2014 IEEE 28th International Parallel & Distributed Processing Symposium

Pipelined Compaction for the LSM-tree

Zigang Zhang[†§], *Yinliang Yue*[†], *Bingsheng He*[‡], *Jin Xiong*[†], *Mingyu Chen*[†], *Lixin Zhang*[†], *Ninghui Sun*[†]
[†] SKL Computer Architecture, ICT, CAS
[§] University of Chinese Academy of Sciences
[‡] Nanyang Technological University

***Abstract*—Write-optimized data structures like Log-Structured Merge-tree (LSM-tree) and its variants are widely used in key-value storage systems like BigTable and Cassandra. Due to deferral and batching, the LSM-tree based storage systems need background compactions to merge key-value entries and keep them sorted for future queries and scans. Background compactions play a key role on the performance of the LSM-tree based storage systems. Existing studies about the background compaction focus on decreasing the compaction frequency, reducing I/Os or confining compactions on hot data key-ranges. They do not pay much attention to the computation time in background compactions. However, the computation time is no longer negligible, and even the computation takes more than 60% of the total compaction time in storage systems using flash-based SSDs. Therefore, an alternative method to speedup the compaction is to make good use of the parallelism of underlying hardware including CPUs and I/O devices.**

In this paper, we analyze the compaction procedure, recognize the performance bottleneck, and propose the *Pipelined Compaction Procedure* (PCP) to better utilize the parallelism of CPUs and I/O devices. Theoretical analysis proves that PCP can improve the compaction bandwidth. Furthermore, we implement PCP in real system and conduct extensive experiments. The experimental results show that the pipelined compaction procedure can increase the compaction bandwidth and storage system throughput by 77% and 62% respectively.

***Index Terms*—storage system; LSM-tree; compaction; pipeline**

I. Introduction

The massive Internet services, like cloud computing, cloud storage, search engine, social networking and so on, generate more and more data over the time. Storage systems must scale out to support the above applications. Key-value storage systems known for scalability and ease of use, including distributed key-value stores, such as BigTable [2], Dynamo [11] and PNUTS [12], and local key-value stores, like LevelDB [10] and Berkeley DB [13], are widely used to support the network service workloads. Since distributed key-value stores comprise many individual local key-value stores, the performance of local key-value stores plays a significant role on the performance of distributed key-value stores. This paper focuses on the performance improvements of the local key-value store.

Many applications have stringent latency requirements. According to the study [1] of Yahoo!, the ratio of writes is increasing in comparison with that of reads. Write-optimized data structures are widely used in storage systems to reduce the write latency, because most read requests are absorbed by multi-level caches [21], implemented by Web browser's cache, CDN, Redis [14], Memcached [15] [16] [17] and OS page cache, and modified data must be written to persistent storage devices, such as HDDs and SSDs (flash-based Solid State Disk), to ensure data persistence. LSM-tree [4] and its variants are the most commonly used write-optimized data structures. For example, LSM-tree, COLA [6] and SAMT [7] are used in LevelDB, Hadoop HBase [9] and Cassandra [18] respectively.

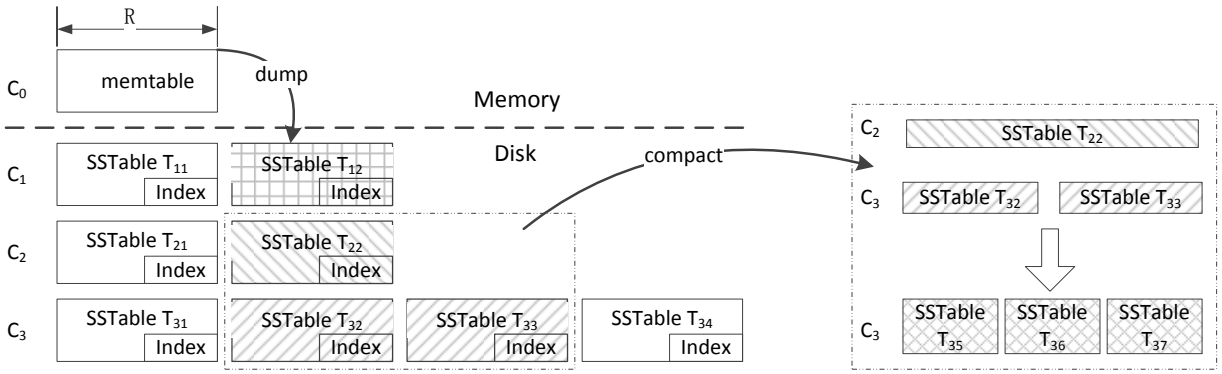
LSM-tree has one memory buffer component and multiple disk resident components. When memory buffer is full, the buffer data is dumped into a SSTable [2] on disks, whose key range may overlap with those of existing SSTables. To bound the latency of upcoming point queries and scans, background compactions are needed to compact SSTables and keep key-value entries sorted. Background compactions move data downwards from upper components to lower components. Slow data movements incur write pauses. That is, the storage system can not serve updates any more until the background compaction completes. So, background compactions play an important role on the LSM-tree based storage systems performance.

Some researches have been done to improve the LSM-tree based storage system performance. VT-tree [5] uses the stitching technique to avoid unnecessary data movement. bLSM [1] uses the replacement-selection sort algorithm to reduce the compaction frequency. PE [19] and bLSM partition the key range to confine compactions in hot key ranges. COLA and FD-tree [8] use forward pointers to improve the lookup performance. bLSM uses bloom filters to avoid unnecessary I/Os. While GTSSL [7] uses the reclamation and re-insert techniques to put data at upper components to expedite lookups, it also uses SSDs as the upper components' storage media and read data cache. However, all these researches ignore the computation time in the compaction procedure and do not exploit the parallelism between the I/O resource and the computation resource.

