An Efficient Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD

Peng Wang Guangyu Sun
Peking University
{wang_peng, gsun}@pku.edu.cn

Song Jiang *
Peking University and
Wayne State University
sjiang@eng.wayne.edu

Jian Ouyang Shiding Lin
Baidu Inc.
{ouyangjian, linshiding}@baidu.com

Chen Zhang
Peking University
chen.ceca@pku.edu.cn

Jason Cong * †
Peking University and
University of California, Los Angeles
cong@cs.ucla.edu

Abstract

Various key-value (KV) stores are widely employed for data management to support Internet services as they offer higher efficiency, scalability, and availability than relational database systems. The log-structured merge tree (LSM-tree) based KV stores have attracted growing attention because they can eliminate random writes and maintain acceptable read performance. Recently, as the price per unit capacity of NAND flash decreases, solid state disks (SSDs) have been extensively adopted in enterprise-scale data centers to provide high I/O bandwidth and low access latency. However, it is inefficient to naively combine LSM-tree-based KV stores with SSDs, as the high parallelism enabled within the SSD cannot be fully exploited. Current LSM-tree-based KV stores are designed without assuming SSD's multi-channel architecture.

To address this inadequacy, we propose LOCS, a system equipped with a customized SSD design, which exposes its internal flash channels to applications, to work with the LSM-tree-based KV store, specifically LevelDB in this work. We extend LevelDB to explicitly leverage the multiple channels of an SSD to exploit its abundant parallelism. In addition, we optimize scheduling and dispatching polices for concurrent I/O requests to further improve the efficiency of data access. Compared with the scenario where a stock LevelDB runs on a conventional SSD, the throughput of storage system can be improved by more than 4× after applying all proposed optimization techniques.

Categories and Subject Descriptors H.3.4 [*Information Storage And Retrieval*]: Systems and Software

Keywords Solid state disk, flash, key-value store, log-structured merge tree

1. Introduction

With the rapid development of Web 2.0 applications and cloud computing, large-scale distributed storage systems are widely deployed to support Internet-wide services. To store the ultra-large-scale data and service high-concurrent access, the use of traditional relational database management systems (RDBMS) as data storage may not be an efficient choice [15]. A number of features and functionalities of RDBMS, such as transaction consistency guarantee and support of complicated SQL queries, are not necessary for many web applications. Therefore, a new storage architecture, key-value (KV) store, has emerged in the era of big data.

A key-value store maps a set of keys to the associated values and can be considered as a distributed hash table. Without having to provide features and functionalities usually required for a database system, the KV stores can offer higher performance, better scalability, and more availability than traditional RDBMS [37][29]. They have played a critical role in data centers to support many Internet-wide services, including BigTable [18] at Google, Cassandra [28] at Facebook, Dynamo [22] at Amazon, and Redis [4] at Github.

The B+ tree is a common structure used in the traditional databases and some KV stores (e.g. CouchDB [1] and Toky-

* This work was done during their visiting professorship at the Center for Energy-Efficient Computing and Applications in Peking University.

† Jason Cong is also a co-director of the PKU/UCLA Joint Research Institute in Science and Engineering.

EuroSys 2014, April 13–16 2014, Amsterdam, Netherlands.

http://dx.doi.org/10.1145/2592798.2592804

oCabinet [8]), because its high fanout helps to reduce the number of I/O operations for a query operation. However, it is highly inefficient for using the data structure to support random insertion and updates. When there are intensive mutations to a data store, using the B+ tree would lead to a large number of costly disk seeks and significantly degraded performance. The log-structured merge tree (LSM-tree) [32] is a data structure optimized for writes, including insertions, modifications, and deletions. The basic idea is to transform random writes to sequential writes by aggregating multiple updates in memory and dumping them to storage as a batch. Incoming data items are stored and sorted according to their keys in a buffer reserved in the main memory. When the buffer is full, all of its data will be written into storage as a whole. There are many popular KV stores adopting the LSM-tree-based data management, including BigTable [18], Cassandra [28], Hbase [9], and LevelDB [10].

When KV stores were initially proposed, hard disk drives (HDDs) were considered as its main target storage devices. Recently, with the development of NAND flash technology, the price per unit capacity of the flash-based solid state disks (SSD) keeps decreasing. Thus, SSDs have become increasingly popular in today's data centers. Compared to HDDs, SSDs provide much higher throughput and lower latency, especially for random operations. However, SSDs also have their own limitations: performance of random write substantially lags behind that of sequential write operations and read operations, mainly due to its incurred expensive garbage collections [31]. As mentioned before, since the LSM-tree-based KV stores can effectively eliminate random writes, it proves promising to integrate LSM-tree-based KV stores with NAND flash-based SSDs so that high throughput can be achieved for both read and write I/O operations.

Although researchers have been aware of the potential advantage of combining LSM-tree-based KV stores with SSDs [11], to the best of our knowledge, this topic has not been well studied in prior works. In fact, a simple integration of these two techniques is not efficient. On the one hand, the process of data movement in LSM-tree-based KV stores is originally designed for HDDs rather than SSDs. Since an HDD has only one disk head, the KV store serially issues I/O requests to the HDD. We need to increase the concurrency level in order to take advantage of SSD's performance advantage on random read. On the other hand, the rich internal parallelism of SSDs has not yet been fully exploited. Although the modern SSD usually consists of multiple channels, the SSD controller hardware provides only one block device interface to the operating system, and the scheduling and dispatching of I/O requests between internal channels are hidden from the software level. This makes the LSM-tree-based KV stores unaware of the SSD's multi-channel architecture, and the scheduling and dispatching decisions made by the SSD controller are not optimized according to the data access patterns from LSM-tree-based KV stores.

In order to address these issues, we propose to employ a customized SSD, called SDF [33] that has become available only recently, to work with a popular LSM-based KV store, LevelDB in this work [10]. The SDF was originally designed and adopted in data centers of Baidu, which is the largest Internet search company in China. It provides a unique feature wherein the access to internal flash channels is open, and can be managed by applications to fully utilize SSD's high bandwidth. In order to leverage this unique feature, we modify LevelDB to apply a number of optimizations. At the same time, we observe that the scheduling and dispatching policies have a critical impact on the I/O performance in this novel system. Thus, we study how to improve throughput with optimized scheduling and dispatching policies, considering the characteristics of access patterns with LevelDB. Our system is called LOCS, which is the abbreviation for "LSM-tree-based KV store on Open-Channel SSD".

The contributions of this work can be summarized as follows:

- This is the first work to integrate an LSM-tree-based KV store with an open-channel SSD whose internal channels can be directly accessed in the applications.
- We extend the LevelDB to support multi-threaded I/O accesses to leverage the abundant parallelism in SDF. In addition, we optimize the write request traffic control mechanism of LevelDB to take advantage of in-device parallelism for a much-improved throughput.
- We study the impact of I/O requests scheduling and dispatching policies and present corresponding optimization techniques to further improve the I/O performance.
- We provide extensive experimental results, which demonstrate that LOCS can outperform the naive integration of original LevelDB with a conventional SSD design with similar hardware organization. The results also show that the I/O performance can be further improved with our optimization techniques for request scheduling and dispatching.

The remainder of this paper is organized as follows. Section 2 provides a review of LevelDB and a brief description of the open-channel SSD design (SDF). In Section 3, we describe how to extend LevelDB so that it can work efficiently with SDF. In addition, we propose several optimization techniques for scheduling and dispatching polices to improve the I/O performance. We also demonstrate the flexibilities provided by LOCS. Section 4 provides extensive experiments and comparisons. We describe related literature in Section 5, followed by conclusions in Section 6.

2. Background

In order to present the details of our design, we first provide some background on the implementation details of LevelDB. Then we provide a review of the design of the customized open-channel SSD.

2.1 LevelDB

LevelDB is an open source key-value store that originated from Google's BigTable [18]. It is an implementation of LSM-tree, and it has received increased attention in both industry and academia [6][34][2]. Figure 1 illustrates the architecture of LevelDB, which consists of two *MemTables* in main memory and a set of *SSTables* [18] in the disk and other auxiliary files, such as the *Manifest* file which stores the metadata of SSTables.

