2017 IEEE International Symposium on High Performance Computer Architecture

KAML: A Flexible, High-Performance Key-Value SSD

Yanqin Jin* Hung-Wei Tseng*† Yannis Papakonstantinou* Steven Swanson*

*Department of Computer Science and Engineering, University of California, San Diego

†Department of Computer Science, North Carolina State University

Abstract

Modern solid state drives (SSDs) unnecessarily confine host programs to the conventional block I/O interface, leading to suboptimal performance and resource under-utilization. Recent attempts to replace or extend this interface with a key-value-oriented interface and/or built-in support for transactions offer some improvements, but the details of their implementations make them a poor match for many applications.

This paper presents the *key-addressable, multi-log SSD* (*KAML*), an SSD with a key-value interface that uses a novel multi-log architecture and stores data as variable-sized records rather than fixed-sized sectors. Exposing a key-value interface allows applications to remove a layer of indirection between application-level keys (e.g., database record IDs or file inode numbers) and data stored in the SSD. KAML also provides native transaction support tuned to support fine-grained locking, achieving improved performance compared to previous designs that require page-level locking. Finally, KAML includes a caching layer analogous to a conventional page cache that leverages host DRAM to improve performance and provides additional transactional features.

We have implemented a prototype of KAML on a commercial SSD prototyping platform, and our results show that compared with existing key-value stores, KAML improves the performance of online transaction processing (OLTP) workloads by $1.1\times$ – $4.0\times$, and NoSQL key-value store applications by $1.1\times$ – $3.0\times$.

I. Introduction

Flash memory is replacing spinning disks as the storage medium of choice for many enterprise storage systems. Solid state drives (SSDs) offer decreased latency, increased bandwidth, and increased concurrency, all of which make them attractive for modern, data-intensive applications. However, the flash memory that SSDs use to store data is a poor match for the block-based, disk-centric interface that dominates storage systems. The key-value interface is an attractive alternative, since SSDs must already maintain a layer of indirection between logical storage addresses (or keys) and physical storage locations. This allows for simpler implementations and makes it possible to exploit the layer of indirection to provide additional services like snapshots or multi-part atomic operations.

However, current proposals for key-value interfaces for SSDs fall short in several respects. First, they provide a single, shared map for all the key-value pairs in the SSD, forcing the SSD to use uniform policies to manage the key-value mapping across all applications. Second, the proposals stop at the SSD and ignore the rest of the storage stack. In particular, they do not provide generic caching facilities to improve the performance of key-value accesses.

Most proposals for multi-part atomic writes in flash-based SSDs also suffer from limitations. Most notably, they support only page-level atomic updates for transactions. While the page-level mapping that most SSDs employ makes this a convenient choice for the implementors, it is a poor match for what many applications require. Providing page-level atomic writes requires the application to perform page-level locking. For applications that deal with smaller units of data (e.g., database records), page-level locking has been shown to provide lower performance than finer-grained approaches.

To address these limitations, this paper presents the *key-addressable, multi-log SSD* (*KAML*). KAML extends existing proposals for key-value-based SSDs and provides a fine-grained multi-part atomic write interface and a host-side caching layer to accelerate accesses to data it contains. The caching layer builds on the SSD's transaction interface and implements a fine-grained locking protocol that minimizes transaction aborts and maximizes concurrency.

KAML allows applications to create multiple key-value *namespaces* that can, depending on application requirements, represent files, database tables, or arbitrary collections of objects.

This new interface leads to three benefits. First, KAML shoulders the responsibility of mapping keys directly to values' *physical* addresses, allowing applications to bypass unnecessary indirection. Without KAML, applications have to maintain their own indices to map keys to file offsets, and rely on the file system to translate file offsets to logical block addresses (LBAs).

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.15

IEEE computer society

Second, applications can create or update multiple key-value pairs atomically using KAML interface. Without this interface, applications must use write-ahead logging (WAL) or other application-level techniques to provide atomic updates, consuming more space and incurring expensive file system operations such as `fsync`. Finally, the fine-grained key-value interface of KAML improves system concurrency in comparison with its coarse-grained counterparts, because it allows for fine-grained locking.

KAML's approach to managing its internal flash memory reflects the requirements of its interface. Modern SSDs have multiple, parallel, semi-independent flash channels and can perform flash operations in parallel across those channels. KAML maps those channels to the logs it uses to record updates to particular namespaces, and allows the application to tune the mapping between namespaces and flash channels to optimize performance and improve quality of service.

We implement the SSD functions of KAML by extending the firmware inside a commercially available flash-based SSD reference design. We have also implemented KAML caching layer that runs on the host machine. The result shows that the KAML caching layer can serve as both a database storage manager and a stand-alone key-value store: on online transaction processing (OLTP) workloads, it outperforms Shore-MT [1], an open-source storage engine, by $1.1\times$ – $4.0\times$. For NoSQL key-value store workloads KAML is $1.1\times$ – $3.0\times$ faster than Shore-MT.

The rest of the paper is organized as follows. Section II provides background for KAML. Section III presents an overview of the system, and Section IV details the implementation. Section V reports experimental results. Section VI places this work in the context of existing literature, while Section VII concludes.

II. Motivation and background

KAML exploits the characteristics and requirements of flash memory to provide a rich transactional key-value-based interface to applications. Previous proposals for transaction support in SSDs have provided transactions for page-level operations, since these coarse-grained transactions are a natural fit for how conventional SSDs manage the flash they contain. However, many applications (e.g., MySQL's InnoDB [2], Oracle Database [3], and Shore-MT [1]) rely on fine-grained transactions to achieve high concurrency. KAML also provides a novel approach to flash management that supports fine-grained transactions while still taking full advantage of modern SSD hardware.

Below, we introduce current SSD and flash, flash translation layer (FTL) design and transactional support in SSDs. The next section describes KAML in detail.

A. Flash memory and SSD

Flash memory has three important characteristics that KAML needs to accommodate. First, flash read, program, and erase operations work on different granularities and have very different latencies. Read and program operate on 4–8 KB pages. Once written, the page becomes immutable unless the *block* that contains it is erased. Blocks typically contain between 64 and 512 pages, and pages within a block must be written sequentially. Reading a page generally takes less than 100 μs, while programming a page takes 100 μs to 2000 μs depending on the underlying flash devices [4]. Erase operations need several milliseconds to complete. Finally, each block can endure only a limited number of erase operations before becoming unreliable.

To address the limitations of flash and improve both performance and reliability of data access, SSDs provide FTLs that map logical block addresses (LBAs) to physical page numbers (PPNs) for each access.

