# Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory

*Virendra J. Marathe* *Margo Seltzer* *Steve Byan* *Tim Harris*
{virendra.marathe,margo.seltzer,steve.byan,timothy.l.harris}@oracle.com
*Oracle Labs*

## Abstract

We report our experience building and evaluating pmemcached, a version of memcached ported to byte-addressable persistent memory. Persistent memory is expected to not only improve overall performance of applications' persistence tier, but also vastly reduce the "warm up" time needed for applications after a restart. We decided to test this hypothesis on memcached, a popular key-value store. We took the extreme view of persisting memcached's entire state, resulting in a virtually *instantaneous* warm up phase. Since memcached is already optimized for DRAM, we expected our port to be a straightforward engineering effort. However, the effort turned out to be surprisingly complex during which we encountered several non-trivial problems that challenged the boundaries of memcached's architecture. We detail these experiences and corresponding lessons learned.

## 1 Introduction

Key-value stores with simple get/put based interfaces have become an integral part of modern data centers. The list of successful key-value stores is long – Cassandra [22], Dynamo [11], LevelDB [23], memcached [14, 24], Redis [29] – to name a few. At the same time, emerging persistent memory technologies [1, 13, 18, 19, 26, 31], such as Intel and Micron's 3D XPoint [1], promise to provide the byte-addressability of DRAM (simple load/store access) and the persistence of traditional storage technologies, at performance 1000X greater than state-of-the-art NAND flash. This can fundamentally change the way applications manage persistent data.

With persistent memory on the horizon, many researchers are developing systems that ensure fast or even instantaneous recovery of application data [3, 5, 7, 27]. The overarching intuition is that by leveraging byte-addressability and the high performance of persistent memory, applications can drastically reduce, or even eliminate, the time needed to recover and "warm up" their state after a restart. While we share this view, we decided to test it in the context of memcached, a key-value store primarily used as a DRAM-resident cache. Warming up memcached's state after a restart can take up to several hours for workloads with large data sets [16]. Persisting that state could drastically reduce the warm up time.

During this exercise we wanted to investigate several important questions. Does persisting memcached's state entail any significant performance overheads? In the early years of persistent memory adoption, programmers will be forced to maintain existing application architectures to continue support for platforms without persistent memory. Minimal variation between this legacy code and the new persistent memory optimized code is desirable. How difficult will it be to persist memcached without changing its high level architecture? What hurdles will we encounter in this effort? Is there a pattern to these problems? Are there common programming practices that could be used to address them? How generic are these problems? Are there issues that cannot be addressed without rearchitecting memcached?

We first summarize the existing structure and operation of memcached (§ 2). We frame the description of our experience developing "pmemcached", our persistent memory port of memcached, in terms of 10 lessons learned (§ 3). Our findings were interesting, and in some cases, quite surprising. A big takeaway was that this exercise can be surprisingly non-trivial. The required lower level changes were contagious and quickly became pervasive. Failure-atomicity – providing all or nothing semantics across a failure boundary – seems fundamental. We found that we needed failure-atomic transactions more widely than we expected [4, 6, 8, 15, 21, 28, 34]. Other high level surprises and lessons learned include the challenges posed by tricky interactions between *persistent* and *non-persistent* objects, co-location of semantically persistent and nonpersistent data, and unexpected critical section inflation.

We evaluated pmemcached on Intel's Software Emulation Platform for persistent memory [12, 36] using the YCSB workload generator [9] (see § 4). We did achieve almost instantaneous warm up. We expected some performance degradation, however it varied significantly across different workloads. Degradation relative to memcached was about 10–15% for YCSB's read heavy workloads, but about 40–60% for YCSB's write heavy workloads.

## 2 memcached Overview

The high level architecture of memcached is typical of many key-value stores: It contains a stateful client re-

quest management module, a centralized data structure to host key-value pairs (a growable hash table), a memory manager called the "slab" allocator, an LRU cache management subsystem, and a set of dedicated background threads for LRU cache management, hash table resize, etc. memcached's key-value pairs are updated using the *copy-on-write* programming idiom – a new copy of the key-value object is created for every update.

