NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store

Leonardo Mármol[‡], Swaminathan Sundararaman[†], Nisha Talagala[†], Raju Rangaswami[‡], Sushma Devendrappa[*], Bharath Ramsundar[*], Sriram Ganesan[*]

[†] *FusionIO* [‡] *Florida International University* [*] *work done at FusionIO*

Abstract

State-of-the-art flash-optimized KV stores frequently rely upon a log structure and/or compaction-based strategy to optimally organize content on flash. However, these strategies lead to excessive I/O, beyond the write amplification generated within the flash itself, with both the application and the flash device constantly rearranging data. In this paper, we explore the other extreme in the design space: minimal data management at the KV store and heavy reliance on the *Flash Translation Layer* (FTL) capabilities. NVMKV is a scalable and lightweight KV store that leverages advanced capabilities that are becoming available in modern FTLs. We demonstrate that NVMKV *(i)* performs KV operations at close to native device access speeds for `get` operations, *(ii)* outperforms state of the art KV stores by 50%-300%, *(iii)* significantly improves performance predictability for the YCSB KV benchmark when compared with the popular LevelDB KV store, and *(iv)* reduces data written to flash by as much as 1.7X and 29X for sequential and random write workloads relative to LevelDB, thereby dramatically increasing device lifetime.

1 Introduction

Key-value (KV) stores are ubiquitous, having become the default data management software for many Internet services [8, 10, 21, 22, 32]. They serve application needs in a variety of different domains that demand high-throughput and low-latency data access [1, 3].

The performance, capacity, and power consumption mix of flash-based storage makes it an attractive medium for KV stores [12, 15, 19, 20, 27]. To get the best performance from both HDDs and low-end SSDs, many KV stores use some form of log structured writing to optimize data layout on media. Since log structured updates require eventual compaction or garbage collection, the consequence is *auxiliary write amplification*, i.e., additional write amplification introduced at the KV store besides the write amplification introduced by the Flash Translation Layer (FTL). For instance, the SILT work introduces an auxiliary write amplification of 5.4 [27]. The recent LevelDB KV store from Google [22] also exhibits rather dramatic auxiliary write amplification. Figure 1 reveals a minimum of 2.5x auxiliary write amplification for sequential asynchronous writes and a maximum of 43x for random synchronous writes. Prior research on

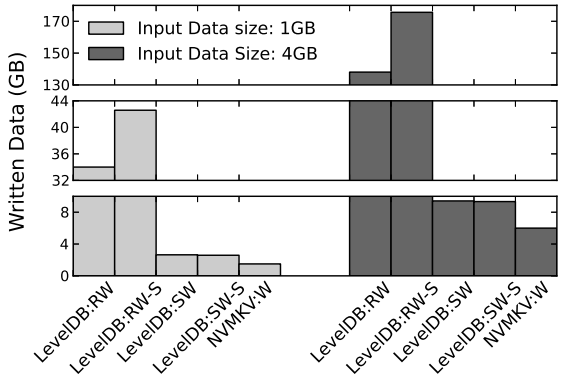

Figure 1: A comparison of write amplification. *LevelDB variants are RW: Random asynchronous writes, RW-S: random synchronous writes, SW: sequential asynchronous writes, SW-S: sequential synchronous writes.*

auxiliary write amplification in caching application data in flash demonstrates that the cumulative write amplification is multiplicative, causing even a small user update to result in massive writing to the flash over time [33].

Three trends force us to rethink KV store design choices. First, NAND flash endurance is getting poorer with every new media generation [25]. With fewer Program/Erase cycles to begin with, the auxiliary write amplification further reduces the device lifetime and increases the KV store's total cost of ownership. Second, the gap between sequential and random write performance has significantly narrowed in state of the art SSDs today [26], calling into question the need for application level log structuring and compaction. Third, modern FTLs are much more powerful than the traditional block devices. New FTL interfaces have recently been developed to provide advanced capabilities to access data to (or from) NAND Flash [7, 9, 26, 29, 30].

