Redesigning Data Structures for Non-Volatile Byte-Addressable Memory

Shivaram Venkataraman[†*], Niraj Tolia[‡], Parthasarathy Ranganathan[†], and Roy H. Campbell[*]
[†]HP Labs, Palo Alto, [‡]Maginatics, and [*]University of Illinois, Urbana-Champaign

1 Introduction

Recent architecture trends show that DRAM density scaling is facing significant challenges and will hit a scalability wall at 40nm [4, 5]. Additionally, power constraints will also limit the amount of DRAM installed in future systems [3]. To support next generation systems, technologies such as Phase Change Memory (PCM) and Memristor are being developed as DRAM replacements. These memories offer latencies that are orders of magnitude lower than either disk or flash and are comparable to DRAM. Not only are they byte-addressable like DRAM but, in addition, they are non-volatile. Projected cost analysis [3] and power-efficiency characteristics of Non-Volatile Byte-addressable Memory (NVBM) lead us to believe that it can replace both disk and memory in data stores (e.g., databases, NoSQL systems, etc.) but not through legacy block or file systems interfaces. The overhead of these interfaces will dominate NVBM's nanosecond access latencies and furthermore, these interfaces impose a two-level logical separation of data, differentiating between in-memory vs. on-disk copies of data. Traditional data stores have to both update the in-memory data and, for durability, sync the data to disk with the help of a write-ahead log. Not only does this data movement use extra power and reduce performance for low latency NVBM, the logical separation also reduces the capacity of an NVBM system.

Instead, we propose a single-level NVBM hierarchy where no distinction is made between a volatile and a persistent copy of data. With a single-level NVBM store, we need to ensure that data structures will never be left in an inconsistent state. Unfortunately, processors today do not provide the necessary extensions to prevent writes from being flushed from cache to memory and given that the memory controller can reorder cache line writes, current mechanisms for updating data structures are likely to cause corruption in the face of failures.

To address the above requirements for NVBM, we propose the use of Consistent and Durable Data Structures (CDDSs), a design that allows for the creation of log-less storage systems on non-volatile memory *without* processor modifications. These data structures allow mutations to be safely performed directly (using loads and stores) on the single copy of data. Instead of using write-ahead logging or shadow paging, we have architected CDDSs to use versioning. Independent of the size of an update, versioning allows the CDDS to atomically move from one consistent state to the next. Failure recovery simply restores the data structure to the most recent consistent version. Garbage collection is run in the background and helps limit the space utilization by eliminating entries which will not be referenced in the future. Further, we have developed primitives for atomic and durable updates using hardware support found in existing processors making CDDSs applicable without any processor modifications.

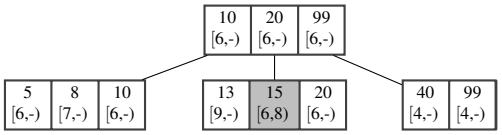


Figure 1: CDDS B-Tree

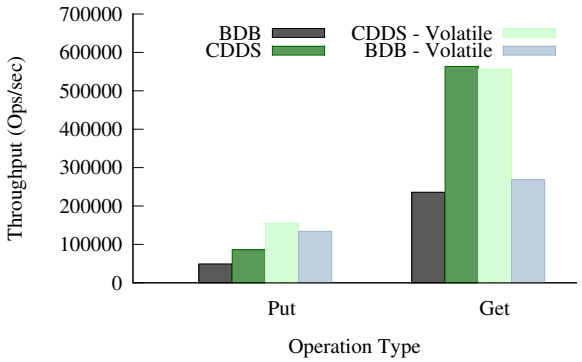
2 Consistent Durable Data Structures

A CDDS is built by maintaining a limited number of versions of the data structure with the constraint that an update should not weaken the structural integrity of an older version and that updates are atomic. This versioning scheme allows a CDDS to provide consistency without the additional overhead of logging or shadowing. A CDDS thus provides a guarantee that a failure between operations will never leave the data in an inconsistent state. As a CDDS never acknowledges completion of an update without safely committing it to non-volatile memory, it also ensures that there is no silent data loss

Internally, a CDDS maintains the following properties:

- There exists a version number for the most recent consistent version. This is used by any thread which wishes to read from the data structure.
- Every update to the data structure results in the creation of a new version.
- During the update operation, modifications ensure that existing data representing older versions are never overwritten. Such modifications are performed by either using atomic operations or copy-on-write style changes.
- After all the modifications for an update have been made persistent, the most recent consistent version number is updated atomically.

