NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store

Leonardo Marmol[‡], Swaminathan Sundararaman[†], Nisha Talagala[†], Raju Rangaswami[‡]
[†]*SanDisk* [‡]*Florida International University*

Abstract

Key-value stores are ubiquitous in high performance data-intensive, scale out, and NoSQL environments. Many KV stores use flash devices for meeting their performance needs. However, by using flash as a simple block device, these KV stores are unable to fully leverage the powerful capabilities that exist within Flash Translation Layers (FTLs). NVMKV is a lightweight KV store that leverages native FTL capabilities such as sparse addressing, dynamic mapping, transactional persistence, and support for high-levels of lock free parallelism. Our evaluation of NVMKV demonstrates that it provides scalable, high-performance, and ACID compliant KV operations at close to raw device speeds.

1 Introduction

Flash-based key-value (KV) stores are becoming mainstream, with the importance of the KV interface to storage and flash technology have been well established through a gamut of implementations [13, 17, 21, 22, 31]. However, best utilizing the high-performance flash-based storage to drive the new generation of key-value stores continues to remain a challenge. The majority of the existing KV stores use a logging-based approach which induces significant additional *write amplification* (WA) at the KV software layer in addition to the internal WA caused by the FTL while managing physical flash.

Modern FTLs offer new capabilities that enable compelling, new design points for KV stores [6, 9]. Integration with these advanced capabilities results in an optimized *FTL-aware* KV store [33]. First, writing to the flash can be optimized to significantly improve both device lifetime and workload I/O performance. Second, modern FTLs already perform many functions that are similar to the functionality built into many KV stores such as log-structuring, dynamic data remapping, indexing, transactional updates, and thin provisioning [29, 35, 37]. Avoiding such replication of functionality can offer significant resource and performance benefits.

In this paper, we present the design, implementation, and evaluation of NVMKV, an FTL-aware KV store. NVMKV has been designed from the ground up to utilize the advanced capabilities found in modern FTLs. It implements a hashing-based design that uses the FTLs sparse address-space support to eliminate all write amplification at the KV layer, improving flash device endurance significantly relative to current KV stores. It is able to achieve single I/O `get/put` operations with performance close to that of the raw device, representing a significant improvement over current KV stores. NVMKV uses the advanced FTL capabilities of *atomic multi-block write*, *atomic multi-block persistent trim*, *exists*, and *iterate* to provide strictly atomic and synchronous durability guarantees for KV operations.

Two complementary factors contribute to increased collocation requirements for KV stores running on a single flash device. First, given the increasing flash densities, the performance points of flash devices are now based on capacity with larger devices being more cost-effective [42]. Second, virtualization supports increases in collocation requirements for workloads. A recent study has shown that multiple independent instances of such applications can have a counterproductive effect on the underlying FTL, resulting in increased WA [42]. NVMKV overcomes this issue by offering a new *pools* abstraction that allows transparently running multiple KV stores within the same FTL. While similar features exist in other KV stores, the FTL-aware design and implementation within NVMKV enables both efficient FTL coupling and KV store virtualization. NVMKV's design also allows for optimized flash writing across multiple KV instances and as a result lowers the WA.

In the quest for performance, KV stores and other applications are trending towards an in-memory architecture. However, since flash is still substantially cheaper than DRAM, any ability to offset DRAM for flash has the potential to reduce Total Cost of Ownership (TCO). We demonstrate how accelerating KV store access to flash can in turn result in similar or increased performance with much less DRAM.

We evaluated NVMKV and compared its performance to LevelDB. We evaluated the scalability of pools, compared it to multiple instances of LevelDB, and also found that NVMKV's atomic writes outperform both async and sync variants of LevelDB writes by up to 6.5x and 1030x respectively. NVMKV reads are comparable to that of LevelDB even when the workloads fit entirely in the filesystem cache, a condition that benefits LevelDB exclusively. When varying the available cache space, NVMKV outperforms LevelDB and more importantly introduces a write amplification of 2x in the worst case, which is small compared to the 70x for LevelDB. Finally, NVMKV improves YCSB benchmark throughput by up to 25% in in comparison to LevelDB.

2 Motivation

In this section, we discuss the benefits of better integration of KV stores with the FTL's capabilities. We also motivate other key tenets of our architecture, in particular the support of multiple KV stores on the same flash device and the need to increase performance with smaller quantities of DRAM.

An FTL-aware KV store: A common technique for performance improvement in flash optimized KV stores is some form of log structured writing. New data is appended to an immutable store and reorganized over time to reclaim space [1, 10, 26, 31]. The reclamation process, also called garbage collection or compaction, generates *Auxiliary Write Amplification (AWA)* which is the application level WA above that which is generated by the FTL. Unfortunately, AWA and the FTL induced WA have a multiplicative effect on write traffic to flash [43]. Previous work highlighted an example of this phenomenon with LevelDB, where a small amount of user writes can be amplified into as much as 40X more writes to the flash device [33]. As another example, the SILT work describes an AWA of over 5x [31]. NVMKV entirely avoids AWA by leveraging native addressing mechanisms and optimized writing implemented within modern FTLs.

Multiple KV stores in the device: The most recent PCIe and SAS flash devices can provide as much as 4-6TB of capacity per device. As density per die increases with every generation of flash driven by the consumer market, the multiple NAND dies required to generate a certain number of IOPs will come with ever increasing capacities as well as reduced endurance [27]. Multiple KV stores on a single flash device become cost effective but additional complexities arise. For instance, recent work shows how applications that are log structured to be flash optimal can still operate in suboptimal ways when either placed above a file system or run as multiple independent instances over a shared FTL [42]. NVMKV provides the ability to have multiple independent KV workloads share a device with minimal AWA.

Frugal DRAM usage: The ever increasing need for performance is driving the in-memory computing trend [7, 11, 24]. However, DRAM cost does not scale linearly with capacity since high capacity DRAM and the servers that support it are more expensive per unit of DRAM (in GB) than the mid-range DRAM and servers. The efficacy of using flash to offset DRAM has also been established in the literature [16]. In the KV store context, similar arguments have been made showing the server consolidation benefits of trading DRAM for flash [1]. A KV store's ability to leverage flash performance contributes directly to its ability to trade off DRAM for flash. NVMKV operates with high performance and low WA in both single and multiple instance KV deployments.

3 Building an FTL-aware KV Store

NVMKV is built using the advanced capabilities of modern FTLs. In this section, we discuss its goals, provide an overview of the approach, and describe its API.