Since pages are immutable once written (until an erase), in-place updates are not possible. Instead, a conventional FTL directs incoming writes to erased pages and updates the map between LBA and PPN. The FTL allows the SSD to expose a generic block device abstraction to the host and conceals the internal details of wear-leveling and garbage collection (GC).

Modern SSDs usually have multiple flash channels providing abundant internal I/O bandwidth. SSDs contain embedded processors to execute firmware for the FTL, wear leveling and GC. SSDs' internal data structures (e.g., the address mapping table, block metadata, and snapshots) reside in an on-board DRAM.

The NVM Express (NVMe) is a new standard interface [5] for high-end SSDs that attach to host systems via PCIe. NVMe is block-oriented, but it allows for vendor-specific extensions.

B. FTL implementation

The two core features that modern FTLs share are a logical-to-physical address mapping table and a facility for GC. Conventional FTLs use the mapping table to keep track of the mapping from logical to physical addresses. The FTL manages flash memory in a log-structured way – it does not write data back to its original physical location and updates a map so it can locate the most recent value. These out-of-place updates lead to stale copies of that data (i.e., "garbage"), and the SSD must reclaim (or "collect") the garbage to make way for new writes.

To maintain the mapping, facilitate garbage collection, and store error correction information, the FTL also stores per-page metadata in the so-called "out-of-band" (OOB) region of each flash page. The OOB region usually occupies between 128 and 256 bytes.

C. Transactions in SSDs

Many applications require some kind of transactional support [6], and several groups have proposed adding transactional support to SSDs [7], [8], [9], [10]. KAML (and these proposals) provides native support for atomicity and durability, two key aspects of transactional ACID support, but leaves isolation and consistency to the application.

Leveraging conventional FTL designs to support atomicity is tempting, since the FTL already relies on atomic copy-on-write (COW). However most conventional FTLs are page-based, and many existing proposals [9], [10], [7] for atomic writes in the FTLs enable transactions to update data in the unit of pages. Another proposal [11] supports finer-grained locking, but relies on a more exotic byte-addressable non-volatile memory rather than flash.

Page-based atomic write interfaces are problematic when the application allows concurrent accesses to records on the same page [12]. If one transaction commits a page successfully, while another transaction with updates on the same page aborts, there is no copy of the page reflecting the correct state. Consequently transactions that use page-based atomic write have to acquire coarse-grained page locks to ensure correctness, which is detrimental to system performance. In contrast, most database storage engines e.g. InnoDB [2], Oracle Database [3], Shore-MT [1], etc. allow transactions to lock individual records. Therefore, they cannot benefit from page-based atomic write interfaces.

III. System overview

The heart of KAML is an SSD that uses a novel FTL structure to manage flash memory. This FTL provides a transactional, key-value interface. The system comprises a customizable SSD and three different pieces of software. First, the new FTL runs on an industrial SSD reference platform. Its interface allows host software (e.g., a database, file system, or other user space application) to create, manage, and access multiple key-value stores using fine-grained transactions. The firmware works in tandem with a kernel driver and a userspace library that run on the host machine. Finally, a host caching layer accelerates access using host DRAM and provides additional transactional facilities.

Figure 1 illustrates the relationship between these components. This section describes KAML's system components while next section describes its implementation in more detail.

KAML presents a key-value interface that supports fine-grained transactions. The interface allows the application to create and destroy key-value *namespaces* that represent logically related key-value pairs. Namespaces allow multiple, independent applications to share a KAML SSD. Table I summarizes the commands included in this interface.

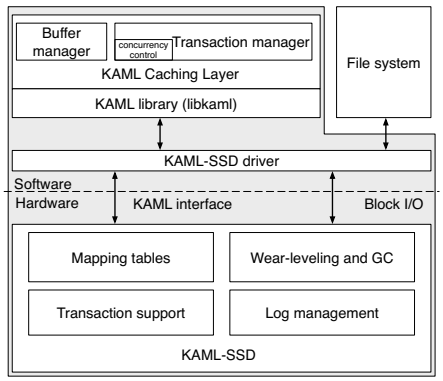


Figure 1: **System architecture** The shaded part represents the KAML system with four components: KAML-SSD, a driver, a user library (libkaml) and KAML caching layer.

A. Keys, values, and namespaces

Keys in KAML are 64-bits, but values can vary in size. Natively supporting variable-sized values lets application store a wide variety of objects in KAML. For instance, a conventional page-based file system could treat keys as block addresses and store 4 KB pages as values in a namespace. Alternately, a database could store individual tuples as values and use the tuple's key or record ID to map it into the SSD.

Exposing a key-value interface allows applications to eliminate redundant layers of indirection. In a conventional system, an application might map keys to locations in a file, the underlying file system would map that file location to a logical block address (LBA), and the FTL would map the LBA to a physical page number (PPN). KAML translates this into a single mapping from key to PPN and eliminates the application and file system overhead associated with the other mappings.

KAML also obviates the need to use log-structured techniques in the file systems, eliminating the performance problems that "log stacking" [13] can cause. The interface allows KAML to consolidate garbage collection operations in the SSD, where the firmware has the most complete information about the flash's performance characteristics and the data layout.

B. Atomicity and durability

KAML's `Put` operation can atomically insert/update multiple key-value pairs, providing atomicity and durability (the "A" and "D" in ACID). The KAML caching layer can provide isolation. We leave consistency up to the application, because while atomicity and durability requirements are similar across applications, consistency requirements (e.g., database cascades, triggers, and constraints) vary widely. Likewise, if an application needs a custom locking protocol it can provide its own rather than rely on the caching layer's facility.

Command	Description
`CreateNamespace`(attributes)	Create a namespace with given attributes, and return a namespaceID.
`DeleteNamespace`(namespaceId)	Delete a namespace with given namespaceID
`Get`(namespaceID, key)	Retrieve a value given its key and returns the value.
`Put`(namespaceIDs[], keys[], values[], lengths[])	Atomically update or insert a list of key-value pairs.

Table I: **KAML commands** These commands allow applications to access data on the SSD with transactional semantics and key-value addressing scheme.

C. Fine-grained locking

Although the KAML SSD does not implement concurrency control, allowing for efficient concurrency control is a central goal of the KAML SSD interface, and the caching layers makes extensive use of it. Since KAML transactions operate on variable-sized key-value pairs [1] rather than fix-sized pages, applications can use fine-grained, record-based locking protocols rather than coarse-grained, page-based protocols.

Fine-grained locking is critical to performance. Without it, a transaction that modifies a record in a page must hold a lock for the entire page until the transaction commits, blocking any other transactions that need access to another record on the same page. We quantify the performance impact of coarse-grained locking in Section V.

