**PERC: Persistent, Efficient, Recoverable, Consistent Research Proficience Evaluation Proposal**

Tony Mason∗

Department of Computer Science, University of British Columbia

# Abstract

This document is a supplement to my main **Re- search Proficiency Evaluation** proposal. It is un- likely to “make sense” outside the context of that document.

# Prior Work

The literature in this area is quite extensive — per- sistent memory has been a long promised technol- ogy that has taken considerably longer to material- ize than expected. As a result, there are quite a few papers that discuss persistent memory but in many cases they are speculative as to the nature and char- acteristics of NVM.

In this section I will explore a subset of the pa- pers from the literature; hopefully I have managed to identify a useful sample. For each paper consid- ered, I have provided a description of the paper and its significance.

# Non-Volatile Memory

Before I review the literature, it seems useful to de- scribe non-volatile memory (NVM) in the context of this proposal. In terms of this discussion, my primary focus is on *mutable* NVM; more recent ad- vances in the field have achieved byte-addressable NVM that can be directly accessed from the CPU.

∗[fsgeek@cs.ubc.ca](mailto:fsgeek@cs.ubc.ca)

There are a variety of technologies used to achieve this, including *phase change memory* (PCM), which has been a research area for almost 50 years. F- RAM is a technology closely related to DRAM, but is not popular due to the achievable density and the destructive nature of reading it, which requires that it be rewritten. **MRAM** technology is also similar to DRAM, but uses magnetic storage, which does not degrade like DRAM’s electronic storage. There are considerable challenges to this type of memory technology, though it certainly is challenging. [[56]](#_bookmark21) One advantage of MRAM is that it is actually faster than DRAM (*almost* as fast as SRAM, which is of- ten used inside CPU cores for cache memory).

In fact, what is emerging as commercially avail- able NVM are things like *3D XPoint*, which is a variant of phase-change memory. Other companies (e.g., SanDisk and HP) claim to also have emerg- ing technologies (in their case based upon *ReRAM*). One thing that is clear from prior industry prediction is that it may take more time to actually materialize than expected.

The primary competitive technologies at this point appear to be based upon *PCM* or *ReRAM*. [[3]](#_bookmark5) While presently Flash is less expensive, over time (5-10 years) that balance is likely to change.

An important driver here is the *size* of NVM. Current NVM solutions are fairly modest in size, with the hybrid DRAM + Flash with battery backup solution consisting of NVDIMM-N modules up to 32GB in size, with most being significantly smaller

(8GB or smaller). One of the promised benefits of NVM is that it is much higher density than DRAM, which ultimately will allow positioning terrabytes of storage on the memory bus.

One important observation here: the **failure se- mantics** around NVM are not particularly well- defined. What I mean is that once you remove the external power source, which is what is used in NVDIMM-N, you must reason about failure. There is nothing explicit thus far that insists there is suffi- cient capacitance in the system to ensure write back of cache lines *in toto* to the NVM. There is evidence this problem is known (I observed a Super Micro patent regarding memory-type range registers on In- tel CPU based motherboards that would allow for guaranteed write back of specific memory ranges.) **Even if** the platform should deal with this class of failure, I will argue that it is likely useful to protect against it in any case since limiting oneself to 64 byte data structures probably overconstrains solu- tions for common problems. Reasoning about *fail- ures* in the overall system is important; the ability to provide consistency guarantees relies upon un- derstanding the failures and validating that we can recover from them.

# Transactional Memory: Architec- tural Support for Lock-Free Data Structures [[18]](#_bookmark13)

This paper describes how CPU hardware can extend the cache coherence protocol to make it possible for software to implement efficient lock-free data struc- tures. The goal of this work is to provide a high per- formance alternative to traditional synchronization. The authors’ note the benefits: “In highly concur- rent systems, lock-free data structures avoid com- mon problems associated with conventional locking techniques, including priority inversion, convoying, and difficulty of avoiding deadlock.” Thus, some of the more serious problems associated with tra- ditional software locking techniques can be elim- inated. To do so and *also* improve performance seems a laudable goal.

This paper defines **lock-free**: “A shared data structure is *lock-free* if its operations do not require

mutual exclusion. If one process is interrupted in the middle of an operation, other processes will not be prevented from operating on that object.”

Software-based lock-free data structures are known to be possible, but are not efficient accord- ing to the authors (and they cite extensively to the literature to make this point.) Thus, they propose a CPU (hardware) architecture that permits efficient implementation of lock-free data structures.

“Transactional memory allows programmers to define customized read-modify-write operations that apply to multiple, independently-chosen words of memory.” To more clearly define this, they es- tablish further terminology. First, they note that a *transaction* is a finite set of machine instructions that can be executed by one process that guaran- tees serialization and atomicity. Thus, a transaction made up of many steps must function so that either **all** or **none** of the operations from the perspective of an actor outside the modifying process. Thus, we say that the given set of operations either *committed*, because all of the changes are made, or it *aborted*, because none of the changes are made.

The authors’ then propose a series of machine level operations for implementing this functionality within the CPU: operations to act on memory, op- erations to commit or abort the transaction, and an operation to determine the outcome of the transac- tion.

The paper describes how they envision this func- tionality being implemented based upon cache co- herency protocols. They observe “[A]ny protocol capable of detecting accessiblity conflicts can also detect transaction conflict at no extra cost.” They also describe what might cause a transaction to abort and suggest multiple ways in which this can be accomplished. Given that my interest is more in how to *use* these transactions, if you are interested in this detail you should refer to the paper.

The authors then describe three examples of how to employ their proposed transactional mem- ory model:

* + - **Shared Counter** - this is a simple monoton- ically increasing counter that can be shared across processors *without* a lock.
    - **Producer/Consumer Queue** - this is a shared producer/consumer queue that permits multi- ple producers and consumers simultaneously accessing the common queue.
    - **Doubly-Linked List** - this is a traditional dou- bly linked list, with a head and tail, and then pointers from one entry to the next.

The authors implement each of these and test it on a simulator; the result of their simulation indicates that each of these lock-free data structure imple- mentations is *faster* than the corresponding mutual- exclusion based mechanism.

While not identical to the model ultimately im- plemented in the Intel architecture machines, it is conceptually similar and the cache coherency pro- tocol based implementation is quite similar to what Intel actually implements for **TSX**.

The authors do not address how this might in- teract with Non-Volatile Memory, likely because it was not a matter of concern for them. However, us- ing transactional memory when working with NVM does seem like a valid use case. In addition, I note that because this involves the cache coherency protocol, the most reasonable expectation for im- plementation is that using a transaction on mem- ory is an implicit *pinning* operation within the CPU cache. Otherwise, the CPU must handle the poten- tially complex problems of unwinding failed trans- actions.

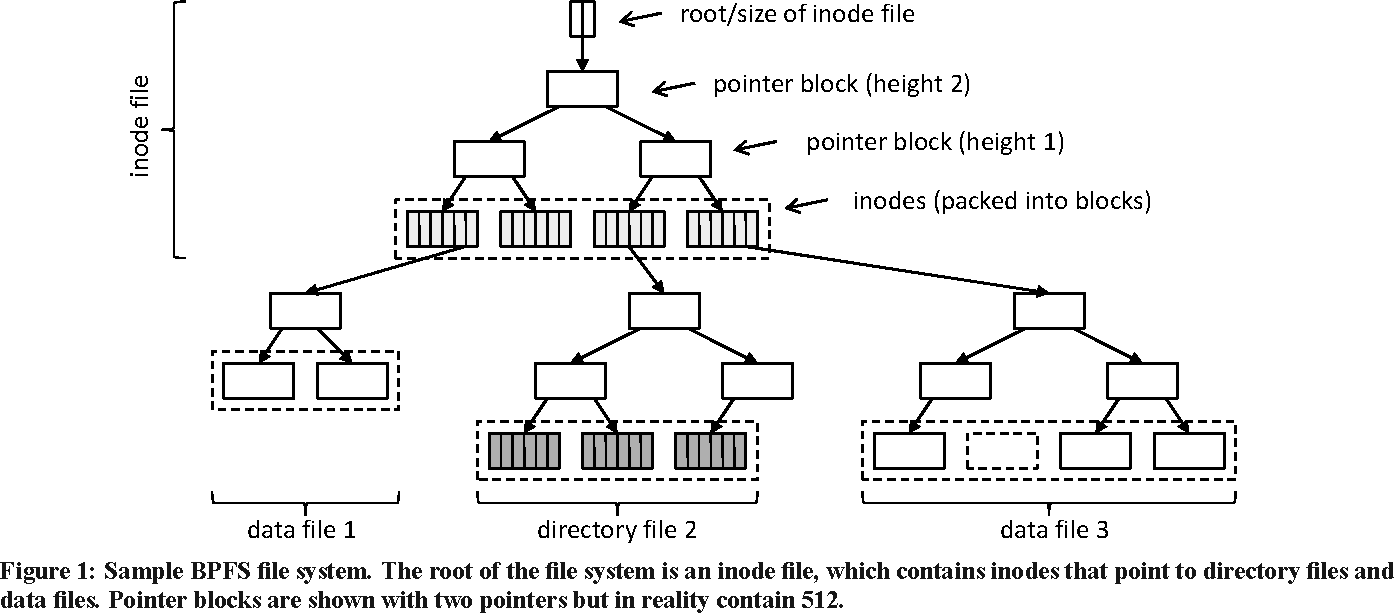
# Better I/O Through Byte- Addressable, Persistent Mem- ory [[10]](#_bookmark8)

This Microsoft Research (MSR) paper presents a study of **BPFS**, a file system developed by the MSR team to evaluate efficient use of byte-addressable persistent memory. They introduce a new tech- nique, which they call “short-circuit shadow pag- ing”. This provides their atomic unit of data man- agement. Logically, this mechanism is similar to the shadow paging mechanism seen in prior systems (e.g., NetApp’s WAFL file system) that utilize B+ trees to implement a transactional recovery model.

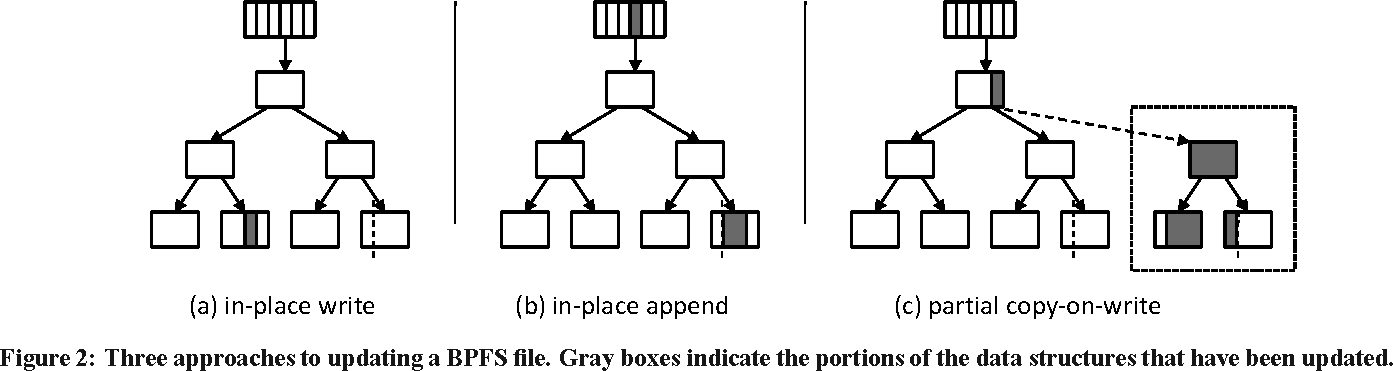
They require only two specific primitives from the underlying hardware: atomic 8-byte write op- erations and “epoch barriers”. I would note that the atomic 8-byte write is in fact assumed to work **in the face of power failure**. While this seems like a reasonable assumption, I have not been able to find any guarantees that this is provided by then *or* current hardware. The authors describe the epoch barrier to be a memory fence that enforces store or- dering; the description sounds much like the Intel **SFENCE** operation.

They use these two mechanisms to implement their short-circuit shadow paging. Specifically, be- cause they can perform an eight byte atomic update, it is often possible to update information *in place*. In other cases it may be possible to limit the number of shadow pages required. This approach substantially minimizes the number of write operations required to maintain consistency — the *write amplification*.

The model for their file system is included in Fig- ure 1 from their paper:



Thus, it is a B+-tree structured file system; this facilitates their use of the shadow paging mecha- nism (which works well for such structure.) Figure 2 from the paper then underscores the “short circuit” of their approach:



Full shadowing is done when the size of the change exceeds the ability of the system to prop- erly handle it (e.g., eight bytes) but the atomic up- date is exploited when the change is small. As it turns out, many operations can be done in this fash- ion and gain the benefit of reducing the size of the change.

Another notable aspect of their system is that it mixed DRAM and NVM usage. Thus, they exploit DRAM for transient data they do not require be per- sistent.

There are some interesting compromises in their file system that indicate one of the restrictions of the atomic update in place approach that they have cho- sen. For example, they note that they cannot update the *write* time simultaneously with the write oper- ation, so it is possible that the data could change without the timestamp being updated, or the times- tamp could be updated without the data being mod- ified; they argue that this is a “reasonable compro- mise” for performance. They point out that some operations have substantial write amplification ver- sus a journaling approach; this seems to be inherent in the shadow paging model.

Interestingly, they do point out that they could eliminate the file system entirely and simply pro- vide a “persistent heap”. They argue against this because it requires changing applications to take ad- vantage of NVM. I find this an interesting argument

— it certainly motivates the interest in DAX file sys- tems support (outside the scope of this paper) but it ignores the high overhead associated with system calls (hearkening back to the work of Leidtke [[33]).](#_bookmark16) I would note that current information suggests the overhead of a system call is around 100 ns [[60].](#_bookmark23) This represents a significant overhead for small op- erations, given the costs of reading from or writing to NVM.

# The Log-Structured Merge Tree (LSM-Tree) [[52]](#_bookmark20)

This paper does not relate to non-volatile mem- ory, but we will see Log-Structured Merge Trees (LSMTs) used in quite a few NVM projects. From the abstract:

*The log-structured merge tree (LSM-tree) is a disk-based data structure designed to provide low-cost indexing for a file expe- riencing a high rate of record inserts (and deletes) over an extended period. The LSM-tree uses an algorithm that defers*

*and batches index changes, cascading the changes from a memory-based compo- nent through one or more disk compo- nents in an efficient manner reminiscent of merge sort. During this process all in- dex values are continuously accessible to retrievals (aside from very short locking periods), either through the memory com- ponent or one of the disk components.*

So LSMTs originate from concerns about the la- tency issues around disk drives.

