storage data formats [24, 25], and (c) the performance overhead of system calls that invoke the kernel [2, 20]. Alternatively, a new programming model for persistence can access NVM via loads and stores [20, 29, 30]. Thus, in-memory data structures can be persisted *in-place* without the need for serialization, assuming some mechanism exists for consistent updates [30]. We term this *Byte-based Persistence* (BYP) (§ 2).

Researchers have tackled the problem of writing new applications for NVM using the BYP model [30]. However, few projects address the conversion of *existing* applications from the BLP model to the BYP model. To convert a BLP program[1] programmers first need to identify the data structures that are serialized and persisted. Next, they need to allocate such structures on the non-volatile heap, and make sure the structures are updated in a crash-consistent manner. Performing this conversion manually is laborious and prone to bugs, and may take a long time. For example, the open-source project porting the key-value store Redis to NVM has been active for over a year, and it still only supports strings for keys [1].

As a first step towards tackling this problem, we present NVMOVE, a tool that uses static analysis techniques to automatically analyze the source code of a BLP application, and identify the user-defined types that are persisted to storage. The key observation behind NVMOVE is that any persistent data structure needs to be instantiated and initialized from persistent storage when the application starts or when it recovers from a crash. We make the following contributions:

- We describe our experience in designing and implementing NVMOVE, and discuss the trade-offs between different design approaches (§ 3).
- We evaluate NVMOVE on Redis [19, 33]. When compared against a manual port of Redis to NVM, NVMOVE correctly identifies *all* persistent types, with no false negatives (§ 4.1).
- Using the analysis results from NVMOVE, we emulate transforming Redis into a BYP program with full durability (no data loss on crash), and show that this program outperforms Redis with synchronous logging by up to 2.2× (§ 4.2).

[*]Work performed as an intern at VMware Research Group.

[1]We use the term *BLP/BYP program* for programs written under the BLP/BYP model.

2 Block-based and Byte-based Persistence

Traditionally, programmers use the BLP model to persist data in two steps: 1) serializing in-memory data, 2) using systems calls such as write to persist the serialized form of data. Consider the code example in Figure 1c. It shows a sample program in the BLP model that updates and persists a simple structure: record. Note how only line 2 actually updates the record; lines 4–9 serialize the record into a temporary buffer buf, and persist it using system calls. Figure 1c omits the code required to recover to a consistent state in the event of a crash.

The block-based persistence model is a poor fit for NVM for two reasons. First, the block-based interface is optimized to hide the high latencies of block-storage devices, and has inherent software overheads that can cause large performance issues with NVM [2, 20]. The performance overhead of providing persistence via the block interface causes programmers to trade-off safety for performance by reducing persistence frequency. This strategy is not only vulnerable to possible loss of valuable data, but also causes problems in program recovery[2]. Second, using the block interface forces serialization of in-memory data into a different format. Maintaining two versions of the same data structure leads to additional complexity and bugs [24, 25]. Both of these problems can be avoided by persisting to NVM using the BYP model.

```
1.  typedef struct {
2.      int id;
3.      long value;
4.  } record;
```

(a) BLP struct

```
1.  @persistent typedef struct {
2.      int id;
3.      long value;
4.  } record;
```

(b) BYP struct

```
1.  void update_persist(record *r, long nval){
2.      r->value = nval; //dram update
3.      //persist to block storage
4.      char *buf = malloc(...);
5.      int fd = open(...);
6.      sprintf(buf, "id:%d, val:%l", r->id, r->value);
7.      write(fd,buf,sizeof(buf));
8.      free(buf);
9.      close(fd);
10. }
```

(c) Updating and persisting data in BLP

```
1.  void update_persist(record *r, long nval){
2.      r->value = nval; //persistent update
3.  }
```

(d) Updating and persisting data in BYP

Figure 1: Comparison of the BLP and BYP models of programming for saving updated state.

The code example of Figure 1d shows a BYP program that provides similar functionality as the BLP program in Figure 1c. The BYP program persists the updated state of record directly in NVM. For this program, the tag @persistent in struct declaration (at line 1 in Figure 1b) is needed to indicate that each instance of struct record should be allocated on the non-volatile heap. For any instance of `record`, updating and persisting its state is as simple as directly updating the instance's field in line 2, and flushing the cache (not shown). Note that a separate mechanism, e.g., transactions, is still needed to guarantee crash consistency [26, 30].