D. KAML caching layer

KAML can improve application performance by caching data in DRAM, just as conventional systems cache page-based SSD or hard drive data. Systems with conventional SSDs cache data either in the operating system's page cache or application-specific caching layers (e.g., database buffer pool). KAML requires a different caching architecture because its interface is based on key-value pairs rather than pages or blocks.

KAML's caching layer provides both host DRAM-based caching and a richer transactional interface that provides isolation in addition to durability and atomicity that the KAML SSD provides.

Caching The caching layer allows KAML to provide fast access to cached data and interact with the SSD via `Get` and `Put` commands. The caching layer differs from the conventional page cache in that it caches variable-length key-value pairs instead of fixed-sized pages (or blocks in file system terminology).

The caching layer uses the namespace ID and the key to form a compound key and probes a hash table. If the hash table has an entry that matches the key, the key-value pair is already in the cache, and the caching layer returns the key-value pair to the application.

If the hash table does not have a matching entry, the caching layer issues a `Get` request to the SSD, and inserts the resulting key-value pair into the hash table.

When a transaction commits (or when the caching layer runs out of space), the caching layer writes key-value pairs back to the SSD with a `Put`. Once the command completes, KAML can re-use the buffer space that the value occupied or keep the key-value pair in the cache and mark it as clean.

[1] In this paper, we use record and key-value pair interchangeably

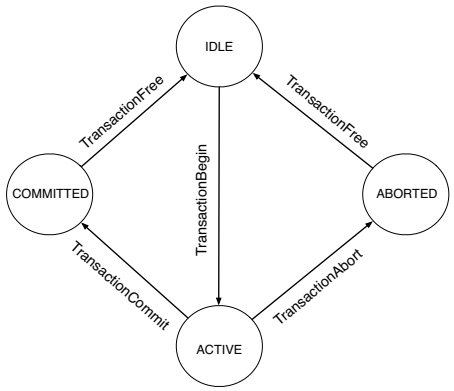


Figure 2: **State transition graph of a transaction** A transaction is in one of these states at any time, and transitions to another state by invoking KAML APIs.

Transactions The caching layer's transaction manager provides support for isolation using strong strict two-phase locking (SS2PL) [14] implemented on the host, and accesses key-value pairs stored on KAML via the caching layer.

The transaction manager maintains an array of transaction control blocks (XCB). When a transaction starts, the transaction manager allocates a XCB for the transaction. The XCB allows the transaction to be in one of the four states: IDLE (i.e., the transaction has not yet started), ACTIVE (i.e., the transaction is in progress), ABORTED, and COMMITTED. Figure 2 shows the possible transitions between these states, and Table II summarizes the actions that cause state transitions as well as transaction manager's API.

Active transactions acquire locks on key-value pairs before accessing them, and create private copies of the key-value pairs for use during the transaction. When the transaction commits, the transaction manager replaces the key-value pairs in the buffer pool with its private copies and then issues `Put` commands to flush the changes to KAML. On abort, the transaction simply discards its private copies and releases its locks.

IV. KAML

KAML uses a customized SSD, custom driver, and lightweight userspace library to implement the KAML interface more efficiently than prior SSDs that provide a key-value interface. The firmware manages in-storage

libkaml API	Description
`TransactionBegin()`	Start a new transaction and allocate resources.
`TransactionRead()`	Read a key-value pair in buffer pool or issue `Get` to bring it from SSD.
`TransactionUpdate()`	Update a key-value pair. The change stays in main memory until transaction commits.
`TransactionInsert()`	Insert a key-value pair. The record stays in main memory until transaction commits.
`TransactionCommit()`	Commit current transaction, persisting updates and releasing locks.
`TransactionAbort()`	Abort current transaction, abandoning updates and releasing locks.
`TransactionFree()`	Release the resources that the current transaction is using.

Table II: **KAML caching layer API** The caching layer provides a transactional interface.

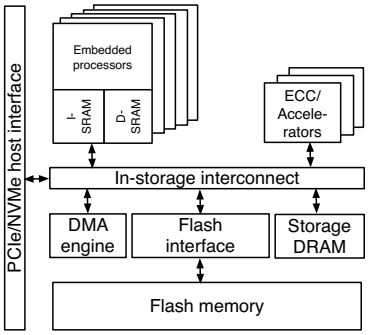


Figure 3: **KAML architecture** KAML exposes its internal computation power to the host via PCIe/NVMe interface.

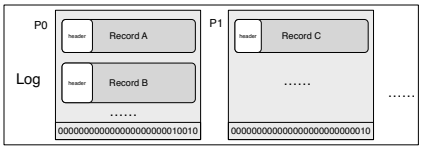


Figure 4: **In-storage log and records** Logs consist of flash pages, each of which stores a bitmap in the OOB region describing variable-sized records in the page. Record A occupies chunk 0 and 1 of page P0, record B occupies chunk 2, 3, and 4 of P0, record C occupies chunk 0 and 1 of P1.

logs, maintains the mapping tables, performs wear leveling, and implements garbage collection (GC). Below, we describe the SSD prototyping hardware we use and key aspects of KAML's firmware and system software.

A. Hardware architecture

We have implemented KAML on a commercial NVMe [5] SSD development board. Like other modern SSDs, it comprises an array of flash chips (totalling 375 GB) arranged in 16 "channels" connected to a multi-core controller that communicates with a host system over PCIe. The firmware running on the controller defines the SSD's interface and implements management policies that aim to provide high performance and maximize flash life.

Figure 3 depicts the internal organization of the flash controller. The key features of the controller architecture are the cores themselves, the flash interface, and the on-board DRAM.

The cores have 64 KB private instruction and data memories. They can communicate with each other, the DRAM, and the flash interface using an on-chip network. To access flash, the cores issue a command to flash controller which reads or writes data from/to a buffer in DRAM. To access the contents of DRAM, the cores explicitly copy between the DRAM and their private data memories.

The flash channels each contain 4 flash chips that share control and data lines and attach to common flash memory controller on the SSD's controller. The flash chips can perform read, program, and erase operations in parallel, but only one chip can transfer data to or from the controller at a time.

For this work, we assume that the DRAM is persistent. In practice, it would either be battery or capacitor-backed or it could be replaced with a non-volatile technology like Intel's 3D-XPoint memory [15].

B. Namespace management

KAML manages the flash memory in each flash target as a log. When a `Put` operation arrives, KAML writes the key-value pair to the tail of the log, and updates an index structure that stores the location of the key-value pair corresponding to each key.

The architecture of the SSD determines the number of logs that the SSD maintains, KAML assigns each key-value namespace to multiple logs. This improves locality and leads to more efficient garbage collection (see below). However, the correspondence between namespaces and logs is not fixed: as workloads change the SSD can assign more or fewer logs to a single namespace to match workload the namespace is experiencing. If a namespace is particularly cold it might share a log with other namespaces. By default, all of the SSD's logs are available to all the namespaces. Multiple logs do not incur extra hardware overhead since modern SSDs already feature multi-channel architecture with internal parallelism.