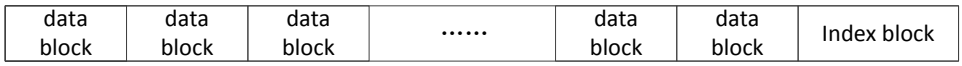
In storage systems, data compression and checksum are always used to reduce the amount of I/Os and verify data integrity, which account for a lot of additional computation cycles. Besides, new types of storage devices like flash-based SSDs have been widely used for their high bandwidth and lower latency, which decrease the I/O time. As a result, the computation time is no longer negligible in comparison with the I/O time. In practice, the computation may take more than 60% of the total compaction time in storage systems using flash-based SSDs.

Each compaction merges the key-value pairs from the

1530-2075/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPS.2014.85



(a) LSM-tree data structure



(b) SSTable layout

Fig. 1. The basic LSM-tree data structure and SSTable layout.

adjacent two components in a specified key range. There are multiple data blocks in the key range. In the existing compaction procedure, the data blocks are scheduled in an ordered manner, and the steps of the compaction procedure of each data block are executed sequentially. We name the above compaction procedure as **S**equential **C**ompaction **P**rocedure (SCP).

There exists two types of resources in storage system, i.e., the computation resource and the I/O resource. Some steps in one data block's compaction procedure utilize the computation resource, and the other steps utilize the I/O resource. Due to the ordered scheduling of data blocks in SCP, either the computation resource or the I/O resource is utilized over a period, which results in the underutilization of both the computation resource and the I/O resource. Motivated by the fact that there is no data dependency among the data blocks in the same component and the steps of each data block can be scheduled on different hardware components, we propose the **P**ipelined **C**ompaction **P**rocedure (PCP). In PCP, we partition the compaction key range into multiple sub-key ranges and each sub-key range comprises one or more data blocks. Thus the compaction is divided into multiple sub-tasks. Each sub-task is in charge of merging the key-value entries in one sub-key range. We exploit the parallelism of the computation resource and the I/O resource to parallelize sub-tasks to further improve the compaction bandwidth. To the best of our knowledge, this is the first paper to exploit the parallelism of the I/O resource and the computation resource to improve the compaction bandwidth.

The compaction bandwidth of the PCP depends on the bottleneck stage, which has the smallest bandwidth in all stages. In the pipelined compaction procedure, either the CPU or the storage device may be the bottleneck. For the above two cases, we propose two parallel variants of PCP, named **C**omputation-**P**arallel **P**ipelined **C**ompaction **P**rocedure (C-PPCP) and **S**torage-**P**arallel **P**ipelined **C**ompaction **P**rocedure (S-PPCP). When the CPU is the bottleneck, C-PPCP can be used for background compactions. Otherwise, S-PPCP is used. The I/O-bound cases can be transformed to CPU-bound cases when excessive storage devices are used for reads and writes, and the CPU-bound cases can be transformed to I/O-bound cases when excessive CPUs are used for computation.

We implemented the pipelined compaction procedure on LevelDB, which is a representative LSM-tree implementation. Compared with LevelDB, the pipelined compaction procedure increases the compaction bandwidth by 77%, and improves the throughput by 62%. The parallel pipelined compaction procedure improves the compaction bandwidth and throughput by 89% and 64% respectively.

The rest of this paper is organized as follows. Background and motivation are presented in Section II. Section III analyzes the SCP and use the (Parallel) Pipelined Compaction Procedure to improve the compaction bandwidth. Implementation and performance evaluation through extensive experiments appear in Section IV. Related work is discussed in Section V. Section VI concludes the paper.

II. Background and Motivation

A. Background

LSM-tree is a disk-based data structure designed to provide indexing for the workloads that have a high rate of records inserts (updates, and deletes) over an extended period. LSM-tree is composed of a memory buffer component C_0 and multiple disk components, C_1, ..., and C_k. Each component size is limited to a predefined threshold, which grows exponentially. Figure 1(a) illustrates one basic LSM-tree data structure. LSM-tree supports inserts, updates, deletes, point queries and scans. LSM-tree uses an algorithm that defers and batches index updates, cascading the changes from a memory buffer through many disk components. The algorithm gathers random, small I/Os into sequential, large I/Os.

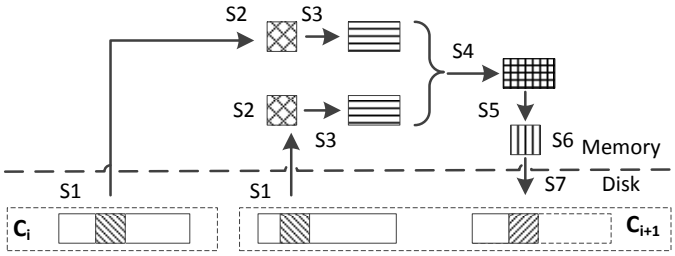


Fig. 2. The compaction procedure.

Each disk component consists of multiple SSTables, whose key ranges do not overlap with each other except those in C_1. Figure 1(b) presents the layout of the SSTable. Each SSTable contains multiple data blocks and one index block. The data blocks contain the sorted key-value pairs. The index block indexes all the data blocks, containing the start key, the end key and the offset of each data block in the SSTable.

As more key-value entries are inserted into memtable, the memtable is filled up and the buffer data is dumped into one C_1 SSTable onto disk. For example, the new built SSTable is T_{12} in Figure 1(a). The key range of T_{12} may overlap the key range of the existing T_{11}. When C_i exceeds its threshold size, the compaction procedure for C_i is triggered. The key-value pairs in a specific key range from the corresponding SSTables in C_i and C_{i+1} are merged into multiple size-limited SSTables in C_{i+1}. Through this way, the data flows downwards from C_i to C_{i+1}. For example, when C_2 exceeds its threshold size, the compaction procedure for C_2 is triggered. The compaction procedure picks T_{22} in C_2 and T_{32} and T_{33} in C_3 whose key ranges overlap with the key range of T_{22}. Then, T_{22}, T_{32} and T_{33} are compacted into T_{35}, T_{36} and T_{37} in C_3. As a result, the data in T_{22} flows downwards from C_2 to C_3.

One compaction merges the key-value pairs in a specific key range, which consists of multiple data blocks. In the existing compaction procedure, the data blocks are scheduled orderly. The compaction procedure of each data block comprises seven steps, illustrated as Figure 2. And the steps of one data block are executed sequentially for data dependency. The compaction procedure iterates the following steps for each data block until the last data block has been compacted.