In a nutshell, the challenge with disk drives are they have mechanical parts that must be moved in order to read the data. Data is written in concen- tric bands around the center. The angular velocity of the disk platter is the same, but of course the sur- face velocity is lowest towards the center and fastest towards the outer edge. The disk drive “head” is moved in and out to read from each of those concen- tric circles. Since it can only read what is beneath the head, it also must wait for the desired data to ro- tate under the head. This is one reason why faster disk drives (usually measured by the rotations-per- minute number) provide faster response times. On the other hand, faster disk drives generate more heat and are more expensive to build.

Thus, an important consideration for file systems working on rotating media is the latency to per- form random access. Tape drives have the high- est latency, since we have to reposition the tape to get to another location and read its data (there are other complexities as well, such as the fact that tape benefits most from streaming write). Hard disk drives (HDDs) are not so bad as tape drives in gen- eral (though SMR drives act much like tape drives, which is one reason I mention that here.) Solid State Disks are even better than disk drives, though even for an SSD random access is slower than sequential access - but both are much faster than HDDs.

Some of those papers that I have yet to cover de- scribe the concept of a log-structured file system. One of the things that I learned when working on the Episode File System was that converting ran- dom I/O to sequential I/O was definitely a win (so was asynchronous logging). It is this observation: converting random I/O to synchronous I/O that pro-

vides the benefit of using journaling techniques (the “log”).

So LSMTs capitalize upon this advantage. Note that an LSMT is not a single data structure; rather it is a general technique for working with systems where insert and delete are the common opera- tions, such as meta-data within a file system, or (key,value) tuples in a key-value store. It also points out that reading large sequential block is generally more efficient; they cite IBM when noting that a sin- gle page read from the DB2 Database takes approx- imately 10 milliseconds. A read of 64 continuous pages costs about 2 milliseconds per page (125ms total). So batching I/O operations is also an impor- tant mechanism for improving performance.

So what is an LSMT? “An LSM-tree is composed of two or more tree-like component data structures.” From there the authors describe their initial case: where one tree is memory resident and the other is disk resident. Note, however, this is the smallest set for a valid LSMT. Systems with more than two have been built - and one way to use non-volatile memory (NVM) is to add it as another layer to an LSMT.

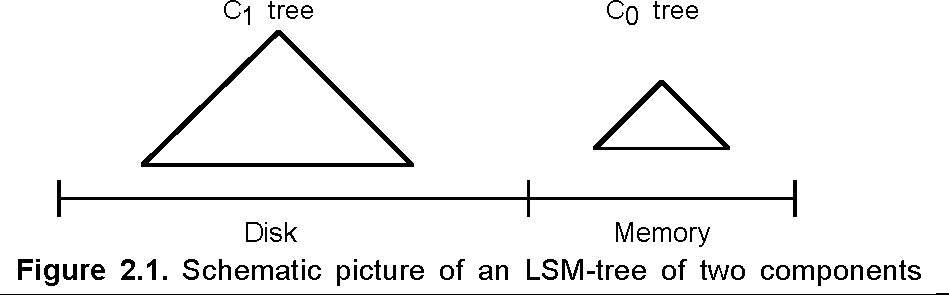


Figure 2.1 (from the paper) shows the high level structure of the LSMT - a tree-like structure at each level of the storage hierarchy; these levels can have different characteristics, such as being ephemeral (in memory) or persistent (on disk, tape, SSD, etc.)

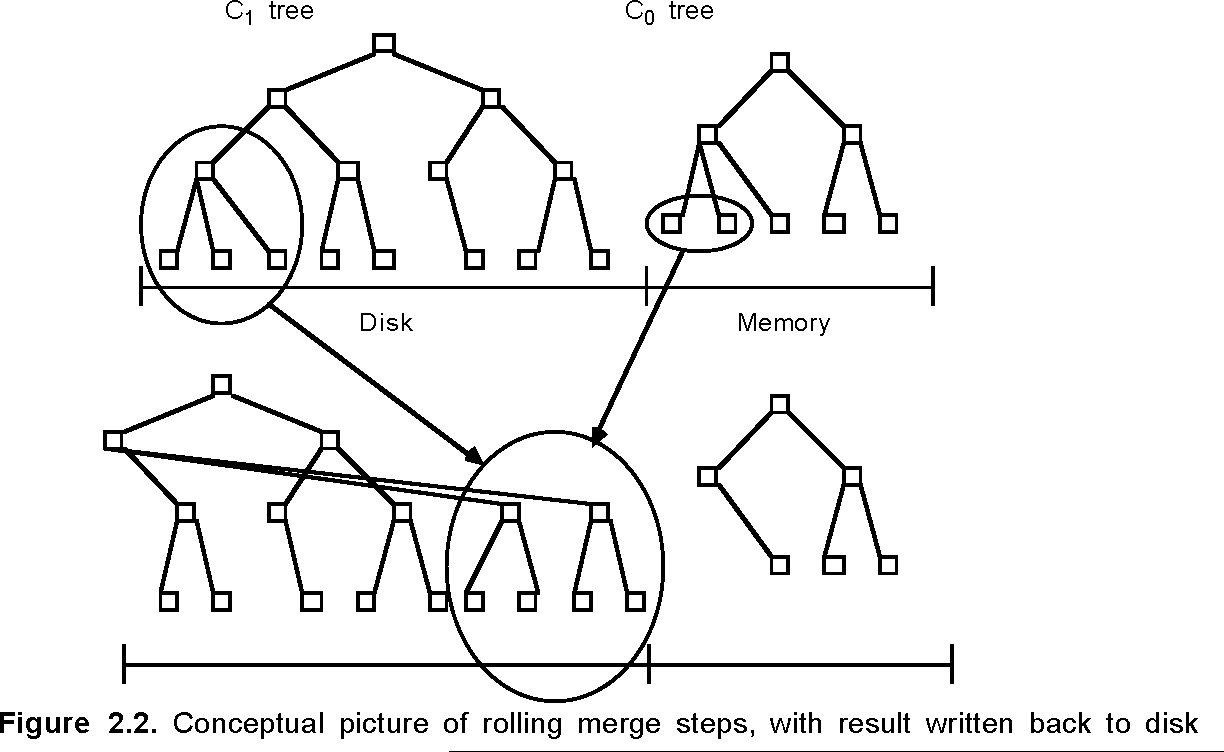


Figure 2.2 provides greater detail, showing how data is merged from one level to the next (”rolling

merge”).

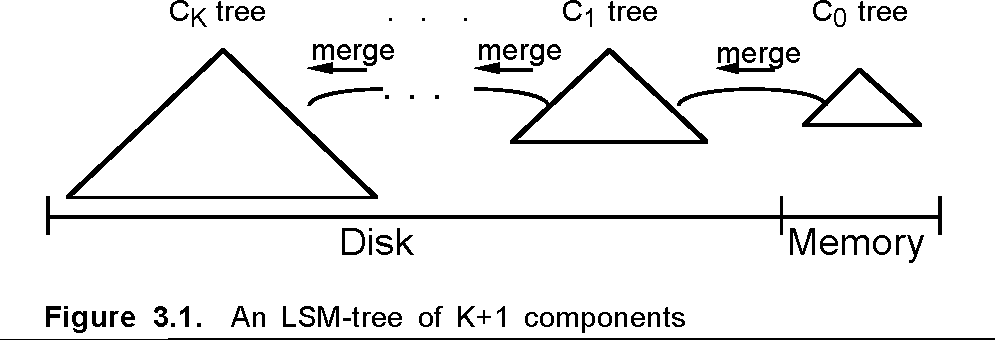


Figure 3.1 shows a generalization of the LSMT, in which data is migrated from one level of the stor- age hierarchy to the next.

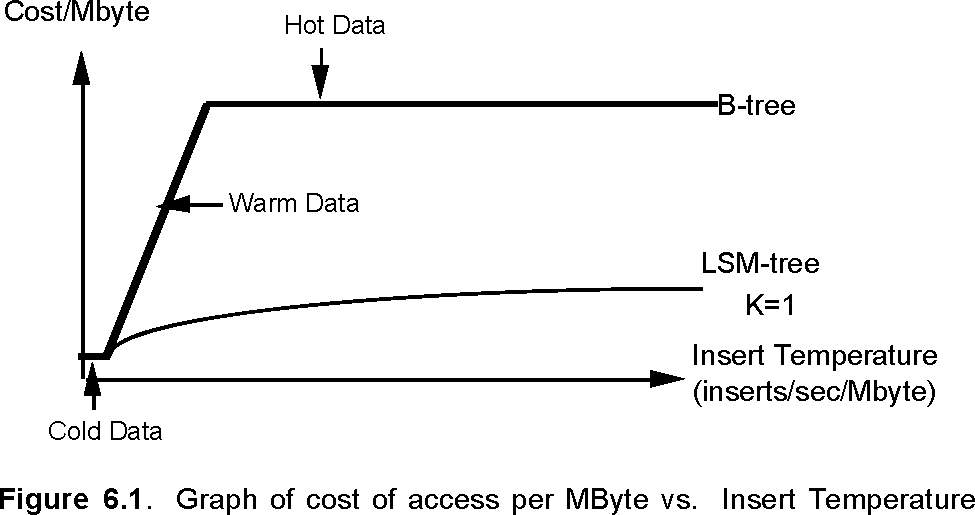


Figure 6.1 then helps motivate this work: data that is seldom accessed (which is most of the data) is “cold” and can be stored in lower cost storage. Data that is frequently accessed (which is a small amount of the data) is “hot” and benefits from be- ing stored in faster but more expensive storage. In- deed, there is a substantial body of work at this point that demonstrates how data tends to cycle from be- ing hot to being cold. Thus, there is a period of migration for “warm” data. This also helps explain why having a multi-stage model makes sense. This behavior is quite general, in fact. Disk drives are constructed with caches on them. The cache is for the hot data, the disk storage for the cold data. SSDs are often structured with multiple classes of NVM; a small amount of expensive but fast NVM and then a larger amount of less expensive (often block ori- ented) NVM. Even CPUs work this way (as I will be discussing ad nauseum), where there are multi- ple levels of caching: L1 cache is small but very fast, L2 cache is larger and slower (and cheaper), L3 cache is again larger and slower. Then we get to memory (DRAM) which is even slower. That’s all before we get to storage!

This is quite a long paper: they describe how data is merged from one level to the next as well as do an in-depth analysis of cost versus performance. But I

will leave ferreting out those details to the interested reader. I got what I came for: a basic description of the tiered nature of LSMTs and how we can use them to make storage more efficient without driving up costs.

# Mnemosyne: Lightweight Persis- tent Memory [[63]](#_bookmark26)

The abstract starts us off in this brave new world:

*New storage-class memory (SCM) tech- nologies, such as phase-change mem- ory, STT-RAM, and memristors, promise user-levelvaccess to non-volatile stor- age through regular memory instructions. These memory devices enable fast user- mode access to persistence, allowing reg- ular in-memory data structures to survive system crashes.*

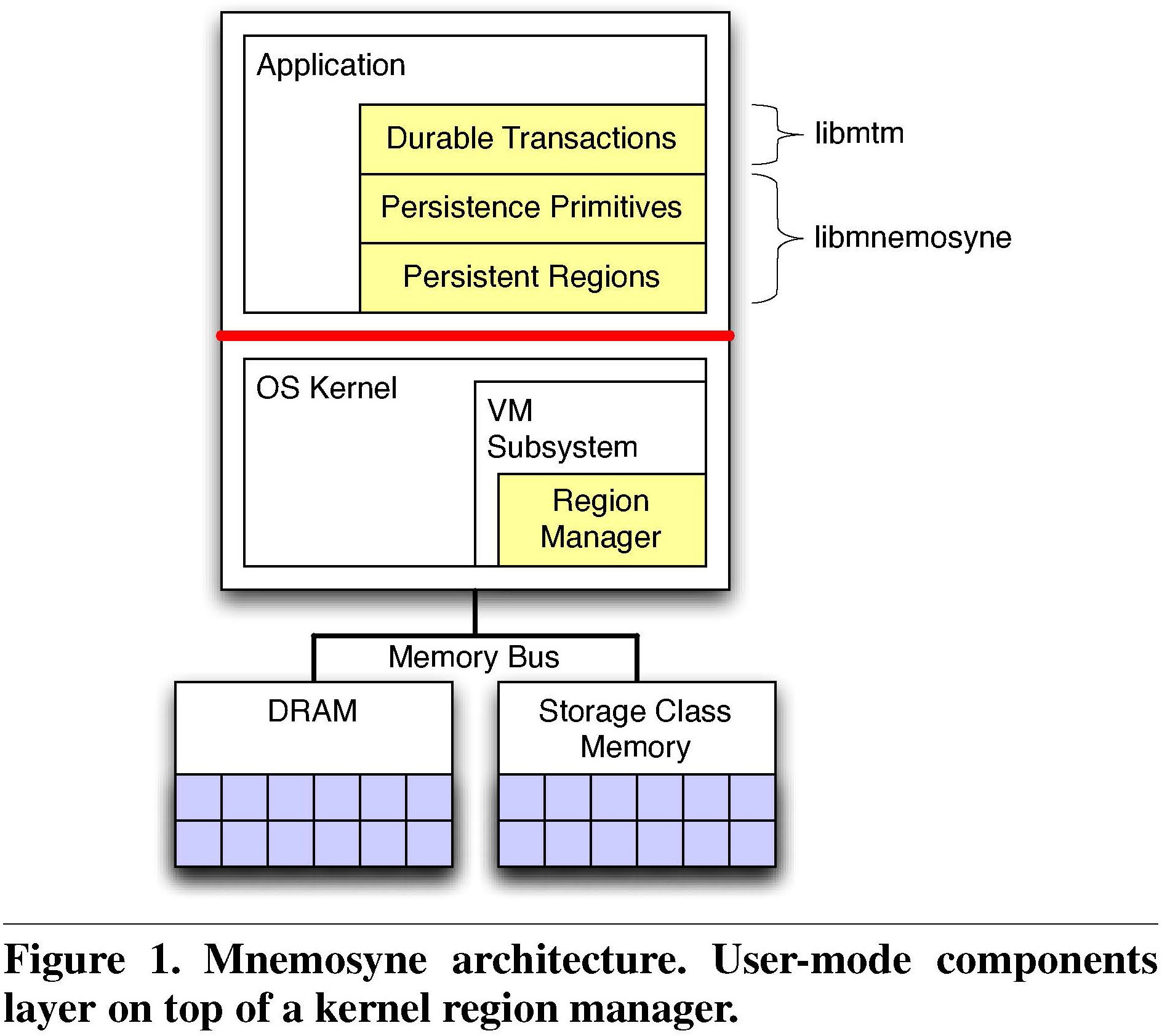
So faster, doesnt require privilege, works like mem- ory, and persistent. Seems like a great invention.