Communication with clients happens over TCP. Request processing is done in multiple stages using a per-request state machine. Different parts of the incoming message are processed in different stages. For instance, a put request first goes through a stage that processes only the size of the key-value pair, by allocating a key-value pair object from the slab allocator. A later stage initializes the newly allocated key-value pair. Similarly, a get request first goes through a stage that retrieves a pointer to the key-value pair in the hash table; a later stage copies the retrieved value into the response buffer. Request processing buffers host pointers to key-value pairs. As a result, these key-value pairs are lazily reclaimed, using reference counters, when they are removed from the hash table (e.g., due to a concurrent put).

## 3 Programming pmemcached

We assume that the OS provides support for naming blocks of persistent memory in the file system [10, 12], and for mapping them in application address spaces via the mmap system call. Applications can thereafter access these blocks with simple load/store instructions. This approach follows SNIA's emerging programming model for persistent memory [32]. pmemcached's entire persistent state resides in a single file, which is mapped as a contiguous *region* in pmemcached's address space.

Starting from this point, we ported memcached's original code (v1.4.25[1]) to a persistent memory friendly version. This turned out to be an educational experience!

**Lesson 1*: Persistence of data structures can be contagious.*** While assessing an application to port to persistent memory, the programmer must be careful to recognize which data structures need to be hosted in persistent memory. Tight coupling between different data structures can quickly lead to an unexpected escalation in the number of data structures that need to be persisted.

Originally, we assumed that persisting memcached's central hash table was sufficient. However, after studying memcached's other data structures, we quickly determined that they were too tightly coupled, even though they logically belonged to different modules. More specifically, although the metadata structures of the slab allocator and LRU cache exist somewhat independently, some of their metadata is sprinkled in the key-value pair objects themselves. These objects also form nodes in the hash table. Making these key-value pairs persistent had a ripple effect on these other data structures. Furthermore, for correct instantaneous warmup, we also needed to persist the slab allocator's state. We could, in principle, keep the LRU cache nonpersistent, but then rebuilding the LRU lists on-the-fly would significantly distrupt memcached's original architecture where the LRU cache's state is consistent with the hash table's state, as well as the slab allocator's state. With these insights, we decided that all these structures needed to reside in persistent memory.

**Lesson 2*: Failure-atomic transactions might become necessary.*** As identified by a number of prior works [4, 6, 8, 15, 21, 28, 34], failure-atomic durable transactions become necessary for some applications to correctly modify persistent data structures, in the presence of failures.

We initially assumed that correctly persisting the data structures of memcached would be straightforward. We were wrong: The code paths for processing different client requests in pmemcached are complex and touch most of the aforementioned modules' data structures in the process. For example, memcached is a copy-on-write system, a put request touches data structures from the slab allocator (to allocate new and free old key-value pair objects), the LRU cache, and the hash table. All this state needs to be persisted correctly, including when there is a failure in the middle of the operation. Ensuring failure-atomicity for all this computation without failure-atomic transactions is practically infeasible, if not impossible.

We used our in-house persistent transaction library in pmemcached. The library contains accessor macros to transactionally read and write persistent objects and macros to begin and commit transactions. We enclosed pmemcached's code that accesses and modifies persistent data structures within transactions. The transactional accessors were hand-coded, which turned out to be a tedious undertaking, producing approximately 7K LOC of instrumented loads and stores.

**Lesson 3*: Code duplication may be unavoidable.*** If the programmer needs to maintain legacy code for platforms that do not support persistent memory, code duplication may be unavoidable. This is mostly borne out of the instrumentation needed for transactional accesses to persistent data. Because the transaction code paths were significantly complex in pmemcached, we ended up duplicating approximately 40% of the functions from memcached. The new versions of these functions executed transactionally instrumented code.

[1] We thank Alan Kasindorf [20] for pointing out errors in our description of memcached's internals in earlier drafts of this paper.

**Lesson 4*: Determining what to persist is hard.*** Modern data center applications are complex, with several disparate modules touching the same data in different ways. If the data moves to persistent memory, all such modules are affected. At the same time, some of these modules could continue processing co-existing nonpersistent data in the same way. At certain program points (e.g., library code), it becomes difficult to determine whether an accessed data object is persistent or not.