1) Step 1(S1): READ. Read one data block along with its checksum from disk into memory.
2) Step 2(S2): CHECKSUM. Calculate the checksum of the data block, and compare it with the original checksum which is read from disk in *Step 1* to verify the integrity of the data block.
3) Step 3(S3): DECOMPRESS. Decompress the data block and restore the original key-value pairs.
4) Step 4(S4): SORT. Merge the key-value pairs from the two components and build new data block.
5) Step 5(S5): COMPRESS. Compress the new built data block to improve the future read and write performance.
6) Step 6(S6): RE-CHECKSUM. Calculate the checksum of each compressed data block for future integrity check in *Step 2*.
7) Step 7(S7): WRITE. Write the compressed data block along with its checksum which is calculated in *Step 6* to the disk.

B. Motivation

In the existing compaction procedure, the data blocks are scheduled in an ordered manner, and the steps of the compaction procedure of each data block are executed sequentially for the data dependency. *Step 1* and *Step 7* utilize the I/O resource, and all the other steps utilize the computation resource. Due to the ordered scheduling of data blocks, only one type of resource, either the computation resource or the I/O resource, is used over a period, which results in the severe underutilization of both the computation resource and the I/O resource. Because the key ranges of different data blocks in the same component do not overlap, there is no data dependency among them. So it's not necessary to schedule the data blocks in an ordered manner. Motivated by the fact that there is no data dependency among different data blocks and the steps of each data block can be scheduled on different hardware components, we propose the Pipelined Compaction Procedure (*PCP*). In *PCP*, we partition the compaction key range into multiple sub-key ranges, thus the compaction is divided into multiple sub-tasks. Each sub-task compacts one or more data blocks. *PCP* exploits the parallelism of the computation resource and the I/O resource to parallelize the sub-tasks to improve the compaction bandwidth.

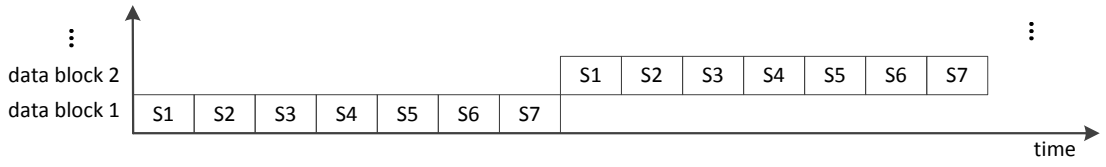
III. Design

This section discusses the design of the Pipelined Compaction Procedure. First, we analyze the existing Sequential Compaction Procedure (SCP), and point out its inefficiency. Then we improve the SCP with pipeline model, and propose the Pipelined Compaction Procedure (PCP). For different hardware configurations, PCP suffers different bottleneck stages, maybe I/O-bound or CPU-bound. Furthermore, we propose two parallel variants of PCP, Storage-Parallel Pipelined Compaction Procedure (S-PPCP) and Computation-Parallel Pipelined Compaction Procedure (C-PPCP), to alleviate the I/O bottleneck and CPU bottleneck. When excessive storage devices are used for I/Os, the I/O-bound cases can be transformed to CPU-bound cases, and when excessive CPUs are used for computation, the CPU-bound cases can be transformed to I/O-bound cases.

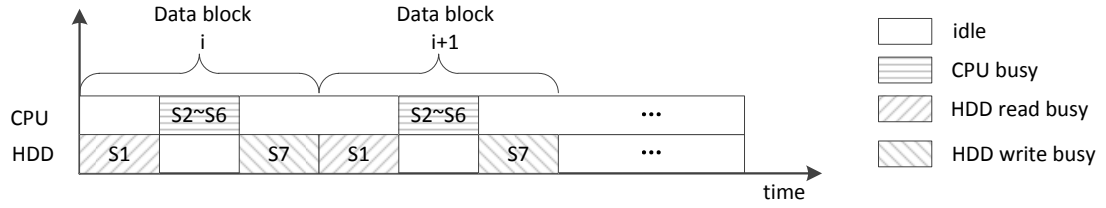
A. Sequential Compaction Procedure (SCP)

As described in Section II-A, one compaction compacts multiple data blocks and the compaction procedure of each data block comprises seven steps. In existing conventional compaction procedure, the data blocks are scheduled orderly as Figure 3(a).

Step 1 and *Step 7* utilize the I/O resource, i.e., HDD or SSD. All the other steps utilize computation resource, i.e., CPU or GPU. Suppose that the storage resource and computation resource in the storage system are HDD and CPU. The



(a) Sequential execution in Sequential Compaction Procedure



(b) CPU and Disk utilization in Sequential Compaction Procedure

Fig. 3. The execution process and utilization of CPU and Disk of the Sequential Compaction Procedure.

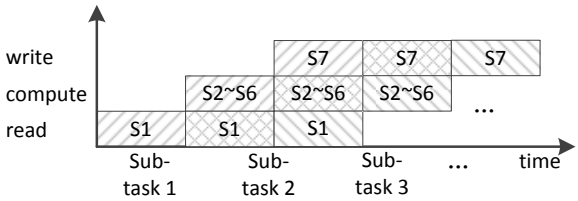


Fig. 4. Ideal Pipelined Compaction Procedure: *Step 1* and *Step 7* are scheduled on disk, and all the other steps are scheduled on CPU.

utilization of resources in Sequential Compaction Procedure is depicted as Figure 3(b).

As one data block is processed on disk and CPUs orderly, when the HDD transfers data from disk to memory in *Step 1* or in the reverse direction in *Step 7* through direct memory access, the CPU is idle. When the CPU does the computation *Step 2* $\sim$ *Step 6* including integrity verifying, decompressing, compressing and merging, the HDD is idle. The HDD and CPU do work sequentially and wait for each other for input. Over a period, only one type of resource is used, which results in severe underutilization of both the I/O resource and the computation resource.