In this paper, we explore a new design for a KV store — one that relies upon cooperative design with an FTL to minimize auxiliary write amplification and maximize application-level performance. The resulting KV store, NVMKV, is lightweight and fully exploits native characteristics of the FTL to achieve `get` performance equivalent to raw device read speeds and `put` operations that are significant fractions of raw device write speeds.

NVMKV makes several novel contributions. While many flash and disk optimized KV stores exist (see

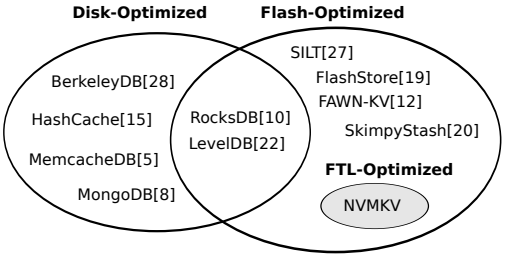

Figure 2: **Categorizing Existing KV stores.** *This figure shows a broad categorization of existing KV stores based on primarily being hard-disk- or flash-optimized and on leveraging capabilities surfaced by modern FTLs.*

Figure 2), NVMKV is the first to leverage native FTL layer primitives such as *atomic multi-block write*, *atomic multi-block persistent trim*, *exists*, and *iterate* to implement KV functionality. NVMKV demonstrates how strong consistency and atomic guarantees provided by the underlying FTL can be used to achieve atomicity and isolation, low write amplification, and performance close to that of the raw device. NVMKV is also the first KV store that uses a small (close to zero), constant, amount of in-memory metadata that is independent of both the number of keys stored and the workload intensity.

Our evaluation of NVMKV reveals the following. NVMKV performs `get` operations at close to the native device access speeds. Relative to LevelDB, a popular KV store in wide use today, NVMKV reduces auxiliary write amplification by as much as 1.7X and 29X for sequential and random write workloads respectively. For the YCSB KV benchmark, NVMKV outperforms LevelDB by 50%-300% besides significantly reducing the variance of KV operation latencies when compared with LevelDB. Finally, NVMKV provides significantly better performance when using a fourth of the user-level DRAM cache size compared to LevelDB and unlike LevelDB, without using any OS-level DRAM caching.

2 Leveraging Flash Devices

Modern FTLs are powerful software layers that include functions such as log-structuring, dynamic data remapping, indexing, transactional updates, and thin provisioning [26, 29], which are superficially similar to the functionality being built into many KV stores. For instance, FTLs implement an indirection map to manage the logical to physical block address mapping and write logging to guarantee durability on a medium that implicitly forces out-of-place writes. There is ongoing effort to surface these advanced capabilities through standardized primitives for use by operating systems and user space software [7, 9]. Table 1 lists some of the primitives that can be surfaced by a modern FTL. Recent work has utilized such primitives for implementing efficient file systems [26], databases [29], and caching [30].

API	Description
`EXISTS`	queries if an address is populated
`ATOMIC-WRITE`	writes an address range as ACID tx.
`ATOMIC-TRIM`	deletes an address range as ACID tx.
`ITERATE`	returns all populated addresses

Table 1: **FTL Primitives.** *These primitives can be used for either individual or ranges of (both sparse and non-sparse) locations. Additionally, batch operations of* `ATOMIC-WRITE`*,* `ATOMIC-TRIM`*, and combinations are also possible, allowing the write of some locations and the deletion of other locations as a single transaction.*

2.1 Dynamic Mapping

FTLs maintain an indirection map translating logical block addresses to physical locations. This mapping is required to organize data for minimal write amplification and best wear leveling. Most KV stores also maintain a mapping engine that converts keys to storage addresses where the values are stored.

To leverage an FTL based remapping engine for mapping key-value pairs, we extend the FTL indirection map to a *sparse map*, similar to that used in previous work [26]. A sparse map provides a few orders of magnitude more addressable logical addresses (LBAs) for the same physical capacity, thinly provisioning physical locations only for LBAs that have been written. Leveraging the underlying sparse addressing, NVMKV replaces the indirection maps found in most KV stores with hashing functions over the sparse address space. Through this approach, `put` and `get` operations are simply mapped to *write* and *read* operations in the FTL, respectively. A `delete` of a key removes the KV pair from the storage device using the *trim* operation.