3.1 Goals

NVMKV is intended for use within single node deployments by directly integrating it into applications. While it is not intended to replace the scale out key-value functionality provided by software such as Dynamo and Voldemort [23, 39], it can be used for single node KV storage within such scale out KV stores. From this point onward, we refer to such single node KV stores simply as KV stores. We had the following goals in mind when designing NVMKV:

Deliver Raw Flash Performance: Convert the most common KV store operations, `GET` and `PUT` into a single I/O per operation at the flash device to deliver close to raw flash device performance. As flash devices support high levels of parallelism, the KV store should also scale with parallel requests to utilize the performance scaling capacity of the device. Finally, when multiple, independent KV instances are consolidated on a single flash device, the KV store should deliver raw flash performance to each instance.

Minimize Auxiliary Write Amplification: Given the multiplicative effect on I/O volume due to WA, it is important to minimize additional KV store writes, which in turn reduces the write load at the FTL and the flash device. Reducing AWA improves KV operation latency by minimizing the number of I/O operations per KV operation as well as improvement of flash device lifetime.

Minimize DRAM Consumption: Minimize DRAM consumption by *(i)* minimizing the amount of internal metadata, and *(ii)* by leveraging flash performance to offset the amount of DRAM used for caching.

Simplicity: Leverage FTL capabilities to reduce code complexity and development time for the KV store.

3.2 Approach

Our intent with NVMKV is to provide the rich KV interface while retaining the performance of a much simpler block based flash device. NVMKV meets its goals by leveraging the internal capabilities of the FTL where possible and complementing these with streamlined additional functionality at the KV store level. The high level capabilities that we leverage from the FTL include:

Dynamic mapping: FTLs maintain an indirection map to translate logical addresses into physical data locations. NVMKV leverages the existing FTL indirection map to the fullest extent to avoid maintaining any additional location metadata. Every read and write operation simply uses the FTL indirection map and thereby operates

Category	API	Description
Basic	get(...)	Retrieves the value associated with a given key.
	put(...)	Inserts a KV pair into the KV store.
	delete(...)	Deletes a KV pair from the KV store.
Iterate	begin(...)	Sets the iterator to the beginning of a given pool.
	next(...)	Sets the iterator to the next key location in a given pool.
	get_current(...)	Retrieves the KV pair at the current iterator location in a pool.
Pools	pool_exist(...)	Determines whether a key exists in a given pool.
	pool_create(...)	Creates a pool in a given NVMKV store.
	pool_delete(...)	Deletes all KV pairs from a pool and deletes the pool from NVMKV store.
	get_pool_info(...)	Returns metadata information about a given pool in a KV store.
Batching	batch_get(...)	Retrieves values for a batch of specified keys.
	batch_put(...)	Sets the values for a batch of specified keys.
	batch_delete(...)	Deletes the KV pairs associated with a batch of specified keys.
	delete_all(...)	Deletes all KV pairs from a NVMKV store in all pools.
Management	open(...)	Opens a given NVMKV store for supported operations.
	close(...)	Closes a NVMKV store.
	create(...)	Creates a NVMKV store
	destroy(...)	Destroys a NVMKV store.

Table 1: **NVMKV API** *The table provides brief descriptions for the NVMKV API calls.*

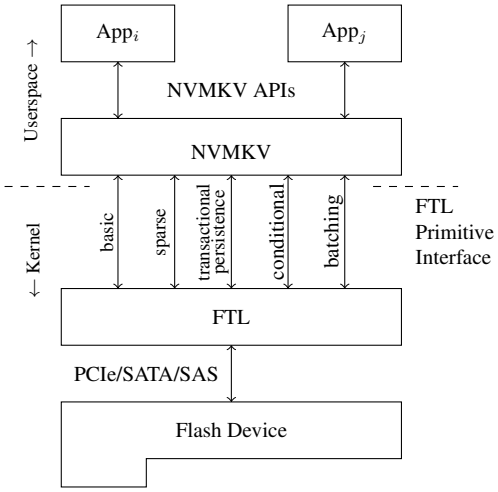

Figure 1: NVMKV System Architecture

at raw flash device performance by definition. This approach also reduces the additional DRAM overhead of NVMKV.

Persistence and transactional support: FTLs access data and metadata, and in particular maintain the persistence of indirection maps, at the speed of raw flash. NVMKV leverages this highly tuned capability to reduce the overhead for metadata persistence, logging, and checkpointing operations. Further, as FTLs operate as non-overwriting redirect-on-write stores, they can easily provide high performance transactional write semantics [35]. NVMKV leverages these capabilities to limit locking and journaling overheads.

Parallel operations: FTLs already implement highly parallel read/write operations while coordinating metadata access and updates. NVMKV leverages this FTL feature to minimize locking, thus improving scalability. We also define batch operations that are directly executed by the FTL to enable parallel KV store requests to be issued with lower I/O stack overhead [40].

3.3 NVMKV Architecture

NVMKV is a lightweight library in user space which interacts with the FTL through a *primitives interface* implemented as IOCTLs to the device driver that manages the flash device. Figure 1 shows the architecture of a system with NVMKV. Consumer applications, such as scale out KV stores, communicate with the NVMKV library using the NVMKV API. The NVMKV API calls are translated to underlying FTL primitives interface calls to be executed by the FTL.

3.4 NVMKV Consumer API

NVMKV's consumer applications interact with the library through the NVMKV API. We held discussions with the creators and vendors of several scale out KV stores to identify a set of operations commonly needed in a KV store. These operations formed the NVMKV API and they fall under five broad categories based on the functionality they provide. The categories are: *basic*, *iterate*, *pools*, *batching*, and *management*.

Table 1 presents the overview of the NVMKV API. We leverage the FTL's ability to provide enhanced operations such as *Atomic Writes* to provide transactional guarantees in NVMKV operations. Most existing KV stores do not offer such guarantees for their operations, and adopt more relaxed semantics such as eventual consistency to provide higher performance. On the other hand, we found that our approach enabled us to provide transactional guarantees with no loss of performance. We believe such guarantees can be of use to specific classes of applications as well as for simplifying the store logic

Category	API	Description
Basic	read(...)	Reads the data stored in the Logical Block Address (LBA).
	write(...)	Writes the data stored in buffer to destination LBA.
	trim(...)	Deletes (or discards) the mapping in FTL for the passed LBA range.
Sparse	exists(...)	Returns the presence of FTL mapping for the passed LBA.
	range_exist(...)	Returns the subset of LBA ranges that are mapped in the FTL.
	ptrim(...)	Persistently deletes the mapping in FTL for the passed LBA range.
	iterate(...)	Returns the next populated LBA starting from the passed LBA.
Transactional Persistence	atomic_read(...)	Executes read for a contiguous LBAs as an ACID transaction.
	atomic_exists(...)	Executes exists for a contiguous LBAs as an ACID transaction.
	atomic_write(...)	Executes write of a contiguous LBAs as an ACID transaction.
	atomic_ptrim(...)	Executes ptrim of a contiguous LBAs as an ACID transaction.
Conditional	cond_atomic_write(...)	Execute the atomic_write only if a condition is satisfied.
	cond_range_read(...)	Returns the data only from a subset of LBA ranges that are mapped in the FTL.
Batching		Operations within each category can be batched and executed in the FTL.