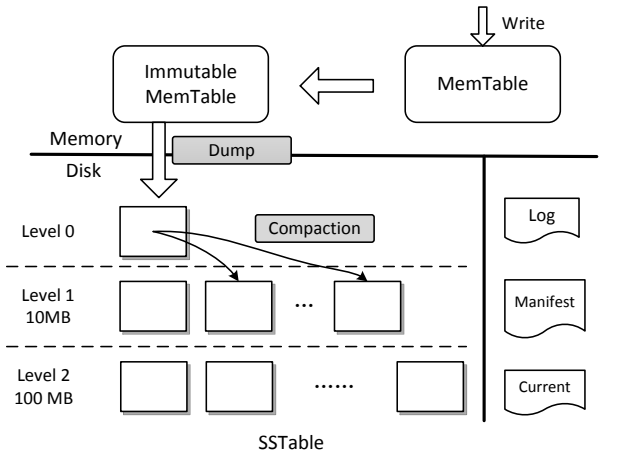


Figure 1. Illustration of the LevelDB architecture.

When the user inserts a key-value pair into LevelDB, it will be first saved in a log file. Then it is inserted into a sorted structure in memory, called *MemTable*, which holds the most recent updates. When the size of incoming data items reaches its full capacity, the MemTable will be transformed into a read-only *Immutable MemTable*. A new MemTable will be created to accumulate fresh updates. At the same time, a background thread begins to dump the Immutable MemTable into the disk and generate a new Sorted String Table file (*SSTable*). Deletes are a special case of update wherein a deletion marker is stored.

An SSTable stores a sequence of data items sorted by their keys. The set of SSTables are organized into a series of levels, as shown in Figure 1. The youngest level, *Level 0*, is produced by writing the Immutable MemTable from main memory to the disk. Thus SSTables in Level 0 could contain overlapping keys. However, in other levels the key range of SSTables are non-overlapping. Each level has a limit on the maximum number of SSTables, or equivalently, on the total amount of data because each SSTable has a fixed size in a level. The limit grows at an exponential rate with the level number. For example, the maximum amount of data in Level 1 will not exceed 10 MB, and it will not exceed 100 MB for Level 2.

In order to keep the stored data in an optimized layout, a *compaction* process will be conducted. The background compaction thread will monitor the SSTable files. When the total size of Level `L` exceeds its limit, the compaction thread will pick one SSTable from Level `L` and all overlapping ones from the next Level `L+1`. These files are used as inputs to the compaction and are merged together to produce a series of new Level `L+1` files. When the output file has reached the predefined size (2 MB by default), another new SSTable is created. All inputs will be discarded after the compaction. Note that the compaction from Level 0 to Level 1 is treated differently than those between other levels. When the number of SSTables in Level 0 exceeds an upper limit (4 by default), the compaction is triggered. The compaction may involve more than one Level 0 file in case some of them overlap with each other.

By conducting compaction, LevelDB eliminates overwritten values and drops deleted markers. The compaction operation also ensures that the freshest data reside in the lowest level. The stale data will gradually move to the higher levels.

The data retrieving, or read operation, is more complicated than the insertion. When LevelDB receives a `Get(Key, Value)` request, it will first do a look up in the MemTable, then in Immutable MemTable, and finally search the SSTables from Level 0 to higher levels in the order until a matched KV data item is found. Once LevelDB finds the key in a certain level, it will stop its search. As we mentioned before, lower levels contain fresher data items. The new data will be searched earlier than old data. Similar to compaction, more than one Level 0 file could be searched because of their data overlapping. A Bloom filter [14] is usually adopted to reduce the I/O cost for reading data blocks that do not contain requested KV items.

2.2 Open-Channel SSD

The open-channel SSD we used in this work, SDF, is a customized SSD widely deployed in Baidu's storage infrastructure to support various Internet-scale services [33]. Currently more than 3 000 SDFs have been deployed in the production systems. In SDF, the hardware exposes its internal channels to the applications through a customized controller. Additionally, it enforces large-granularity access and provides lightweight primitive functions through a simplified I/O stack.

The SDF device contains 44 independent channels. Each flash channel has a dedicated channel engine to provide FTL functionalities, including block-level address mapping, dynamic wear leveling, bad block management, as well as the logic for the flash data path. From an abstract view of software layer, the SDF exhibits the following features.

First, SDF exposes the internal parallelism of SSD to user applications. As mentioned previously, each channel of an SDF has its exclusive data control engine. In contrast to the conventional SSD, where the entire device is considered as a single block device (e.g., `/dev/sda`), SDF presents each channel as an independent device to the applications (e.g., from `/dev/ssd0` to `/dev/ssd43`). With the capability of directly accessing individual flash channels on SDF, the user

application can effectively organize its data and schedule its data access to fully utilize the raw flash performance.

Second, SDF provides an asymmetric I/O interface. The read unit size is different from write unit size. SDF aggressively increases the write unit size to the flash erase block size (2 MB) and requires write addresses to be block-aligned. Therefore, write amplification is almost eliminated since no flash block will contain both valid and invalid pages at the time of garbage collection. The minimal read unit is set to the size of a flash page (8 KB), which keeps SSD's inherent capability of random read. In other words, SDF discards its support for small random writes while keeping the ability of random read, which is well matched to the access pattern of LSM-tree-based KV store.

Third, the erase operation is exposed to the software as a new command to the device. Erase is an expensive operation compared to read or write. For example, it takes about 3 *ms* to erase a 2 MB block. When an erase operation is in process in a channel, it can significantly delay the service of regular requests issued to the channel. Erase operations scheduled by the conventional SSD controller are hidden from the applications. They can cause unpredictable service quality fluctuation, which is especially harmful for performance-critical workloads. With this erase command, the software is responsible for conducting erase operations before a block can be overwritten. But it also gives software the capability to schedule erase operations to minimize delay, improve throughput, and alleviate collision with the service of high-priority requests.

Fourth, a simplified I/O stack is designed specifically for SDF. Linux builds a complicated I/O stack which is mainly designed for a traditional low-speed disk. The layer of I/O stack, such as the block layer, has become a bottleneck for today's high-performance SSD [16][17]. Experiments show that the additional delay introduced by the software layer could be as high as 12 μs on our servers. This overhead is substantial with high-speed flash data accesses. Since the SDF is customized for the LSM-tree-based KV store, most functionalities of the file system become unnecessary. For the sake of efficiency, we bypass most of the I/O layers in the kernel and use the `ioctl` interface to directly communicate with the SDF driver. The latency of SDF's I/O stack is only about 2 μs to 4 μs. SDF provides a user-space storage API library for applications to exploit the features stated here.

3. Design and Implementation

In this section we first introduce the architectural overview of our LOCS system. Then we describe how to extend the original LevelDB design to facilitate concurrent accesses to the multiple channels in the SDF. Following that, we analyze the impact of request scheduling in LevelDB and dispatching to the channels of SDF, and propose policies to improve access efficiency.

3.1 Overall Architecture

As shown in Figure 2, the LOCS system is illustrated with both software and hardware levels. The software level is composed of four main layers: the LevelDB, the proposed scheduler, the layer of storage API, and the SSD driver. As mentioned in the previous section, LevelDB is a popular LSM-tree-based KV store and is used in this work as a case study. In order to enable LevelDB to work with the open-channel SDF and improve its efficiency for high concurrent accesses to SSD, we introduce several necessary modifications to LevelDB. The details are described in Section 3.2. Note that the techniques proposed in this work can also be applied to other LSM-tree-based KV stores.

Different from the traditional system design, a scheduler is added between LevelDB and the layer of storage API. The scheduler is dedicated to scheduling and dispatching requests from LevelDB to various channels of SDF. Note that the scheduler is not like the OS scheduler in the traditional I/O stack. As introduced in Section 2.2, most functions in the traditional I/O stack have been removed when the SDF is employed. In addition, the OS scheduler is responsible for I/O requests from all processes. On the contrary, the scheduler in Figure 2 only takes the requests from LevelDB into consideration. Moreover, the erase operations for garbage collection are also managed by the scheduler. The detailed designs and request management policies will be described in Section 3.3.