File systems arent really constructed to have di- rect access to the disk from user applications. Gen- erally it is done via an I/O interface: open, close, read, and write. But memory isnt accessed in that fashion at all. So, how does this affect things? What do the services look like? What does it mean to take something everyone thinks of as transient and make it persistent?

Lets start exploring!

Mnemosyne provides an explicit mechanism for exposing persistent memory to applications. This is done by extending the programming tools so they can declare something should be stored in persistent memory, or so that it can be dynamically allocated with the proviso that it be allocated from this persis- tent memory.

Thus, the default is that an existing application retains the same behavior it does not use persistent memory. If an application wishes to use persistent memory it must be modified to do so. Mnemosyne will provide a basic service level, but it wont change the behavior of existing applications (technical debt really does follow us around in this business).



Its impressive: Mnemosyne can persist data as fast as 3 microseconds. It makes existing applica- tions modified to use it much faster. Figure 1 (from the paper) describes the architecture the authors cre- ated for Mnemosyne.

Mnemosyne ArchitectureThis architecture envi- sions the persistent memory being exposed to the application through a persistence interface; the mo- tivation for this is that merely having persistent memory is not enough. It requires additional work to ensure that it is crash resistant. In other words, the system can restore the state of the contents in memory to some well-defined consistent state.

This is something file systems routinely handle the issues of persistence and recoverability. I often try to think about failure: how does failure mani- fest? How do I know that I can recover the state to a consistent spot and then proceed?

This is an uncommon concept for most applica- tion developers: they dont need to worry about the contents of memory being consistent in the face of crashes because when the application crashes, the memory is lost.

Mnemosyne provides a model of consistency for applications by creating an explicit mechanism for providing crash consistence. Note that Mnemosyne wont define those consistent states the application must define what it means for its data structures to be consistent. What Mnemosyne offers are certain guarantees about the contents of memory.

The authors decision to virtualize their SCM is

an interesting one: [V]irtualization prevents a mem- ory leak in one program from monopolizing a finite amount of SCM. Thus, they stage SCM content to disk between processes. Consistency of data is pro- vided by ordering writes. The authors identify four consistency mechanisms:

* + - Atomic variable update update the data in place as a single all-or-nothing operation. Ap- pend updates data is not written in place, but rather a new copy is written, such as it might be to the end of a log (such updates are ordered).
    - Shadow updates data is written to a new loca- tion and once done, the pointer to the old copy is updated to point to the new copy (e.g., via an atomic variable update). The authors point out there is a potential leak here that must be handled properly.
    - In-place updates used for data structures that can be modified in place; provided the opera- tions are ordered.

Consistency guarantees for persistent memory are accomplished using processor semantics and mechanisms:

*A write through operation (e.g., a tem- poral move) that is written directly to memory. Memory fences that ensure strict ordering of operations before the fence relative to operations after the fence. Cache line flushes. The CPU stores mem- ory inside the processor while it is acting upon it. In fact, a modern CPU has multi- ple levels of memory. The most expensive (and smallest) will be the Level 1 cache. Its also the fastest. L2 cache is larger and slower than L1 cache. L3 cache is typ- ically shared with all CPUs on the pro- cessor; it is the largest and slowest of the caches.*

For storage people, some of this is familiar and some of it is different instead of worrying about storage stack semantics were now worrying about

processor cache semantics. One upside is that pro- cessor semantics are more rigidly enforced than storage semantics (e.g., disk drives that lie and say that the data has been written when it hasnt.) One downside is that its a new failure domain. For any- one used to working with persistent storage, under- standing the failure domain is vital. I suspect it is also different for people used to thinking about the processor perspective, since persistence isnt usually something you have to reason about.

Mnemosyne implemented a persistent heap allo- cator, a modified version of Intels STM Compiler (well see later that others had to move that work to other compilers because it is now abandoned), a logging mechanism, a persistent region mechanism, a transactional system (based upon TinySTM).

Their results are, of course, good. After all, if they had not been good, they wouldnt have been published. They outperform BerkeleyDB (for some metrics). They demonstrated a fast and persistent red-black tree implementation. They show the ben- efits of asynchronous truncation.

Mnemosyne was a useful contribution because it was an early exploration into considering how we should use byte-addressable non-volatile memory. The library they built is used in future work as well, and this is a heavily cited paper.

# NV-Heaps: Making Persistent Ob- jects Fast and Safe with Next- Generation, Non-Volatile Memo- ries

This paper was presented at the same conference as Mnemosyne. The authors explore a different use of Non-Volatile Memory (NVM): using it for storing persistent objects. The authors sum up the motiva- tion for this:

Creating these data structures requires a system that is lightweight enough to expose the perfor- mance of the underlying memories but also ensures safety in the presence of application and system failures by avoiding familiar bugs such as dangling pointers, multiple free()s, and locking errors. In ad- dition, the system must prevent new types of hard- to-find pointer safety bugs that only arise with per-

sistent objects. These bugs are especially dangerous since any corruption they cause will be permanent.

Thus, their motivation is to enable the use of these nifty persistent, user-defined objects that are not practical when backed by disks ([T]he slow speed of persistent storage (i.e., disk) has restricted their design and limited their performance.)

The authors make some important observations that are just as applicable today as they were in 2011. These include the observation that persis- tent objects in NVM cannot reasonably be treated like disk based objects because the gap between memory and storage performance drove many de- sign decisions that shaped them. Nor can they be treated like volatile memory: To guarantee consis- tency and durability, non-volatile structures must meet a host of challenges, many of which do not exist for volatile memories.

They also observe that NVMs greatly expand the possibility of bug sources from having disparate ad- dress spaces. In other words, while you have a sin- gle address space, part of it is ephemeral and if you store a reference to the ephemeral part in the per- sistent part, it will be inconsistent after the current process terminates.

I found their observation about the ability of programmers to reason about this also apropos: [t]rusting the average programmer to get it right in meeting these challenges is both unreasonable This is consistent with more than 50 years of experience in systems. Personally, I dont think this is an indict- ment of the programmer so much as it is a burden on the system (a perspective the authors appear to endorse as well). To make this viable, we need to make it easy to get it right.

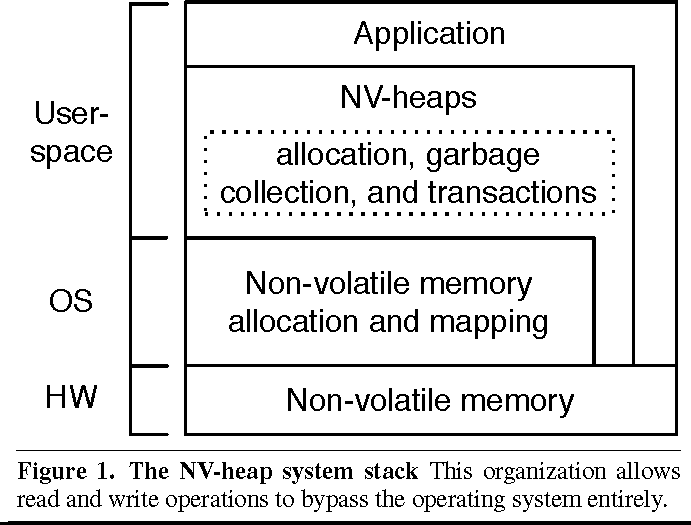


Figure 1 shows the general architecture of NV- Heaps: It is envisioned as a library of useful services layered on top of the operating system provided ab- stractions. One important observation here is that this model completely avoids the need to interact with the operating system in ordinary program ex- ecution. Persistence no longer relies upon utilizing the standard file systems interface.

The authors explanation of their goals looks like a veritable wish list to me: prevent programmer er- rors, transactions, referential integrity, performance and scalability, and ease of use. Im not sure how referential integrity is different than programmer er- rors, but clearly it is a very important aspect of their persistent system.

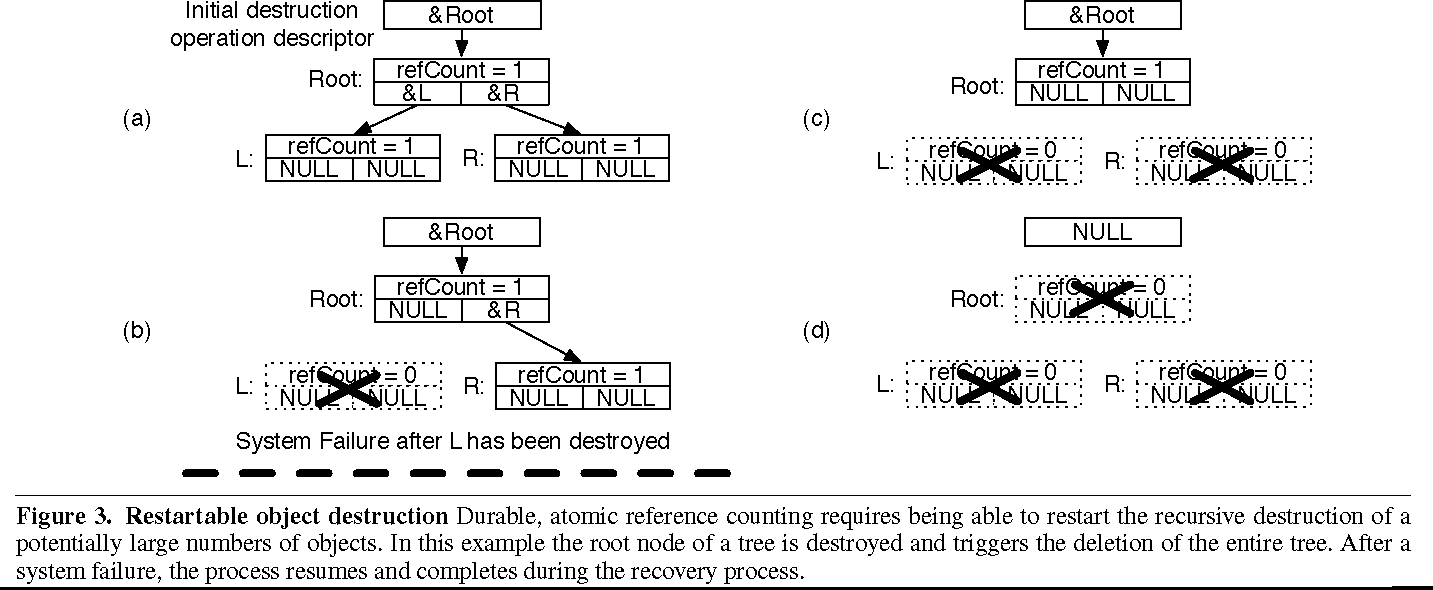


Figure 3 shows how they handle one of the com- plex consistency cases inherent in managing NVM: the need to ensure that operations can be safely restarted. For example, when deleting a large data structure, such as a tree, it must be removed in a way that it can be stopped and restarted (e.g., if the system were to crash, it must then be able to resume removal). To resume after a crash, they use a log of operations and replay it a classic solution to the problem.

To make their goal of referential integrity work

properly they utilize the programming language constructs to do this. The authors note they achieve this by using 128 bit pointer values (on a 64 bit sys- tem).

The paper describes their implementation in con- siderable detail. Again, as we would expect, the implementation yields substantially better perfor- mance than comparable systems backed by disks this really shouldnt come as a surprise, given the performance differential between disks and non- volatile memory. Even if they had used solid state disks (which existed but were rare in 2011) their re- sults would have still be notably better.

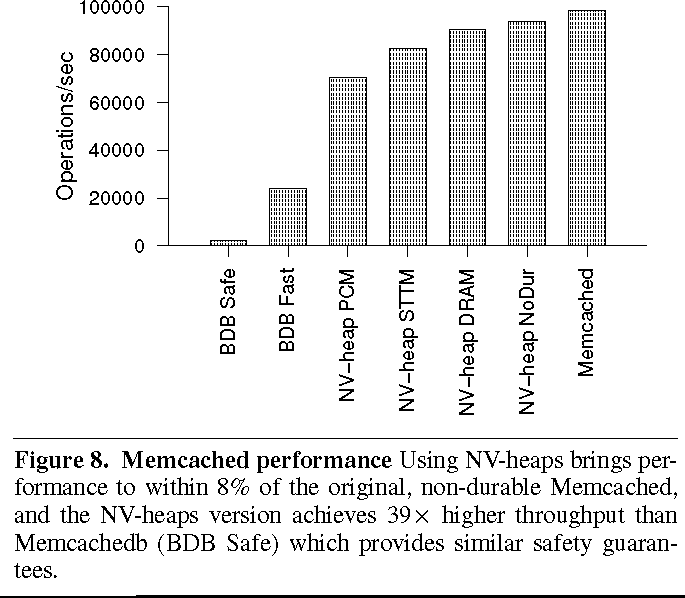


Figure 8 shows their performance information, comparing themselves against several other sys- tems. One thing to note: they do not have NVM memory. They use a memory simulator to model the behavior of the system. The performance fig- ures they provide surprised me: they are substan- tially faster than I would have expected. For PCM, they used a 67 nano-second (ns) read time and 215 ns write time. The paper explains how they ob- tained these values and how they validated them. For STTM (a different NVM technology) they re- ported 29 ns read and 95 ns write. As a baseline, their DRAM read time was 25 ns, and write time was 35 ns.

While these numbers were lower than I would have expected, the relative ratio is close to what I expected from other things that I have read: PCM memory is about 2.5 times slower for reads, and 10

times slower for writes. This is consistent with what the paper reports. I guess its time to update my men- tal Jeff Dean numbers. And indeed, it turns out that DRAM latency is around 15 ns. [[12]](#_bookmark10)

The authors were able to modify memcached to use their library for persistence. They report that they were able to get within 8% of the original memcached. That seems like an excellent outcome.

All we need now are NVMs.

# Consistent and Durable Data Structures for Non-Volatile Byte- Addressable Memory [[61]](#_bookmark24)

In this paper the authors turn their attention to data structure considerations for Non-Volatile Memory (NVM). Unlike the previous papers I have cov- ered (Mnemosyne and NV-Heaps) they look at data structures specifically optimized to exploit the ca- pabilities of NVM. From the abstract:

*For these systems, where no distinction is made between a volatile and a persistent copy of data, we present Consistent and Durable Data Structures (CDDSs) that, on current hardware, allows program- mers to safely exploit the low-latency and non-volatile aspects of new memory tech- nologies. CDDSs use versioning to al- low atomic updates without requiring log- ging.*

Some other aspects of this paper that stand out:

*They are looking at NVM replacing both DRAM and storage thus, they view this as a single level store. They use version- ing to protect their data structures, ver- sus logging. They describe how to achieve this without hardware changes.*

The paper has a good review of NVM memory technologies that may emerge.