In this paper, we use compaction bandwidth as the metric to evaluate the performance of compaction procedure. Compaction bandwidth indicates the amount of data compacted in one time unit. l represents the length of one data block. t_{S_i} represents the execution time of *Step i* for one data block. We can get the compaction bandwidth of SCP as Equation 1.

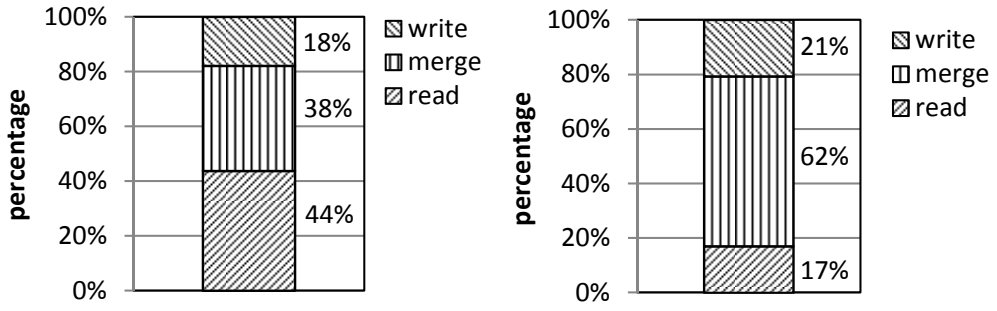
$$B_{scp} = \frac{l}{\sum_{i=1}^{7} t_{S_i}} \tag{1}$$

B. Pipelined Compaction Procedure

A naive approach to improve the compaction bandwidth for SCP is to parallelize the steps of one data block. For example, one can do the *Step 1* and *Step 2* simultaneously to reduce the execution time. However, the steps of one data block can not be parallelized due to data dependency. The good news is that although there are many dependencies within the steps of one data block, there are no data dependencies across subsequent data blocks in the same component due to that their key ranges do not overlap. Motivated by the fact that there are no data dependencies among different data blocks in the same component and the steps of each data block can be scheduled on different hardware components, we exploit the pipeline model to SCP and propose the Pipelined Compaction Procedure (*PCP*). Hence our approach is to exploit the inter-data block parallelism to overlap the I/O time and the computation time of one data block with that of other data blocks. PCP partitions the compaction key range into multiple sub-key ranges. Each sub-key range consists of one or more data blocks. *PCP* exploit the parallelism between the disk and CPU.

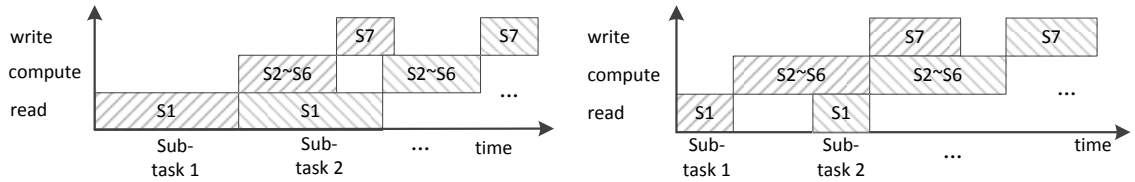
A natural question is that how many stages the Pipelined Compaction Procedure should be divided into. A straightforward way is that *Step 1* and *Step 7* are two individual stages, and the other steps are partitioned into multiple stages evenly according to the execution time. That is, $Step2 \sim Step6$ are partitioned into multiple stages. Different stages are executed on different CPUs. Unfortunately, the above method has the following disadvantages. Firstly, it is difficult to divide the adjacent compaction steps evenly, because the execution times of different steps differ significantly. And it is meaningless to make nonadjacent steps into a stage. Secondly, this results in low dcache performance. Data blocks must flows through multiple processors. The processors may not share the dcache, or the data blocks are evicted from dcache when they are waiting for processing. Besides, if nonadjacent stages are assigned to one processor, this worsen the dcache performance further. Thirdly, it will result in load imbalance. Lastly, the above method does not scale well. So, we do not divide $Step2 \sim Step6$ further and let them as one stage. Thus the pipelined compaction procedure is divided into three stages, i.e., stage *read*, stage *compute*, and stage *write*.

We schedule *Step 1* and *Step 7* on disk, and the other



(a) on HDD (b) on SSD

Fig. 5. The execution time breakdown of sequential compaction procedure into three parts.



(a) on HDD (b) on SSD

Fig. 6. The practical Pipelined Compaction Procedure on HDD and SSD

steps on CPU. However, the difference with SCP is that the compaction procedures of adjacent sub-tasks are parallelized, as Figure 4.

For PCP, we can get the compaction bandwidth as Equation 2 when one sub-task consists of only one data block.

$$B_{pcp} = \frac{l}{\max\{t_{S_1}, \sum_{i=2}^{6} t_{S_i}, t_{S_7}\}} \tag{2}$$

From Equation 1 and Equation 2, we get the ideal performance speedup of *PCP* as Equation 3. Equation 3 shows that in comparison with SCP, PCP improves the compaction throughput obviously.

$$\frac{B_{pcp}}{B_{scp}} = \frac{\sum_{i=1}^{7} t_{S_i}}{\max\{t_{S_1}, \sum_{i=2}^{6} t_{S_i}, t_{S_7}\}} \tag{3}$$