After the scheduling and dispatching, requests from LevelDB will call the corresponding API for different operations, including read, write, and erase. Then, the SSD driver is invoked to send these requests to the SSD controller in the hardware device. Since the scheduling and dispatching have been taken over by the scheduler in the software level, the corresponding unit in the traditional hardware SSD controller is reduced. The SSD controller is only responsible for sending the requests to the corresponding channels according to the instructions of the software scheduler. In addition, since the erase operation is also explicitly issued from our scheduler in the software level, the garbage collection function is not needed either. The other basic functions, such as wear-leveling, ECC, and DRAM cache control, are still kept in the SSD controller.

3.2 Extension of LevelDB

In this subsection we discuss how to extend the current LevelDB to make it efficiently access the multiple channels exposed by SDF.

3.2.1 Enabling Concurrent Access to Multiple Channels

Although LevelDB supports multiple concurrent user queries, there is only one background thread that dumps MemTables to the storage device and handles the compaction process. Such a design is sound for traditional systems with HDD

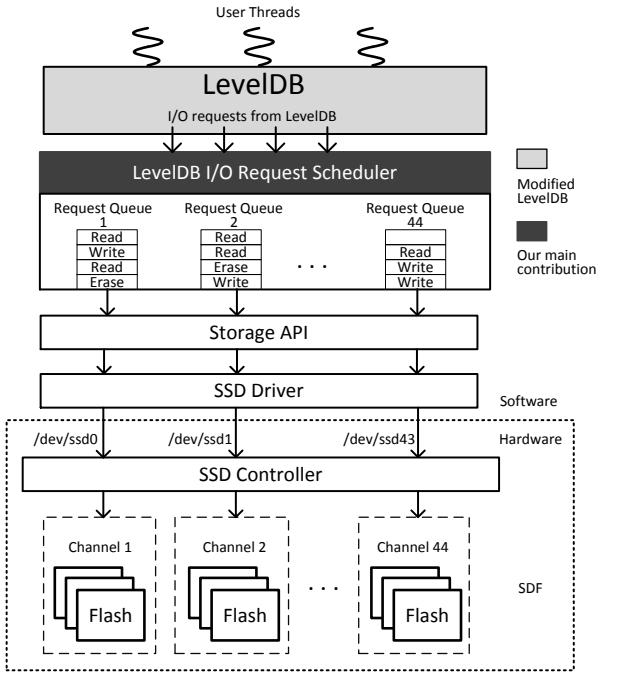


Figure 2. The overall architecture of LOCS.

because there is only one access port in HDD. Due to the latency of moving the disk head, using multiple threads for SSTable writing and compaction can introduce interference among requests from different threads and degrade I/O performance. However, for storage systems based on SSD, the seek time can be removed. For the SDF used in this work, since the internal flash channels have been exposed to the software level, it is necessary to extend the LevelDB design to enable concurrent accesses to these channels.

First, we increase the number of Immutable MemTables in memory to make full use of SDF's 44 channels. As introduced in Section 2, in the stock LevelDB there are only two Memtables: one working MemTable and one Immutable MemTable. The entire Immutable MemTable is flushed to SSD in a single write request. If the working MemTable is full while the Immutable MemTable is still being flushed, LevelDB will wait until the dumping ends. Since one write request cannot fill up all channels, we increase the upper limit on the number of Immutable MemTables to store more incoming data. When the working MemTable has accrued enough updates and reached the size threshold (2 MB in this work), it will generate one Immutable MemTable, set it ready for write, and send it to the scheduler. With this modification, write requests produced from multiple Immutable MemTables can be issued concurrently. We will show the impact of the number of Immutable MemTables in Section 4.

When the scheduler receives a write request, it will insert the write request into a proper request queue according to the dispatching policy. As shown in Figure 2, there is one I/O request queue for each flash channel. It means that all requests in the queue are exclusively serviced by a single flash channel. With an appropriate dispatching policy, the access concurrency to multiple channels can be effectively exploited. For the read and erase requests, they will be inserted into corresponding queues depending on the locations of data to be accessed. For the write requests, they will be inserted into the suitable queues according to the dispatching policy. It should be addressed that multiple compaction processes can also be conducted in parallel if they are applied to SSTables distributed over different channels.

3.2.2 Write Traffic Control Policy

The second modification of the LevelDB is concerned with the write traffic control policy. The traffic control means that LevelDB has the mechanism of limiting the rate of write requests from users when the number of Level 0 SSTables reaches a threshold. The purpose of this mechanism is to limit the number of Level 0 SSTables, thereby reducing the search cost associated with multiple overlapping Level 0 SSTables. In other words, the write throughput is traded for the read performance.

There are several thresholds that control the number of Level 0 SSTables. When the threshold `kL0_Compaction Trigger` (4 by default) is reached, the compaction process will be triggered. If the number of Level-0 SSTables cannot be efficiently decreased in the compaction process and reaches the second threshold `kL0_SlowdownWritesTrigger` (set as 8 in the stock LevelDB), the LevelDB will enter the sleep mode for one millisecond to reduce the receiving rate of write requests. However, if the number of Level-0 SSTable keeps increasing and exceeds the third threshold `kL0_StopWritesTrigger` (12 by default), all write requests will be blocked until the background compaction completes.

The write traffic control policy affects the write throughput significantly. When the insertion of KV pair is blocked, the write throughput degrades substantially. Considering SDF's multiple channel architecture, we adjust the write traffic control policy to improve the throughput. First, we increase the value of these thresholds, which are originally optimized for HDDs. When there are multiple MemTables flushed to SSD simultaneously, the threshold should not be triggered too frequently. By increasing the threshold, we can enlarge the intervals between the pausing. Second, we introduce an additional background thread when the slowdown condition is triggered. In traditional HDD, all read and write I/O requests share the only disk head. Running multiple compactions concurrently will cause random I/O problem. However, the multiple channels in SDF make it feasible to trigger multiple compactions at the same time. When the user request is blocked, a single compaction thread cannot utilize all channels effectively, i.e., some channels will be idle. By creating an additional thread to do the compaction during the pausing, we can reduce the number of files in the Level 0 faster and mitigate the throughput loss brought by the pausing. It should be mentioned that, if too many

threads for compaction are introduced, they will interfere with the normal data access. Our experiments show an additional compaction thread is good enough to reduce throughput fluctuation. Additionally, we do not need to block write requests until all Level 0 SSTables are compacted. Instead, we modify LevelDB to make sure that it will accept user write requests again when the number of Level 0 SSTables is less than the half of `kL0_CompactionTrigger`.

3.2.3 Write Ahead Log

LevelDB maintains one single log file to recover the MemTables in main memory when accidents like power-down happen. In the original LevelDB design, the log is written to HDD using the memory-mapped file. The new updates are appended to the current log file as continuous I/O requests. This log strategy of using a single log file can seriously impact the write throughput when there are intensive concurrent write requests. In this work we assume that we have a small part of high-performance non-volatile memory (NVM), such as PCM or battery-backup DRAM, to hold these logs. Since the logs can be discarded whenever a MemTable is dumped to the SSD, the demand for storage space on the non-volatile memory is moderate. For the example of using SDF with 44 channels, we only need about 100 MB.[1]

3.3 Scheduling and Dispatching Policies

With the extension introduced in the last subsection, the LevelDB can now work with the SDF. In LOCS, we find that the scheduling and dispatching policies of write requests have an impact on the throughput and working efficiency of SSD. For example, if there are a large number of I/O requests in one channel while other channels are idle, the performance will be severely degraded. However, since we can directly control the accesses to multiple channels of SDF, it is possible to optimize scheduling and dispatching with the guiding information from LevelDB. Thus, in this subsection we study several different policies and discuss how to improve the performance of our system.

3.3.1 Round-Robin Dispatching

Our baseline dispatching policy is the simple round-robin (RR) dispatching, which uniformly distributes write requests to all channels in the SSD. The write requests from LevelDB are in the granularity of an SSTable, the size of which is 2 MB. We dispatch all write requests to each channel in circular order.

This is similar to the design in the traditional hardware-based SSD controller. For example, in the Huawei SSD, which is used in our experiments, each large write request from LevelDB is stripped over multiple channels to benefit from parallel accesses. In our system, with the multiple request queues, the case is a little different. A large write request from LevelDB is issued to a channel, and its data will not be striped. Thus, the requests are distributed to the channels in the granularity of 2MB.