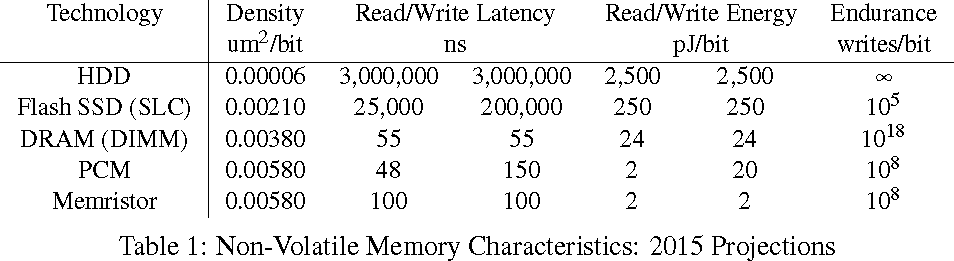


Table 1 (from the paper) underscores the dramatic decrease in latency. This is why weve been waiting for this for more than 10 years now. It really does change the platform in ways that we have not yet realized.

But it is not just the speed aspect that matters, it is also the persistence aspect. The density is much larger for these memories as well. Anyone that has looked at an NVMe M.2 drive can notice how few and small the components on it.

Do we treat it as storage? If so, perhaps we should look to the file systems world for some in- sight. The authors turn to the WAFL shadow page mechanism. They point to BTRFS and their use of this technique with B-trees. They dismiss this ap- proach, concluding that they have fewer data-copies in CDDS. They distinguish this work because it is byte addressable versus prior work that was block oriented (page addressable). Again, the lessons learned from working with it arent directly appli- cable. They do point out that using NVM makes sense in a world of large, persistent storage backed data farms. So there is a need, and one they see fulfilled by NVM. It just needs efficient use of that NVM.

Thus, the authors walk their own path.

The speed of NVM is such that direct access is the only mechanism that makes sense. System calls impose too much overhead, doubling the cost of ac- cessing the NVM itself. Thus they posit that it will be direct access (and indeed that is what seems to come to pass).

They observe that one of the challenges for per- sistent data is that CPUs do not provide mechanisms for ordering persistent writes (though they do, but at a fairly coarse granularity of a fence.) So they de- scribe the issues that must be handled:

* + - Reordering of writes from the caching behav- ior of the CPU itself as well as a multi-level cache hierarchy.
    - Failure semantics of atomic operations across power failures.

They summarize the various approaches available to them for ensuring data has been stored properly in

NVM. This includes memory fences, cache write- back and invalidate operations, marking memory as non-cacheable (which forces write-back), cache line flushes, and atomic processor operations. They point out that this is not sufficient for more complex updates, such as tree rebalancing operation. This leads them to versioning.

I found it interesting that their goals were similar to those I have seen previously: durability, consis- tency, scalability, and ease-of-use for the program- mer. They also note that they focus on physical con- sistency of the data contents in memory. Logical consistency of higher level meta-data structures is not addressed in the context of this work.

Thus, the authors point out that a CDDS is an abstract idea; they demonstrate how they envision using it by implementing a b-tree structure.

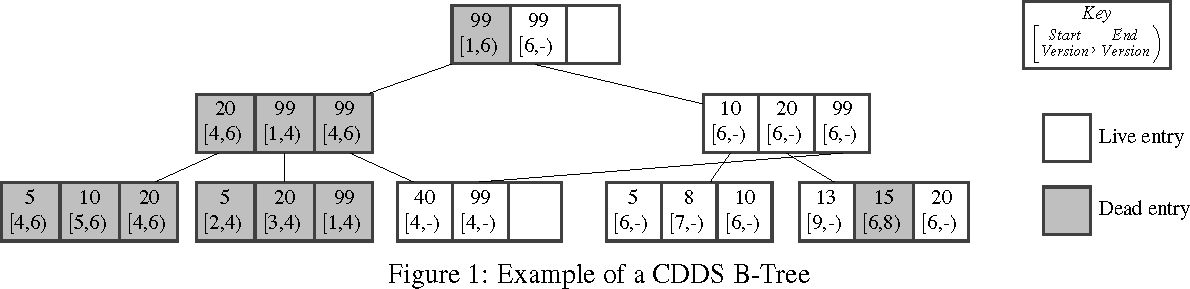


Figure 1 is from the paper as they describe their B-tree.

In versioning, changes are not made in place to the current version; instead, a new version is writ- ten. The current version is immutable (at least as long as it is the current version). Atomic opera- tion and copy-on-write techniques are used to make changes persistent. Once done, a new version num- ber is assigned and the new version becomes the current version. The old version can be recycled once its reference count drops to zero.

Failure recovery then becomes a function of cleaning up any in-progress operations that had not been written to disk.

The authors then walk through their B-tree ex- ample. They explain their use of versioning, they provide pseudo-code for the various B-tree opera- tions (lookup, insert, delete) as well as the internal operations needed to support them.

They evaluate their solution by simulating NVM on top of DRAM (a common solution as we have seen). They compare against BerkeleyDB, STX B- Tree, Tembo, Cassandra, and Redis. They were slower than the entirely in-memory STX B-Tree, presumably due to the cost overhead. They are

much faster than BerkeleyDB (even BerkeleyDB runing on a RAM disk.) They also tested using YCSB as their end to end benchmark.

In the end, they do demonstrate that it is possible to rebuild existing data structures preserving the in- terface so they work efficiently with NVM. They even point out it does not require processor changes to do so. Given that better processor cache con- trol mechanisms have been introduced since then, I would expect that exploiting them will lead to even better performance.

# Cache Craftiness for Fast Multi- core Key-Value Storage [[40]](#_bookmark18)

This paper does not deal with non-volatile mem- ory; it deals with key-value stores. However, it is applicable to some of the considerations of NVM because it organizes data for *cache optimal- ity*, which is an important consideration when look- ing at NVM.

*MassTree* is the name of the authors’ key-value store. In this case, it focuses on utilizing a “trie- like concatenation of B+-trees”. Each of these data structures handles a fixed length slice of the variable-length key, which permits support of ar- bitrary length keys and the keys can be arbitrary data. Note that keys may have shared prefixes in this scheme as well; such shared keys will utilize com- mon data structures (hence the “trie-like” aspect). The B+-tree fanout is chosen to minimize memory delay when walking the tree and it utilizes prefetch- ing for individual nodes.

Key aspects of MassTree:

* + - Supports range queries over keys, due to the organizational structure.
    - Lookup is performed unlocked
    - Wide-fanout trees to minimize DRAM fetch time
    - Node prefetching from DRAM
    - Optimize data layout in a cache line aware fashion, e.g., “collocate important information on a small number of cache lines”
    - Persistence is obtained using batch logging and checkpointing
    - Avoid cache line pollution from **get** operations (“... its get operation must never dirty shared cache lines by writing shared data structures”)

Coordination in MassTree is carefully crafted: this is a shared memory system and they use tra- ditional mutual exclusion for interlocking activity across cores. The paper clearly lays out their lock- ing rules, which are non-naive. They define a strict locking hierarchy because they have cases that must hold multiple locks simultaneously.

They indicate they looked at other locking mech- anisms, including compare-and-swap (CAS), and found that it did not perform better. “[T]he major cost of locking, namely the cache coherence proto- col, is also incurred by lock-free operations...”

For in-place updated, they use aligned write op- erations, noting that these have atomic effect. The utilize read-copy-updte (RCU) techniques for han- dling garbage collection for deleted values. To pre- vent issues with reordering keys in a node (they are sorted) they utilize this same aligned write tech- nique to write a “permutation” field within the node (See §4.6.2 of the paper for details).

Thus, their system has lock-free **get** operations. They use a combination versioning/hazard mecha- nism to permit readers to detect a state change and retry the operation (a simple form of software trans- actional memory, perhaps?)

The paper has an interesting discussion about per- formance tuning across machines of the same archi- tecture (§4.8) that points out how optimizations can be very CPU specific, as they describe difference in performance optimization between Intel and AMD based multi-core systems.

They evaluate the contribution of each feature to the overall performance of the final system, scal- ability of the system, impact of partitioning, and traditional comparison measurements against exist- ing systems (Mongo, Volt, memcached, and Redis). They compare favorably.

One interesting observation on their evaluation: they chose to evaluate on 16 cores, despite having 48 cores available on their machine. Their rationale

for doing this seems dubious to me (“The goal is to mimic the configuration of a machine more like those easily purchasable today.”)

Their performance does better than the compet- ing systems, in some cases by a factor of 5x, except for the 16 core **get** where they are slightly slower than memcached. An interesting note here is that the Anna paper [(§1.13)](#_bookmark4) reports much lower results for MassTree than are reported here.

The other interesting note here is that they found the performance of MassTree was consistently sus- tained *even in the face of significantly skewed work- load*. It made me curious how Anna (which is shared nothing) would do with such a workload.

# Logic and Lattices for Distributed Programming [[11]](#_bookmark9)

This is definitely a different direction than weve had in prior papers, though I do have an ulterior motive in presenting this particular paper we will see it used later.

*In recent years there has been interest in achieving application-level consistency criteria without the latency and availabil- ity costs of strongly consistent storage in- frastructure. A standard technique is to adopt a vocabulary of commutative op- erations; this avoids the risk of inconsis- tency due to message reordering. Another approach was recently captured by the CALM theorem, which proves that logi- cally monotonic programs are guaranteed to be eventually consistent. In logic lan- guages such as Bloom, CALM analysis can automatically verify that programs achieve consistency without coordination. In this paper we present BloomL, an ex- tension to Bloom that takes inspiration from both of these traditions. BloomL generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lat- tices. We show how the Bloom inter- preter can be generalized to support effi-*

*cient evaluation of lattice-based code us- ing well-known strategies from logic pro- gramming. Finally, we use BloomL to develop several practical distributed pro- grams, including a key-value store simi- lar to Amazon Dynamo, and show how BloomL encourages the safe composi- tion of small, easy-to-analyze lattices into larger programs.*

Notice they do mention key-value stores, so you have another hint on how Ill be referring back to this work in a future post.

This tends more to the theoretical side of sys- tems. It is not a theory paper (there just isnt enough formalism, let alone proofs!) It has performance graphs, which you certainly expect from a systems paper, but not from a theory paper.

The driving factor behind this is the issue of dis- tributed consistency. At a high level, distributed consistency is concerned with ensuring that a group of communicating computers, with some temporal separation, agree on the outcome of operations even when things go wrong. Perhaps the most famous example of distributed consistency is Paxos. These days we refer to these as consensus protocols. I gen- erally describe there being several such: two-phase commit is certainly one of the older ones. Quorum protocols are another (e.g., weighted voting, which I described previously). Viewstamped Replication is another. These days, the popular consensus pro- tocols are Raft and Blockchain.

The paper starts by pointing out that monotonic consisency provides a valuable mechanism for rea- soning about distributed consistency. Prior work by the authors establishes that all monotonic pro- grams are invariant to message reordering and retry, a property they call confluent. This matters for dis- tributed systems because such a system only moves forward (the operations are durable.)

They point out some weaknesses in the prior def- inition and motivate improving it by explaining one such obvious case that does not fit within the model (a voting quorum in a distributed protocol.)

Hence, they introduce the lattice. They do this within the context of their language (Bloom*L*),

which works on top of Ruby. I will not dwell on the details.

The authors define a *bounded semijoined lattice*. My reading of what they are saying is that in such a set, there is a unique element that happened first. They define this formally as a set S, with an operator

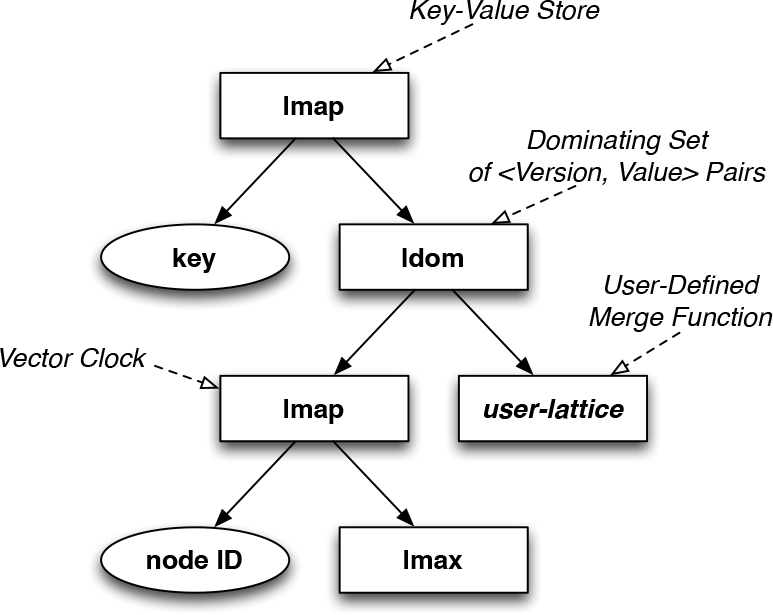
LJ that defines a partial ordering. There is a unique element ⊥ that represents the least element.

From this definition, they construct their model; the paper drops the bounded semijoined part of the definition and simply discusses lattices from that point forward, but it is this partial ordering property that imparts the key characteristics to their subse- quent operations.

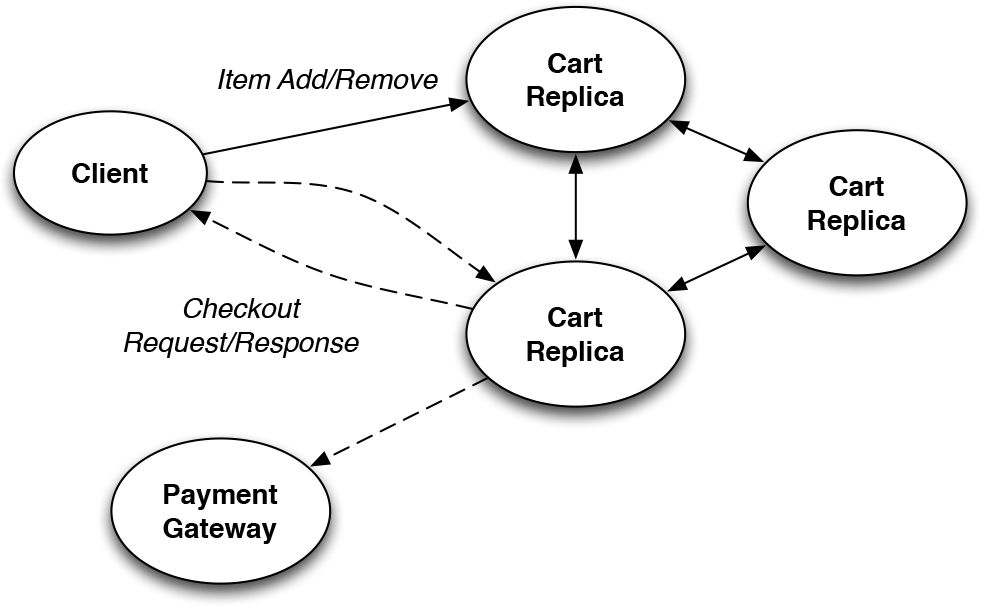
Why is this important? Because it demonstrates that these lattices which are going to turn out to be equivalent to key operations in distributed systems have consistency guarantees that are desirable.