Table 2: **FTL Primitive Interface** *Enhanced FTL capabilities that NVMKV builds upon.*

contained within such applications. For instance, atomic KV operations imply that applications no longer need to be concerned with partial updates to flash.

The *Basic* and *Iterate* categories contain common features provided by many KV stores today. The *Pools* category interfaces allow for grouping KV pairs into buckets that can be managed separately within an NVMKV store. Pools provide the ability to transparently run multiple KV stores within the same FTL (discussed in more detail in § 6). The *Batching* category interfaces allow for group operations both within and across *Basic*, *Iterate*, and *Pools* categories, a common requirement in KV stores [18]. Finally, the *Management* category provides interfaces to perform KV store management operations.

4 Overview and FTL Integration

NVMKV's design is closely linked to the advanced capabilities provided by modern FTLs. Before describing its design in more detail, we provide a simple illustrative example of NVMKV's operation and discuss the advanced FTL capabilities that NVMKV leverages.

4.1 Illustrative Overview

To illustrate the principles behind NVMKV's design simply, we now walk through how a `get`, a `put`, and a `delete` operation are handled. We assume the sizes of keys and values are fixed and then address arbitrary sizes when we discuss design details (§5).

By mapping all KV operations to FTL operations, NVMKV eliminates any additional KV metadata in memory. To handle `put`s, NVMKV computes a hash on the key and uses the hash value to determine the location (i.e., LBA) of the KV pair. Thus, a `put` operation gets mapped to a write operation inside the FTL.

A `get` operation takes a key as input and returns the value associated with it (if the key exists). During a `get` operation, a hash of the key is computed first to determine the starting LBA of the KV pair's location. Using the computed LBA, the `get` operation is translated to a read operation to the FTL wherein the size of the read is equal to the combined sizes of the key and value. The stored key is matched with the key of the `get` operation and in case of a match, the associated value is returned.

To handle a `delete` operation, the given key is hashed to compute the starting LBA of the KV pair. Upon confirming that the key stored at the LBA is the key to be deleted, a discard operation is issued to the FTL for the range of LBAs containing the KV pair.

In this simplistic example, translating existing KV operations to FTL operations is straightforward and the KV store becomes a thin layer offloading most of its work to the underlying FTL with no in-memory metadata. However, additional work is needed to handle hash collisions in the LBA space and persisting discard operations.

4.2 Leveraging FTL Capabilities

We now describe the advanced FTL capabilities that are available and also extended to enable NVMKV. Many of these advanced FTL capabilities have already been used in other applications [20, 29, 35, 37, 43]. The FTL interface available to NVMKV is detailed in Table 2.

4.2.1 Dynamic Mapping

Conventional SSDs provide a dense address space, with one logical address for every advertised available physical block. This matches the classic storage model, but forces applications to maintain separate indexes to map items to the available LBAs. *Sparse address spaces* are available in advanced FTLs which allow applications to address the device via a large, thinly provisioned, virtual address space [35, 37, 43]. Sparse address entries are allocated physical space only upon a write. In the NVMKV context, a large address space enables simple mapping techniques such as hashing to be used with manageable collision rates.

Additional primitives are required to work with sparse address spaces. `EXISTS` queries whether a particular sparse address is populated. `PTRIM` is a persistent and atomic deletion of the contents at a sparse address. These primitives can be used for individual, or ranges of, locations. For example, `RANGE-EXISTS` returns a subset of a given virtual address range that has been populated. The `ITERATE` primitive is used to cycle through all populated virtual addresses, whereby `ITERATE` takes a virtual address and returns the next populated virtual address.

4.2.2 Transactional Persistence

The transactional persistence capabilities of the FTL are provided by `ATOMIC-WRITE` and `PTRIM` [20, 29]. `ATOMIC-WRITE` allows a sparse address range to be written as a single ACID compliant transaction.

4.2.3 Optimized Parallel Operations

The FTL is well-placed to optimize simultaneous device-level operations. Two classes of FTL primitives, *conditional* and *batch*, provide atomic parallel operations that are well-utilized by NVMKV. For example, `cond_atomic_write` allows for an atomic write to be completed only if a particular condition is satisfied, such as the LBA being written to is not already populated. This primitive removes the need to issue separate `exists` and `atomic_write` operations. Batch or vectored versions of all primitives are also implemented into the FTL (such as `batch_read`, `batch_atomic_write`, and `batch_ptrim`) to amortize lock acquisition and system call overhead. The benefits of batch (or vector) operations have been explored earlier [41].

5 Design Description

NVMKV implements novel techniques to make sparse addressing practical and efficient for use in KV stores and for providing low-latency, transactional persistence.

5.1 Mapping Keys via Hashing

Conventional KV stores employ two layers of translations to map keys to flash device locations, both of which need to be persistent [8, 10, 13]. The first layer translates keys to LBAs. The second layer (i.e., the FTL) translates the LBAs to physical locations in flash device. NVMKV leverages the FTLs sparse address space and encodes keys into sparse LBAs via hashing, thus collapsing an entire layer.

NVMKV divides the sparse address into equal sized virtual slots, each of which stores a single KV pair. More specifically, the sparse address space (with addressability through N bits) is divided into two areas: the *Key Bit Range* (KBR) and the *Value Bit Range* (VBR). This division can be set by the user at the time of creating the

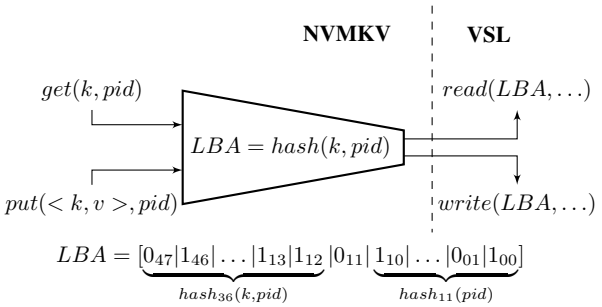

Figure 2: Hash model used in NVMKV. The arguments to the functions represent k:key, v:value, and pid:pool_id.