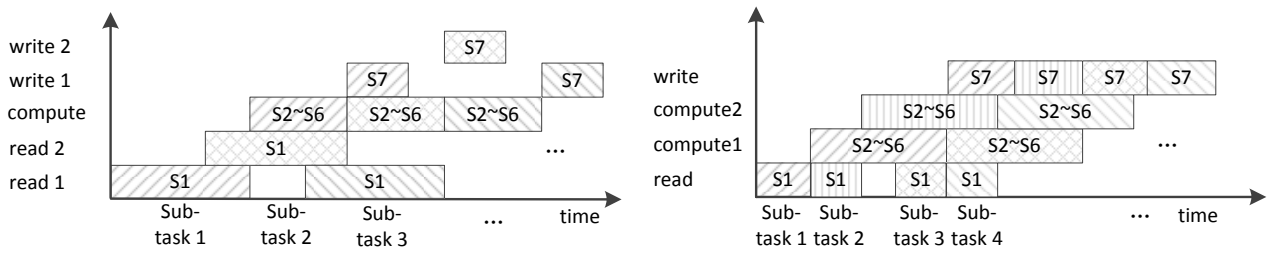
Suppose that *Step 1* and *Step 7* are scheduled on HDDs. The bandwidth of HDD is two order of magnitude smaller than DRAM, and becomes even worse for random I/Os. Although *Step 2* $\sim$ *Step 6* are scheduled on CPU, CPU may also wait for the HDD. For HDD, one data transfer may consume more than ten milliseconds, due to mechanical disk head seek and rotation latency. Besides, *Step 1* and *Step 7* are scheduled on HDD, and the read and write requests contend to the same disk, which aggravates the randomness. We profile the sequential compaction procedure on HDD and break down the compaction time into three parts as Figure 5(a). The *Step 1* takes more than 40% of the compaction time, *step 7* takes less than 20% and all the computation steps take about 40%. So *read* is the bottleneck operation. The *Step 1* and *Step 7* take about 60% of the compaction time, so HDD is the bottleneck. So, the pipelined compaction procedure resembles Figure 6(a). Under this scenario, the pipelined compaction procedure is I/O-bound.

Suppose that *Step 1* and *Step 7* are scheduled on flash-based solid state drives, which are known for high throughput and low latency in comparison with hard disk drives. Because there are no mechanical parts in flash-based SSDs, the bandwidth of SSD may be over five times larger than HDD, especially for random I/Os. We profile the sequential compaction procedure on SSD and break down the compaction time into three parts as Figure 5(b). The computation steps take more than 60% of the compaction time. Both *Step 1* and *Step 7* take less than 40% of the compaction time totally. Obviously, the CPU becomes the bottleneck, as Figure 6(b). Under this scenario, the pipelined compaction procedure is CPU-bound.

C. Parallel Pipelined Compaction Procedure (PPCP)

In this section, we propose two parallel variants of PCP to remove the performance bottleneck. Present data center nodes are always equipped with multi-core processors or multiple processors and multiple disks, including HDDs and SSDs. In the *PCP*, for the I/O-bound case, we exploit the parallelism of multiple storage devices to improve the I/O bandwidth to alleviate the I/O performance bottleneck, which is named as Storage-Parallel Pipelined Compaction Procedure (*S-PPCP*). For the CPU-bound case, we exploit the parallelism of multiple cores or processors to improve the computation bandwidth to alleviate the computing performance bottleneck, which is named as Computation-Parallel Pipelined Compaction Procedure (*C-PPCP*).

1) S-PPCP: In *S-PPCP*, we use multiple disks for *Step 1* and *Step 7*. *Step 1* and *Step 7* of different sub-tasks are scheduled on different disks. For example, we use 2 disks to do the I/Os, as Figure 7(a). *Step 1* of sub-task 1 is scheduled



(a) S-PPCP (b) C-PPCP

Fig. 7. Parallel Pipelined Compaction Procedure

on disk 1 and *Step 1* of sub-task 2 is scheduled on disk 2. Thus I/Os of different sub-tasks parallelize.

Suppose that there are *k* disks for *Step 1* and *Step 7* and one sub-task consists of one data block, we can get the compaction bandwidth of *S-PPCP* as Equation 4.

$$B_{s-ppcp} = \frac{l}{\max\{\frac{t_{S_1}}{k}, \sum_{i=2}^{6} t_{S_i}, \frac{t_{S_7}}{k}\}} \tag{4}$$

When $k < \frac{\max\{t_{S_1}, t_{S_7}\}}{\sum_{i=2}^{6} t_{S_i}}$, the pipelined compaction procedure is still I/O-bound. When $k > \frac{\max\{t_{S_1}, t_{S_7}\}}{\sum_{i=2}^{6} t_{S_i}}$, the pipelined compaction procedure becomes CPU-bound. For the latter case, even if we use more storage devices, the compaction bandwidth doesn't increase any more. Note that when using excessive storage devices in the pipelined compaction procedure, the pipelined compaction procedure may become CPU-bound.

From Equation 2 and Equation 4, we can get the ideal performance speedup of S-PPCP as Equation 5. The ideal speedup is at most $\min\{k, \frac{\max\{t_{S_1}, t_{S_7}\}}{\sum_{i=2}^{6} t_{S_i}}\}$.

$$\frac{B_{s-ppcp}}{B_{pcp}} = \frac{\max\{t_{S_1}, \sum_{i=2}^{6} t_{S_i}, t_{S_7}\}}{\max\{\frac{t_{S_1}}{k}, \sum_{i=2}^{6} t_{S_i}, \frac{t_{S_7}}{k}\}} \tag{5}$$

2) C-PPCP: For the CPU-bound case, we don't increase the pipeline depth to increase the parallelism as described in Section III-B. Instead we schedule the computation steps of different sub-tasks on different cores or processors. That is, multiple sub-tasks are processed at the same time, but the computation steps of one sub-task are done on the same processor. For example, there are two cores to do the computation, as Figure 7(b). Then $S_2 \sim S_6$ of sub-task 1 are scheduled on core 1 and $S_2 \sim S_6$ of sub-task 2 are scheduled on core 2.

Suppose that there are *k* cores to do the computation and each sub-task consists of one data block. We get the compaction bandwidth of *C-PPCP* as Equation 6.

$$B_{c-ppcp} = \frac{l}{\max\{t_{S_1}, \frac{\sum_{i=2}^{6} t_{S_i}}{k}, t_{S_7}\}} \tag{6}$$

When $k < \frac{\sum_{i=2}^{6} t_{S_i}}{\max\{t_{S_1}, t_{S_7}\}}$, the CPU is still the performance bottleneck. When $k > \frac{\sum_{i=2}^{6} t_{S_i}}{\max\{t_{S_1}, t_{S_7}\}}$, the storage device becomes the performance bottleneck. For the latter case, even if we use more processors, the compaction bandwidth can't increase any more. The pipelined compaction procedure becomes I/O-bound.