3 NVMove

NVMove analyzes a given application, and identifies user-defined types — structs for C programs — that represent semantically persistent state. The types that represent the persistent state may be different from the types that are syntactically persisted. For example, in Figure 1c, we would like to identify record as the persistent type, and not char* (the type of buf). NVMove achieves this goal via static analysis of the source code.

Analyzing Recovery Code. We observe that an application that persists state must restore it during initialization or recovery after a crash. This leads to the main insight of NVMove: *any user-defined type that is created or updated in the recovery/initialization phase must be persisted*. NVMove requires the programmer to provide the name(s) of the top-level source function(s) that initializes the in-memory application state by reading the previously persisted state.[3] Such functions are called *load functions*. This implies that the programmer needs some knowledge of the application, but this knowledge is minimal. Without NVMove, the programmer still needs to identify the load functions when manually porting the code to NVM. A programmer could manually follow this approach to identify semantically persistent types, but the effort is prone to errors and takes significant time. For example, our optimized implementation of NVMove visits 62 functions in Redis, and parses thousands of lines of code.

We restrict our focus to applications that implement the recovery/initialization mechanism in static code and currently do not support applications that initialize their state using function pointers that are dynamically followed.

[2]Just recently, Delta Airlines experienced a major service disruption due to a malfunction that lasted only an hour, but the total recovery took more than 13 hours and affected more than 45% of their flights [28].

[3]For example, the database state in Redis is loaded by the function rdbLoad in source file rdb.c. Using the Redis documentation, one can easily find this function.

NVMove starts the static analysis with an empty set T_s used to store the types that are candidates for persistence, and a FIFO queue F_q that stores the function names to visit. At startup, it prompts the programmer to provide the name(s) of the load function, and inserts it in F_q. It then performs the following steps in a loop until F_q is empty: it scans the source, and finds the definition of the function f at the head of the queue F_q. It then collects all the variables in f that are assigned or modified through assignment operators, or library calls such as memcpy/memmove. For each such variable, it inspects its type t and adds it to T_s if and only if t is defined in the application source. If t was already present in the set T_s, then it does not need to be added again. NVMove then removes f from F_q, and goes to the beginning of the loop. This analysis is guaranteed to terminate as the application source is finite. Upon termination, T_s contains all the types defined in the application program that NVMove identifies as candidates for persistence.

Note that marking all the types that are created/modified during initialization is likely to overestimate the persistent variables and may lead to sub-optimal performance. We want to focus on correctness, and hence compromise on performance. We optimize this approach by: (a) not parsing through application source functions that either accept no arguments, or accept only non-pointer built-in types, (b) using our knowledge of libc API and treating strcpy/memcpy/memmove and similar functions as assignment operations, (c) not visiting bodies of functions defined in libc API (fopen/fclose for example) whose results and side-effects are known to us, (d) analyzing each function body only once.

Back-tracking from Writes. An alternative approach is to statically identify all the system calls that write to the block-device[4] and follow the data flow backwards to the persistent data structures. The steps of this method are as follows. First, system calls can be easily found by parsing the code. Next, the intermediate buffers used to serialize the data can be retrieved from the arguments of the system calls.[5] Using the knowledge of library functions, such as sprintf (and its variants), we can then back-track through the source-code — recursively visiting caller function definitions — to identify the structures that were serialized into such buffers. This approach identifies struct record as persistent in Figure 1c. However, programmers use write calls not only for serializing data to block-devices, but also for logging debug and error messages, writing to pipes, and network sockets. Therefore, this solution produces a large number of false positives for real applications.

NVMove is implemented using Clang [13] and it currently works on C programs. However, Clang supports many C-like languages, and thus our approach could be extended to programs in other languages.

4 Evaluation

We evaluate NVMove on the source code of Redis [19, 33] (version 3.2.0, 64-bit), a widely-used fast data-structure store that persists its data to block storage. The Redis codebase currently has $\approx$ 50K lines of code.