NVMKV store. The VBR defines the amount of contiguous address space (i.e., maximum value size or virtual slot size) reserved for each KV pair. The KBR determines the maximum number of such KV pairs that can be stored in a given KV store. In the expected use cases, the sparse virtual address range provided by the KBR will still be several orders of magnitude larger than the number of KV pairs as limited by the physical media.

The keys are mapped to LBAs through a simple hash model as shown in Figure 2. User supplied keys can be of variable length up to the maximum supported key size. To handle a `put` operation, the specified key is hashed into an address which also provides its KBR value. The maximum size of the information (Key, Value, metadata) that can be stored in a given VBR is half of the size addressed by the VBR. For example, if the VBR is 11 bits and each address represents a 512B sector, a given VBR value can address 2 MB.

The above layout guarantees the following two properties. First, each VBR contains exactly one KV pair, ensuring that we can quickly and deterministically search and identify KV pairs stored in the flash device. Second, no KV pairs will be adjacent in the sparse address space. In other words, there is always unpopulated virtual addresses between every KV pair. This deliberately wasted virtual space does not translate into unutilized storage since it is in virtual and physical space. These two properties are critical for NVMKV as the value size and exact start location of the KV pair are not stored as part of NVMKV metadata but are inferred via the FTL. Doing so helps in significantly reducing the in-memory metadata footprint of NVMKV. Non-adjacent KV pairs in the sparse address space help in determining the value size along with the starting virtual address of each KV pair. To determine the value size, NVMKV issues a `range_exist` call to the FTL.

A direct consequence of this design is that every access pattern becomes a random pattern, losing any possible order in the key space. The decision to not preserve sequentiality was shaped by two factors: metadata overhead and flash I/O performance. To ensure sequential

writes for contiguous keys, additional metadata would be required. This metadata would have to be consulted when reading, updated after writing, and cached in RAM to speed up the lookups. While straightforward to implement, doing so is unnecessary since the performance gap between random and sequential access for flash is ever decreasing; for current high performance flash devices, it is practically non-existent.

5.2 Handling Hash Collisions

Hashing variable sized keys into fixed size virtual slots could result in collisions. Since each VBR contains exactly one KV pair, hash conflicts only occur in the KBR. To illustrate the collision considerations, consider the following example. A 1TB Fusion-io ioDrive can contain a maximum of 2^{31} (2 billion) keys. Given a 48 bit sparse address space with 36 bits for KBR and 12 bits for VBR, NVMKV would accommodate 2^{36} keys with a maximum value size of 512KB. Under the simplifying assumption that our hash function uniformly distributes keys across the value ranges, for a fully-utilized 1TB ioDrive, the chances of a new key insertion resulting in a collision is $1/2^5$ or a little under 3 percent.

NVMKV implicitly assumes that the number of KBR values is sufficiently large, relative to the number of keys that can be stored in a flash device, so that the chances of a hash collision are small. If the KV pair sizes are increased, the likelihood of a collision reduces because the device can accommodate fewer keys while preserving the size of the key address space. If the size of the sparse address space is reduced, the chances of a collision will increase. Likewise, if the size of the flash device is increased without increasing the size of its sparse address space, the likelihood of a collision will increase.

Collisions are handled deterministically by computing alternate hash locations using either linear or polynomial probing. By default, NVMKV uses polynomial probing and up to eight hash locations are tried before NVMKV refuses to accept a new key. With this current scheme, the probability of a `put` failing due to hash failure is vanishingly small. Assuming that the hash function uniformly distributes keys, the probability of a `put` failing equals the probability of 8 consecutive collisions. This is approximately $(1/2^5)^8 = 1/2^{40}$, roughly one failure per trillion `put` operations. The above analysis assumes that the hash function used is well modeled by a uniformly distributing random function. Currently, NVMKV uses the FNV1a hash function [5] and we experimentally validated our modeling assumption.

5.3 Caching

Caching is employed in two distinct ways within NVMKV. First, a read cache speeds up access to frequently read KV pairs. NVMKV's read cache implementation is based on LevelDB's cache [26]. The read cache size is configurable at load time. Second, NVMKV uses a collision cache to improve collision handling performance. It caches the key hash (the sparse LBA) along with the actual key which is used during `put`s (i.e., inserts or updates). If the cached key matches the key to be inserted, the new value can be stored in the corresponding slot (the key's hash value). This significantly reduces the number of additional I/Os needed during collision resolution. In most cases, only a single I/O is needed for a `get` or a `put` to return or store the KV pair.

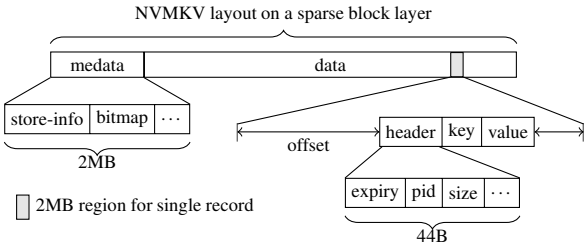

Figure 3: NVMKV layout

5.4 KV Pair Storage and Iteration

KV pairs are directly mapped to a physical location in the flash device and addressable through the FTL's sparse address space. In our current implementation, the minimum unit of storage is a sector and KV pairs requiring less than 512B will consume a full 512B sector. Each KV pair also contains metadata stored on media. The metadata layout is shown in Figure 3; it includes the length of the key, the length or the value, pool identifier (to be discussed further in §6), and other information.

To minimize internal fragmentation, NVMKV packs and stores the metadata, the key, and the value in a single sector whenever possible. If the size of the KV pair and the metadata is greater than a sector, NVMKV packs the metadata and key into the first sector and stores the value starting from the second sector. This layout allows for optimal storage efficiency for small values and zero-copy data transfer into the users buffer for larger values.

NVMKV supports unordered iteration through all KV pairs stored in the flash device. Key iteration is accomplished by iterating across the populated virtual addresses inside the FTL in order. The iterator utilizes the `ITERATE` primitive in the FTL, which takes in the previously reported start virtual address and returns the start address of the next contiguously populated virtual address segment in the sparse address space. Note that this approach relies on the layout guarantee that each KV pair is located contiguously in a range of virtual addresses, and that there are unpopulated virtual addresses in between each KV pair.

5.5 Optimizing KV Operations

NVMKV's design goal is one I/O for a `get` or a `put` operation. For `get`, this is achieved with the KV data layout and the `CONDITIONAL-RANGEREAD` primitive. The layout guarantees that individual KV pairs occupy a single contiguous section of the sparse address space, separated from other KV pairs by unpopulated virtual addresses. Given this, a `CONDITIONAL-RANGEREAD` can retrieve the entire KV pair in one operation without knowing the size of the value up front. Second, collisions induce a minimal number of additional operations. Since `get` and `put` operations for a given key map to hash addresses in a deterministic order, and since `put` places the new KV pair at the first available hash address (that is currently unused) in this order, subsequent `get`s are guaranteed to retrieve the most recent data written to this key. Finally, `DELETE` operations may require more than one I/O per operation, since they are required to read and validate the key before issuing a `PTRIM`. It also needs to check multiple locations to ensure that previous instances of a particular key have all been deleted.