From Equation 2 and Equation 6, we can get the ideal performance speedup of C-PPCP as Equation 7. The ideal speedup can not exceed $\min\{k, \frac{\sum_{i=2}^{6} t_{S_i}}{\max\{t_{S_1}, t_{S_7}\}}\}$.

$$\frac{B_{c-ppcp}}{B_{pcp}} = \frac{\max\{t_{S_1}, \sum_{i=2}^{6} t_{S_i}, t_{S_7}\}}{\max\{t_{S_1}, \frac{\sum_{i=2}^{6} t_{S_i}}{k}, t_{S_7}\}} \tag{7}$$

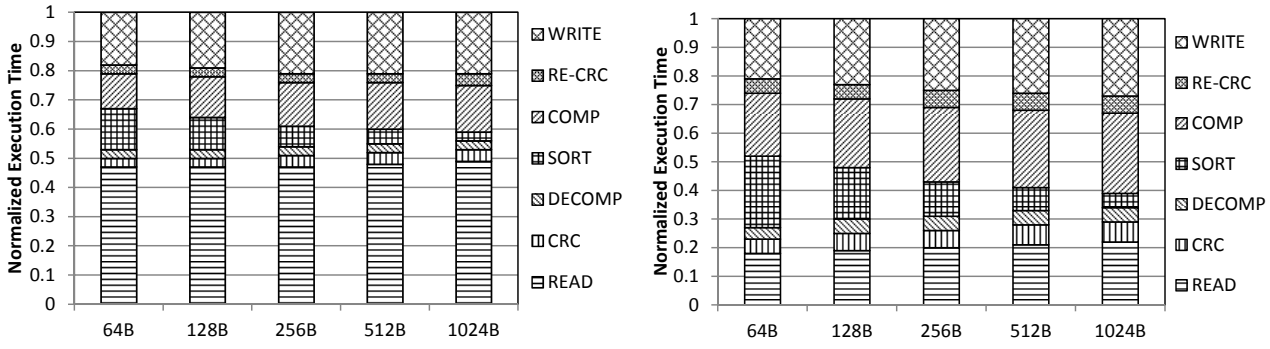
IV. Evaluation

In this section, we present the experimental methodology and workload. Then, we analyze the profiling result for the sequential compaction procedure to claim the necessity of pipelined compaction procedure. Thirdly, we evaluate the performance improvement of the pipelined compaction procedure and the impacts of sub-task size and sub-task count on the compaction bandwidth. Last, we evaluate the performance improvement of the parallel pipelined compaction procedure, and present that I/O-bound and CPU-bound cases of Pipelined Compaction Procedure may be transformed.

A. Experimental Setup

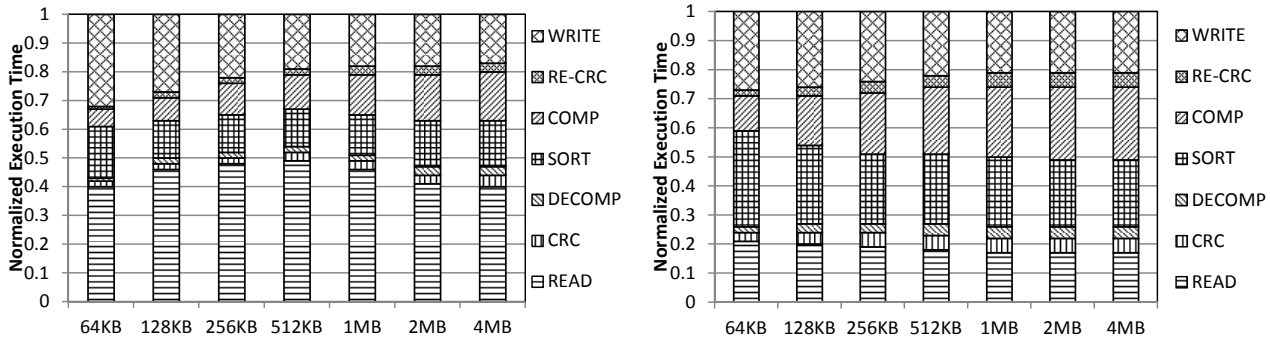
Implementation Details. The pipelined compaction procedure is implemented based on LevelDB, which is a representative LSM-tree implementation. We leave out the details of LevelDB, and instead focus on components specific to our optimization.

We use multiple threads to execute the pipelined compaction procedure. In current implementation, the pipeline depth is three, i.e., stage *read*, stage *compute* and stage *write*. For *PCP*, we assign one thread for each stage. For *S-PPCP*, we use multiple disks with the *md* driver to build RAID0 and create multiple threads for I/Os. For *C-PPCP*, we create multiple additional threads for the computation stage, and assign one thread for each sub-task. Between the adjacent stages, we create a queue for data communication.



(a) Execution time breakdown on HDD (b) Execution time breakdown on SSD

Fig. 8. The execution time breakdown of sequential compaction procedure for different key-value sizes.



(a) Execution time breakdown on HDD (b) Execution time breakdown on SSD

Fig. 9. The execution time breakdown of sequential compaction procedure for different sub-task sizes.

Measurement Methodology. We conducted experiments on one machine running Linux CentOS 6.0 *final* with kernel *2.6.32*. The machine includes a two-socket Intel Xeon E5645 (6 cores with hyper-threading, 2.4GHz, 12MB L3 cache). The machine has 16GB of DRAM. To test out of DRAM performance, we booted it with 512MB. The machine has 11 1TB 7200RPM SATA III disks, one as the system disk and ten as data disks. Besides, it also has a 240GB SATA III 2.5in Intel X25-M Solid State Drive.

Except that we claim explicitly, we used the following parameter values for all experiments. That is, the memtable size is 4MB, the SSTable size is 2MB, the data block size is 4KB and the compression algorithm is *snappy*. In all experiments, we use insert-only workloads to do the compaction profiling and evaluate the compaction bandwidth improvements. The key and value size is 16B and 100B respectively. The number of key-value pairs is fifty million.