We ran into this situation when memcached used a common function to populate the value of a key-value pair. The function was at the end of deeply nested chain of function calls that were initiated for two different purposes: (i) to read a put request from a client, and (ii) to populate the response buffer for a get request. The former case used a persistent key-value pair and the latter case used a nonpersistent buffer. Discriminating between the two required either changing the signatures of every function in the call path, or passing state information to this function, breaking modularity. To expedite development, we employed a simple hack that issues persistence primitives (cache line writebacks/flushes and persist barriers) for the value buffer irrespective of whether the buffer was persistent or not. This hack worked, because the scenario was simple – populate and persist a buffer. It may not work in more complex circumstances, requiring more disruptive changes.

**Lesson 5*: Persistent pointers are tricky.*** Relying on the mmap interface to map persistent files in application address spaces has its challenges, one of which relates to pointer implementations. mmap does not guarantee that a given file will always map to the same virtual address range. As a result, pointer implementations as virtual addresses may not work correctly across restarts.

There are several workarounds to this problem most of which boil down to implementing pointers as offsets from some specified location in the virtual address space. We chose to use *self-relative* pointers. Self-relative pointers store the offset of the target from the pointer's own virtual address. We require special getter/setter functions to read/write these pointers. While the concept sounds simple, programmers must be careful – most of our program crashes happened when self-relative pointers were dereferenced as virtual addresses. Furthermore, the default assignment operator for structures, which typically does a direct bitwise copy, does not work correctly with self-relative pointer fields. We needed a special copy constructor implementation. This drawback does not exist in other base+offset style pointer implementations.

**Lesson 6*: Bypassing transactional accessors is risky.*** Transactional instrumentation can lead to significant performance overheads (e.g., logging for transactional writes). This instrumentation can be avoided in select cases, such as initializing a newly allocated persistent object before linking it into globally visible data structures. However, programmers must ensure that cache line writebacks/flushes follow the "naked" stores to correctly order them relative to surrounding stores and transactional writes; not doing so can lead to inconsistencies in the face of failures. Additionally, once a naked store persists, rollback of the enclosing transaction may not be able to restore the target's old value.

We encountered several instances where the instrumentation bypass optimization was effective. However, there were some tricky cases where such bypasses would lead to data inconsistencies. For instance, we first thought we could bypass all initialization writes to a newly allocated key-value pair within a transaction (the slab allocator is used to allocate all the key-value pairs). Initialization writes default values in various fields of the key-value pair (e.g., key/value size, zero out pointers fields). However, we subsequently realized that the slab allocator and the LRU cache data structures were using the same pointer fields to link objects in free lists and LRU lists respectively. Applying a naked store to initialize the pointers to NULL breaks the slab allocator's free list structure if the enclosing transaction subsequently aborts.

**Lesson 7*: Persistent and nonpersistent objects interact in unexpected ways.*** Persistent and nonpersistent objects can exhibit non-trivial dependencies. We discovered a dependency between the client request processing session objects and persistent key-value pairs. The session objects directly reference key-value pairs. In return, the key-value pairs themselves track these *nonpersistent* references (the session objects are nonpersistent) using an internal reference counter for lazy reclamation. This dependency between the request session objects and key-value pairs leads to memory leaks if a key-value pair, $K$, is removed from the hash table and there is a power failure before outstanding nonpersistent references to $K$ retire and decrement $K$'s reference counter to 0.

We need to persistently track such outstanding nonpersistent references. To that end, we added a per-thread persistent list of "pending reclamation" objects. The transaction that removes a key-value pair from the hash table, puts it in the thread's pending reclamations list, which can be used to reclaim objects correctly during recovery. The object is taken out of that list when the reference counter of the object goes down to 0.

**Lesson 8*: Initialization of semantically nonpersistent data colocated with persistent data is tricky.*** Programmers frequently find it convenient to co-locate nonpersis-

tent data in persistent objects. Identifying whether such nonpersistent fields need to be re-initialized after a restart event may be important for correctness. In pmemcached, the reference counters discussed above are semantically nonpersistent – they primarily track outstanding references from client request session objects. Resetting the reference counters of key-value pairs in the hash table to 1 is important to avoid memory leaks similar to the one described above. We do so by adding a persistent *generation number* [8] to each key-value pair. This must be equal to pmemcached's global persistent generation number, which is incremented (and persisted) at each restart. A generation number mismatch forces *lazy* reinitialization of a key-value pair's nonpersistent fields.