NVMKV is intended to be zero copy and avoid memory comparison operations wherever possible. First, for any value that is large enough to start at its own sector, the data retrieved from a `get` (or written during a `put`) operation will be transferred directly to (or from) the user provided memory buffer. Second, no key comparisons occur unless the key hashes match. Given that the likelihood of collisions is small, the number of key comparisons that fail is also correspondingly small.

6 Multiple KV Instances Via Pools

Pools in NVMKV allow applications to group related keys into logical abstractions that can then be managed separately. Besides simplifying KV data management for applications, pools enable efficient access and iteration of related keys. The ability to categorize or group KV pairs also improves the lifetime of flash devices.

6.1 Need for Pools

NVMKV as described thus far, can support multiple independent KV stores. However, it would need to either partition the physical flash device to create multiple block devices each with its own sparse address space or logically partition the single sparse address space to create block devices to run multiple instances of KV stores.

Unfortunately, both approaches do not work well for flash. Since it is difficult to predict the number of KV pairs or physical storage needed in advance, static partitioning would result in either underutilization or insufficient physical capacity for KV pairs. Further, smaller capacity physical devices would increase pressure on the garbage collector, resulting in both increased write amplification and reduced KV store performance. Alternatively, partitioning the LBA space would induce higher key collision rates as the KBR would be shrunk depending on the number of pools that need to be supported.

6.2 Design Overview

NVMKV encodes pools within the sparse LBA to avoid any need for additional in-memory pool metadata. The encoding is done by directly hashing both the pool ID and the key to determine the hash location within the KBR. This ensures that all KV pairs are equally distributed across the sparse virtual address space regardless of which pool they are in. Distributing KV pairs of multiple pools evenly across the sparse address space not only retains the collision probability properties but also preserves the `get` and `put` performance with pools.

Pool IDs are also encoded within the VBR to optimally search or locate pools within the sparse address space. Encoding pool IDs within the VBR preserves the collision properties of NVMKV. The KV pair start offset within the VBR determines the Pool ID. The VBR size determines the maximum number of pools that can be addressed without hashing, while also maintaining the guarantee that each KV pair is separated from neighboring KV pairs by unpopulated sparse addresses. For example, with a 12 bit VBR, the maximum number of pools that can be supported without pool ID hashing is 1024. If the maximum number of pools is greater than 1024, the logic of `get` is modified to also retrieve the KV pair metadata that contains the pool ID now needed to uniquely identify the KV pair.

6.3 Operations

Supporting pools requires changes to common operations of the KV store. We now describe three important operations in NVMKV that have either been added or significantly modified to support pools.

Creation and Deletion: Pool creation is a lightweight operation. The KV store performs a one-time write to record the pool's creation in its persistent configuration metadata. On the other hand, pool deletion is an expensive operation since all the KV pairs of a pool are distributed across the entire LBA space, each requiring an independent `PTRIM` operation. NVMKV implements pool deletion as an asynchronous background operation. Upon receiving the deletion request, the library marks the pool as invalid in its on-drive metadata, and the actual deletion of pool data occurs asynchronously.

Iteration: NVMKV supports iteration of all KV pairs in a given pool. If no pool is specified, all key-value pairs on the device are returned by the iteration routines. Iteration uses the `ITERATE` primitive of the FTL to find the address of the next contiguous chunk of data in the sparse address space. During pool iteration, each contiguous virtual address segment is examined as before.

However, the iterator also examines the offset within the VBR of each starting address, and compares it to the pool ID, or the hash of the pool ID, for which iteration is being performed. Virtual addresses are only returned to the KV store if the pool ID match succeeds.

NVMKV guarantees that each KV pair is stored in a contiguous chunk, and that adjacent KV pairs are always separated by at least one empty sector, so the address returned by ITERATE locates the next KV pair on the drive (see §5.1). When the maximum number of pools is small enough that each pool ID can be individually mapped to a distinct VBR offset, the virtual addresses returned by the ITERATE primitive are guaranteed to belong to the pool currently being iterated upon. When the maximum number of pools is larger, the ITERATE uses the hash of the pool ID for comparison. In this case, the virtual addresses that match are not guaranteed to be part of the current pool, and a read of the first sector of the KV pair is required to complete the match.

7 Implementation

NVMKV is implemented as a stand-alone KV store written in C++ using 6300 LoC. Our current prototype works on top of ioMemory VSL and interacts with the FTL using the IOCTL interface [25]. The default subdivision for KBR and VBR used in the current implementation is 36 bits and 12 bits respectively, in a 48 bit address space. The KBR/VBR subdivision is also configurable at KV store creation time. To accelerate pool iteration, we implemented filters inside the ITERATE/BATCH-ITERATE FTL primitives. During the iteration of keys from a particular pool, the hash value of the pool is passed along with the IOCTL arguments to be used as a filter for the iteration. The FTL services (BATCH-)ITERATE by returning only populated ranges that match the filter. This reduces data copying across the FTL and NVMKV.

7.1 Extending FTL Primitives

We extended the FTL to better support NVMKV. ATOMIC-WRITE and its vectored forms are implemented in a manner similar to what has been described by Ouyang *et al.* [35]. Atomic operations are tagged within the FTL log structure, and upon restart, any incomplete atomic operations are discarded. Atomic writes are also not updated in the FTL map until they are committed to the FTL log to prevent returning partial results. ITERATE and RANGE-EXISTS are implemented as query operations over the FTL indirection map. CONDITIONAL-READ and CONDITIONAL-WRITE are emulated within NVMKV in the current implementation.

7.2 Going Beyond Traditional KV Stores

NVMKV provides new capabilities with strong guarantees relative to traditional KV stores. Specifically, it provides full atomicity, isolation, and consequently serializability for basic operations in both individual and batch submissions. Atomicity and serializability guarantees are provided for individual operations within a batch, not for the batch itself. The atomicity and isolation guarantees provided by NVMKV rely heavily on the ATOMIC-WRITE and PTRIM primitives from the FTL.