Restricting the mapping between namespaces and logs serves as a locality optimization allows for the SSD to control the allocation of resources – especially bandwidth – to namespaces. However, since the mapping is flexible, KAML's GC and flash management algorithms do not assume that data for a particular namespace resides in its assigned logs.

To bridge the gap between application requirements and flash memory characteristics, KAML stores

variable-sized key-values in fixed-sized flash pages. Physical flash pages in our SSD are 8 KB + 256 bytes in size. Each page is divided into 64 fixed-sized chunks and each record can occupy variable number of chunks. The firmware uses 8-bytes of OOB data to store a bitmap that records the start and end chunks of each record on the page, as shown in Figure 4.

If a record's last chunk is the i-th chunk, the i-th bit of the bitmap is set. The first record always starts from chunk 0 of the page, and there are no unused chunks between two records on the same page. Other components of the firmware such as GC can use this bitmap to parse the content of the page (see Section IV-E).

Flash supports only whole-page program operations, but it must commit individual `Put` operations even if they do not fill a page. To overcome this problem, KAML uses a non-volatile, DRAM-like buffer to buffer multiple `Put` operations until a page's-worth of data is ready to write or an internal timer times out. Existing SSDs use capacitor- or battery-backed memory for this purpose, but emerging non-volatile, byte-addressable memories like 3D XPoint would also be suitable.

C. Mapping tables

KAML uses per-namespace indices to locate the key-value pairs associated with each key. KAML indices are different from the maps that the FTLs in conventional SSDs in two ways. First, conventional FTLs map LBAs to PPNs, and the LBAs form a continuous block address space. This means that the meaning of an LBA is fixed (i.e., it corresponds to a particular logical address in the SSD's logical storage space), so it cannot contain useful application-level data (e.g., they cannot represent record IDs in a database table). It also means that LBAs correspond to fixed-sized chunks of data.

Second, in conventional FTLs, there is only a single mapping table, rather than many. This further limits the flexibility in how applications can use LBAs, since LBAs reside in a global, shared namespace. It also prevents the SSD from managing indices differently depending on access patterns or application needs.

For instance, KAML could limit the size of the mapping table for a namespace or even use different data structures (e.g., a tree instead of the hash tables KAML uses) to store the mapping tables.

We store the indices in the SSD's DRAM when the namespace is in use. Like other modern, high-end SSDs ours has several GBs of DRAM that we use to store the KAML indices. Since the KAML indices can be finer-grained they may be larger than the conventional LBA-to-PPN map. For example, a hash table for 100 million key-value pairs and a load factor of 75% requires approximately 2 GB in-storage DRAM.

To support larger data sizes, either the amount of DRAM the SSD provides may need to increase or the firmware may need to "swap" index information between flash and DRAM. KAML currently loads the index for a namespace when an application accesses it, but it does not swap parts of the index. KAML employs a simple policy to swap unused mapping tables out to flash to make room for those in use before the application starts.

D. Transaction support

KAML's transaction mechanism supports fine-grained data access with high concurrency. This design decision separates KAML from most previous proposals for multi-part atomic write support in SSDs [9], [10]. Specifically, KAML-SSD provides native support for fine-grained, multi-record atomic writes, while the KAML caching layer provides support for data buffering and locking. KAML's native atomic write provides durability and atomicity, and the caching layer implements isolation.

The `Put` command provides the transactional interface. It accepts arrays of namespace IDs, keys, values, and value sizes that specify a set of updates/insertions to apply, and the SSD guarantees that all the updates/insertions occur atomically.

`Put` executes in three phases. In the first phase, the firmware receives a list of key-value pairs to update or create. The KAML firmware transfers the data to the SSD's non-volatile RAM. Then, the firmware checks if each key already exists in the namespace's index. If it does, it locks that entry. If the key does not exist, the firmware reserves and locks an empty entry in the index. After the firmware has acquired the locks, the operation has logically committed, and the firmware notifies the host.

In the second phase, the firmware initiates flash write operations to write the key-value pairs to flash. When a flash write completes, the firmware records its physical address. If a failure occurs before all flash write operations succeed, the firmware recovers using the data in the non-volatile buffers.

Once all the flash write operations are complete, the firmware has the information of all new flash addresses. The firmware updates the indices with the physical addresses of the newly-written key-values in flash memory. Once the firmware finishes applying the changes to the mapping tables, it releases locks, frees the buffers and marks the command as success. If a failure occurs, the firmware recovers using the flash addresses recorded in the second phase.

E. Wear-leveling and GC

KAML's GC algorithm operates on variable-sized key-value pairs rather than fixed-sized pages, but the basic operations are similar to other SSD GC schemes. The GC subsystem maintains a list of erased flash blocks for each log and selects a new one when the

block at the end of the log is full. After writing all pages in a block, KAML firmware moves the block to a sorted list that orders blocks by their erase count and the amount of valid data (i.e., bytes that have not been replaced by a newer version of the same record).

KAML's GC algorithm starts reclaiming space when the free block count falls below a threshold. It selects blocks to clean that have low erase counts and small amounts of valid data. This serves to spread erases evenly across the blocks and minimize the work needed to copy the valid data to a new block.

Once it has selected a flash block to clean, the firmware reads the pages from flash memory into the storage DRAM, and uses the per-page bitmap described in Section IV-B to extract all the key-value pairs. For each key-value pair, the firmware searches for the key in the index. If the search result matches the physical address of the current key-value pair being scanned, the key-value pair is valid and the firmware writes it back to a new location. If the firmware cannot find an entry for the key-value pair or the search result points to a different physical location, the key-value pair is invalid and the firmware discards it. After examining all pages in a block, the firmware erases the block and adds it to the list of free blocks.

V. Evaluation

We use microbenchmarks to measure basic operation performance of KAML, discussing factors that affect KAML performance. Then we run more complex OLTP and NoSQL workloads to quantify how KAML design decisions impact application-level performance. Below we detail our experimental setup and the results of our testing.

A. Experimental setup

We built KAML using an industrial flash-based SSD reference platform that connects to the host machine via four-lane PCIe Gen 3. The controller of the SSD consists of multiple embedded processors running at 500 MHz. Data structures such as address mapping tables reside in the 2 GB on-board DRAM. We assume that the SSD uses a capacitor or battery to protect it from abrupt power failures. We implemented KAML by modifying a reference firmware provided by the board manufacturer. The original reference firmware supports only block-based I/O, while our implementation exposes the KAML commands as extensions to the standard NVMe command set. We make corresponding extensions to the Linux NVMe driver. Thus KAML is fully compatible with block devices. Before running the experiments, we preconditioned the device by filling the SSD with random data multiple times.