**Lesson 9*: Use of atomics needs to be rethought.*** It can be problematic to use atomic read-modify-write instructions, such as compare-and-swap, on persistent data if the update needs to be rolled back when the enclosing transaction aborts. memcached manages its hash table size counter using a tiny critical section. This can potentially be optimized to use a compare-and-swap instruction. However, that optimization is obviated in pmemcached since the counter needs to be updated transactionally. We anticipate that many legacy applications will contain uses of atomic read-modify-write instructions, which will need to be replaced with transactional updates, mediated through synchronization primitives such as mutual exclusion locks.

**Lesson 10*: Critical sections can pose significant problems.*** Lock-based critical sections are widely used in multithreaded applications to mediate access to shared data. Similar practices will be required for shared access to data hosted in persistent memory. This poses significant challenges in some cases. In particular, code that gets wrapped in a persistent transaction can end up executing critical sections that access persistent objects. Furthermore, the transaction's scope can be much larger than that of the critical section, complicating the question of when the thread should release the lock guarding a critical section – the locks cannot be released when the critical section ends since its transactional writes will not take effect until the enclosing transaction commits. Releasing the locks can lead to data races.

One "safe" approach, which we applied in pmemcached, postpones the lock release to the end of the enclosing transaction (our transactional API provides hooks to do so). However, this leads to expansion of the critical section, which in turn can lead to deadlocks and scalability bottlenecks. We ran into instances of threads deadlocking on themselves, which we resolved using reentrant pthread locks. In other cases, we resolved the inter-thread deadlocks by ensuring that locks are acquired in a specific order (by address of the lock).

For the scalability problems, we were able to eliminate expansion of some critical sections by moving the critical section to the end of the transaction; e.g., hash table size increment/decrement. However, other critical sections were harder to postpone because the computation in them was significantly tangled up with the enclosing transaction (e.g., memory allocation). In the end, we could not entirely resolve the scalability problems, the effects of which are visible in our evaluation. This problem can be addressed by significantly changing pmemcached's high level architecture, which we wanted to avoid. Alternately programmers can leverage advanced transactional features, such as *open nesting* [2, 4, 25] or *transaction boosting* [17], which adds complexity to manage *compensating actions* to correctly handle aborts. Our transaction library does not support such features.

# 4 Evaluation

Our pmemcached port turned out to be a major reengineering exercise, requiring about 4 months of an experienced researcher's time. It persists enough of its state during execution that a warm up requires only the following steps: mmap the persistent region file, recover various in-flight transactions (which typically takes a few hundred microseconds at most), load the persistent root pointer that identifies the root structure that hosts pmemcached's state information, and initialize pmemcached's background and foreground worker threads. All this work leads to sub-millisecond warm up intervals, irrespective of the dataset size.

We however wanted to measure common-case performance overheads required to make this near instant warm up possible. To that end, we conducted performance experiments on Intel's Software Emulation Platform [12, 36] using the YCSB [9] workload generator. The emulator is a dual-socket processor with 8 cores per socket and 512GB of DRAM, 384GB of which is emulated as persistent memory and the rest operates as conventional DRAM. In addition, we can configure load latency for the emulated persistent memory (300 nanoseconds for our experiments), as well as the persist barrier latency (100 nanoseconds). The latter aligns with Intel's recent deprecation of their pcommit instruction and requirement of the Asynchronous DRAM Refresh (ADR) feature, which ensures that on-chip memory controller buffers are flushed on power failure [30]. We simulated support for asynchronous cache-line writebacks, not flushes, in our experiments since they lead to significantly better cache locality (and performance) than cache line flushes, which evict flushed cache lines from processor caches.