The round-robin dispatching has the advantage of simplicity. Since the size of each write request is fixed, it works efficiently with the case where the write requests are dominant. However, when there are intensive read and erase requests in the queues, the round-robin dispatching becomes less efficient. This is because the dispatching of both types of requests is fixed. Especially in the case where there are several read/erase requests waiting in the same queue, it can result in unbalanced queue lengths.

Figure 3(a) gives an example of round-robin dispatching. Suppose that there are three channels in the SSD. The traces of 11 I/O requests are shown in Figure 3(c), including 6 write requests and 5 read requests. The third row shows the channel address for a read request. Note that the channel addresses of read requests are already determined and cannot be changed, but we have the flexibility to decide at which channel each write request should be serviced. When the round-robin dispatching is applied, all write requests are dispatched to the channels uniformly in circular order. However, the round-robin dispatching causes the request queue of Channel 2 to be much longer than the other two channels, since four continuous read requests all fall into Channel 2. Therefore, the request queues are unbalanced, which means Channel 2 will take a longer time to handle I/O requests than Channel 1 and Channel 3, and it is possible that Channel 1 and 3 will be idle while Channel 2 is still busy. In this situation, parallelism enabled by the multiple channels is not fully exploited. In fact, we have observed that such an unbalanced queue can severely impact the performance. The detailed experimental results are shown and discussed in Section 4.

3.3.2 Least Weighted-Queue-Length Write Dispatching

In order to mitigate the unbalanced queue problem of round-robin dispatching, we propose a dispatching policy based on the length of request queues. The basic idea is to maintain a table of weighted-queue-length to predict the latency of processing all requests in these queues. Since there are three types of requests (read, write, and erase) with various latencies, we should assign different weights to the three types of requests. Then, the weighted length of a queue can be represented as follows,

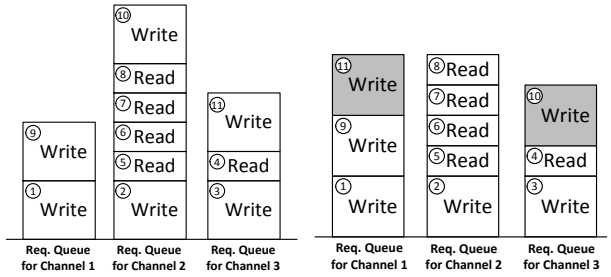
$$Length_{weight} = \sum_{1}^{N} W_i \times Size_i \qquad (1)$$

Note that N denotes the total number of requests in a queue, and W_i and $Size_i$ represent the weight and size of each request, respectively. The weights are determined according to the corresponding latency of each type of requests. The sizes of both write and erase requests are fixed, and only the size of a read request may vary. Then we select

[1] Currently the write ahead logs are stored in DRAM.

a channel with the least value of the weighted-queue-length, and insert the current write request from LevelDB to this channel. Unlike the round-robin dispatching policy, the least weighted-queue-length (denoted as LWQL) dispatching policy takes all three kinds of I/O requests into consideration. Figure 3(b) is an example of using LWQL dispatching.

In this example, the weighted-queue-lengths of read and write requests are 1 and 2, respectively. We notice that the dispatching of the 10th and the 11th requests is different from the RR policy, because the weighted queue length of Channel 2 is not the least when the 10th request arrives. Compared to the round-robin dispatching, it is apparent that the queues become more balanced in terms of weighted queue length. And the throughput of SDF can be improved compared to the RR dispatching.



(a) Round-Robin (b) Least Weighted-Queue-Length

Trace #	1	2	3	4	5	6	7	8	9	10	11
Op.	W	W	W	R	R	R	R	R	W	W	W
Channel	–	–	–	3	2	2	2	2	–	–	–

(c) Trace of the example

Figure 3. Illustration of RR and LWQL dispatching.

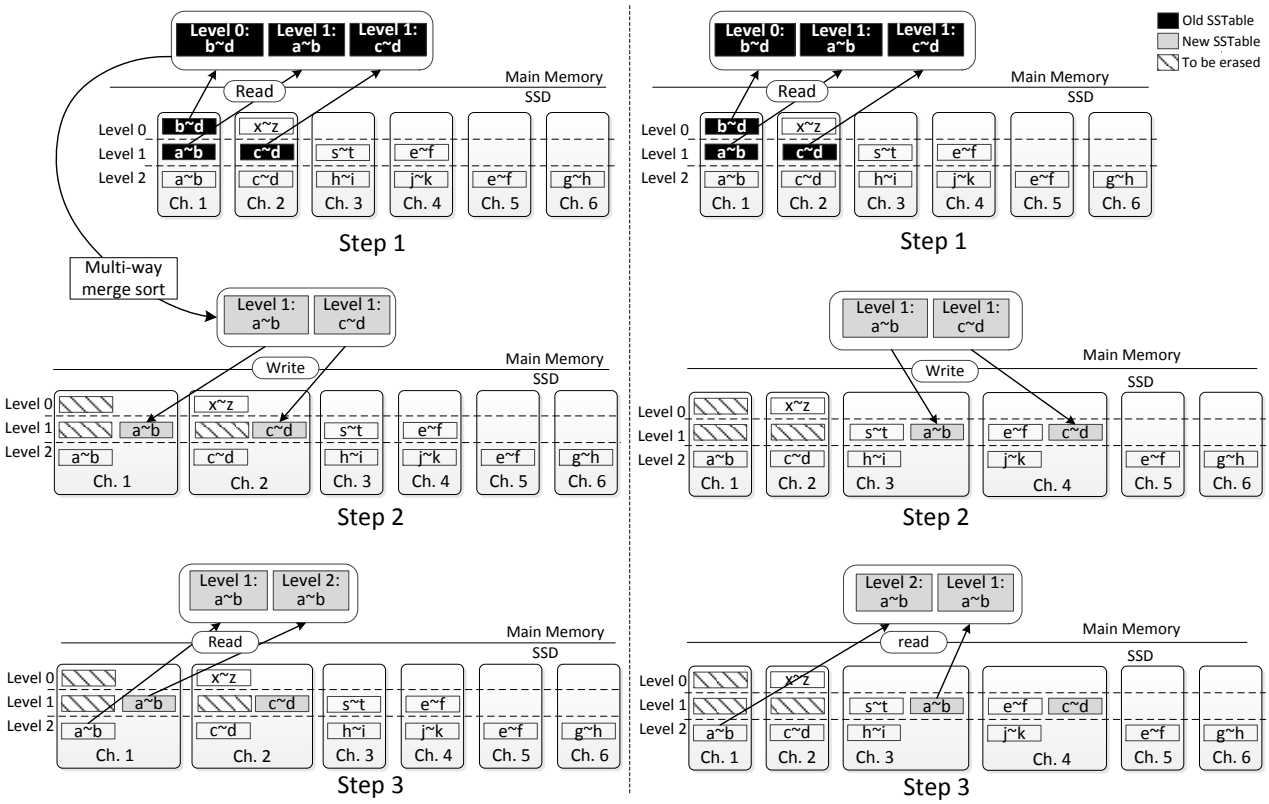
3.3.3 Dispatching Optimization for Compaction

There are two types of write requests generated by the LevelDB. Besides the write requests of dumping MemTables into Level 0, the other write requests are generated by the compaction. We notice that the data access pattern of the compaction is predictable. Therefore, we propose a dispatching optimization for compaction. As mentioned in Section 2.1, the compaction will merge SSTables within a certain key range. The process includes both read and write requests. Since the channel addresses of read requests are fixed, if some input SSTables for one compaction are allocated in the same channel, they have to be read in sequence, resulting an increase of the latency of compaction. Figure 4(a) shows an example of this case. First, when a compaction process is conducted, three SSTables, denoted as Level 0 "b∼d" (i.e., the SSTable whose key range is "b∼d" in Level 0), Level 1 "a∼b" and Level 1 "c∼d" will be read out from SDF to the main memory, as shown in Step 1. Then a multi-way merge sort is performed on these SSTables to generate two new SSTables, Level 1 "a∼b" and Level 1 "c∼d." After the merge operation, they will be written back to the SDF, as illustrated in Step 2. If we do not allocate these SSTables carefully, it is possible that the Level 1 "a∼b" will be assigned to the Channel 1, which contains a Level 2 "a∼b" SSTable. This means we have to read both of these SSTables from the same channel in the next compaction, as shown in Step 3. Thus the efficiency of compaction is compromised.