We seek to answer the following questions:

- How effective is NVMove in identifying user-defined types that should be persistent?
- How is Redis performance affected if all the types identified by NVMove are persisted in NVM?

4.1 Type Identification

Running NVMove on Redis source code takes five minutes on average, which suggests that NVMove can easily handle large code bases. Apart from pointing out the load function (rdbLoad), we did not provide NVMove with any other information about the code base.

To see how effective NVMove is at identifying persistent types, we compare the results of NVMove with a manual port of Redis performed by an independent industrial team of developers. They provided us with a list of structs that they treat as persistent in their ported NVM-compatible version of Redis. Table 1 summarizes the comparison of these manually identified types with the results of NVMove.

Total types (structs) in Redis source	122
NVMove identified persistent types	25
True positives (manually identified)	14
False positives	11
False negatives	0

Table 1: Comparing manually identified persistent types to the results of NVMove.

Without any in-built knowledge of Redis source code, NVMove identifies *all* persistent types, and produces *no* false negatives. Among the 11 false positives, four are iterators over persistent types, another four are variants of the same type with different data alignments. We believe that results produced by NVMove can be of significant value to programmers, as they would be able to prune out these false positives upon closer examination.

[4]e.g., write, fwrite, pwrite.

[5]In Figure 1c, buf would be identified as the argument that was persisted.

4.2 Emulated Performance

We evaluate the performance impact of transforming Redis to a BYP program, by emulating non-volatile memory latencies when accessing persistent variables identified by NVMove. The transformed program offers full persistence [6], but does not have any mechanism for crash consistency. For full persistence, every NVM write must be followed by a cache line flush (clwb instruction) as well as a PCOMMIT instruction, as described in Intel's recent ISA extensions [10].

We consider two primary NVM technologies: (1) Phase-Change Memory (PCM) representing slower NVM with a read/write latency of 300ns; and (2) Spin-Transfer-Torque MRAM (STT-RAM) representing fast NVM with read/write latency of 100 ns. We set the PCOMMIT latency to 500 ns for PCM and 200 ns for STT-RAM. The latency of clwb is kept constant at 40 ns. These latencies are per cache-line, and based on recent literature [32].

We simulate these delays by injecting configurable delay functions in Redis source code after each read/write operation of candidate variables. This is done by an automated source-to-source transformation using Clang. Overall, roughly around 4000 delay calls are injected in the source. Our emulation indicates the worst-case performance. We expect that performance would be better using real hardware for the following reasons. First, many reads can be served from the cache, but our emulation assumes that all PCM reads are served from main memory. Second, writes to the same cache line often need to incur only the cost of a single cache line flush, whereas in our evaluation every write incurs a cache line flush. Third, multiple cache lines written together often only need to incur the cost of a single PCOMMIT. Finally, the latencies of non-volatile memory reads and writes can be overlapped better by a memory controller that performs intelligent scheduling [17, 22, 34], which we do not model. Our emulation is conservative in these regards and provides an estimated performance that errs towards the worst-case.

We consider two scenarios for performance comparison as Redis provides two modes of persistence:

Redis Database (RDB): RDB files are point-in-time snapshots of data at specified intervals. Redis forks a background process to perform serialization of data to storage. Redis does not allow sub-second intervals for this mode and thus can lose one second worth of data.

Append Only File (AOF): Redis appends every write command received by the server to the end of a log. The log is played at startup to reconstruct the original data.

[6]On a crash, only the last write in progress could be lost.

We first obtain practical upper bounds on performance by running Redis with both of these modes disabled (In-Memory Redis). We then run Redis in a RDB-only mode that takes a snapshot every second. Next, we disable RDB and run it with AOF that logs every write command and flushes it immediately to block storage (by calling fsync). Finally, we disable both RDB and AOF and run Redis under the two NVM emulation settings listed above. We report the throughput results on a machine with 56 Intel Xeon (2.2GHz) cores, and 500 GB DRAM. In the interest of space, we show baseline results with only SSD block-storage (throughput of AOF with a hard-disk is around one-third of that with SSDs). For each experiment, we report mean values of five runs.