2.2 Persistence and Transactional Support

FTLs maintain both data and metadata, and in particular, persistent indirection maps to recover data upon restart. Since FTLs operate as copy-on-write, they can provide high performance transactional write semantics [29]. KV stores can leverage this capability to ensure that *puts* of KV pairs are atomic without additional journaling, and providing nearly the same performance as that of conventional writes. Access to the transactional persistence capabilities of the FTL can be provided through two primitives, `ATOMIC-WRITE` and `ATOMIC-TRIM`.

2.3 Highly parallel operations

FTLs support highly parallel read/write operations to match the parallelism available inside the NAND die of most flash devices. By utilizing the atomic operations, we can minimize locking within the KV store and better leverage the inherent parallelism within the flash device.

3 NVMKV Design

NVMKV is designed as a user space library that exports a KV API and leverages the publicly available FTL prim-

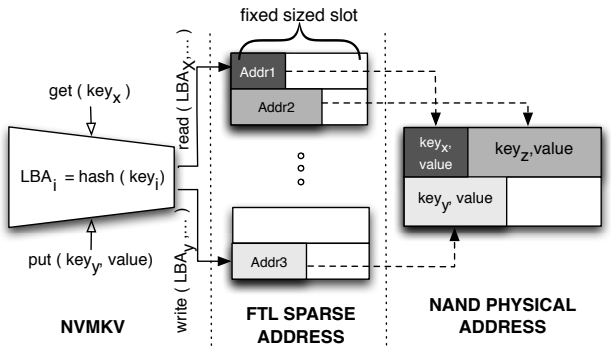

Figure 3: NVMKV Hash model. *How NVMKV retrieves and stores KV pairs in a flash device. First, for both* `get` *and* `put` *operations, a hash function computes the starting LBA (i.e., slot number) in the FTL's sparse address space. Second, NVMKV issues a read or write operation to the LBA range, which in turn gets translated (via the FTL) to the mapped physical address(es). On* `put` *operations, NVMKV adds metadata to each KV pair to help identify and resolve collisions.*

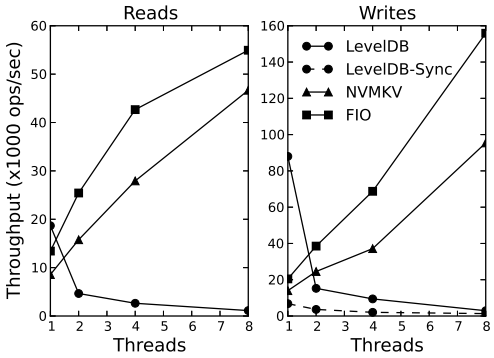

Figure 4: Microbenchmark comparing LevelDB, NMVKV and Raw Block Device (FIO).

itives (see Table 1) to access flash devices [9]. These primitives are implemented within Fusion-io's ioMemory FTL [23] and exported as a set of IOCTLs.

3.1 Sparse Addressing

Leveraging sparse addressing is central to NVMKV since it allows minimizing I/O amplification during both `get` and `put` operations. In the absence of collisions, each `get` or `put` operation translates into exactly one I/O to flash. The elimination of an indirection map results in fixed (nearly zero) metadata at the KV store.

To effectively utilize the sparse address space, NVMKV divides the sparse address into two parts: the *Key Bit Range* (KBR) and the *Value Bit Range* (VBR). By default, NVMKV uses 36 bits for the KBR and 12 bits for the VBR in a 48 bit address space, with alternatives configurable by the user when the KV store is created. The VBR defines the amount of contiguous address space (i.e., maximum value size) reserved for each KV pair, ensuring that KV pairs mapped into the sparse address space to not overlap each other in logical address space. The KBR determines the maximum number logical hash slots that each KV pair can be placed into. User supplied keys are mapped to LBA addresses through a simple hash model (Figure 3). Keys can be variable length up to the maximum supported key size (2MB for a 12 bit VBR).