It is obvious that the allocation of new SSTables generated in current compaction will affect the efficiency of future compaction. Thus, write dispatching optimization for compaction is needed to improve the efficiency of compaction. The goal is to make sure that SSTables are carefully dispatched so that the SSTables with adjacent key ranges are not allocated in the same channel. Consequently, we propose a technique to optimize dispatching for SSTables generated from compaction. The dispatching policy is based on the LWQL policy and is described as follow.

- We record the channel location for each SSTable in the Manifest file. The Manifest file also contains the set of SSTables that make up each level, and the corresponding key ranges.
- For each SSTable generated from compaction, we first look for the queue with the shortest weighted-queue-length as the candidate. If there are any SSTables in the next level, of which the keys fall in the range of the four closest SSTables, then this candidate queue is skipped.
- We then find the queue with the shortest weighted-queue-length in the rest of the queues and check for the condition of key ranges, as in the previous step. The step is repeated until the condition is met.

Figure 4(b) shows an example of this policy. Step 1 is the same as 4(a). After Step 1, the compaction thread selects the candidate queue for the newly generated SSTables in Step 2. Supposing Channel 1 is selected for the Level 1 "a∼b" SSTable according to the LWQL policy, then the thread will search for four nearest neighbors of Level 2. Consequently, it will find Level 2 "a∼b" which means they will probably be merged in future compactions. Thus Channel 1 is skipped, and it will find another candidate queue with the least weighted-queue-length. Then Channel 2 is chosen, where no neighbouring SSTables at Level 2 reside. As mentioned before, the compaction operation will read all involved neighboring SSTables into memory. Thus avoiding SSTables with neighboring key ranges to stay in the same channel effectively improves read performance, as shown in the Step 3 of Figure 4(b). Similarly, the Level 1 "c∼d" SSTable in memory will evade Level 2 "a∼b," "c∼d," "e∼f," and "g∼h" SSTables. A statistical experiment shows that over 93% compaction operation in our workloads involve no more than five SSTables, which means one SSTable will be merged with four or fewer SSTables in the next level, in most cases. Considering this fact, we set the searching range to four nearest SSTables in both left and right directions. Note that such a technique cannot be applied to the tra-



(a) Compaction operation without dispatching optimization

(b) Compaction operation with dispatching optimization

Figure 4. Illustration of dispatching optimization for compaction.

ditional hardware-based SSD controller because it does not have enough information from the software level. It demonstrates the flexibility of software-hardware co-optimization provided by the scheduler in LOCS.

3.3.4 Scheduling Optimization for Erase

So far, the dispatching policy is only optimized for write requests. Besides the dispatching, the scheduling of requests can also impact the performance of the system. In this subsection we will discuss how to schedule the erase requests to improve the throughput.

As mentioned in previous subsections, the dispatching of erase requests cannot be adapted. However, since the erase process is not on the critical path, the scheduling can be adapted dynamically. For the LSM-tree-based KV stores, the erase operations only happen after compaction. The SSTables used as inputs of compaction are useless and should be erased. A straightforward method is to recycle these storage spaces by erasing the SSTables right after the compaction. However, such an erase strategy can degrade performance, especially when there are intensive read operations. First, the read operation may be blocked by the erase for a long period of time due to long erase latency. Second, the queues can become unbalanced because the dispatching policy for both erase and read requests is fixed. An example is shown in in Figure 5(a). The existence of one erase operation in Channel 1's request queue results in a long delay of the following two read operations.

The solution to this problem is to delay the erase requests and schedule them when there are enough write requests. This is because the write requests can help balance the queue length. The key to this solution is to determine whether there are enough write requests. In LOCS, we set up a threshold TH_w for the ratio of write requests. The erase requests are scheduled when the ratio of write requests reaches the threshold. Note that the erase requests are forced to be scheduled when the percentage of free blocks are lower than a threshold. This design is similar to the policy of garbage collection in a traditional hardware-based SSD controller.

Figure 5(b) shows that by removing this erase operation from the request queue of Channel 1, the overall read delay will be greatly reduced. Figure 5(c) shows a scenario where the queues contain seven write operations. Each queue is scheduled by the LWQL policy. It takes 12 time slots to process the requests. When a delayed erase operation is inserted into the request queue, as shown in Figure 5(d), the LWQL policy ensures that the four write requests arriving after the erase request are inserted to the shortest queues, making the

queues balanced without increasing the processing time. We can see that the total time to complete all operations in Figure 5(a) and 5(c) is 19 without the erase scheduling, while it reduces to 15 in Figure 5(b) and 5(d) when the optimization is applied. Thus the overall throughput is improved.

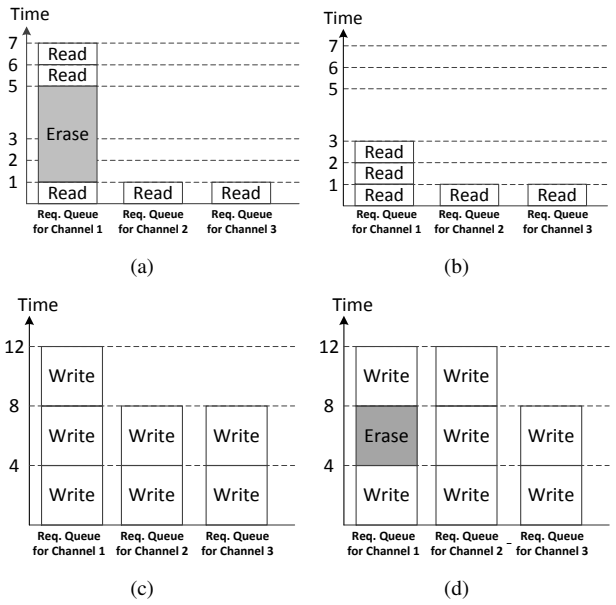


Figure 5. Illustration of scheduling optimization for erase.

4. Evaluation

In this section we describe the experimental setup for evaluation of LOCS and comprehensive evaluation results and analysis.

4.1 Experimental Setup

We conducted experiments on a machine equipped with SDF, the customized open-channel SSD. The configuration of the machine is described in Table 1.

CPU	2 Intel E5620, 2.4 GHz
Memory	32 GB
OS	Linux 2.6.32

Table 1. Evaluation platform configuration.

We used a PCIe-based SSD, Huawei Gen3, for comparison in the evaluation. In fact, the Huawei SSD is the predecessor of SDF, which shares the same flash chips with SDF, except that SDF has a redesigned controller and interface. In Huawei SSD, data are stripped over its 44 channels with a stripping unit of 8 KB. Table 2 shows its specification.

In the following experiments, "RR" denotes the round-robin dispatching policy described in Section 3.3.1, "LWQL" denotes the least weighted-queue-length dispatching policy described in Section 3.3.2, and "COMP" denotes the dispatching optimization for compaction described in Section 3.3.3.

Host Interface	PCIe 1.1 x8
Channel Count	44
Flash Capacity per Channel	16 GB
Channel Interface	Asynchronous 40 MHz
NAND Type	25 nm, MLC
Page Size	8 KB
Block Size	2 MB

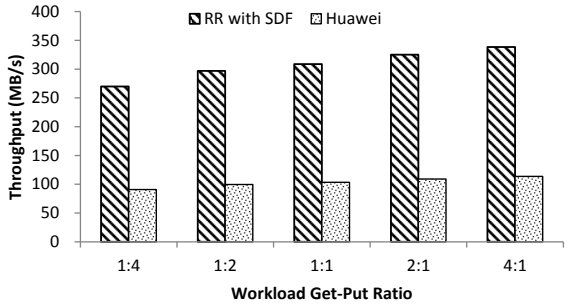
Table 2. Device specification of SDF and Huawei Gen3.