As an example of a CDDS, we have designed and implemented a consistent and durable version of a B-



Mean of 5 trials. Max. standard deviation: 2.2% of the mean.

Figure 2: Berkeley DB Comparison

Tree [7]. While inspired by previous work on multiversion data structures [6], our focus on durability required changes to the design and impacted our implementation. In a CDDS B-Tree, the key and value stored in an unmodified B-Tree entry is augmented with a start and end version number. A simplified example of a CDDS B-Tree is shown in Figure 1. An entry is considered 'live' if it does not have an end version (displayed as a '$-$' in the figure). To bound space utilization, in addition to ensuring that a minimum number of entries in a B-Tree node are used, we also maintain a minimum number of live entries in each node. The CDDS B-Tree supports lookup, insert, and delete operations. Detailed algorithms for each of these operations can be found in our paper [7]. Additionally, our implementation supports features such as leaf iterators and range scans. We also believe that CDDS versioning lends itself to other powerful features such as instant snapshots and integrated NVBM wear-leveling. Our current CDDS B-Tree implementation uses a multiple-reader, single-writer model. However, the use of versioning lends itself to more complex concurrency control schemes including multi-version concurrency control (MVCC) and we are exploring different concurrency control schemes for CDDSs as a part of our ongoing work.

Finally, apart from multi-version data structures, CDDSs have also been influenced by Persistent Data Structures (PDSs) [2]. The "Persistent" in PDS does not actually denote durability on persistent storage but, instead, represents immutable data structures where an update always yields a new data structure copy and never modifies previous versions. The CDDS B-Tree presented above is a weakened form of semi-persistent data structures. We modify previous versions of the data structure for efficiency but are guaranteed to recover from failure and rollback to a consistent state. However, the PDS concepts are applicable, in theory, to all linked data structures. Using PDS-style techniques, we have implemented a proof-of-concept CDDS hash table and we are confident that CDDS versioning techniques can be extended to a wide range of data structures.

2.1 Evaluation

As NVBM is not commercially available yet, we used DRAM-based servers to evaluate our proposed design. Previous studies have shown that DRAM-based results are a good predictor of NVBM performance. To compare the benefits of versioning over logging, we compare the CDDS B-Tree performance for puts, and gets to a memory backed Berkeley DB's (BDB) B-Tree implementation. For this experiment, we insert and fetch 1 million key-value tuples into each system. After each operation we flush the CPU cache to eliminate any variance due to cache contents. The results, displayed in Figure 2, show that, for memory-backed BDB in durable mode, the CDDS B-Tree improves throughout by 74% and 138% for puts and gets respectively. These gains come from not using a log (extra writes) or the file system interface (system call overhead). If zero-overhead epoch-based hardware support [1] was available, the CDDS volatile numbers show that performance of puts would increase by 80% as `flush`es would never be on the critical path. We do not observe any significant change for gets as the only difference between the volatile and durable CDDS is that the `flush` operations are converted into a noop. Finally, to measure the versioning overhead, we compared a volatile CDDS B-Tree to a normal B-Tree. While not shown in Figure 2, volatile CDDS's performance was lower than the in-memory B-Tree by 24% and 13% for puts and gets.

References

[1] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–146, Big Sky, MT, Oct. 2009.

[2] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[3] R. F. Freitas and W. W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4):439–447, 2008.

[4] International Technology Roadmap for Semiconductors: Process integration, Devices, and Structures, 2007. `http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_PIDS.pdf`.

[5] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens. Challenges and future directions for the scaling of dynamic random-access memory (DRAM). *IBM Journal of Research and Development*, 46(2-3):187–212, 2002.

[6] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.

[7] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*, San Jose, CA, Feb. 2011. To appear.