The authors then turn their attention to utiliz- ing lattices for key-value stores. They describe data structure versioning and vector clocks. Vec- tor clocks have a property they desire for lattices: they are partially ordered. They combine this with a quorum voting protocol, to provide the distributed consensus for their system.

Figure 8 (from the paper) shows the general structure of their key-value store implementation, which is implemented in Bloom*L* and Ruby:



Their sample usage for this is a shopping cart, which they graphically describe in Figure 9 (from the paper):



As one would expect in a distributed system, the key benefit here is that there is no centralized au- thority deciding on the order of things. They point out that prior work argues shopping carts are non- monotonic and thus cannot be solved in a distributed systems setting. The authors point out that using the lattice structure, they achieve a monotonic ordering, which permits them to implement it without a cen- tralized decision maker; in fact the decision maker in this case is really the client itself, as it has all the information from all the servers sufficient to com- plete the operation. While a shopping cart might not be the killer application for a distributed systems technology, this paper does describe a powerful tool for providing distributed consensus in a system that can be implemented in a modest amount of code; compared to Paxos, Raft, or Viewstamped Replica- tion, that is a significant contribution.

It does not appear to have byzantine protection, however, so if you live in a hostile environment it might not be the right protocol. Similarly, if you need stronger consistency guarantees, this might not be the best model either. But for many applica- tions slightly relaxed consistency guarantees are of- ten more than adequate.

We will see how this can be applied in the future.

# SILT: A Memory-Efficient, High-Performance Key-Value Store [[34]](#_bookmark17)

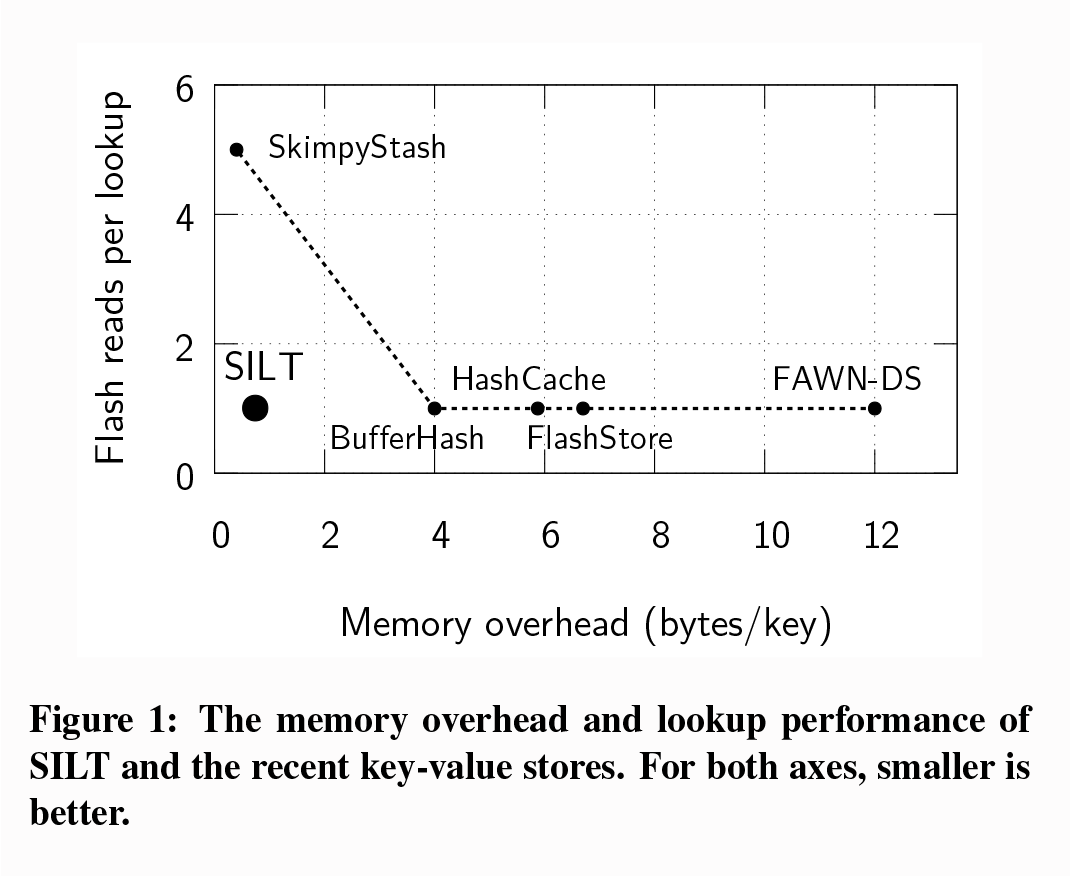
In this paper, we have an interesting hybrid key- value store; part of the store is in DRAM, while the rest of it is stored in flash storage. The authors have focused on optimizing the performance of the

store across the specific media chosen, which leads to some interesting choices and observations.

*SILT (Small Index Large Table) is a memory-efficient, high performance key- value store system based on flash stor- age that scales to serve billions of key- value items on a single node. It requires only 0.7 bytes of DRAM per entry and retrieves key/value pairs using on aver- age 1.01 flash reads each. SILT com- bines new algorithmic and systems tech- niques to balance the use of memory, stor- age, and computation. Our contribu- tions include: (1) the design of three ba- sic key-value stores each with a differ- ent emphasis on memory-efficiency and write-friendliness; (2) synthesis of the ba- sic key-value stores to build a SILT key- value store system; and (3) an analyti- cal model for tuning system parameters carefully to meet the needs of different workloads. SILT requires one to two or- ders of magnitude less memory to provide comparable throughput to current high- performance key-value systems on a com- modity desktop system with flash storage.*

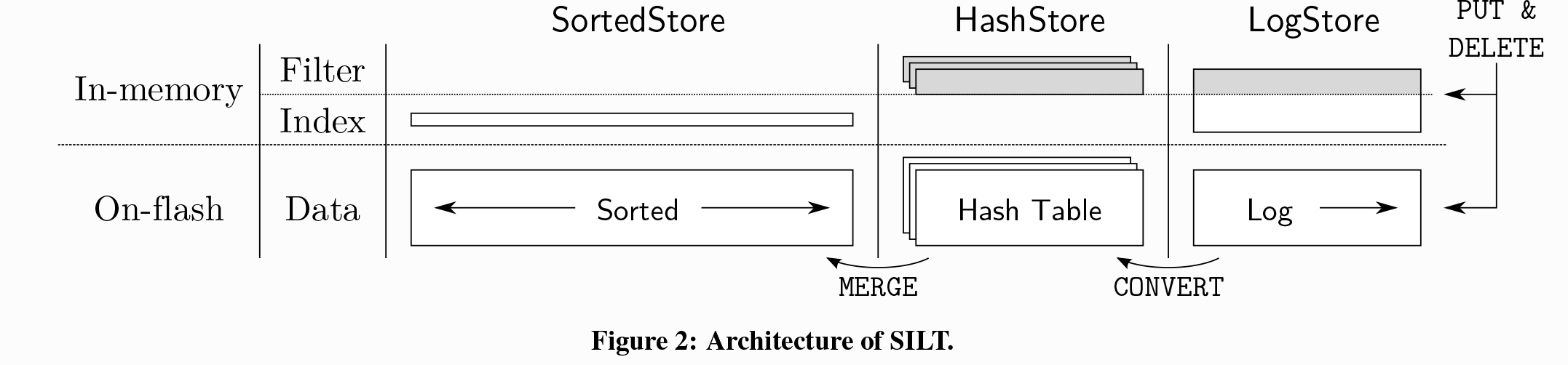
Thus, to achieve optimal performance they con- struct ahybrid ensemble of key-value stores that are optimized to particular underlying media.

They start early with their performance observa- tions, and continue this pace throughout the paper. Thus, they point out that they have a small mem- ory requirement (0.7 bytes per key value) and only a single read from flash required:



This fits well with their observation that memory efficiency is critically important for scalable key- value stores. Their system saturated the capabilities of their hardware with 46,000 lookups per second, though this doesn’t guarantee that it will do so on newer, faster hardware (and we will get to several current KV stores.

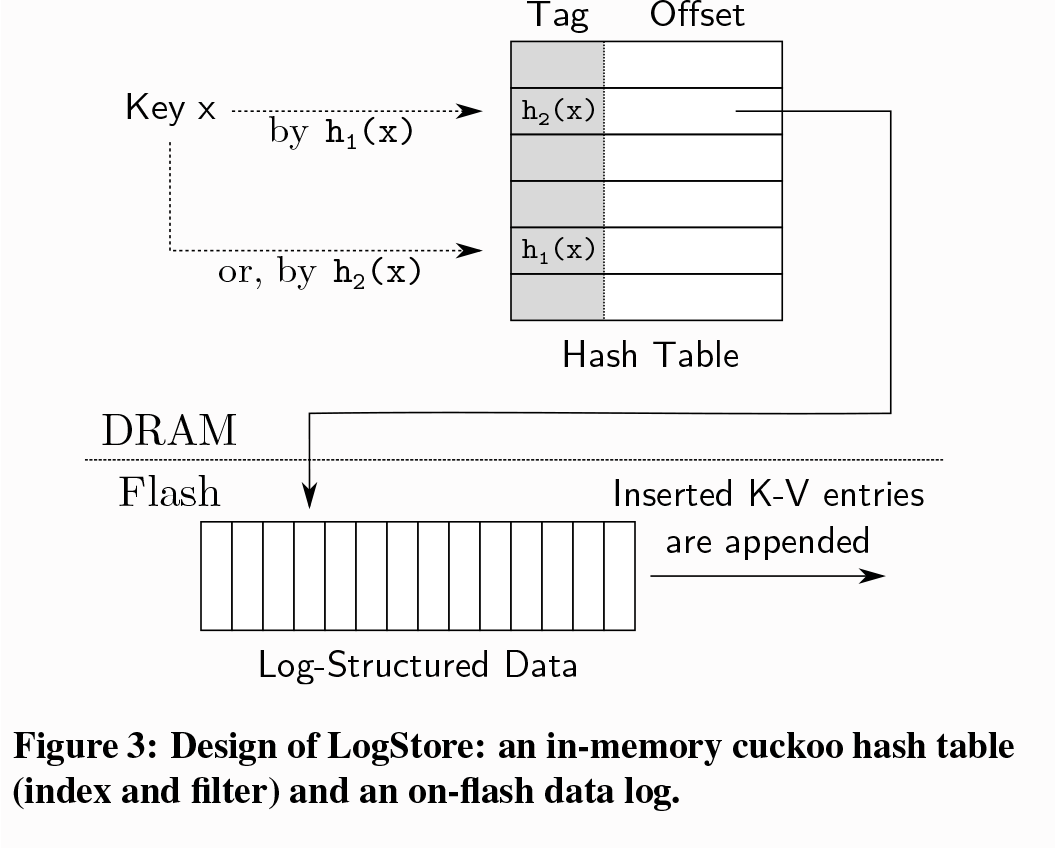
Figure 2 (from the paper) provides a basic de- scription of the SILT architecture.From this, we can see the three distinct types of stores that are used within their implementation:



* + - **LogStore** - uses a CPU-efficient cuckoo-hash based only upon atag derived from the full key (20 bytes). The benefit of this approach is that it avoids accessing the flash memory if the key is not present. It is probabilistic, in that a key might not be present in Flash with some low probability.
    - **HashStore** - uses a memory-efficient storage model for their cuckoo flash table. These stores areimmutable and are stored in flash.
    - **SortedStore** - They use anentropy-coded trie, which provides a very space efficient index- ing mechanism: the average according to the authors is 0.4 bytesper key. Given that keys are 160 bits (20 bytes), which means the in- dex is 64 bits (8 bytes). The authors provide

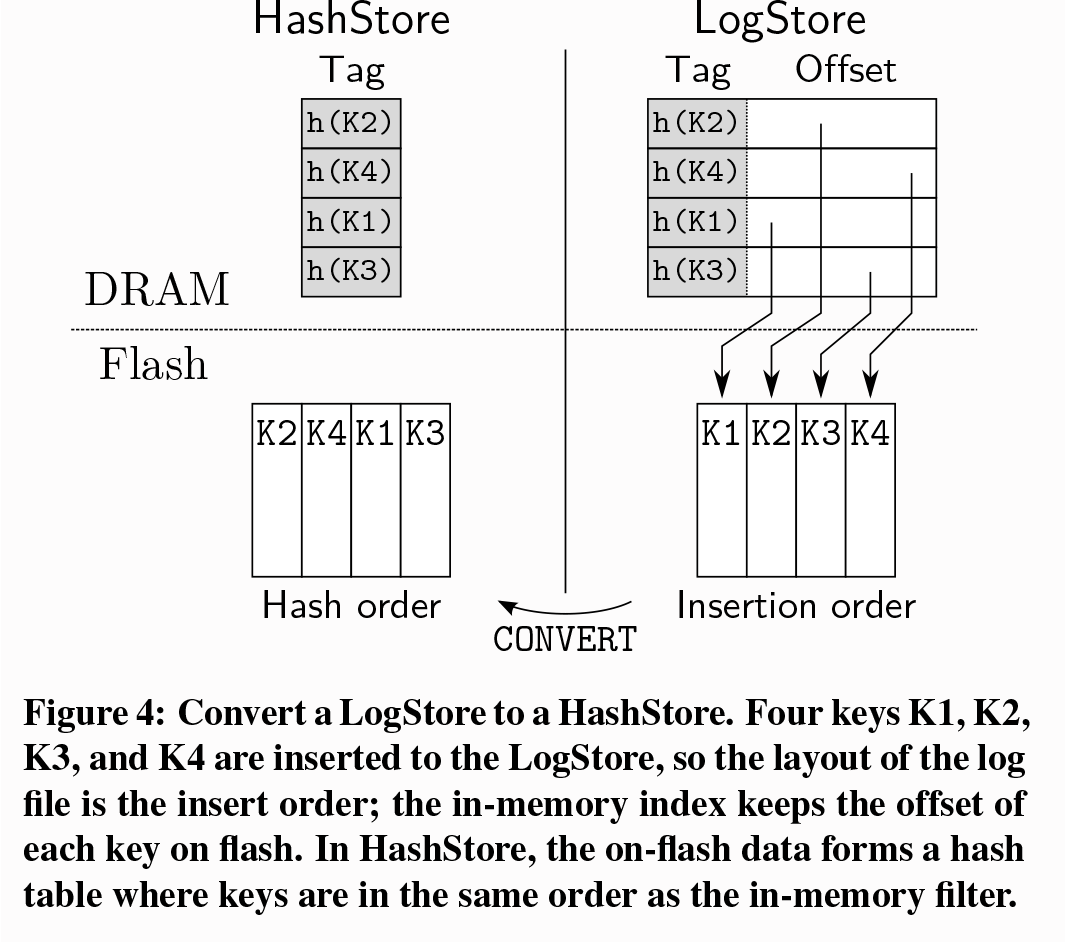
us with an impressive list of specific optimiza- tions they have applied to sorting data, pack- ing indexes, intelligent (non-pointer) based ad- dressing,

They also provide us with pretty figures to graph- ically lay out their data structures. First, Figure 3, which shows the in-memory cuckoo hash table:



Note the fact the key-value insertions are logged to the flash storage.