3.2 Hashing and Collision Handling

Since each VBR will contain exactly one KV pair, hash conflicts only occur in the KBR. The design of NVMKV assumes that the KBR is kept sufficiently large, relative to the number of keys that can be stored in a flash device, to reduce the chances of a hash collision. For example, 1TB of flash can contain a maximum of 2 billion 512B KV pairs. With the default KBR of 36 bits which supports 64 billion hash slots, and uniform hashing of KV pairs across KBR space, the chances of a new KV pair inserted into even a full device causing a collision is $\tilde{3}\%$.

Collisions are handled within the library by deterministically computing an alternate hash location (via polynomial probing) within the KBR. Up to eight hash locations are tried before the KV Store refuses to accept a new key. Assuming that the hash function uniformly distributes keys, the probability of a `PUT` failing in the above example equals the probability of 8 consecutive collisions and is approximately $(1/(64\,billion\,/\,2\,billion)^8 = 1/2^{40}$ is vanishingly small. The sparse address bits can be increased proportionately as device capacities increase to maintain low hash collision probability.

3.3 KV Storage and Caching

The minimum unit of storage in NVMKV is a sector where keys and values smaller than 512B will consume a full 512B sector. Each KV pair will also contain some metadata in a header stored on media. NVMKV packs and stores the metadata, the key, and the value in a single sector if the sum of their individual sizes is less than or equal to the sector size. Our instance of NVMKV implements a DRAM cache which can be used to hold KV pairs. Separately, a collision cache holds information about recent hash collisions, reducing the need for additional flash lookups at collision time.

4 Evaluation

We evaluated the performance, overhead, and auxiliary write amplification of NVMKV, comparing it to the raw device and LevelDB 1.14 [22]. We used the FIO benchmark, the YCSB KV benchmark [17], and the LevelDB suite of micro-benchmarks for our workloads. Our experiments were performed on a system with a Quad-Core 2.5 GHz AMD Opteron(tm) Processor, 8GB of DDR2 RAM, and a 825GB Fusion-io ioScale2 drive running Linux Ubuntu 12.04 LTS.

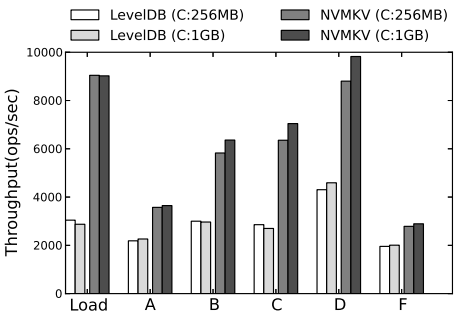

Figure 5: LevelDB vs. NVMKV for built-in YCSB workloads A-F. *Load populates data and precedes all workloads. A is a mix of 50/50 reads and writes. B is 95/5 reads/write mix. C is read only. D is a read latest workload, and F is a read-modify-write workload.*

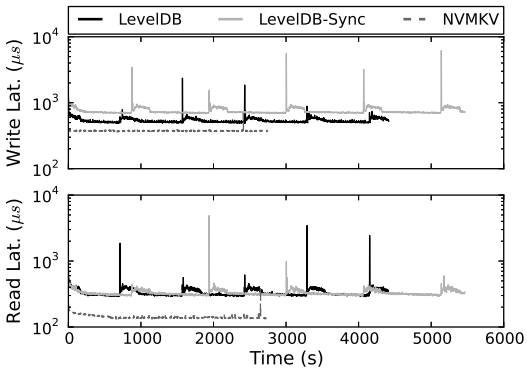

Figure 6: KV operation latency over time for YCSB-A. *The two charts depict the update (top), and read (bottom) phases of the workload.*

4.1 Raw Device Performance Comparison

Our first experiment evaluates the overhead of the NVMKV stack and LevelDB relative to the raw flash device. `Get` and `put` used 512B values and key sizes ranging from 1 byte to 128 bytes and for these sizes, NVMKV issues I/O operations of size 1KB. The FIO tool was configured to generate 1KB I/Os to the raw device. Figure 4 shows that as the thread count increases, the throughput for NVMKV's `get` operations tracks the FIO benchmark's `read` rate. For `put` operations, NVMKV significantly outperforms both the asynchronous and synchronous versions of LevelDB. Additional overheads in NVMKV such as checking for collisions cause performance to be lower than the underlying native device `write` performance extracted by FIO.