4.2 Evaluation Results

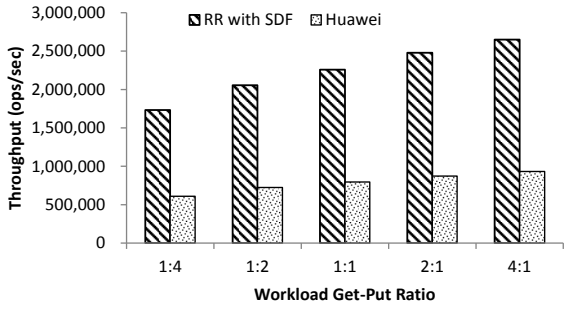
We compare the performance of the stock LevelDB running on the Huawei SSDs and the optimized LevelDB running on SDF. Figure 6(a) shows the comparison of I/O throughput. The round-robin dispatching policy is used for the SDF. The channel dispatching for Huawei SSD is implemented within its firmware, and the SSTables generated by LevelDB are stripped and dispatched to all the channels uniformly. The results show that the I/O throughput can be significantly improved on SDF with the optimized LevelDB for different benchmarks with various get-put request ratios. On average, the I/O throughput can be improved by about 2.98×. It shows that LOCS can exploit high access parallelism even without any optimization of scheduling and dispatching policies. In the Huawei SSD, each large write request (2MB) from LevelDB is stripped over its 44 channels, with a stripping unit size of 8 KB, and distributed over different channels to benefit from the parallel accesses to multiple channels. In SDF, the case is different with the multiple request queues. The large write request from LevelDB is issued to a channel without stripping. Thus, the requests are uniformly distributed to the channels in the granularity of 2MB. Large requests are split into multiple sub-requests in the Huawei SSD, and serviced in different channels. Accordingly, requested data has to be split (for write) or merged (for read) in the request service, and each channel serves a larger number of smaller requests. This adds to the overhead and reduces throughput. Furthermore, there is additional write amplification overhead caused by the garbage collection in the Huawei SSD. The expensive garbage collection can compromise performance stability.

Figure 6(b) compares the performance of LevelDB in terms of number of operations per second (OPs). The trend is similar to that of I/O throughput. The improvement is about 2.84× on average, which is a little bit lower than that of I/O throughput. The reason is that writes generated by compaction are counted into I/O throughput but not considered in the calculation of OPs of LevelDB. Note that we only show results for data with a value size of 100 Bytes due to space constraint. In fact, our LOCS system consistently achieves performance improvement across different data value sizes.

In Figure 7, we study the impact of Immutable MemTables in main memory on the I/O throughput of SDF. In this experiment, the get-put data ratio is set to be 1:1, and the



(a)



(b)

Figure 6. Comparison of sustained performance with Huawei Gen3 SSD.

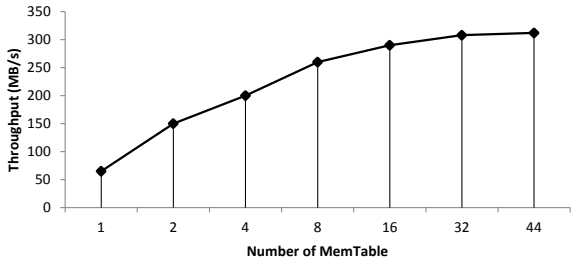


Figure 7. Impact of number of MemTables.

key value size is 8 KB. Since the number of Immutable MemTables determines the number of concurrent write requests to the SDF, the I/O throughput increases proportionally to the number of MemTables. The results show that the I/O throughput saturates when MemTables count reaches the number of flash channels. In fact, if we further increase the count, the I/O throughput of SDF could be reduced because of competition of service by excessive numbers of concurrent write requests at each flash channel. Thus, we always set the number of MemTables to the number of flash channels. The results for other configurations show the same trend.

We analyze the impact of the thresholds for write request traffic control in LevelDB in the following experiments. The dispatching policy is round-robin, and the value size and get-put ratio are set to be 8 KB and 1:1, respectively.

In Figure 8, the fluctuation of I/O throughput is illustrated and compared to two cases when threshold `kL0_Slowdown WritesTrigger` is set to be 8 and 68, respectively. Threshold `kL0_StopWritesTrigger` is set to be $1.5 \cdot$`kL0_Slowdown WritesTrigger`. The periodic fluctuation and significant throughput degradation are caused by traffic control on write requests. It is apparent that the period of fluctuation becomes larger with a higher threshold. It increase from 30 seconds to about 70 seconds, and accordingly, the average throughput increases from 221 MB/s to 349 MB/s. This suggests that the threshold should set to be much larger value in SDF to accommodate its much higher access parallelism.

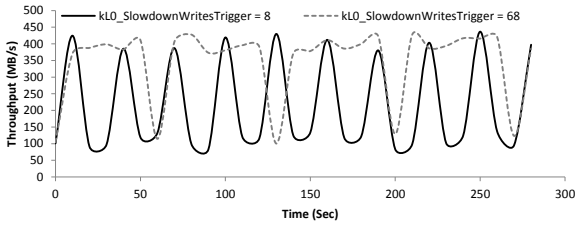


Figure 8. Illustration of throughput fluctuation with different slowdown thresholds.

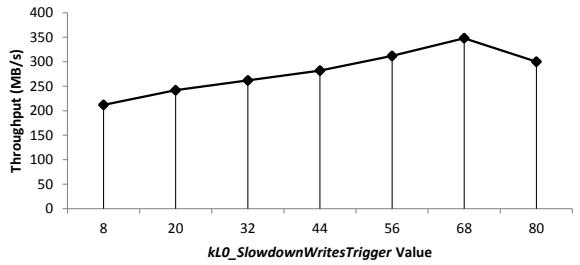


Figure 9. Impact of SlowdownWritesTrigger threshold.

Note that the throughput is not always improved when the threshold increases. This is because a higher threshold results in more SSTables in Level 0. Since the latency of searching data in Level 0 SSTable is increased, the overall throughput may be decreased with a too-large threshold. In order to confirm this finding, we measure the throughput with different values of the threshold and compare them in Figure 9. We find that the sweet point appears around the value of 68.

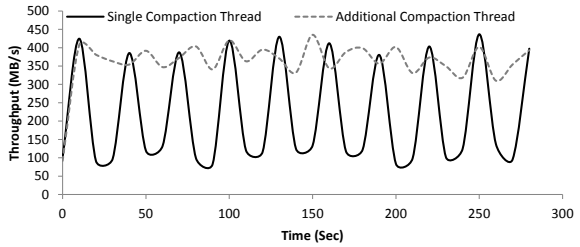


Figure 10. Illustration of throughput fluctuation with additional compaction thread.

We study the impact of introducing an additional compaction thread in Figure 10. We find that the fluctuation of throughput can be mitigated with this extra thread. This is because the SSTables in Level 0 can be compacted at a higher rate so that the slowdown of write requests is alleviated. On average, the throughput can be increased from 221 MB/s to about 361 MB/s. Note that the `kL0_Slowdown WritesTrigger` threshold is still set to be 8 in order to highlight the impact of adding one extra thread. The results of putting together these two techniques are shown in Figure 11. The throughput can be improved up to about 375 MB/s on average.

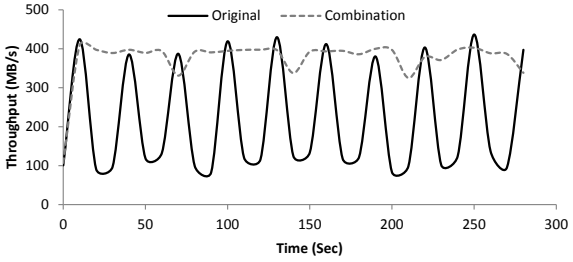


Figure 11. Illustration of throughput fluctuation after applying two techniques.

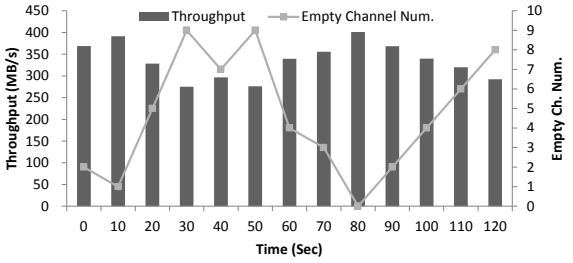


Figure 12. Impact of empty queues.