When we move ontoHashStore, they show us how the structure of the data is moved from inser- tion order to hash sorted order as the data is moved from memory to flash.



Recall that once written to Flash, theHashStore is immutable - so this order will remain fixed.

Of course, one of the important aspects of a key- value store is the ability tolook up the data quickly. Thus, they also show us how they sort data using a

prefix tree (Trie). This is a key aspect of theirSort- edStore:

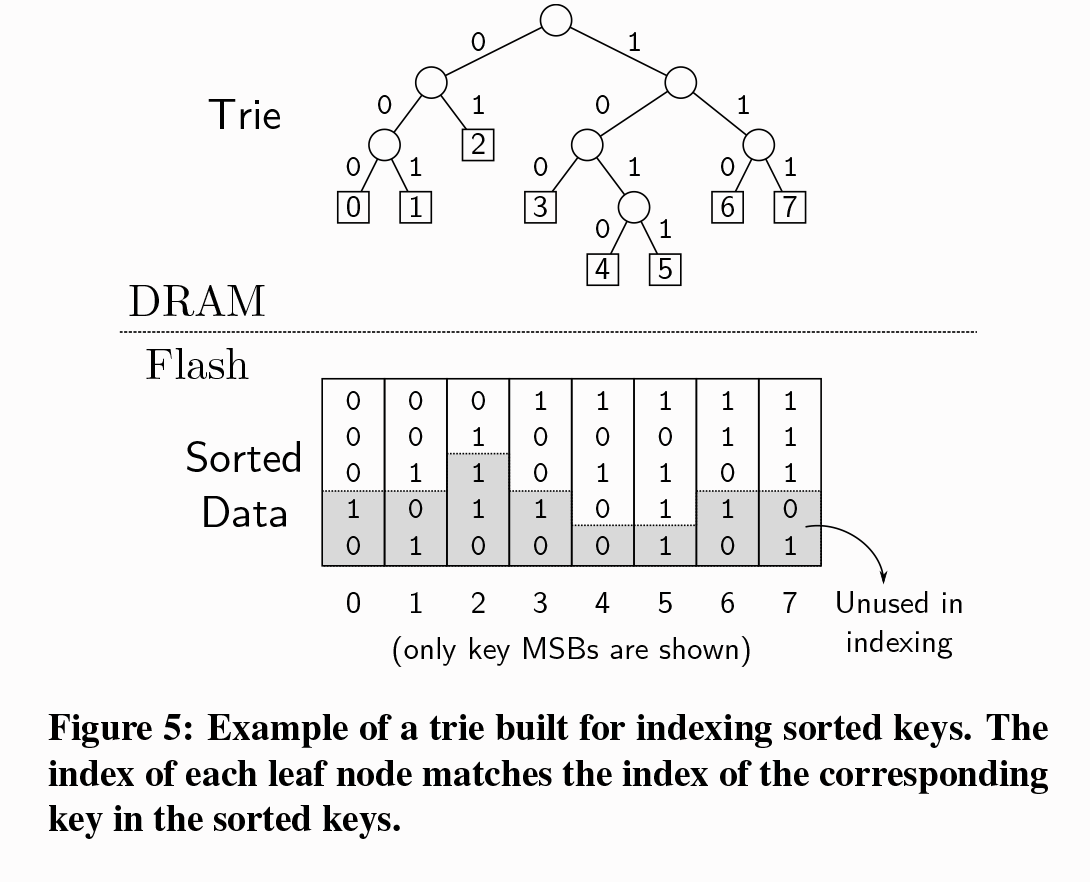
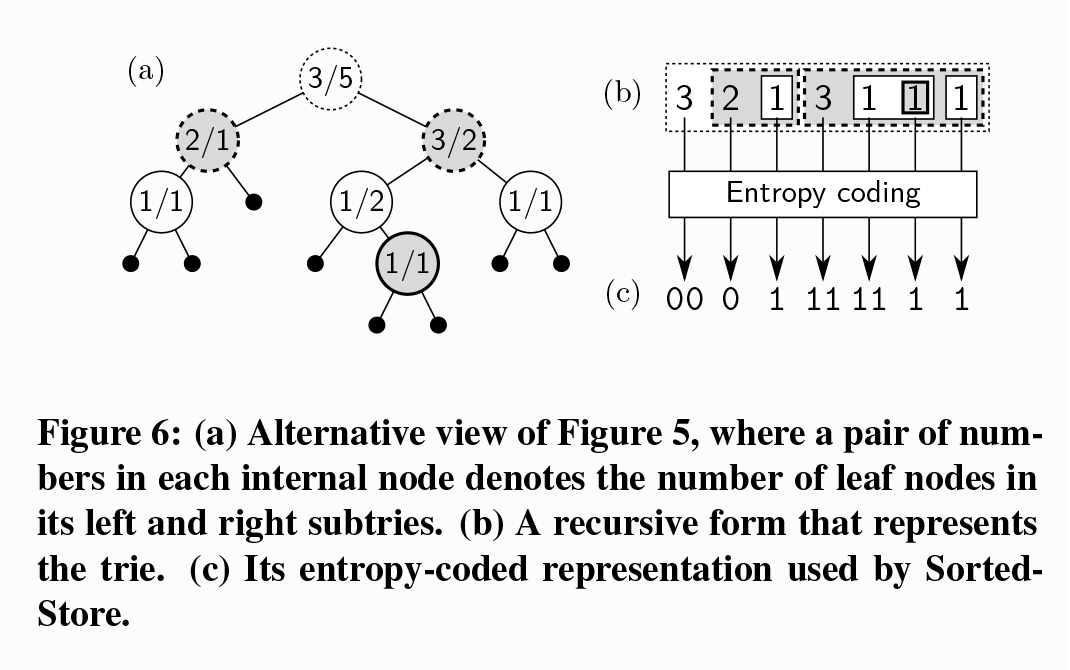
The in-memory Trie, permits them to rapidly find the corresponding data in Flash. This is one of their key motivations for very efficient encoding of the Trie: it ensures they can store it in memory. Once they can no longer store the indices in memory the lookup system will become I/O bound (to the Flash, in this case).

Figure 6 then shows how they actually encode the length of the tree in each direction, and how it is entropy encoded:



There is some serious thought that has gone into their layout. They do admit, however, that they are not providing crash tolerance: there will be data that was stored in DRAM that cannot be recovered after a crash. They do not discuss the use of NVM for achieving this. Instead, they suggest a synchronous write mechanism in which the write blocks until the log data has been written to the flash-based log.

One of the interesting aspects of their perfor- mance evaluation is the recognition that they have quite a few parameters that must be chosen. Thus,

they spend more than a page of the paper just discussing the relative merits of the trade-offs in- volved in selecting various parameters. They con- sider**write amplification** (the need to write more than just the data being preserved),read amplifica- tion(performing reads only to discover the data that you seek is not present),**memory overhead**(how much DRAM is used), andupdate rate versus flash life time. This latter point is an interesting one to keep in mind when reading about non-volatile memory: it*wears out*. There is a certain amount of degradation that occurs in the underlying physical medium each time it is erased and re-used. The spe- cific details are dependent upon the characteristics of the medium, but itis an issue to keep in mind.

The evaluation of SILT is certainly interesting. They choose not to compare to any other KV system and instead focus on the performance of their sys- tem using a variety of fairly standard loads (YCSB) as well as simple get/put key operations as micro- benchmarks. They end up claiming they they can lookup almost 45k keys per second in theslowest of their hybrid stores. When combined, they indicate they can achieve almost 58k**get** operations per sec- ond on a single core. As they move to multiple cores they see some further improvement (180k**get** opera- tions per second for data that is already in memory). SILT offers some interesting observations, par- ticularly about highly efficient use of memory and

novel indexing schemes.

# Exploiting Hardware Transac- tional Memory in Main-Memory Databases [[32]](#_bookmark15)

I have not spent much time discussing transactional memory previously, though I have touched upon it in prior work. By the time this paper was presented, transactional memory had been fairly well explored from a more theoretical perspective. Intel hardware with transactional memory support was just starting to emerge around the time this paper was released. I would note that Intel had substantial challenges in getting hardware transactional memory (HTM) cor- rect as they released and then pulled support for it in several different CPU releases. Note that HTM

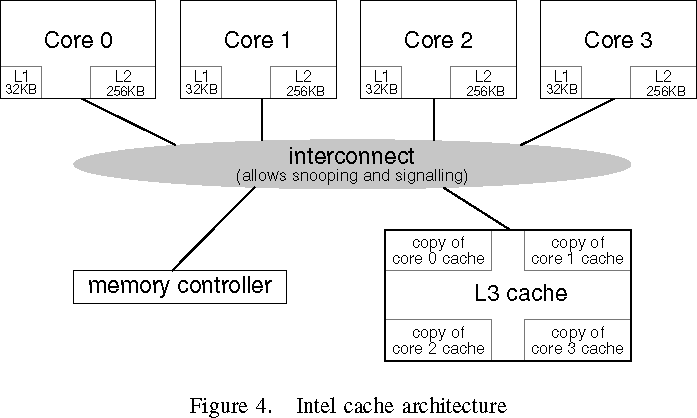
was not new, as it had been described in the litera- ture to an extent that earlier papers (e.g.,Virtualizing Transactional Memory, which I have decided not to write about further, discusses the limitations of HTM back in 2005).

Logically, it extends the functionality of the pro- cessor cache by tracking what is accessed by the processor (and driven by the program code). Cache lines are read from memory, any changed made to the cache line, and then written back to memory. This is in turn all managed by the cache coherency protocol, which provides a variety of levels of co- herency.

The idea behind HTM is thatsometimes you want to change more than a single element of memory. For example, you might use a mutual exclusion, then add something to a linked list, and increment a counter indicating how many elements are in the linked list before you release the mutual exclusion. Even if there is no contention for the lock, you will pay the lock cost. If the platform requires a fence operation (to ensure memory has been flushed prop- erly) you will also stall while the memory is written back. In a surprising number of cases, you need to do multiple fences to ensure that operations are se- quentially consistent (which is a very strong form of consistency).

With HTM you can do this all speculatively: start the transaction, add something to the linked list, in- crement the counter, thencommit the transaction. Once this has been followed with an appropriate fence, the change is visible to all other CPUs in the system. Thegoal then is to avoid doing any memory operations unless absolutely necessary.

The authors point out that thefastest option is partitioning (ignoring hot spots). They graphically demonstrate this in Figure 1 (from the paper):



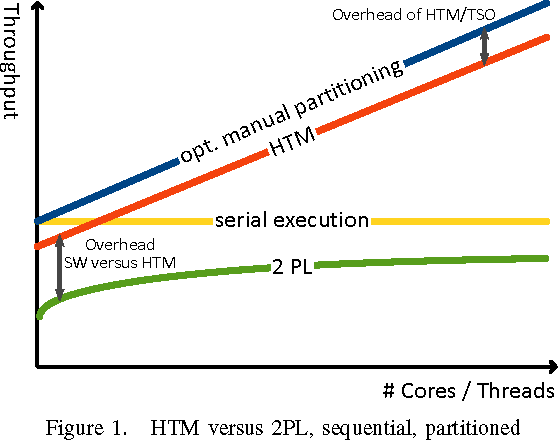
HTM hassomeoverhead, but it tracks with par- titioning fairly linearly. This difference is theover- head of HTM.

They compare this toserial execution, which just means performing them one at a time. The tra- ditional mechanism for doing this kind of par- alleism is thetwo phase commit protocol. That’s the lock/work/unlock paradigm.

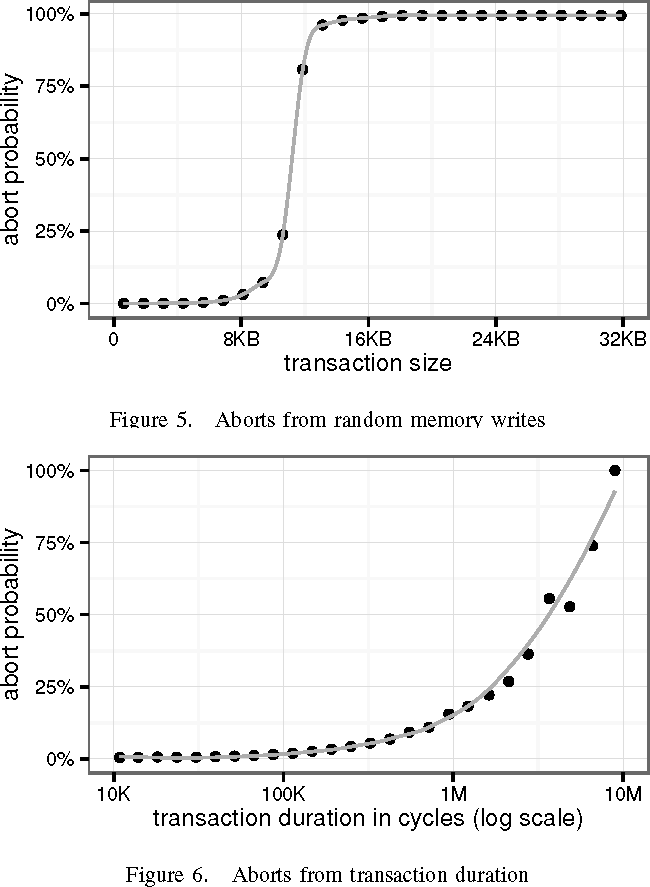
If we only considered this diagram, we’d stick with strong partitioning - and we’re going to see this observation reflected again in future work. Of course thereason we don’t do this is because it turns out that the database (and it shows up in file sys- tems as well) is not being uniformly accessed. In- stead, we havehot spots. This was a particular con- cern in the MassTree paper, where they supported novel data structures to spread the load around in a rather interesting fashion. There’s quite a bit of discussion about this problem in the current paper - “[A] good partitioning scheme is often hard to find, in particular when workloads may shift over time.” Thus, their observation is: “we have to deal with this problem”.