Our host system has two quad-core Intel Xeon E5520 CPUs on a dual-socket motherboard. This machine contains 64 GB of DRAM as the host main memory. The machine runs a Linux 3.16.3 kernel. We obtained all the results without using hyper-threading.

For microbenchmarks that measure bandwidth, we use eight threads running on the host machine to continuously issue I/O requests to the SSD. To eliminate the overhead of Linux virtual file system (VFS), the driver and the user-space library allow the baseline program to issue `read` and `write` commands directly to the SSD. The KAML version of microbenchmarks use `Get` and `Put` commands to manipulate a single record in each command. To measure KAML's latency, we issue commands using a single thread.

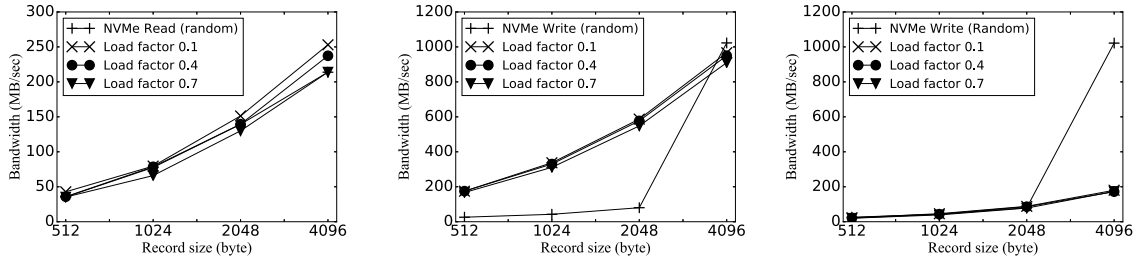
For OLTP and NoSQL key-value store applications, we compare KAML with Shore-MT [1], an open-source storage engine that offers ACID guarantees using a combination of ARIES-style [16] logging and two-phase locking (2PL). In our experiments, Shore-MT provides the same functionality as KAML with techniques used in many popular databases, thus serves as an optimized baseline with low overhead. Shore-MT relies on a file system to store user data and logs. Unless otherwise noted, we configure Shore-MT to use record-level locks, since our measurements show that they provide better performance than page-level locks.

B. Microbenchmarks

We use `Fetch`, `Update` and `Insert` microbenchmarks to understand the performance characteristics of KAML. The baseline version of `Fetch` uses NVMe `read` commands to retrieve sectors from the SSD. The baseline version of `Insert` uses NVMe `write` commands to write sectors of data to previously unmapped LBAs, and `Update` uses NVMe `write` commands to write new versions of data to mapped LBAs. The KAML version of `Fetch` uses `Get` commands to retrieve records by their keys directly. For `Update` and `Insert`, we use `Put` commands to modify existing records and create new ones, respectively.

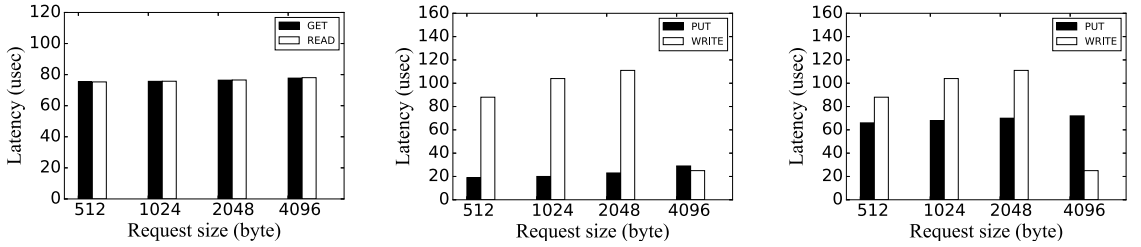
Figure 5 compares the throughput of `read`/`write` with that of `Get`/`Put` commands while varying the value size. The test uses one 1024 MB mapping table that can hold up to 64 million entries. Since the number of elements in the mapping table can affect KAML performance, Figure 5 reports the throughput of KAML requests when the mapping table has different number of elements, i.e. load factors.

The experimental results demonstrate that when the load factor of the mapping table is 0.1, the bandwidth of `Get` can be up to 1.2× of `read`. When the load factor is 0.4, `Get` and `read` achieve the same bandwidth. `read` starts to outperform `Get` after the load factor surpasses 0.7. `Get` achieves greater throughput than `read` because `Get` avoids the cost of LBA indirection. The SSD firmware has to obtain locks on LBA ranges to protect the data within the LBA range from changing or migrating during the `read`



(a) Fetch (b) Update (c) Insert

Figure 5: **Bandwidth comparison between KAML and block IO** Get outperforms read by up to 20% in Fetch. Put outperforms write by up to 7.9× for small requests in Update and 10% in Insert. Put's overhead is greater than that of write when Put inserts *new* elements whose sizes are 4KB.



(a) Fetch (b) Update (c) Insert

Figure 6: **Latency comparison between KAML and block IO** Get has almost the same latency as read for Fetch. Put latency is much lower than that of write (20%) for small requests in Update since writing small records incurs flash read-modify-write. In Insert the latency of Put is between 63% and 75% of that of write for small requests due to the same reason.

command. The performance gain of Get diminishes when the mapping table has more elements, since the firmware has to scan more mapping table entries to search for an element.

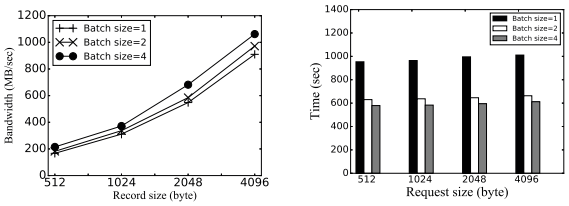
In Update, Put outperforms write by 6.7× – 7.9× when the request size is smaller than 4 KB. write achieves marginal improvement over Put when the record size reaches 4 KB. The performance of write sees a huge leap when the record size reaches 4 KB because in the baseline, write requests smaller than 4 KB are treated as read-modify-write, indicating the command cannot return before the firmware reads 4 KB from flash into the RAM and modifies a portion of it. When the request size is 4 KB, the command can return after writing the new data into persistent DRAM in the SSD. KAML does not suffer from this limitation for small records since it adopts a log-structured approach and never overwrites previous data.

In Insert, the throughput of Put is close to that of write for requests smaller than 4 KB. write outperforms Put for 4 KB requests. On the one hand, 4 KB write does not have to perform long-latency read-modify-write operation. On the other hand, Put needs to insert new address mapping entries into the hash-based mapping table while write updates an element in an array.