4.2 LevelDB Comparison using YCSB

YCSB is a framework for comparing the performance of KV stores and implements six workload personalities A-F [17]. The YCSB dataset size was 10GB and we evaluated both KV stores with caches of size 256MB (C: 256 MB) and 1GB (C: 1 GB). LevelDB implements write buffering and utilizes the OS page cache (both active during the experiment) while NVMKV does neither. LevelDB was configured to perform asynchronous writes.

Figure 5 demonstrates throughput performance gains of 50%-300% with NVMKV relative to LevelDB when running the YCSB workloads. These performance gains are even more significant when we consider that LevelDB does not provide durability while NVMKV does, and that LevelDB uses both a write buffer and the OS buffer cache for additional DRAM caching/buffering, while NVMKV does neither. We do not report results for YCSB-E because it performs short range scans (short sequential scans at randomly chosen locations), an operation not currently supported by the YCSB Java binding for NVMKV.

We measured how KV operation latencies varied over time for each of the workloads (Figure 6). We include data for both *async* (default) and *sync* modes for LevelDB writes. There are a few interesting insights here. First, NVMKV, which delivers atomic and durable updates, significantly outperforms even the LevelDB's best performing, but weaker, *async* mode. Second, we note that the performance variance in LevelDB is greater relative to NVMKV with significant latency spikes. We measured average/maximum KV read latencies of 0.33/1.38 and 0.33/1.39 ms for LevelDB and LevelDB-S respectively, relative to 0.14/0.66 ms for NMVKV. Since the LevelDB latency spikes seem to occur periodically, we believe these are correlated with the internal compaction mechanism. On the other hand, NVMKV offers much more consistent latency performance over time.

Revisiting Figure 1, we see that for the LevelDB suite of microbenchmarks [22], NVMKV incurs 1.7X to 29X lower auxiliary write amplification than LevelDB. The performance gains in NVMKV can be attributed to this reduction, as well as eliminating layers of indirection and metadata management overhead.

5 Discussion and Future Work

Besides NVMKV's performance improvements and endurance gains, atomic durability guarantees for KV operations are an added benefit. The need for such guarantees is a topic of debate within the NoSQL community with KV stores implementing different levels of eventual to strict consistency and varying degrees of durability. In NVMKV, we were able to provide strictly atomic and synchronous durability guarantees by leveraging the underlying FTL capability. Thus, rather than adding complexity and sacrificing performance to achieve strict guarantees (and thereby feeling pressurized to give them up), we found that leveraging their presence in the underlying FTL helped us simplify the NVMKV design without sacrificing performance. We also observed that by extracting more of the native performance of the flash device, we were able to deliver more KV operation performance than LevelDB while consuming less DRAM. We

believe that NVMKV represents a sound building block on which scale-out KV stores can be built. This is an area we intend to explore further in the future.

Due to lack of space, we did not include any descriptions of how the FTL supported the primitives that NVMKV relies upon. However, the implementations of similar primitives have been discussed with descriptions of possible FTL implementation designs [26, 29, 30]. In particular, FlashTier discusses sparse maps, showing how an incremental extension to an existing FTL data structure can enable dramatic DRAM reductions in applications while only causing moderate additional DRAM consumption at the FTL.

A limitation in our current design is the requirement to map individual KV pairs to separate sectors. NVMKV is best utilized for KV pairs which consume over 256 bytes. While many workloads fit this criterion, there are also many that do not. For the second group of workloads, NVMKV will have poor capacity utilization. One way to manage efficient storage of small KV pairs is to follow a multi-level storage mechanism, as provided in SILT [27], where small items are initially indexed separately and later compacted into larger units such as sectors. This is also a target area for future work.