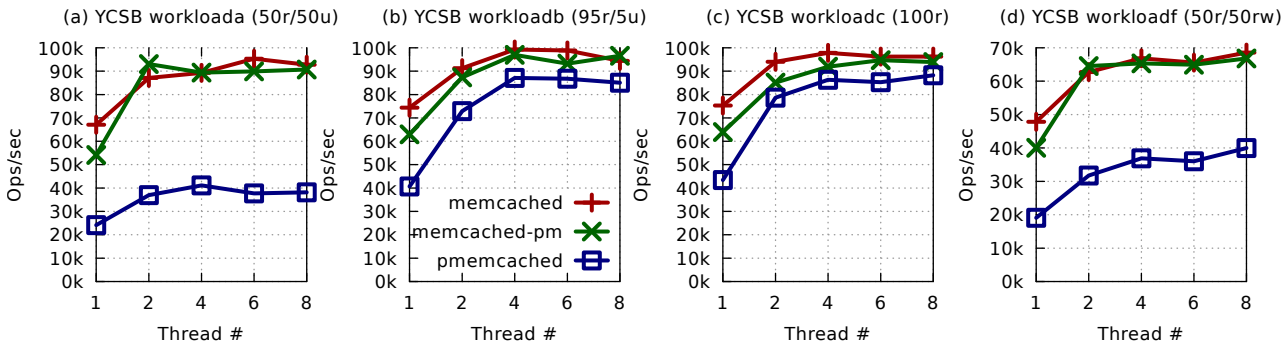


**Figure 1:** Scalability results for YCSB workloads (workloade, not included here, contains range queries that are not supported by memcached; workloadd's performance is similar to workloadb's). $r$ = read percentage, $u$ = update percentage, $rw$ = read-modify-write percentage. This is the default configuration where the key is a string ranging from 5 to 23 bytes, and value is a 1KB block; request distribution is zipfian. All the key-value stores were initialized to contain 10 million key-value pairs. The key-value store threads were all hosted on one socket (8 threads at most, one per core). The 8 YCSB threads were all hosted on the other socket. Each YCSB thread posts one operation at a time to the memcached server, and waits for the response before proceeding.

We measured the performance of three implementations: 1) original memcached 2) memcached-pm which is identical to memcached, but runs on the (slower) persistent memory, and 3) pmemcached, which uses undo logging for failure-atomicity [4, 8, 28]. This three way comparison lets us decouple the overhead due to slower persistent memory from that of managing persistence.

Figure 1 depicts the performance of YCSB workloads. When running a single thread, memcached-pm suffers a 10–15% performance degradation relative to memcached because persistent loads are configured to take 300 nanoseconds. pmemcached performs significantly worse, by 40–60%, due to the transactional instrumentation required for failure-atomic updates to internal state. The write cost is higher, due to the cost of writing and persisting the transaction log. As a result, we observe greater degradation for pmemcached on write-heavy workloads (workloada and workloadf) compared to the other read-heavy ones (workloadb and workloadc).

As we grow the number of worker threads, lock contention becomes the primary scalability bottleneck, and memcached-pm catches up with memcached. pmemcached has somewhat mixed results in that its throughput comes within 10–15% of memcached's throughput for read heavy workloads, whereas its throughput remains 40–60% that of memcached for write heavy ones. This again reflects the overheads of transactional writes, but more importantly, it reflects the scalability problems we discussed in § 3. More specifically, critical section inflation leads to significantly higher contention for the slab allocator and LRU cache locks in pmemcached compared to memcached. The higher latency of transactional writes further exacerbates the problem. While these performance results are not comprehensive, we believe they are representative of expected overheads in pmemcached.

## 5 Conclusion

We presented our unexpectedly complex effort making memcached persistent without changing its high level architecture. In the process, we encountered several surprises and learned many important lessons related to programmability challenges that we believe will be applicable in other application contexts. Overall, a major port of any complex application will likely be a non-trivial undertaking even though it may not seem so in the beginning. Many of the challenges we faced (e.g., failure atomicity requirements, code duplication) can be entirely eliminated with language support for transactions [4, 35, 33]. Programmers will still need to address other challenges on their own (e.g., interactions between persistent and nonpersistent objects, critical section inflation).