Figure 6 compares the latency of read/write with that of Get/Put commands with different value size when the hash table load factor is 0.4.

Get and read have the same latency. For Update, Put latency is only 20% of that of write for small requests. The latency of write for small requests is high due to the read-modify-write described before. In contrast, Put does not suffer from the latency increase when the record size is smaller than 4 KB. For Insert, the latency of Put is between 63% and 75% of write when request size is smaller than 4 KB. For 4 KB requests, Put latency is 2.9 × of write. Our experiments also show that for Get, 98% of the latency comes from hardware including the PCIe link and SSD internal latency, while the remaining 2% comes from software including user space and OS kernel. In Update, hardware contributes 92% of the latency. In Insert, hardware latency is 97% of the total.

In addition to updating/inserting a single record with each command, the Put command supports updating/inserting a *batch* of records atomically. The number of records in a batch is the *batch size*. Figure 7 measures the effect of batch sizes on performance and shows that increasing batch size leads to larger bandwidth. Specifically, increasing the batch size from 1 to 4 leads to 1.2× – 1.3× increase in throughput for Update. It also reduces the amount of time required to add data to the namespace by 40%.



(a) `Update` (b) `Insert`

Figure 7: **Effect of batch size on KAML bandwidth** Increasing the number of key-value pairs to update or insert by a `Put` command improves throughput of `Update` by 1.2× - 1.3×, and reduces the amount of time by 40% to populate an empty namespace to 70% full (mapping table load factor reaches 0.7).

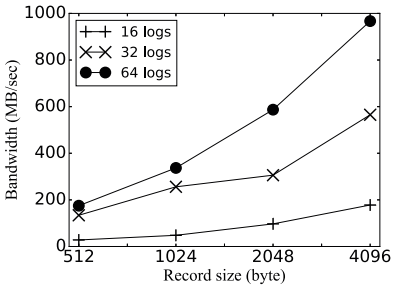


Figure 8: **Effect of multi-logs** Increasing the number of logs from 16 to 64 leads to 5.8 × increase in throughput.

C. Effect of number of logs

KAML-SSD allows for the configurable allocation of internal write bandwidth to user applications in terms of the number of logs associated with each namespace to append. Figure 8 shows the effect of the number of logs on bandwidth of `Put` in the `Update` benchmark. When the number of logs increases from 16 to 64, the maximum bandwidth that the namespace can achieve increases by 5.8× since more logs can support more concurrent commands.

D. OLTP

To quantify the benefits of KAML for OLTP workloads, we implemented both TPC-B [17] and a subset of TPC-C [18] using the API that the KAML caching layer provides. The caching layer effectively serves as a database storage engine in these implementations. Shore-kits [19], an open-source benchmark suite using Shore-MT, provides a reference implementation for TPC-B and TPC-C. For the sake of fair comparison, our implementation of TPC-B and TPC-C uses the same lock manager as Shore-MT, and leverages Shore-kits' code to serialize/deserialize database records. Due to current hardware limitation and to comply with the TPC specifications, all values are 512 bytes except TPCC CUSTOMER table, whose values are 1024 bytes. Shore-MT's default page size is 8 KB. The scaling factor for both benchmarks is 100, yielding

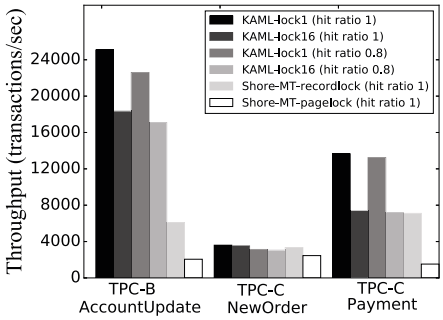


Figure 9: **Throughput of OLTP workloads** KAML outperforms Shore-MT with record-level lock by 4.0× for TPC-B, 1.1× for TPC-C `NewOrder` and 2.0× for `Payment`. Coarse-grained locks negatively impact throughput.

10 million values for TPCB and 60 million values for TPCC so that the mapping tables of each benchmark can fit in the in-SSD DRAM. We configure the amount of memory allocated to KAML caching layer to compare the performance when hit ratios are 0.8 and 1.0 respectively. Shore-MT allocates sufficient memory to the buffer pool so that the entire working set can fit in host main memory. This is ideal for conventional SQL databases and storage engines. To guarantee atomicity and durability, both KAML and Shore-MT must flush data to the SSD when transactions commit. For KAML, we vary the number of records protected by a lock to be 1 and 16. For Shore-MT, we run the benchmarks with both record-level and page-level locks.

Figure 9 shows the throughput of running TPC-B's `AccountUpdate` transaction, TPC-C's `NewOrder` transaction and `Payment` transaction. These three transactions are the most important (i.e., frequently executed) queries in the benchmarks. The results show two things: First, KAML outperforms Shore-MT with record-level lock by up to 4.0× for TPC-B's `AccountUpdate`, 1.1× for TPC-C's `NewOrder`, and 2.0× for TPC-C's `Payment`. Second, coarse-grained locks negatively impact transaction throughput. For example, KAML's throughput drops by up to 47% when the number of records protected by a lock increases from 1 to 16. Likewise, Shore-MT's performance drops by up to 80% when it switches from record-level locks to page-level locks.

1) Performance advantage over Shore-MT

Several factors account for KAML performance advantage over Shore-MT with record-level locks. First, centralized, synchronous logging is the major bottleneck in most conventional storage engines [20] with ARIES-style logging, and only a single transaction can acquire the global lock and flush the log at the same time. This transaction will block other transactions even if there is no data conflict among these transactions, and this transaction cannot commit before

data becomes persistent in the SSDs via an `fsync`. Consequently, the system under-utilizes the bandwidth of modern SSDs. In contrast, the KAML's `Put` command allows multiple transactions to commit in parallel if they do not contend for the same records, making full use of SSD's I/O bandwidth.

Second, conventional storage engines have to perform checkpointing and copy dirty data out of the log to limit log size. Although this happens in the background, it can interfere with foreground activity. The resulting degradation in performance is in addition to the performance impact of SSD garbage collection. In KAML, the log cleaning happens in the SSD, so the application only suffers the effect of one layer of garbage collection rather than two.

Finally, KAML avoids the extra layers of indirection that the file system adds and that Shore-MT (and other conventional storage engines) must pay the price for. In this regard, KAML's transactions are lighter-weight and allow for faster commit.

2) Impact of locking granularity

Support for fine-grained locking is one of the key advantages of KAML over existing key-value SSDs. Intuitively, the cost of coarse-grained locking results from the contention between multiple transactions for exclusive access to data – especially "hot" data. We analyze the performance impact of locking granularity.