6 Conclusions

We explored a novel concept of a KV store designed cooperatively with an FTL to reduce redundant work across the two layers. The result, NVMKV, is able to extract significant fractions of the raw device performance and outperform a state of the art KV store while minimizing auxiliary write amplification. NVMKV is open source, available at `https://github.com/opennvm/nvmkv`.

7 Acknowledgments

We thank the reviewers for their feedback and comments. We also would like to thank the people who helped build NVMKV and the FTL interface primitives.

References

[1] Aerospike: High performance KV Store use cases. http://www.aerospike.com/.

[2] Cassandra. http://cassandra.apache.org/.

[3] Couchbase: NoSQL use cases. http://www.couchbase.com.

[4] fio performance measurement tool. http://freecode.com/projects/fio.

[5] memcachedb. http://memcachedb.org/.

[6] Native Flash Support for Applications. http://www.flashmemorysummit.com/.

[7] SBC-4 SPC-5 Atomic writes and reads. http://www.t10.org/cgi-bin/ac.pl?t=d&f=14-043r2.pdf, 2013.

[8] MongoDB. http://mongodb.org, 2014.

[9] NVM Primitives Library. http://opennvm.github.io/nvm-primitives-documents/, 2014.

[10] RocksDB. http://rocksdb.org, 2014.

[11] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proc. of USENIX ATC*, 2008.

[12] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. of ACM SOSP*, 2009.

[13] E. Anderson, X. Li, M. Shah, J. Tucek, and J. Wylie. What consistency does your key-value store actually provide? *Proc. of USENIX HotDep*, 2010.

[14] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large scale key-value store. In *Proc. of ACM SIGMETRICS*, 2012.

[15] A. Badam, K. Park, V. Pai, and L. Peterson. Hashcache: cache storage for the next billion. In *Proc. of USENIX NSDI*, 2009.

[16] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *IGCC*, 2011.

[17] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. 2010.

[18] B. Debnath, S. Sengupta, and J. Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *Proc. of USENIX ATC*, 2010.

[19] B. Debnath, S. Sengupta, and J. Li. Flashstore: high throughput persistent key-value store. *Proc. of VLDB Endow*, 2010.

[20] B. Debnath, S. Sengupta, and J. Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proc. of ACM SIGMOD*, 2011.

[21] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *Proc. of ACM SIGOPS*, 2007.

[22] S. Ghemawat et al. LevelDB. https://code.google.com/p/leveldb/.

[23] Fusion-io, Inc. ioMemory Virtual Storage Layer (VSL). http://www.fusionio.com/overviews/vsl-technical-overview.

[24] R. Geambasu, A. Levy, T. Kohno, A. Krishnamurthy, and H. Levy. Comet: An active distributed key-value store. *Proc. of USENIX OSDI*, 2010.

[25] L. Grupp, J. Davis, and S. Swanson. The bleak future of nand flash memory. In *Proc. of USENIX FAST*, 2012.

[26] W. Josephson, L. Bongo, K. Li, and D. Flynn. Dfs: A file system for virtualized flash storage. In *Proc. of USENIX FAST*, 2010.

[27] H. Lim, B. Fan, D. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. of ACM SOSP*, 2011.

[28] M. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Proc. of USENIX ATC*, 1999.

[29] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *Proc. of IEEE HPCA*, 2011.

[30] M. Saxena, M. Swift, and Y. Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *Proc. of ACM ECCS*, 2012.

[31] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: reliable transactional p2p key/value store. In *Proc. of ACM SIGPLAN*, 2008.

[32] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with project voldemort. In *Proc. of USENIX FAST*, 2012.

[33] J. Yang, N. Plasson, G. Gillis, N. Talagala, S. Sundararaman, and R. Wood. Hec: Improving endurance of high performance flash-based cache devices. In *Proc. SYSTOR*, 2013.