Results on YCSB. Figure 2 compares the throughput of Redis when run under the above configurations on the Yahoo Cloud Serving Benchmark (YCSB) [6]. The three workloads tested are: read-heavy (90% reads, 10% updates), balanced (50% reads, 50% updates), and write-heavy (10% reads, 90% updates). Each run first inserts one million records (of 1 KB each) in an empty database before starting throughput computations. The speedup in the plot is the ratio of throughput for each setting divided by the throughput of our baseline, which uses the AOF mode on SSD, for the same workload. The actual values (in operations/second) of baseline SSD throughputs are: 27946 for read-heavy, 17612 for balanced, and 6605 for write-heavy workload. As expected, Redis without persistence gives the highest throughput. RDB mode performance is close to optimal since checkpointing is triggered once every second at the most. However, RDB mode could lose at least 46 MB (read-heavy), 47 MB (balanced), and 27 MB (write-heavy) of data due to its coarse-granularity approach to persistence. Also, for many runs, the background process saving the snapshot takes more than ten seconds to save the final snapshot after the client disconnects. Hence, in case of a crash/power-outage the size of lost data can be much larger. Persisting types identified by NVMove yields better performance than baseline, while providing strong durability guarantees: a crash may leave only the latest write incomplete in persistent memory. Even under the unfavorable delays we test that approximate the worst case, our NVM-emulated persistent Redis with slow PCM has higher throughput than the baseline, and has up to $2\times$ higher throughput with faster STT-RAM.

Results on Redis-benchmark. We also ran the Redis-benchmark, which ships with the Redis source-code. There are 17 workloads, many of which are read-heavy/read-only. On several write-heavy workloads (such as MSET,LPUSH,RPUSH), our NVM-emulated

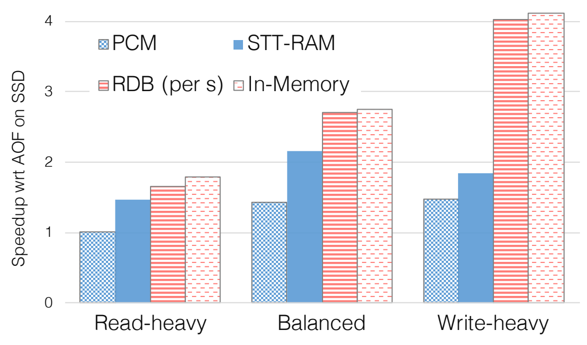

Figure 2: Throughput comparison on YCSB benchmark.

persistent Redis is faster by as much as 45% than the baseline (AOF on SSD). On read-only workloads, such as GET, our Redis is slower by 2–3%, mainly because NVMOVE falsely detects iterators as persistent types, which leads to a minor degradation in read performance.

5 Limitations and Future Work

NVMOVE may over-approximate results because it identifies types, not specific variables. Even if only one variable among many instances needs to be persisted, the entire type is marked as persistent. Another problem is that many types contain volatile state in addition to non-volatile state. For example, many applications keep reference counts of objects for memory management. If an object's reference count is initialized during the load of its persisted state, then our analysis will flag the count variable as a persistent pointer. Despite these limitations, we believe that our tool can be of significant value to programmers in porting a large codebase to byte-based persistence.

Future Work. We plan to explore other approaches to identifying persistent types to decrease the number of false positives. We are also working on precise identification of persistent variables, not just types. In order to do so, we plan to combine our static analysis approach with dynamic/taint analysis. This would allow us to follow the progression of the persistent variables through the execution of the program.

Converting a BLP program into a BYP program involves many changes, such as removing serialization code and replacing allocation calls for types identified as persistent. We also would like to update persistent structures in a consistent manner, using techniques such as write-ahead logging [8] and transactions [7].

6 Related Work

The vision behind NVMOVE is to allow legacy applications to use persistent memory without significant programmer effort. NVMOVE currently only identifies persistent data structures. Updating such structures in a consistent manner without significant programmer involvement is a difficult problem. We aim to address this problem in future work.

Atlas [3] atomically persists all updates between `lock()` and `unlock()` in multi-threaded programs. Atlas targets legacy applications that do *not* durably store state, and transparently provides atomic persistence for such programs. In contrast, NVMOVE targets legacy applications that already have protocols to durably store state to a block-device. NVMOVE is suitable for both single-threaded and multi-threaded applications, and is not limited to applications that use a locking discipline (*e.g.,* those employing lock-free data structures [9]).