Assume there are K keys divided into pages that hold l keys each. A lock protects each page. If N updates that each target key i with probability p_i arrive at about the same time, we can reduce the problem to the classic "balls into bins" problem [21] which gives the expected number of conflicts (i.e., the number of requests that will contend for a page lock) as:

$$\mathbb{E}[conflicting\ requests] = N - \frac{K}{l} + \sum_{i=1}^{\frac{K}{l}} (1-p_i)^N$$

If the choice of keys satisfy uniform distribution, i.e. $p_i = \frac{1}{K}$ for $0 \leq i \leq K-1$, the formula becomes

$$\mathbb{E}[conflicting\ requests] = N - \frac{K}{l}[1 - (1 - \frac{1}{K})^N]$$

So, as l increases, the number conflicts increases as well. As a result, KAML and many databases adopt fine-grained record-level locking.

E. NoSQL key-value store

The KAML caching layer can serve as a NoSQL key-value store. This section compares the performance of YCSB benchmark [22] on KAML and Shore-MT. In both cases, we populate the key-value store with 20 million 1024-bytes records and allocate 8 GB main memory to the buffer pool. We choose not to cache the entire data set in memory since we want to test the performance of `Get`.

Table III summarizes YCSB workloads, and Figure 10 shows the throughput of those workloads running on KAML and Shore-MT. KAML outperforms Shore-MT by between 1.1× and 3.0×. KAML achieves more performance improvement over Shore-MT for write-intensive workloads than for read-intensive ones.

workload	read	update	insert	read-modify-write
a	0.5	0.5	0	0
b	0.95	0.05	0	0
c	1	0	0	0
d	0.95	0	0.05	0
f	0.5	0	0	0.5

Table III: **YCSB workloads summary** The ratio of different operations in each workload.

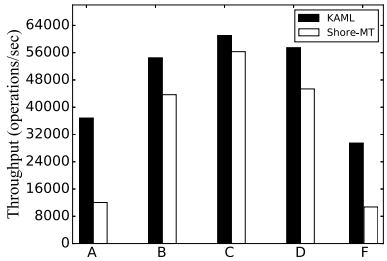


Figure 10: **YCSB throughput** KAML achieves up to 3.0× throughput gain relative to Shore-MT, while the average improvement is 2.3×.

VI. Related work

Many systems have explored SSD architecture and key-value store optimization. Below we describe these efforts and place KAML in context with them.

A. Innovations in SSD architecture

The architecture of SSDs is an active field of study, and researchers have addressed several opportunities and challenges that SSDs present [23], [24], [25], [26], [27], [10], [28], [29]. These include devising new interfaces to take advantage of flash memory's characteristics and refining the design of the FTL. Below we discuss key developments in each of these areas and how KAML differs from previous work.

Novel interfaces The flash memory that SSDs use to store data suggests a wide range of alternatives to the conventional block-based interface for storage.

The multi-streamed SSD [23] allows host applications to categorize accesses into "streams" that the SSD can manage differently depending on their characteristics. KAML can manage different namespaces according to different policies, providing a wider range of policy options than multi-stream SSDs, and it makes those parameters explicit in the SSD's interface. Nameless Writes [24] remove the indirection from LBA to PPN by allowing the file system to access flash memory via physical address and exposing the data movement that occurs during GC to the host. KAML goes a step further, mapping user-specified keys (not just LBAs) to PPNs, so that applications can more easily exploit the mapping inside the SSD.

SDF [30] targets datacenter workloads. SDF's minimum write size is a flash block, and it exposes individual channels to applications which must explicitly

manage erasure and GC. LOCS [25] is an LSM-tree-based key-value store [31] exploiting the internal bandwidth of SDF. Data analytics can also benefit from new SSD interfaces that provide in-storage data filtering [26]. Besides exposing new interfaces, people have proposed adding programmability to SSDs, making it more convenient to change the interaction between the host and the SSD [32].

Seagate Kinetic Open Storage Platform [33] supports access to remote storage via key-value interface over Ethernet. Seagate currently offers only hard drives with the interface but SSDs are an obvious extension. While Kinetic targets data transfer to/from remote disks, current KAML prototype focuses on local SSDs with lower latency and higher throughput. Moreover, KAML provides a key-value interface with transactional semantics supporting fine-grained isolation.

Transactions on SSDs Several projects add transaction support to SSDs by leveraging the copy-on-write facilities that FTLs already employ. TxFlash [7] enables applications to write multiple pages atomically using a novel commit protocol. Atomic-write [8] extends a log-based FTL and uses one bit per block to track if the block is part of an atomic-write. Atomic-write can eliminate double-write that induces considerable overhead in InnoDB [2]. X-FTL [9] and Möbius [10] atomically update multiple pages. However, they perform operations on entire pages, and, therefore require page-level lock. MARS [11] supports fine-grained atomic-write, but assumes the SSD uses byte-addressable non-volatile memory which is not commercially available yet. In contrast, KAML uses flash which will remain the dominant solid state storage technology in the foreseeable future.

FTL Design There have been a number of efforts focusing on the design of efficient FTL [34], [35], [36], [37], [38], [39], [28], [40]. DFTL [37] selectively caches page-level address mappings to improve the performance for random writes. CAFTL [28] uses the hash value of the data to detect duplicates to improve the lifetime and performance of SSD. CA-SSD [41] exploits the value locality of data access pattern to reduce SSD response time with small additional hardware support. Compared with these efforts, KAML generalizes its FTL to a set of mapping tables from user-specified keys to physical addresses. Instead of a single array, KAML supports variable number of mapping tables.

Virtual Storage Layer (VSL) [42] is a software FTL between the SSD and applications tailored specifically to FusionIO's SSDs. It supports sparse addressing, dynamic mapping and transactional persistence. VSL keeps FTL data structures in the host memory. KAML adopts an alternative approach that offloads the FTL to the SSD. Therefore KAML consumes less host resources but requires careful budgeting of SSD's internal resources. Manufacturers have put more powerful processors and larger RAMs into SSDs and this trend will likely continue in the foreseeable future.

B. Key-Value stores in SSDs

Researchers have built key-value stores optimized for SSDs. FlashStore [43] and SkimpyStash [44] store key-value pairs on an SSD and apply various optimizations to improve performance and memory consumption, but they rely on conventional SSDs resulting in stacked logs and the inefficiencies that those entail. Similarly, using Bw-Tree [45] as a key-value database leads to similar issues. In contrast, KAML implements a key-value interface in the SSD, eliminating stacked logs and reducing complexity while providing fast access via its caching layer.