Each put is executed as a single ACID compliant ATOMIC-WRITE, which guarantees that no get running in parallel will see partial content for a KV pair. The get operation opportunistically retrieves the KV pair from the first hash location using cond_range_read to guarantee the smallest possible data transfer. In the unlikely event of a hash collision, the next hash address is used. Since the hash address order is deterministic, and every get or put to the same key will follow the same order, and every write has atomicity and isolation properties, get is natively thread safe requiring no locking.

When ATOMIC-WRITEs are used, put operations require locking for thread safety because multiple keys can map to the same KBR. When a CONDITIONAL-WRITE (which performs an atomic EXISTS check and WRITE of the data in question) is used, put operations can also be made natively thread safe. Individual iterator calls are thread safe with respect to each other and to get/put calls; thus, concurrent iterators can execute safely.

The ITERATE primitive is also supported in batch mode for performance. BATCH-ITERATE returns multiple start addresses in each invocation, reducing the number of IOCTL calls. For each LBA range returned, the first sector needs to be read to retrieve the key for the target KV pair.

8 Evaluation

Our previous work established performance of the basic approach used in NVMKV, contrasting it relative to block device performance [33]. Our evaluation addresses a new set of questions:

(1) How effective is NVMKV in supporting multiple KV stores on the same flash device? How well do NVMKV *pools* scale?

(2) How effective is NVMKV in trading off DRAM for flash by sizing its read cache?

(3) How effective is NVMKV in improving the endurance of the underlying flash device?

(4) How sensitive is NVMKV to the size of its collision cache?

8.1 Workloads and Testbed

We use LevelDB [26], a well-known KV store as the baseline for our evaluation of NVMKV. LevelDB uses a logging-based approach to write to flash and uses compaction mechanisms for space reclamation. Our eval-

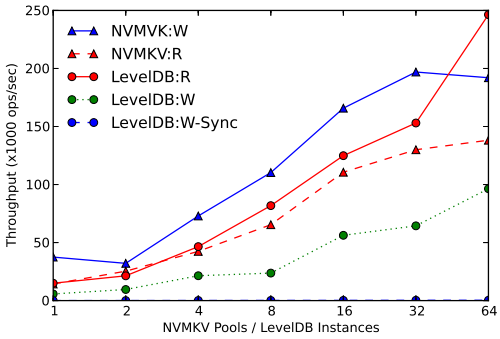

Figure 4: Comparing multiple identical instances of LevelDB with a single instance of NVMKV with equal number of pools.

uation consists of two parts. We answer question (1) above using dedicated micro-benchmarks for NVMKV and LevelDB. We then answer questions (2), (3), and (4) using the YCSB macro-benchmark [19]. We used the YCSB workloads A, B, C, D, and F with a data set size of 10GB; workload E performs short range-scans and the YCSB Java binding for NVMKV does not support this feature currently. We also used a raw device I/O micro-benchmark which was configured so that I/O sizes were comparable to the sizes of the key-value pairs in NVMKV. Our experiments were performed on two testbeds, I and II. Testbed I was a system with a Quad-Core 3.5 GHz AMD Opteron(tm) Processor, 8GB of DDR2 RAM, and a 825GB Fusion-io ioScale2 drive running Linux Ubuntu 12.04 LTS. Testbed II was a system with a 32 core Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz with 128 GB DDR3 RAM, and a 1.2TB Fusion-io ioDrive2 running Linux Ubuntu 12.04.2 LTS.

8.2 Micro-Benchmarks

Our first experiment using Testbed II answers question (1). We ran a single instance of NVMKV and measured the throughput of reads and writes as functions of the number of NVMKV pools. NVMKV also used as many threads as pools. We compared its performance against multiple instances of LevelDB. Both KV stores were configured to use the same workload, sizes of key-value pairs, and accessed a total of 500 MB of data. In addition, LevelDB used both its own user-level cache of size 1GB and the operating system's file system cache as well. On the other hand, NVMKV used neither. LevelDB provides two options for writes, a low-performing but durable *sync* and the high performing *async*, and we include them both here. NVMKV, on the other hand, performs all writes synchronously and atomically, and thus only a synchronous configuration is possible.

Figure 4 provides a performance comparison. Due to its low-latency flash-level operations, NVMKV almost equals LevelDB's primarily in-memory performance for up to 32 pools/instances. LevelDB continues scaling beyond 32 parallel threads; its operations continue to be memory-cache hits while NVMKV must perform flash-level accesses (wherein parallelism is limited) for each operation. When writing, NVMKV outperforms LevelDB's *sync* as well as *async versions* despite not using the filesystem cache at all. Even when LevelDB was configured to use async writes, it was about 2x slower than NVMKV in the best case, and about 6.5x slower at its worst. For synchronous writes, a more comparable setup, NVMKV outperforms LevelDB between 643x (64 pools) and 1030x faster (1 pool).

8.3 DRAM Trade-off and Endurance

This second experiment using Testbed I addresses questions (2) and (3). NVMKV uses negligible in-memory metadata and does not use the operating system's page cache at all. It implements a read cache whose size can be configured, allowing us to trade-off DRAM for flash, thus providing a tunable knob for trading off cost for performance. To evaluate the effectiveness of the collision cache, we evaluate two variants of NVMKV, one without the collision cache and the other when it uses 64MB of collision cache space. We used the YCSB benchmark for this experiment. We present the results from both the *load phase*, that is common to all workload personalities implemented in YCSB, and the *execution phase*, that is distinct across the workloads.

Figure 5 (top) depicts throughput as a function of the size of the application-level read cache available to LevelDB and NVMKV. Unlike NVMKV, LevelDB accesses use the file system page cache as well. Despite this, NVMKV outperforms LevelDB during both phases of the experiment, *load* and *execution*, by a significant margin. Further, the gap in performance increases as the size of the cache increases for every workload. This is because YCSB's workloads favor reads in general, varying from 50%, in the case of workload A, all the way to 100% in the case of workload C. Furthermore, the YCSB workloads follow skewed data access distributions, making even a small amount of cache highly effective.