So, how can HTM be exploited to provide ro- bust scalability without partitioning. The authors do a good job of explaining how HTM works on Intel platforms. Figure 4 (from the paper) shows a fairly standard description of how this is done on the Intel platform:

It has a bus snooping cache, an on-chip memory management unit (MMU), a shared Level 3 cache, and percore Level 1 and Level 2 caches (in case you are interested, the two caches do have somewhat dif- ferent roles and characteristics.) Level 1 cache is the fastest to access, but the most expensive to provide. Level 2 cache is slower than Level 1, but because it is alsocheaper we can have more of it on the CPU. Level 3 cachemight be present on the CPU, in which case it is shared between all three cores. Note that none of this isrequired. It just happens to be how CPUs are constructed now.

The benefit of HTM then is that it exploits the cache in an interesting new way. Changes that are made inside a transaction arepinned inside the cache so they are not visible outside the current core. Note, however, that this could mean just the L1 cache. In fact, the functional size permitted is even smaller than that, as shown in Figure 5 (from the paper). Transactions below 8KB have a low probability of aborting (and if it aborts, the operation failed so it must be tried again, either using HTM or the fallback mechanism with software). That probabil- ity approaches 100% as the size goes above above 8KB. Interestingly, the primary reason for this is not so much thesize of the cache as theassociativ- ity of the cache. What that means is the cache uses some bits from the address to figure out where to store data from that particular cache line. The paper points out that 6 bits (7-12) are used for determining the cache location, and each cache location (so each unique value of bits 7 through 12) are has a fixed number of cache lines (e.g., 8 entries in the Haswell chips the authors are evaluating). If we need to use a ninth, we evict one of the existing pages in the cache.

Similarly, when theduration of the transaction goes up, the probability of it aborting also rises. This is shown in Figure 6 (from the paper).



This is because the chance that various systems events will occur, which cause the transaction to abort. This includes various types of interrupts: hardwareand software.

Thus, these two graphically demonstrate that to exploit HTM effectively we need to keep our trans- actions small in both duration and the number of cache lines modified by them.

Wealso note that we should take steps to mini- mize the amount of sharing of data structures that might be required - the point that not sharing things is more efficient. The authors discuss a variety of approaches to this issue: segmenting data struc- tures, removing unnecessary conflict points (e.g., counters), and appropriate choice of data structures.

Recall the Trie structures fromMassTree?These authors offer usAdaptive Radix Trees, which seem to have a similar goal: they are “[A]n effi- cient ordered indexing structure for main memory databases.” They combine this with a spin lock; the benefit now is that HTM doesn’t require the spin lock normally, so even if some parts of the tree are

being read shared, the lock is not being acquired and thus it does not force a transactional abort for other (unrelated) nodes.

They put all of this insight together and that forms the basis for their evaluation. Figure 11 in the paper makes the point that HTM scales much better than traditional locking for small lookups (4 byte keys) with a uniform distribution once there is more than one thread:

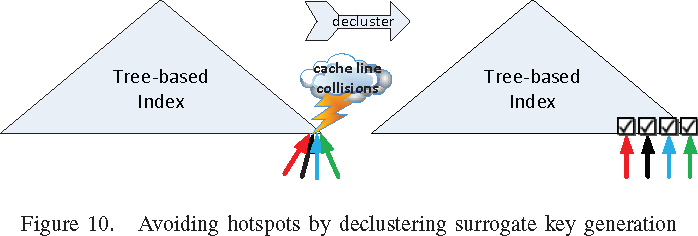


Figure 12 (from the paper) evaluates theTPC-C Benchmarkagainst their resulting system to demon- strate that it scales well . Note they stick with four threads, which are all likely on a single physical CPU, so there are no NUMA considerations in this aspect of the evaluation. They address this a bit later in the paper.

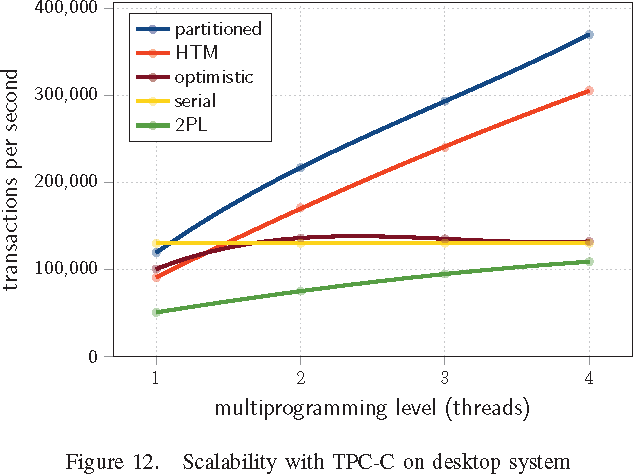


Figure 13 (from the paper) compares their per- formance against apartitionedsystem. Because they cannot prevent such cross-partition access, they must “live with” the inherent slowdown. One of the amazing benefits of HTM is thus revealed: as more operations cross partition boundaries, HTM contin- ues to provide a constant performance. This seems to be one of the key lessons: no sharing is great, but once you find that you must share, synchronizing optimistically works surprisingly well.

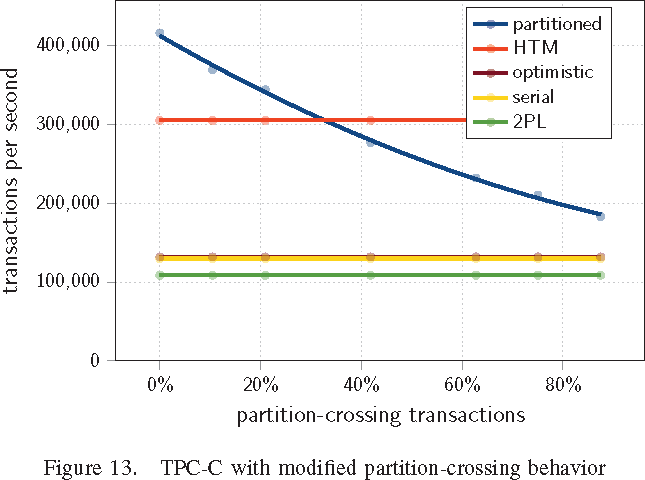
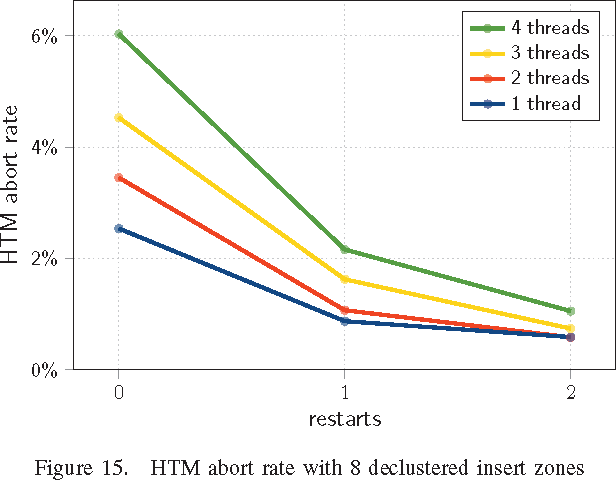
 

Figure 14 (from the paper) attempts to address my comment earlier abut Figure 12: they really don’t have a multiprocessor system under evalua- tion. They admit as much in the paper:the hardware just isn’t available to them. They provide their sim- ulation results to defend their contention that this does continue to scale, projecting almost 800,000 transactions per second with 32 cores.

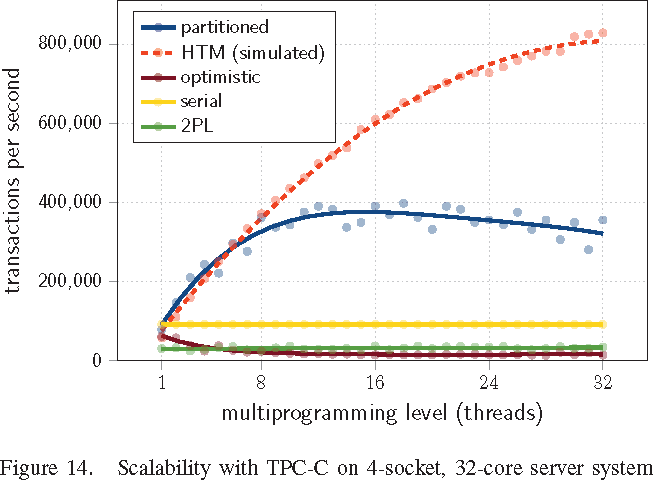


Figure 15 (from the paper) finally demonstrates the reproducibility of HTM abort operations. If an HTM isretried, many will complete with one or two tries. Thus, it seems that even with multiple threads, they tend to converge towards thehardware limita- tions.

Bottom line: hardware transactional memory can be a key aspect of improving performance in a shared memory systems with classical synchroniza- tion.

# Nvmcached: An nvm-based key- value cache [[69]](#_bookmark28)

* 1. **Anna: A KVS For Any Scale** [**[68]**](#_bookmark27)

1. **Take-Aways**

From the prior literature, here are important take- aways to consider in a new design:

* Cache exploitation is an important aspect of performance. [(§1.2)](#_bookmark0)
* NUMA awareness is important [(§1.8)](#_bookmark3)
* Minimizing data amplification is beneficial. [(§1.3)](#_bookmark1)
* NVM can permit very fast persistence [(§1.5)](#_bookmark2)

# Historical

This section includes historical information that may be useful in understanding the context.

# Research Proposal

My original application included a research pro- posal that has provided the initial direction of my re- search. While the direction has continued to evolve,

I capture that original text here; it has been refor- matted to fit the current document structure.

## Introduction

My research interests have revolved around operat- ing systems and ways to improve them, either by identifying inefficiencies and correcting them or by identifying services they could provide to enhance their usefulness. As computer hardware continues to evolve, operating systems must also evolve, both to support those new hardware features as well as utilize them to provide both robustness and new generally useful features.

## Summary of Previous and Ongoing Work

One area in which I have worked is the area of file systems. This is generally considered to be a ma- ture field, yet my own experience suggests there are opportunities for innovation in this area, whether it is utilizing new storage modalities such as Solid State Disks (SSDs), Memory-class storage, or en- hanced security. For example, the concept of a layered file system allows integration of new func- tionality on top of existing file systems. [[16,](#_bookmark11) [17]](#_bookmark12) In DCE/DFS [[24]](#_bookmark14) we used a vnode operations over- lay technique because UNIX operating systems had no model for layering functionality. Linux sup- ports a stackable file system interface [[72]](#_bookmark29) though it does not appear to be heavily utilized. Similarly, Windows NT provided a model for layered drivers which can be used for implementing layered file systems that I have used and taught to others for over 20 years. [[62]](#_bookmark25)

File systems are notoriously difficult to replace. Windows has NTFS, first introduced in 1993 and fully adopted as the default file system in 2001 as part of Windows XP. Linux typically uses ext3 as its default file system. Newer file systems certainly do exist (ZFS, ReFS) but are typically not the boot file system and not used on the desktop.

This makes innovation in file systems challeng- ing, despite the rapidly changing nature of storage in typical hardware platforms. When I helped de- velop the Episode File System [[8]](#_bookmark7) for example, we focused on providing a number of features which,

at the time, were novel. The ability to support block level copy-on-write, POSIX security, quotas, exten- sible properties, and journaling were all state-of-the art. A large disk drive was 1GB and we postulated that we might need to deal with drives up to 1TB in size.

Systems have changed. A single rotating me- dia device can provide 8TB of storage. Multi- TB solid state drives are commercially available. Further challenging the traditional file system lay- ered on top of storage is the Object Storage Device (OSD) model. These devices challenge the funda- mental idea that unstructured storage should be or- ganized in a hierarchical fashion. Despite this, oper- ating systems continue to cling to the file/directory metaphor. Some of this is structural: the VFS layer is inherently hierarchically structured. The OS in- terfaces presume that data is organized in a hier- archical fashion. Hierarchical organization breaks down even at the level of a single computer system. In my own work I routinely find that locating infor- mation is a challenge because I do not know where I stored that information. There are a number of solutions to this, including localized search mech- anisms that index content, but they suffer from a variety of problems, notably performance. There is a modest body of research in this area but few of these solutions investigate ways in which the lo- cal file system within the OS itself can be enhanced to provide native support for the contextual clues that are so useful for supporting search-friendly in- terfaces. [[42,](#_bookmark19) [57,](#_bookmark22) [7]](#_bookmark6)

## Future Research

Undertaking any file systems work is challenging. Because I have extensive experience in this area, my plans for future research are to look at ways in which file systems can be augmented or replaced entirely to provide native support for efficient sup- port of data management. I can envision a number of potential research projects in this area:

* + - 1. Layering additional attribute functionality on top of an existing file system. In this model, rather than replace the native file system, a sep- arate component would be used to interpose it-

self between the native file system. It would take over some aspects of the organizational interface of the underlying file system in or- der to implement the augmented functional- ity. Because the existing file system remains present, this will limit functionality somewhat, but will be compatible with existing systems. I have used this approach before when imple- menting transparent per-file encryption, for ex- ample, where without the layered component the file is still present but the data is encrypted.

* + - 1. Designing and developing a new file system that provides OSD-like features, a program- ming interface that can be integrated with higher level indexing facilities, and provide a backwards compatible hierarchical structure.
      2. Evaluating specific indexing systems in order to establish efficient ways of augmenting the file system to support them. For example, using the Linux ext4 file system but adding explicit additional support for attribute tag- ging. Evaluating models for attribute informa- tion and how they can be efficiently supported by the file system will provide further insight into what native file systems can to do effi- ciently provide this functionality.