SILT [46] is a high-performance, memory-efficient storage system on a single node. It has three key-value stores with different optimization goals for memory-efficiency and write performance. KAML combines application index with original FTL by mapping keys to physical addresses directly. The new mapping tables reside in battery-backed DRAM inside the SSD, thus reducing the amount of host memory usage.

NVMKV [27] builds upon VSL [42] to implement key-value stores. Its mapping table resides in host memory and keys in different stores are evenly distributed across the entire LBA space. Furthermore, NVMKV relies on LevelDB's [47] read cache. In contrast, KAML provides separate namespaces for different key-value stores and a generic caching layer.

VII. Conclusion

Existing proposals to modernize the SSD interface by providing key-value semantics present a single inflexible key space, while proposals for transactional SSDs require inefficient coarse-grained locking. KAML solves these problems by supporting multiple, independent namespaces for key-value pairs and allow applications to tune the performance of each namespace to meet application requirements. It also enables fine-grained locking by treating key-value pairs as the basic unit of transactional operations. Finally, KAML provides a caching layer analogous to a conventional page cache to improve performance. These changes allow KAML to outperform conventional designs in a wide range of workloads.

Acknowledgment

We would like to thank the reviewers for their helpful suggestions. This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA. It was also sponsored in part by Samsung and NSF award 1219125.

REFERENCES

[1] Shore-MT, http://research.cs.wisc.edu/shore-mt/.

[2] InnoDB, http://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html.

[3] Oracle Database, https://www.oracle.com.

[4] L. Grupp, A. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. Siegel, and J. Wolf, "Characterizing Flash Memory: Anomalies, Observations, And Applications," in *MICRO-42*, 2009, pp. 24 –33.

[5] NVM Express, http://www.nvmexpress.org/.

[6] J. Gray *et al.*, "The transaction concept: Virtues and limitations," in *VLDB*, 1981, pp. 144–154.

[7] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional Flash," in *OSDI*, 2008, pp. 147–160.

[8] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. Panda, "Beyond Block I/O: Rethinking Traditional Storage Primitives," in *HPCA*, 2011, pp. 301–311.

[9] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min, "X-FTL: Transactional FTL for SQLite Databases," in *SIGMOD*, 2013, pp. 97–108.

[10] W. Shi, D. Wang, Z. Wang, and D. Ju, "Mobius: A High Performance Transactional SSD with Rich Primitives," in *MSST*, 2014, pp. 1–11.

[11] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives," in *SOSP*, 2013, pp. 197–212.

[12] G. Graefe, "Write-optimized B-trees," in *VLDB*, 2004, pp. 672–683.

[13] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman, "Don't Stack Your Log on My Log," in *INFLOW*, 2014.

[14] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control And Recovery in Database Systems*. Addison-wesley New York, 1987, vol. 370.

[15] 3D Xpoint, https://www.micron.com/about/emerging-technologies/3d-xpoint-technology.

[16] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging," *TODS*, vol. 17, pp. 94–162, 1992.

[17] TPC-B, http://www.tpc.org/tpcb/.

[18] TPC-C, http://www.tpc.org/tpcc/.

[19] Shore-kits, https://bitbucket.org/shoremt/shore-kits/src.

[20] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki, "Aether: A Scalable Approach to Logging," *VLDB*, vol. 3, no. 1-2, pp. 681–692, 2010.

[21] M. Raab and A. Steger, "Balls into Bins – A Simple and Tight Analysis," in *Randomization and Approximation Techniques in Computer Science*. Springer, 1998.

[22] YCSB, https://github.com/brianfrankcooper/YCSB.

[23] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The Multi-Streamed Solid-State Drive," in *HotStorage*, 2014.

[24] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "De-indirection for Flash-based SSDs with Nameless Writes," in *FAST*, 2012, p. 1.

[25] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD," in *EuroSys*, 2014, pp. 16:1–16:14.

[26] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query Processing on Smart SSDs: Opportunities and Challenges," in *SIGMOD*, 2013, pp. 1221–1230.

[27] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "NVMKV: A Scalable, Lightweight, FTL-aware Key-value Store," in *USENIX ATC*, 2015, pp. 207–219.

[28] F. Chen, T. Luo, and X. Zhang, "CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives," in *FAST*, 2011.

[29] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson, "Hippogriffdb: Balancing I/O And GPU Bandwidth in Big Data Analytics," *VLDB*, vol. 9, pp. 1647–1658, 2016.

[30] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "SDF: Software-defined Flash for Web-scale Internet Storage Systems," in *ACM SIGPLAN Notices*, vol. 49, 2014, pp. 471–484.

[31] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The Log-Structured Merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[32] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A User-programmable SSD," in *OSDI*, 2014, pp. 67–80.

[33] Seagate Kinetic, http://www.seagate.com/solutions/cloud/data-center-cloud/platforms/.

[34] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *CSUR*, vol. 37, pp. 138–163, 2005.

[35] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "System Software for Flash Memory: A Survey," in *EUC*, 2006, vol. 4096, pp. 394–404.

[36] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: Locality-Aware Sector Translation for NAND Flash Memory-based Storage Systems," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 6, pp. 36–42, 2008.

[37] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," in *ASPLOS XIV*, 2009, pp. 229–240.

[38] Y. Lu, J. Shu, and W. Zheng, "Extending the Lifetime of Flash-based Storage Through Reducing Write Amplification from File Systems," in *FAST*, 2013, pp. 257–270.

[39] T. Wang, D. Liu, Y. Wang, and Z. Shao, "FTL2: A Hybrid Flash Translation Layer with Logging for Write Reduction in Flash Memory," in *LCTES*, 2013, pp. 91–100.

[40] Y. Kang, R. Pitchumani, T. Marlette, and E. L. Miller, "Muninn: A Versioning Flash Key-Value Store Using an Object-based Storage Model," in *SYSTOR*, 2014.

[41] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging Value Locality in Optimizing NAND Flash-based SSDs," in *FAST*, 2011, pp. 91–103.

[42] VSL, http://www.fusionio.com/products/vsl.

[43] B. Debnath, S. Sengupta, and J. Li, "FlashStore: High Throughput Persistent Key-value Store," *VLDB*, vol. 3, no. 1-2, pp. 1414–1425, 2010.

[44] ——, "SkimpyStash: RAM Space Skimpy Key-value Store on Flash-based Storage," in *SIGMOD*, 2011, pp. 25–36.

[45] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The Bw-Tree: A B-Tree for New Hardware Platforms," in *ICDE*, 2013, pp. 302–313.

[46] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A Memory-efficient, High-performance Key-value Store," in *SOSP*, 2011, pp. 1–13.

[47] LevelDB, https://github.com/google/leveldb.