Performance results of pmemcached, in failure-free executions, are mixed. Slowdown for read dominated loads is modest (10–15% in multithreaded runs), whereas slowdown in write dominated loads is high (40–60% in multithreaded runs). Though transactional instrumentation, critical section inflation, etc. seem to be the primary reasons for the slowdown, we believe the fundamental problem is related to our strategy of not rearchitecting memcached. A more careful restructuring of the architecture could lead to less inter-module dependency, and more efficient and scalable transaction execution. That does not guarantee that we would avoid the programming challenges we encountered with pmemcached. While the findings in our pmemcached work may not apply verbatim to other application contexts, the lessons we learned should be broadly pertinent. In the end, architecting such systems from scratch is likely the best approach. A more modest goal of *fast warm up*, instead of *instantaneous warm up*, may also be a reasonable compromise.

## References


[1] 3D XPoint Technology Revolutionizes Storage Memory. http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-technology-animation.html, 2015.

[2] K. Agrawal, C. E. Leiserson, and J. Sukha. Memory Models for Open-nested Transactions. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, pages 70–81, 2006.

[3] J. Arulraj, A. Pavlo, and S. Dulloor. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 707–722, 2015.

[4] B. Bridge. NVM-Direct library. https://github.com/oracle/NVM-Direct, 2015.

[5] Z. Caklovic. Bringing Persistent Memory Technology to SAP HANA: Opportunities and Challenges. In *Annual SNIA Persistent Memory Summit*, 2017.

[6] D. R. Chakrabarti, H. Boehm, and K. Bhandari. Atlas: leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 433–452, 2014.

[7] A. Chatzistergiou, M. Cintra, and S. D. Viglas. REWIND: recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8(5):497–508, 2015.

[8] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–118, 2011.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, 2010.

[10] J. Corbet. Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/, 2014.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007*, pages 205–220, 2007.

[12] S. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *9th EuroSys Conference*, page 15, 2014.

[13] Everspin: The MRAM Company. https://www.everspin.com/.

[14] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124), 2004.

[15] E. Giles, K. Doshi, and P. J. Varman. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015*, pages 1–14, 2015.

[16] A. Goel, B. Chopra, C. Gerea, D. Mátáni, J. Metzler, F. Ul Haq, and J. Wiener. Fast Database Restarts at Facebook. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 541–549, 2014.

[17] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, pages 207–216, 2008.

[18] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM. *International Electron Devices Meeting*, pages 459–462, 2005.

[19] Y. Huai. Spin-Transfer Torque MRAM (STT-MRAM): Challenges and Prospects. *AAPPS Bulletin*, 18(6):33–40, 2008.

[20] Personal Communication, *Alan Kasindorf, Facebook*, 2017.

[21] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-Performance Transactions for Persistent Memories. In *21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[22] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[23] LevelDB – a fast and lightweight key/value database library. `http://leveldb.org/`.

[24] Memcached – a distributed memory object caching system. `https://memcached.org/`.

[25] J. E. B. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1981.

[26] Nantero. `http://www.nantero.com/`.

[27] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main memory databases. In *7th Biennial Conference on Innovative Data Systems Research*, 2015.

[28] pmem.io: Persistent Memory Programming. `http://pmem.io/`, 2015.

[29] Redis – in-memory data structure store, `http://redis.io/`.

[30] A. Rudoff. Deprecating the PCOMMIT instruction. `https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction`.

[31] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing Memristor found. *Nature*, 453:80–83, 2008.

[32] The SNIA NVM Programming Technical Working Group. NVM Programming Model (Version 1.0.0 Revision 10), Working Draft. `http://snia.org/sites/default/files/NVMProgrammingModel_v1r10DRAFT.pdf`, 2013.

[33] Transactional Memory in GCC. `https://gcc.gnu.org/wiki/TransactionalMemory`.

[34] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 91–104, 2011.

[35] M. Wong (Editor). Working Draft, Technical Specification for C++ Extensions for Transactional Memory. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4301.pdf`, 2014.

[36] Y. Zhang and S. Swanson. A study of application performance with non-volatile main memory. In *IEEE 31st Symposium on Mass Storage Systems and Technologies*, pages 1–10, 2015.