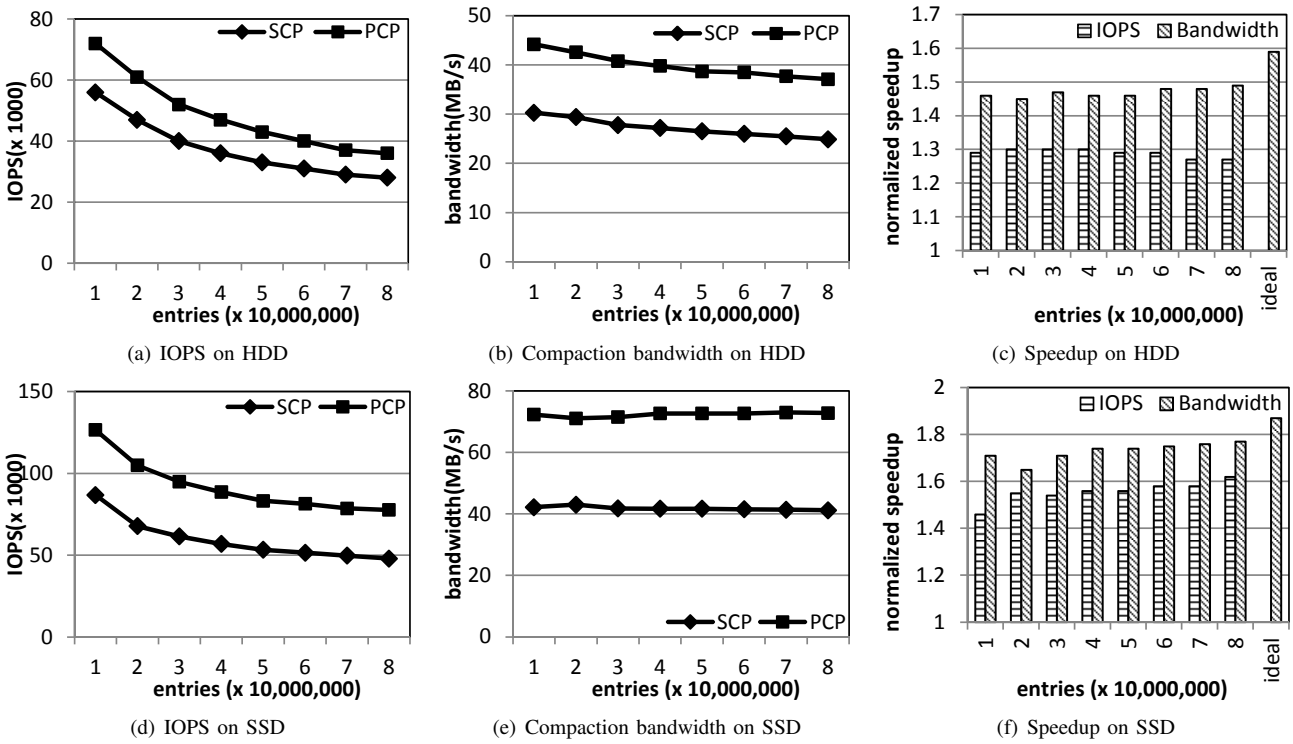
B. Sequential Compaction Procedure Profiling

In this section, we profile the compaction procedure to evaluate the execution time of the seven compaction steps. Figure 8 depicts the breakdown of the execution time for different key-value sizes from 64B to 1024B. Figure 9 depicts the execution time for different sub-task sizes from 64KB to 4MB. To avoid the impact of page cache, we use direct I/O for reads and writes.

As depicted in Figure 8(a) and Figure 9(a), the step *read* takes more than 40% of the compaction time. Firstly, due to the mechanical characteristics of hard disk drives, one data transfer may consume more than 10ms. Secondly, the SSTables are dynamically allocated. As a result, the data can not be placed on disk sequentially. Although the I/O size in the compaction is large, the disk arm may suffer seeks, due to that there are multiple sub-tasks in one compaction. Thirdly, the step *write* contends to use the same disk. However, the write bandwidth is better than step *read* as the write request is considered completed after the data has been written into the disk write buffer rather than the disk. Besides, the read requests and write requests interleave, which gives the disk a chance to write back the buffer data to disk. With the addition of the step *write*, the input and output of the compaction take more than 60% of the compaction time, which illustrates that the HDD is the performance bottleneck.

As depicted in Figure 8(b) and Figure 9(b), the step *write* takes more time than step *read*, which is due to the write-after-erase feature of flash-based solid stage disks. All the computation steps take about 60% of the time. Obviously, the CPU may be the performance bottleneck in storage systems using flash-based SSDs.

As the key-value size increases, step *sort* takes less time due to the decreasing amount of key-value entries. The execution time of step *write* decreases as the sub-task size increases because of that larger I/O size can exploit the internal parallelism of SSD and increase the bandwidth of HDD. No matter on SSD or HDD, either step *crc* or step *re-crc* takes less than



(a) IOPS on HDD (b) Compaction bandwidth on HDD (c) Speedup on HDD

(d) IOPS on SSD (e) Compaction bandwidth on SSD (f) Speedup on SSD

Fig. 10. The performance of Sequential Compaction Procedure and Pipelined Compaction Procedure on HDD and SSD.

5% of the compaction time, and step *decomp* takes the least amount of time. In all the computational steps, step *comp* is almost the most costly.

C. Pipelined Compaction Procedure

In this section, we evaluate the performance improvement of the pipelined compaction procedure. We increase the size of working set from ten million to eighty million entries as Figure 10. Figure 10(a) and Figure 10(d) show that when the data set size increases, the throughput of both Sequential Compaction Procedure and Pipelined Compaction Procedure on HDD and SSD decreases. As more key-value entries are inserted into LSM-tree based storage system, there are more components in the storage system. Thus key-value entries flow downwards through more components, which results in lower throughput.