# References

1. ARULRAJ, J., AND PAVLO, A. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 1753–1758.
2. ARULRAJ, J., PAVLO, A., AND DULLOOR, S. R. 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* (2015), ACM, pp. 707–722.
3. CHEN, A. A review of emerging non-volatile memory (nvm) technologies and applications. *Solid-State Electron- ics 125* (2016), 25–38.
4. CHEN, G., ZHANG, L., BUDHIRAJA, R., SHEN, X., AND WU, Y. Efficient support of position independence on non-volatile memory. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitec- ture* (New York, NY, USA, 2017), MICRO-50 ’17, ACM, pp. 191–203.
5. CHEN, S., AND JIN, Q. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment 8*, 7 (2015), 786–797.
6. CHEN, X., SHA, E. H.-M., ABDULLAH, A., ZHUGE, Q., WU, L., YANG, C., AND JIANG, W. Udorn: A design framework of persistent in-memory key-value database for nvm. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2017 IEEE 6th* (2017), IEEE, pp. 1– 6.
7. CHOU, J. *FindFS: adding tag-based views to a hierarchi- cal filesystem*. PhD thesis, University of British Columbia, 2015.
8. CHUTANI, S., ANDERSON, O. T., KAZAR, M. L., LEV- ERETT, B. W., MASON, W. A., SIDEBOTHAM, R. N., ET AL. The episode file system. In *Proceedings of the USENIX Winter* (1992).
9. COHEN, N., FRIEDMAN, M., AND LARUS, J. R. Efficient logging in non-volatile memory by exploiting coherency protocols. *Proc. ACM Program. Lang. 1*, OOPSLA (Oct. 2017), 67:1–67:24.
10. CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceed- ings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 133–146.
11. CONWAY, N., MARCZAK, W. R., ALVARO, P., HELLER- STEIN, J. M., AND MAIER, D. Logic and lattices for dis- tributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing* (2012), ACM, p. 1.
12. CRUCIAL. Speed vs latency: Why cas latency isn’t an accurate measure of memory performance, 2017. <http://www.crucial.com/usa/en/memory-performance-> speed-latency (accessed April 16, 2018).
13. FAN, B., ANDERSEN, D. G., KAMINSKY, M., AND MITZENMACHER, M. D. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (2014), ACM, pp. 75–88.
14. FRIEDMAN, M., HERLIHY, M., MARATHE, V., AND PE- TRANK, E. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Sym- posium on Principles and Practice of Parallel Program- ming* (New York, NY, USA, 2018), PPoPP ’18, ACM, pp. 28–40.
15. GILES, E., DOSHI, K., AND VARMAN, P. Continuous checkpointing of htm transactions in nvm. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management* (2017), ACM, pp. 70–81.
16. HEIDEMANN, J. S., AND POPEK, G. J. *A layered ap- proach to file system development*. Citeseer, 1991.
17. HEIDEMANN, J. S., AND POPEK, G. J. File-system devel- opment with stackable layers. *ACM Transactions on Com- puter Systems (TOCS) 12*, 1 (1994), 58–89.
18. HERLIHY, M., AND MOSS, J. E. B. *Transactional mem- ory: Architectural support for lock-free data structures*, vol. 21. ACM, 1993.
19. HSU, T. C.-H., BRU¨ GNER, H., ROY, I., KEETON, K., AND EUGSTER, P. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys ’17, ACM, pp. 468–482.
20. HU, X., WANG, X., LI, Y., ZHOU, L., LUO, Y., DING, C., JIANG, S., AND WANG, Z. Lama: Optimized locality- aware memory allocation for key-value cache. In *USENIX Annual Technical Conference* (2015), pp. 57–69.
21. HUANG, K., ZHOU, J., HUANG, L., AND SHEN, Y. Nvht: An efficient key-value storage library for non- volatile memory. *Journal of Parallel and Distributed Com- puting* (2018).
22. JOSHI, A., NAGARAJAN, V., CINTRA, M., AND VIGLAS,

S. Efficient persist barriers for multicores. In *Microarchi- tecture (MICRO), 2015 48th Annual IEEE/ACM Interna- tional Symposium on* (2015), IEEE, pp. 660–671.

1. KANNAN, S., ARPACI-DUSSEAU, A. C., ARPACI- DUSSEAU, R. H., WANG, Y., XU, J., AND PALANI, G. Designing a true direct-access file system with devfs. In *16th USENIX Conference on File and Storage Technolo- gies* (2018), p. 241.
2. KAZAR, M. L., LEVERETT, B. W., ANDERSON, O. T., APOSTOLIDES, V., BOTTOS, B. A., CHUTANI, S., EV- ERHART, C., MASON, W. A., TU, S.-T., AND ZAYAS,

E. R. Decorum file system architectural overview. In

*USENIX Summer* (1990), vol. 90, pp. 151–163.

1. KIM, J., LEE, S., AND VETTER, J. S. Papyruskv: a high- performance parallel key-value store for distributed nvm architectures. In *Proceedings of the International Confer- ence for High Performance Computing, Networking, Stor- age and Analysis* (2017), ACM, p. 57.
2. KIM, W.-H., SEO, J., KIM, J., AND NAM, B. clfb-tree: Cacheline friendly persistent b-tree for nvram. *ACM Trans- actions on Storage (TOS) 14*, 1 (2018), 5.
3. KOLLI, A. Architecting persistent memory systems.
4. KOLLI, A., GOGTE, V., SAIDI, A., DIESTELHORST, S., CHEN, P. M., NARAYANASAMY, S., AND WENISCH,

T. F. Language-level persistency. *SIGARCH Comput. Ar- chit. News 45*, 2 (June 2017), 481–493.

1. KOLLI, A., GOGTE, V., SAIDI, A., DIESTELHORST, S., CHEN, P. M., NARAYANASAMY, S., AND WENISCH,

T. F. Language-level persistency. In *Proceedings of the 44th Annual International Symposium on Computer Archi- tecture* (New York, NY, USA, 2017), ISCA ’17, ACM, pp. 481–493.

1. LEE, S. Write-optimal radix tree: A deterministic indexing structure for persistent memory storage systems.
2. LEE, S. K., LIM, K. H., SONG, H., NAM, B., AND NOH,

S. H. Wort: Write optimal radix tree for persistent memory storage systems. In *FAST* (2017), pp. 257–270.

1. LEIS, V., KEMPER, A., AND NEUMANN, T. Ex- ploiting hardware transactional memory in main-memory databases. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on* (2014), IEEE, pp. 580–591.
2. LIEDTKE, J. *On µ-kernel construction*, vol. 29. ACM, 1995.
3. LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY,

M. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 1–13.

1. LIU, D., ZHONG, K., WANG, T., WANG, Y., SHAO, Z., SHA, E. H.-M., AND XUE, J. Durable address translation in pcm-based flash storage systems. *IEEE Transactions on Parallel and Distributed Systems 28*, 2 (2017), 475–490.
2. LIU, H., HUANG, L., ZHU, Y., AND SHEN, Y. Librekv: A persistent in-memory key-value store. *IEEE Transactions on Emerging Topics in Computing* (2017).
3. LIU, H., HUANG, L., ZHU, Y., ZHENG, S., AND SHEN,

Y. Hmfs: A hybrid in-memory file system with version consistency. *Journal of Parallel and Distributed Comput- ing 117* (2018), 18–36.

1. LU, G., NAM, Y. J., AND DU, D. H. Bloomstore: Bloom- filter based memory-efficient key-value store for indexing of data deduplication on flash. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on* (2012), IEEE, pp. 1–11.
2. MAENG, K., COLIN, A., AND LUCIA, B. Alpaca: In- termittent execution without checkpoints. *Proc. ACM Pro- gram. Lang. 1*, OOPSLA (Oct. 2017), 96:1–96:30.
3. MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache crafti- ness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), ACM, pp. 183–196.
4. MARATHE, V. J., SELTZER, M., BYAN, S., AND HAR- RIS, T. Persistent memcached: bringing legacy code to byte-addressable persistent memory. In *9th USENIX Work- shop on Hot Topics in Storage and File Systems (HotStor- age 17)* (2017).
5. MARTIN, B. Formal concept analysis and semantic file systems. In *International Conference on Formal Concept Analysis* (2004), Springer, pp. 88–95.
6. MEMARIPOUR, A., BADAM, A., PHANISHAYEE, A., ZHOU, Y., ALAGAPPAN, R., STRAUSS, K., AND SWAN- SON, S. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys ’17, ACM, pp. 499–512.
7. NALLI, S., HARIA, S., HILL, M. D., SWIFT, M. M., VO- LOS, H., AND KEETON, K. An analysis of persistent mem- ory use with whisper. *SIGOPS Oper. Syst. Rev. 51*, 2 (Apr. 2017), 135–148.
8. NALLI, S., HARIA, S., HILL, M. D., SWIFT, M. M., VO- LOS, H., AND KEETON, K. An analysis of persistent mem- ory use with whisper. *SIGPLAN Not. 52*, 4 (Apr. 2017), 135–148.
9. NALLI, S., HARIA, S., HILL, M. D., SWIFT, M. M., VO- LOS, H., AND KEETON, K. An analysis of persistent mem- ory use with whisper. *SIGARCH Comput. Archit. News 45*, 1 (Apr. 2017), 135–148.
10. NALLI, S., HARIA, S., HILL, M. D., SWIFT, M. M., VO- LOS, H., AND KEETON, K. An analysis of persistent mem- ory use with whisper. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Pro- gramming Languages and Operating Systems* (New York, NY, USA, 2017), ASPLOS ’17, ACM, pp. 135–148.
11. NAWAB, F., IZRAELEVITZ, J., KELLY, T., MORREY III,

C. B., CHAKRABARTI, D. R., AND SCOTT, M. L. Dal´ı: A periodically persistent hash map. In *LIPIcs-Leibniz International Proceedings in Informatics* (2017), vol. 91, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

1. OUKID, I., BOOSS, D., LEHNER, W., BUMBULIS, P., AND WILLHALM, T. Sofort: A hybrid scm-dram stor- age engine for fast data recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware* (New York, NY, USA, 2014), DaMoN ’14, ACM, pp. 8:1–8:7.
2. OUKID, I., BOOSS, D., LESPINASSE, A., LEHNER, W., WILLHALM, T., AND GOMES, G. Memory manage- ment techniques for large-scale persistent-main-memory systems. *Proc. VLDB Endow. 10*, 11 (Aug. 2017), 1166– 1177.
3. OUKID, I., AND LEHNER, W. Data structure engineer- ing for byte-addressable non-volatile memory. In *Proceed- ings of the 2017 ACM International Conference on Man- agement of Data* (2017), ACM, pp. 1759–1764.
4. ONEIL, P., CHENG, E., GAWLICK, D., AND ONEIL, E. The log-structured merge-tree (lsm-tree). *Acta Informatica 33*, 4 (1996), 351–385.
5. PAPAGIANNIS, A., SALOUSTROS, G., MARAZAKIS, M., AND BILAS, A. Iris: An optimized i/o stack for low- latency storage devices. *ACM SIGOPS Operating Systems Review 50*, 1 (2017), 3–11.
6. PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Mem- ory persistency. In *ACM SIGARCH Computer Architecture News* (2014), vol. 42, IEEE Press, pp. 265–276.
7. PILLAI, T. S., ALAGAPPAN, R., LU, L., CHI- DAMBARAM, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Application crash consistency and performance with ccfs. *ACM Trans. Storage 13*, 3 (Sept. 2017), 19:1–19:29.
8. SBIAA, R., MENG, H., AND PIRAMANAYAGAM, S. Ma- terials with perpendicular magnetic anisotropy for mag- netic random access memory. *physica status solidi (RRL)- Rapid Research Letters 5*, 12 (2011), 413–419.
9. SELTZER, M. I., AND MURPHY, N. Hierarchical file sys- tems are dead. In *HotOS* (2009).
10. SHA, E. H.-M., JIANG, W., DONG, H., MA, Z., ZHANG, R., CHEN, X., AND ZHUGE, Q. Towards the design of efficient and consistent index structure with minimal write activities for non-volatile memory. *IEEE Transactions on Computers 67*, 3 (2018), 432–448.
11. SHIN, S., TIRUKKOVALLURI, S. K., TUCK, J., AND SOLIHIN, Y. Proteus: a flexible and fast software sup- ported hardware logging approach for nvm. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (2017), ACM, pp. 178–190.
12. SOLLER, S. Measurements of system call performance and overhead, January 2017. <http://arkanis.de/weblog/2017-> 01-05-measurements-of-system-call-performance-and- overhead (accessed April 16, 2018.
13. VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., CAMPBELL, R. H., ET AL. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST* (2011), vol. 11, pp. 61–75.
14. VISCAROLA, P., AND MASON, W. A. *Windows NT device driver development*. New Riders Publishing, 1998.
15. VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 91–104.
16. WANG, C., WEI, Q., WU, L., WANG, S., CHEN, C., XIAO, X., YANG, J., XUE, M., AND YANG, Y. Persisting rb-tree into nvm in a consistency perspective. *ACM Trans- actions on Storage (TOS) 14*, 1 (2018), 6.
17. WANG, T., SAMBASIVAM, S., SOLIHIN, Y., AND TUCK,

J. Hardware supported persistent object address transla- tion. In *Proceedings of the 50th Annual IEEE/ACM Inter- national Symposium on Microarchitecture* (2017), ACM, pp. 800–812.

1. WANG, Z., QIAN, H., LI, J., AND CHEN, H. Us- ing restricted transactional memory to build a scalable in- memory database. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 26.
2. WEI, Q., WANG, C., CHEN, C., YANG, Y., YANG, J., AND XUE, M. Transactional nvm cache with high per- formance and crash consistency. In *Proceedings of the In- ternational Conference for High Performance Computing, Networking, Storage and Analysis* (2017), ACM, p. 56.
3. WU, C., FALEIRO, J. M., LIN, Y., AND HELLERSTEIN,

J. M. Anna: A kvs for any scale.

1. WU, X., NI, F., ZHANG, L., WANG, Y., REN, Y., HACK, M., SHAO, Z., AND JIANG, S. Nvmcached: An nvm- based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems* (2016), ACM, p. 18.
2. WU, X., XU, Y., SHAO, Z., AND JIANG, S. Lsm-trie: an lsm-tree-based ultra-large key-value store for small data. In *Proceedings of the 2015 USENIX Conference on Usenix*

*Annual Technical Conference* (2015), USENIX Associa- tion, pp. 71–82.

1. YU, S., XIAO, N., DENG, M., LIU, F., AND CHEN, W. Redesign the memory allocator for non-volatile main mem- ory. *ACM Journal on Emerging Technologies in Computing Systems (JETC) 13*, 3 (2017), 49.
2. ZADOK, E., AND BADULESCU, I. A stackable file system interface for linux. In *LinuxExpo Conference Proceedings* (1999), vol. 94, p. 10.
3. ZHANG, M., LAM, K. T., YAO, X., AND WANG, C.-L. Simpo: A scalable in-memory persistent object framework using nvram for reliable big data computing. *ACM Trans- actions on Architecture and Code Optimization (TACO) 15*, 1 (2018), 7.
4. ZHANG, W.-Z., LU, K., LUJA´ N, M., WANG, X.-P., AND ZHOU, X. Fine-grained checkpoint based on non-volatile memory. *Frontiers of Information Technology & Electronic Engineering 18*, 2 (2017), 220–234.