To better understand these results, we also collected how much data was written to the media while the experiments were running. All workload were configured to use 10GB of data, so any extra data that is written to the media is overhead introduced by NVMKV or LevelDB. Figure 5 (bottom) depicts the results of the write amplification. By the end of each experiment, LevelDB has written anywhere from 42.5x to 70x extra data to the media. This seems to be a direct consequence of its internal design which migrates the data from one level to the next, therefore copying the same data multiple times as it ages. NVMKV on the other hand, introduces a write amplification of 2x in the worst case. We believe this to be

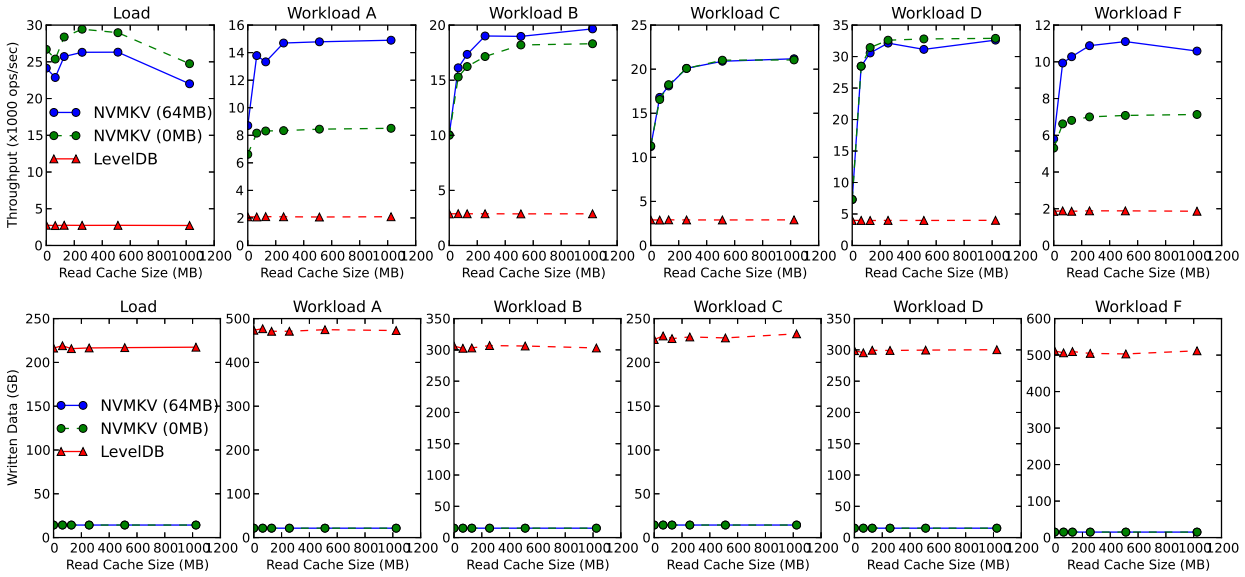

Figure 5: Throughput comparison between NVMKV and LevelDB using YCSB workloads (above). Write Amplification Comparison between NVMKV and LevelDB using YCSB workloads (below).

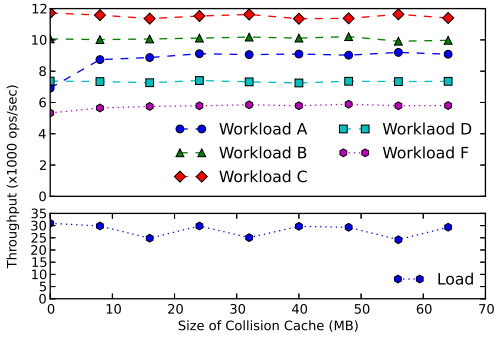

Figure 6: Collision cache impact on YCSB workloads.

the main reason for the performance difference between the two KV Stores.

Finally, the effect of NVMKV's collision cache is highly sensitive to the workload type. As expected, for read-intensive workloads (i.e., B, C, and D) the presence of collision cache has little to no impact at all. On the other hand, the update heavy workloads (i.e., A and F) benefit significantly from the collision cache, increasing the performance up to 76% and 56% respectively. Surprisingly the load phase is negatively affected by the collision cache, and performance decreases by up to 11%.

8.4 Effectiveness of the Collision Cache

We used Testbed II to address question (4). We measured YCSB workload throughput when varying the size of the collision cache in NVMKV. During this experiment, the read cache was disabled to eliminate other caching effects. As shown in Figure 6, the presence of the collision cache benefits workloads A and F with a throughput improvement of 28% and 10% respectively. Read-mostly workloads (B, C, and D) do not benefit from the collision cache since the probability of collision is low and a single flash-level read is necessary to service KV `GET` operations. A and F involve writes and these benefit from the collision cache. The collision cache optimizes the handling repeated writes to the same location by eliminating the reading of the location (to check for collisions) prior to the write. Finally, the loading phase does not demonstrate any benefit from the collision cache mainly because of YCSB's key randomization during inserts.

9 Discussion and Limitations

Through the NVMKV implementation, we were able to achieve a majority of our design goals of building a flash-aware lightweight KV store that leverages advanced FTL capabilities. We made several observations through the design and development process.

It is valuable for atomic KV operations, such as those described by Ouyang *et al.* [35], to be fully ACID compliant. The usage described in Ouyang *et al.*'s work only required the durable writes to have the atomicity property. We found that having *isolation* and *consistency* enables reduced locking and in some cases, fully lock free operation, at the application level. For example, updating multiple KV pairs atomically as a single batch can help provide application-level consistent KV store state without requiring additional locks or logs.

Many primitives required by NVMKV are the same as those required by other usages of flash. FlashTier, a primitives based solid state cache leverages the sparse

addressing model, and the EXISTS and PTRIM FTL primitives [37], as does DirectFS, a primitives based filesystem [4, 29].

NVMKV suffers from internal fragmentation for small KV pairs. Since we map individual KV pairs to separate sectors, NVMKV will consume an entire sector (512B) even for KV pairs smaller than the sector size. While this does not pose a problem for many workloads, there are those for which it does. 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 [31], where small items are initially indexed separately and later compacted into larger units such as sectors. We believe that implementing similar methods within the FTL itself can be valuable.

10 Related Work

Most previous work on FTL-awareness has focused on leveraging FTL capabilities for simpler and more efficient applications, focusing on databases [35], file systems [29] and caches [37]. NVMKV is the first to present the complete design and implementation of an FTL-aware KV store and explains how specific FTL primitives can be leveraged to build a lightweight and performant KV store. NVMKV is also the first to provide support for multiple KV instances (i.e., *pools*) on the same flash device. Further, NVMKV trades-off main memory for flash well as evidenced in the evaluation of a read cache implementation. Finally, NVMKV extends the use of the FTL primitives in a KV store to include conditional-primitives and batching.

There is substantial work on scale-out KV stores and many of the recent ones focus on flash. For example, Dynamo [23] and Voldemort [39] both present scale out KV stores with a focus on predictable performance and availability. Multiple local node KV stores are used underneath the scale out framework and these are expected to provide `get`, `put`, and `delete` operations. NVMKV complements these efforts by providing a lightweight, ACID compliant, and high-performance, single-node KV store.