In the next several sets of experiments, we study the impact of dispatching policies on the throughput of SDF and the performance of LevelDB. The value size is 8 KB and the get-put ratio is 1:1. As shown in Figure 12, in order to demonstrate the importance of balancing the lengths of weighted queues, we first illustrate the relationship between transiently empty queues and the I/O throughput. The average number of empty queues is obtained by sampling queue length over a long execution period of LevelDB. We find that the throughput of SDF has significant fluctuation and is inversely proportional to the number of empty queues. It is easy to understand that the transiently empty queues are caused by unbalanced dispatching of requests. On average, an empty queue may result in about 14 MB/s loss of I/O throughput.

In order to address this problem, we have proposed the LWQL dispatching policy. In Figure 13, we compare the standard deviation of queue length for all the 44 channels over a long execution period of LevelDB. It is easy to see

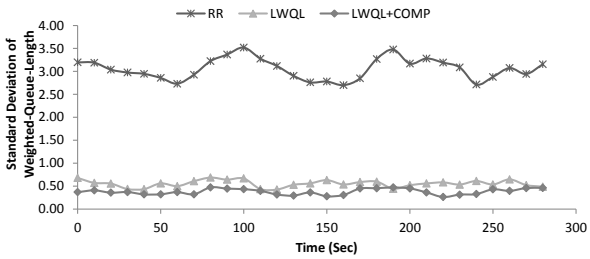


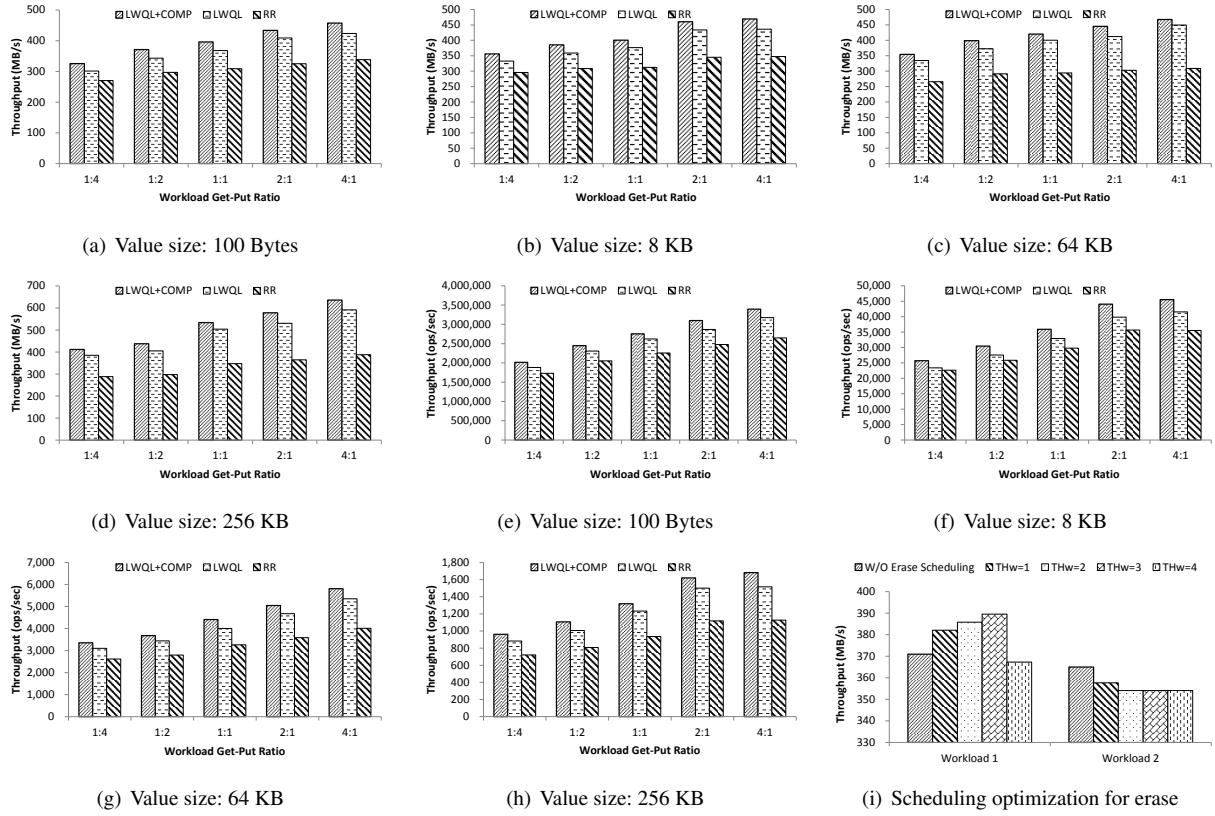
Figure 13. Standard deviations of weighted-queue-lengths.

that, after using the LWQL dispatching policy, the deviation of queue length is reduced significantly, compared to the baseline round-robin policy. This indicates that an optimized dispatching policy for write requests can effectively help to balance the I/O intensity across channels.

The third set of results in Figure 13 shows the deviation of queue length when the optimization of the dispatching for compaction is applied. As discussed in Section 3.3.3, this technique can help to address the problem of dispatching multiple SSTables, which are likely to be read for later compaction, into the same channel. In other words, it can help to balance the intensity of read requests across different channels. Thus, the deviation of queue length can be further reduced after using this technique compared with the LWQL policy. Note that the improvement is not very significant because the optimization can only be applied to the channels with the shortest queue lengths.

We evaluate the performance of our LOCS system in Figure 14. In order to get a sustained performance result, the total amount of data written in each workload is at least 1 500 GB, twice as large as the SDF's capacity. The results of I/O throughput for different value sizes, get-put data ratios, and various benchmarks are shown in Figures 14(a)–(d). All of these results demonstrate similar trends and support prior discussion about dispatching optimization for balanced queue lengths. For example, the I/O throughput of SDF is increased by 30% on average after using the LWQL policy, compared to the baseline case of using the round-robin dispatching policy. For the same setup, after applying the optimization technique for compaction, the improvement to throughput of SDF is further increased by 39%. Note that the improvement not only comes from balancing queue length but also benefits from the more efficient compaction operations. Another observation is that the improvement decreases as the get-put request ratio decreases. As we mentioned in Section 3, this is because the round-robin policy works well when the write intensity is high. Compared with the baseline that simply runs the stock LevelDB on SSDs, the throughput is improved by more than 4×.

In Figures 14(e)–(h), we compare the performance of LevelDB in terms of LevelDB OPs for three dispatching policies with different value sizes, get-put data ratios, and benchmarks. We can find a throughput trend similar to that of the SSD throughput. On average, the number of OPs is



(a) Value size: 100 Bytes (b) Value size: 8 KB (c) Value size: 64 KB

(d) Value size: 256 KB (e) Value size: 100 Bytes (f) Value size: 8 KB

(g) Value size: 64 KB (h) Value size: 256 KB (i) Scheduling optimization for erase

Figure 14. Comparison of performance: sustained throughput and LevelDB OPs/sec.

improved by 21% after using the LWQL dispatching policy and is further improved to 31% after applying the optimization technique for compaction. It means that our optimization techniques can not only increase the throughput of SSD but also improve the performance of LevelDB.

In Figure 14(i), we illustrate the impact of erase scheduling operations with two workloads of varying read/write ratios. The first workload (Workload 1) can be divided into two phases: the equal-read-write phase with a get-put request ratio of 1 and the write-dominant phase with a get-put request ratio of 1/4. The second workload (Workload 2) has the read-modify-write pattern with a get-put ratio of around 1. The baseline policy is to erase blocks right after the compaction without intelligent erase scheduling. This is compared to the erase scheduling policy described in Section 3.3.4 with various thresholds TH_w on the ratio of write requests. We can see that the throughput is improved for Workload 1 when using our scheduling policy with a TH_w less than 4. This is because significantly varied read/write ratios provide the opportunity for the policy to clearly identify a time period to perform erase operations for higher throughput. However, for Workload 2, the throughput is reduced with our scheduling policy, as the write intensity rarely reaches the threshold for performing erase operations. Thus, most erase operations are postponed until free blocks are used up and have to be performed on the critical path of the request service.