ThyNVM [26] transparently checkpoints state of legacy applications onto persistent memory. ThyNVM requires special hardware support. In contrast, NVMOVE does not require new hardware support, and persisting the data structures identified by NVMOVE can be done using a number of other approaches (*e.g.,* [4, 30]) that also do not require specialized hardware.

Existing work that builds new systems and APIs on top of persistent memory [4, 5, 18, 21, 23, 29, 30] is complementary to NVMOVE: once NVMOVE identifies the data structures to be made persistent, existing work can be leveraged to persist them in a crash-consistent manner. All of these systems require significant programmer effort in rewriting the applications using a new API. A future version of NVMOVE could help alleviate this task.

7 Conclusion

The emergence of a new technology leads to a flurry of activity to create new applications that exploit its benefits. It is equally important to consider how *existing* applications and use-cases can benefit from the new technology as seamlessly as possible. Many research/open-source projects enable new applications for NVM, but there is little work to port current applications to NVM. Given the large number of applications written for block storage, easing this transition is vitally important. We believe NVMOVE is the first step towards easing the transition of programs from block storage to byte-addressable non-volatile memory. Our preliminary results show promise, and can already be of significant value to programmers. Continued research and development in this direction will lead to improved solutions that can further reduce the programming effort in adopting NVM.

References

[1] Pmem Redis. `https://github.com/pmem/redis`.

[2] Adrian M Caulfield, Arup De, Joel Coburn, Todor I Mollow, Rajesh K Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *MICRO*, 2010.

[3] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *OOPSLA*, 2014.

[4] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.

[5] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[6] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[7] Jim Gray et al. The transaction concept: Virtues and limitations. In *VLDB*, 1981.

[8] Robert Hagmann. Reimplementing the Cedar file system using logging and group commit. In *SOSP*, 1987.

[9] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[10] Intel. Intel architecture instruction set extensions programming reference. `https://software.intel.com/sites/default/files/managed/69/78/319433-025.pdf`.

[11] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating STT-RAM as an energy-efficient main memory alternative. In *ISPASS*, 2013.

[12] Stefan Lai. Current status of the phase change memory and its future. In *IEDM*, 2003.

[13] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, 2008.

[14] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory As a Scalable DRAM Alternative. In *ISCA*, 2009.

[15] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Phase Change Technology and the Future of Main Memory. *CACM*, 2009.

[16] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase Change Technology and the Future of Main Memory. *IEEE Micro*, 2010.

[17] Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems. In *HPS Technical Report*, 2010.

[18] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. Loose-ordering consistency for persistent memory. In *ICCD*, 2014.

[19] Tiago Macedo and Fred Oliveira. *Redis Cookbook*. O'Reilly Media, Inc., 2011.

[20] Justin Meza, Yixin Luo, Samira Khan, Jishen Zhao, Yuan Xie, and Onur Mutlu. A Case for Efficient Hardware-Software Cooperative Management of Storage and Memory. In *5th Workshop on Energy-Efficient Design (WEED)*, 2013.

[21] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 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*, 2013.

[22] Onur Mutlu and Thomas Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *ISCA*, 2008.

[23] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory Persistency. In *ISCA*, 2014.

[24] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *OSDI*, 2014.

[25] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Crash Consistency. *ACM Queue*, 2015.

[26] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *MICRO*, 2015.

[27] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. The missing memristor found. *Nature*, 2008.

[28] New York Times. Delta malfunction on land keeps a fleet of planes from the sky. `http://www.nytimes.com/2016/08/09/business/delta-air-lines-delays-computer-failure.html`, August 2016.

[29] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, 2011.

[30] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.

[31] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase change memory. *Proceedings of the IEEE*, 2010.

[32] Jian Xu and Steven Swanson. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, 2016.

[33] Jeremy Zawodny. Redis: Lightweight key/value store that goes the extra mile. *Linux Magazine*, 2009.

[34] Jishen Zhao, Onur Mutlu, and Yuan Xie. FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems. In *MICRO*, 2014.