Most flash-optimized KV stores use a log structure on block-based flash devices [10, 14, 17, 18, 21, 31]. FAWN-KV [14] focused on power-optimized nodes and uses an in-memory map for locating KV pairs at they rotate through the log. FlashStore [21] and SkimpyStash [22] take similar logging-based approaches to provide high-performance updates to flash by maintaining an in-memory map. SILT [31] provides a highly memory optimized multi-layer KV store, where data transitions between several intermediate stores with increasing compaction as the data ages. Unlike the above mentioned systems, NVMKV eliminates an entire additional layer of mapping along with in-memory metadata management by utilizing the FTL mapping infrastructure.

There are several popular disk optimized KV stores [3, 8, 26]. Memcachedb [3] provides a persistent back end to the in-memory KV store, memcached [2], using BerkeleyDB [34]. BerkeleyDB, built to operate on top a black-box block layer, caches portions of the KV map in DRAM to conserve memory and incurs read amplification on map lookup misses. MongoDB, a cross-platform document-oriented database, and LevelDB, a write-optimized KV store, are HDD based KV stores. Disk-based object stores can also provide KV capabilities [28, 30, 32, 36]. Disk-based solutions do not work well on flash because the significant AWA that they induce reduces the flash device lifetime by orders of magnitude [33].

Finally, we examine the role of consistency in KV stores in the literature. Anderson *et al.* analyze the consistency provided by different KV stores [15]. They observe that while many KV stores offer better performance by providing a weaker form of (eventual) consistency, user dissatisfaction when violations do occur is a concern. Thus, while many distributed KV stores provide eventual consistency, others have focused on strong transactional consistency [38]. NVMKV is a unique KV store that leverages the advanced capabilities of modern FTLs to offer strong consistency guarantees and high-performance simultaneously.

11 Conclusions

Leveraging powerful FTL primitives provided by a flash device allows for rapid and stable code development; application developers can exploit features present in the FTL instead of re-implementing their own mechanisms. NVMKV serves as an example of leveraging and enhancing capabilities of an FTL to build simple, lightweight but highly powerful applications. Through the NVMKV design and implementation, we demonstrated the impact to a KV store in terms of code and programming simplicity and the resulting scalable performance that comes from cooperative interaction between the application and the FTL. We believe that the usefulness of primitives for FTLs will only grow. In time, such primitives will fundamentally simplify applications by enabling developers to quickly create simple but powerful, feature-rich applications with performance comparable to raw devices.

Acknowledgments

We thank the many developers of NVMKV, an open source project [12]. We would like to thank Akshita Jain for her help with our experimental setup. And we thank our shepherd, Haryadi Gunawi and the anonymous reviewers for their insightful feedback.

References

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

[2] memcached. http://memcached.org/.

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

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

[5] FNV1a Hash. http://www.isthe.com/chongo/tech-/comp/fnv/index.html, 1991.

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

[7] MemSQL. http://www.memsql.com, 2014.

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

[9] NVM Primitives Library. http://opennvm.github.io/, 2014.

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

[11] What is SAP HANA? http://www.saphana.com/community/about-hana, 2014.

[12] NVMKV. https://github.com/opennvm/nvmkv, 2015.

[13] David Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proc. of ACM SOSP*, 2009.

[14] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. Technical report, 2009.

[15] Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. What Consistency Does Your Key-Value Store Actually Provide? october 2010.

[16] Anirudh Badam and Vivek S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *Proc. of the USENIX NSDI*, 2011.

[17] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. HashCache: Cache Storage for the Next Billion. In *Proc. of the USENIX NSDI*, 2009.

[18] Mateusz. Berezecki, Eitan. Frachtenberg, Michael. Paleczny, and Kenneth Steele. Many-Core Key-Value Store. In *Green Computing Conference and Workshops (IGCC)*, july 2011.

[19] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. June 2010.

[20] Dhananjoy Das, Dulcardo Arteaga, Nisha Talagala, Torben Mathiasen, and Jan Lindström. NVM Compression—Hybrid Flash-Aware Application Level Compression. In *Proc. of the USENIX INFLOW)*, October 2014.

[21] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. *Proc. of VLDB*, September 2010.

[22] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. In *Proc. of the ACM SIGMOD*, 2011.

[23] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *Proc. of the ACM SIGOPS*, October 2007.

[24] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proc. of the ACM SIGMOD*, 2013.

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

[26] Sanjay Ghemawat and Jeff Dean. LevelDB. https://code.google.com/p/leveldb/, 2011.

[27] Laura M. Grupp, John D. Davis, and Steven Swanson. The Bleak Future of NAND Flash Memory. In *Proc. of the USENIX FAST*, 2012.

[28] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software Persistent Memory. In *Proc. of USENIX ATC*, 2012.

[29] William Josephson, Lars Bongo, Kai Li, and David Flynn. DFS: A File System for Virtualized Flash Storage. In *Proc. of the USENIX FAST*, February 2012.

[30] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore Database System. *Commun. ACM*, 34:50–63, October 1991.

[31] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proc. of the ACM SOSP*, October 2011.

[32] Barbara Liskov, Atul Adya, Miguel Castro, Sanjay Ghemawat, Umesh Gruber, Robert a nd Maheshwari, Andrew C. Myers, Mark Day, and Liuba Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. of ACM SIGMOD*, 1996.

[33] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. In *Proc. of the USENIX HotStorage*, June 2014.

[34] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. Berkeley DB. In *Proc. of the USENIX ATC*, 1999.

[35] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond Block I/o: Rethinking Traditional Storage Primitives. In *High Performance Computer Architecture*, Feb 2011.

[36] Mahadev Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steer, and James J. Kistler. Lightweight Recoverable Virtual Memory. *Proc. of ACM SOSP*, December 1993.

[37] Mohit Saxena, Michael Swift, and Yiying Zhang. FlashTier: A Lightweight, Consistent and Durable Storage Cache. In *European Conference on Computer Systems*, April 2012.

[38] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: Reliable Transactional P2P Key/Value Store. In *Proc. of the ACM SIGPLAN workshop on ERLANG*, 2008.

[39] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving Large-scale Batch Computed Data with Project Voldemort. In *Proc. of the USENIX FAST*, 2012.

[40] Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. Using Vector Interfaces to Deliver Millions of IOPS from a Networked Key-value Storage Server. In *Proc. of the ACM SoCC*, 2012.

[41] Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. Using Vector Interfaces to Deliver Millions of IOPS from a Networked Key-value Storage Server. In *Proc. of the ACM Symposium on Cloud Computing*, 2012.

[42] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't Stack Your Log On My Log. In *Proc. of the USENIX INFLOW*, October 2014.

[43] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, Swaminathan Sundararaman, and Robert Wood. HEC: Improving Endurance of High Performance Flash-based Cache Devices. In *Proc. of the SYSTOR*, 2013.