5. Related Work

Key-Value Stores. Distributed key-value systems, such as BigTable [18], Dynamo [22], and Cassandra [28], use a cluster of storage nodes to provide high availability, scalability, and fault-tolerance. LevelDB [10] used in this work runs on a single node. By wrapping the client-server support around the LevelDB, LevelDB can be employed as the underlying single-node component in a distributed environment, such as Tair [6], Riak [5] and HyperDex [23]. In order to meet the high throughput and low latency demands of applications, several recent works, such as FlashStore [20], SkimpyStash [21], and SILT [30], explore the key-value store design on the flash-based storage. But all of the existing works use the conventional SSDs that hide its internal parallelism to the software. Our work focuses on how to use the open-channel SSD efficiently with the scheduling and dispatching techniques, and is complementary to the techniques aimed at consistency and node failure.

Log-Structured Merge Trees. LSM-tree [32] borrows its design philosophy from the log-structured file-system (LFS) [35]. The LSM-tree accumulates recent updates in memory, flushes the changes to the disk sequentially in batches, and merges on-disk components periodically to reduce the disk seek costs. LevelDB [10] is one of the representative implementations of LSM trees. TableFS [34] uses LevelDB as part of the stacked filesystem to store small

metadata. Recently, several similar data structures on persistent storage were proposed. TokuDB [7] and TokuFS [24] use Fractal Tree, which is related to cache-oblivious streaming B-trees [12]. VT-trees [36] were developed as a variant of LSM-tree that avoids unnecessary copies of SSTables during compaction by adding pointers to old SSTables. Other LSM-tree-based KV stores, such as Cassandra [28] and HBase [9], can also make full use of software-defined flash. The scheduling and dispatching techniques proposed in Section 3.3.3 can also be applicable to them.

NAND Flash-Based SSD. NAND Flash-Based SSDs have received widespread attention. To enhance the I/O performance, most SSDs today adopt the multi-channel architecture. It is necessary to exploit parallelism at various layers of the I/O stack for achieving high throughput of SSD in the high concurrency workloads. Chen et al. investigate the performance impact of SSD's internal parallelism [19]. By effectively exploiting internal parallelism, the I/O performance of SSDs can be significantly improved. Hu et al. further explore the four levels of parallelism inside SSDs [26]. Awasthi et al. [11] describe how to run HBase [9] on a hybrid storage system consisting of HDDs and SSDs to minimize the cost without compromising throughput. However, their design doesn't utilize the parallelism of SSD. Our design successfully exploits the channel-level parallelism in a software manner.

Some researchers also tried to make use of the internal parallelism. Bjorling et al. propose a Linux block I/O layer design with two levels of multi-queue to reduce contention for SSD access on multi-core systems [13]. Our design also uses multiple queues, but the queues are in the user space instead of in the kernel space. In addition, we propose a series of scheduling and dispatching policies optimized for the LSM-tree-based KV stores. Wang et al. present an SSD I/O scheduler in the block layer [38] which utilizes the internal parallelism of SSD via dividing SSD logical address space into many subregions. However, the subregion division is not guaranteed to exactly match the channel layout. Moreover, they use only a round-robin method to dispatch the I/O request, which, as shown in our work, is not optimal.

More closely related to our design is SOS [25], which performs out-of-order scheduling at the software level. By rearranging I/O requests between queues for different flash chips, SOS can balance the queue size to improve the overall throughput. However, their "software-based scheduler" is actually located in the flash translation layer (FTL) and implemented inside an embedded FPGA controller, which is under the OS and part of the hardware. So they cannot get application-level information as we do. What is more, the scheduling policy they adopt only considers the number of requests of a queue, not the weighted-queue-length, as we do in this work. Since the three types of NAND flash operations have sharply different costs, this scheduling policy make it hard to balance the queues if there are both reads and writes in the request queue. By setting all weights to 1 in the LWQL policy, LWQL essentially degenerates into SOS and has a reduced throughput.

Several SSD designs similar to the open-channel SDF are also proposed by industry. FusionIO's DFS [27] employs a virtualized flash storage layer. It moves the FTL in the hardware controller into the OS kernel to allow direct access to the NAND flash devices. The NVM Express specification [3] defines a new interface, in which a NVM storage device owns multiple deep queues to support concurrent operations.

6. Conclusions

The combination of LSM-tree-based KV stores and SSDs has the potential to improve I/O performance for storage systems. However, a straightforward integration of both cannot fully exploit the high parallelism enabled by multiple channels in the SSDs. We find that the I/O throughput can be significantly improved if the access to channels inside SSD can be exposed to KV stores. The experimental results show that the I/O throughput can be improved by up to 2.98×. With such a storage system, the scheduling and dispatching policies for requests from the KV stores have an important impact on the throughput. Thus, we propose several optimization techniques of scheduling and dispatching policies for the scheduler at the software level. With these techniques, the I/O throughput of LOCS can be further improved by about 39% on average.

Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, Christian Cachin, for their valuable feedback and comments. We thank the engineers at Baidu, especially Chunbo Lai, Yong Wang, Wei Qi, Hao Tang, for their helpful suggestions. This work was supported by the the National Natural Science Foundation of China (No. 61202072), National High-tech R&D Program of China (No. 2013AA013201), as well as by generous grants from Baidu and AMD.

References

[1] Apache CouchDB. `http://couchdb.apache.org/`.

[2] HyperLevelDB. `http://hyperdex.org/performance/leveldb/`.

[3] NVM Express explained. `http://nvmexpress.org/wp-content/uploads/2013/04/NVM_whitepaper.pdf`.

[4] Redis. `http://redis.io/`.

[5] Riak. `http://basho.com/leveldb-in-riak-1-2/`.

[6] Tair. `http://code.taobao.org/p/tair/src/`.

[7] TokuDB: MySQL performance, MariaDB performance. `http://www.tokutek.com/products/tokudb-for-mysql/`, .

[8] Tokyo Cabinet: A modern implementation of DBM. `http://fallabs.com/tokyocabinet/`, .

[9] Apache HBase. `http://hbase.apache.org/`.

[10] LevelDB – a fast and lightweight key/value database library by Google. `http://code.google.com/p/leveldb/`.

[11] A. Awasthi, A. Nandini, A. Bhattacharya, and P. Sehgal. Hybrid HBase: Leveraging flash SSDs to improve cost per throughput of HBase. In *Proceedings of the 18th International Conference on Management of Data (COMAD)*, pages 68–79, 2012.

[12] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the 19th ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 81–92, 2007.

[13] M. Bjorling, J. Axboe, D. Nellans, and P. Bonnet. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 22:1–22:10, 2013.

[14] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[15] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[16] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, 2010.

[17] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, 2012.

[18] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[19] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 266–277, 2011.

[20] B. Debnath, S. Sengupta, and J. Li. FlashStore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2): 1414–1425, Sept. 2010.

[21] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 25–36, 2011.

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, 2007.

[23] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A distributed, searchable key-value store. *SIGCOMM Comput. Commun. Rev.*, 42(4):25–36, Aug. 2012.

[24] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. The TokuFS streaming file system. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '12, pages 14:1–14:5, 2012.

[25] S. Hahn, S. Lee, and J. Kim. SOS: Software-based out-of-order scheduling for high-performance NAND flash-based SSDs. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 13:1–13:5, 2013.

[26] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren. Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance. *Computers, IEEE Transactions on*, 62(6):1141–1155, 2013.

[27] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. DFS: A file system for virtualized flash storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST '10, pages 85–100, 2010.

[28] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[29] N. Leavitt. Will NoSQL databases live up to their promise? *Computer*, 43(2):12–14, 2010.

[30] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, 2011.

[31] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST '12, pages 139–154, 2012.

[32] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[33] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 471–484, 2014.

[34] K. Ren and G. Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 USENIX Annual Technical Conference*, USENIX ATC '13, pages 145–156, 2013.

[35] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[36] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building workload-independent storage with VT-trees. In *Proccedings of the 11th Conference on File and Storage Technologies*, FAST '13, pages 17–30, 2013.

[37] M. Stonebraker. SQL databases v. NoSQL databases. *Commun. ACM*, 53(4):10–11, Apr. 2010.

[38] H. Wang, P. Huang, S. He, K. Zhou, C. Li, and X. He. A novel I/O scheduler for SSD with improved performance and lifetime. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 6:1–6:5, 2013.