Figure 10(b) and Figure 10(e) depict the compaction bandwidth. Figure 10(b) presents that as more entries are inserted, the compaction bandwidth deceases slightly on HDD. Because the data set size increases on HDD, the seek cost increases. However, as there are no mechanical parts in solid state drives, the compaction bandwidth on SSD does not decrease in Figure 10(e).

We normalize the IOPS and compaction bandwidth speedup of *Pipelined Compaction Procedure* against *Sequential Compaction Procedure* as Figure 10(c) and 10(f). *PCP* improves the throughput at least by 25% on HDD in Figure 10(c) and 45% on SSD in Figure 10(f). *PCP* improves the compaction bandwidth at least by 45% on HDD and 65% on SSD. Compared with the ideal compaction bandwidth speedup which is calculated using the values in section IV-B, the practical compaction bandwidth speedup is lower by about 10%. This is due to the overhead of the pipeline compaction procedure filling and draining. The throughput speedup is lower than the speedup of the compaction bandwidth by 20% approximatively. Because in the storage system, there are other operations, including database consistence maintaining, garbage collecting, and other operations, which are not pipelined by now.

Besides, we also evaluate the performance improvement of *Pipelined Compaction Procedure* for different sub-task sizes and for different compaction sizes on SSD. As depicted in Figure 11(a), the sub-task size increases from 64KB to 4MB with fixed compaction size, in which the input size of upper component is 4MB. While the sub-task size increases, the compaction bandwidth of *Sequential Compaction Procedure* increases. Due to that the I/O size is equal to the sub-task size, large I/O size can exploit the internal parallelism of SSDs and improve the bandwidth of SSDs. However, the compaction bandwidth of *Pipelined Compaction Procedure* first increase and then decrease. Because too small I/O size can not exploit the internal parallelism of SSDs. Too large I/O size decreases the sub-task count, which decreases the efficiency of PCP. The compaction bandwidth of PCP using 512KB sub-task size is the highest. In Figure 11(b), we increase the input size of upper component from 1MB to 10MB while the sub-task size is set 1MB. For *Sequential Compaction Procedure*, the compaction bandwidth does not increase as the compaction size increase. The compaction bandwidth of PCP keeps on increasing until the sub-task count reach 6. Large compaction size increase the efficiency of PCP. Figure 11 shows that *Pipelined Compaction Procedure* can improve the compaction bandwidth for all sub-

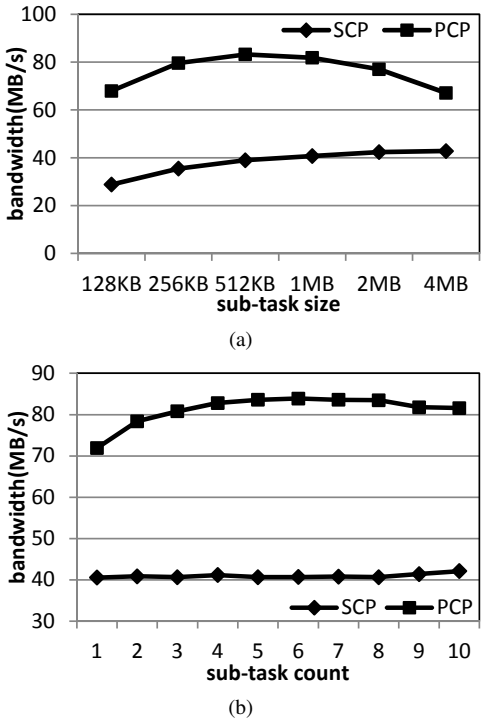


Fig. 11. Compaction bandwidth with different subtask sizes and compaction sizes.

task sizes and compaction sizes.

D. Parallel Pipelined Compaction Procedure

In this section, we evaluate the performance of *Parallel Pipelined Compaction Procedure*. Figure 12 presents the performance improvement on throughput, compaction bandwidth, and the corresponding speedup. For *S-PPCP*, we increase the number of disks to alleviate the disk bottleneck. The multiple disks are built into RAID0. As depicted in Figure 12(a), the throughput increases when more disks are used for input and output. The throughput does not increase any more when the disk count reaches 5 since the CPU becomes the performance bottleneck, and then *Pipelined Compaction Procedure* becomes CPU-bound. For *C-PPCP*, we increase the number of threads to do the computation work. As depicted in Figure 12(d), the throughput increases when another thread is added to do the computation work. After that, the *Pipelined Compaction Procedure* becomes I/O-bound. When more threads are added to do the computation work, the throughput and the compaction bandwidth decrease. This is due to the overhead of creation and synchronization of multiple threads.

V. Related Work

Improve the lookup performance. COLA [6] and FD-tree [8] exploit fractional cascading technique [20] to confine the search data extent. COLA puts the every eighth element of the $(k+1)^{st}$ array in the k^{th} array as the forward pointer. FD-tree puts the first element of every page of the $(k+1)^{st}$ array in the k^{th} array as the search fence. Thus, in each level, point queries just do one I/O. bLSM [1] uses bloom filter to avoid disk I/Os for the level which does not contain the sought-after key. GTSSL [7] uses the reclamation and re-insert techniques to put key-value pairs in upper levels to decrease the count of levels that point queries go through. bLSM [1] and the partitioned exponential file [19] partition the key range into multiple sub-key ranges, and then point queries just search one smaller sub-key range.

Improve the compaction performance. bLSM [1] uses the replacement-selection sort algorithm in the compaction procedure of component C_0 to increase the length of sorted key-value pairs, which decreases the frequency of compactions for all the components. VT-tree [5] uses the stitching technique to avoid unnecessary disk I/Os for sorted and non-overlap key range. But the stitching technique may incur fragmentation which degrades the performance of scans and compactions. bLSM [1] and PE [19] partition the key range into multiple sub-key range and confine compactions in hot data key ranges, which accelerate the data flow.

Exploit flash-based SSDs. GTSSL [7] targets hybrid HDD-SSD systems and uses SSDs as the storage media of lower components and the cache of hot data. Based on the observation that random writes with good locality have performance similar to sequential writes on flash-based SSDs, FD-tree [8] uses SSDs as the storage media.

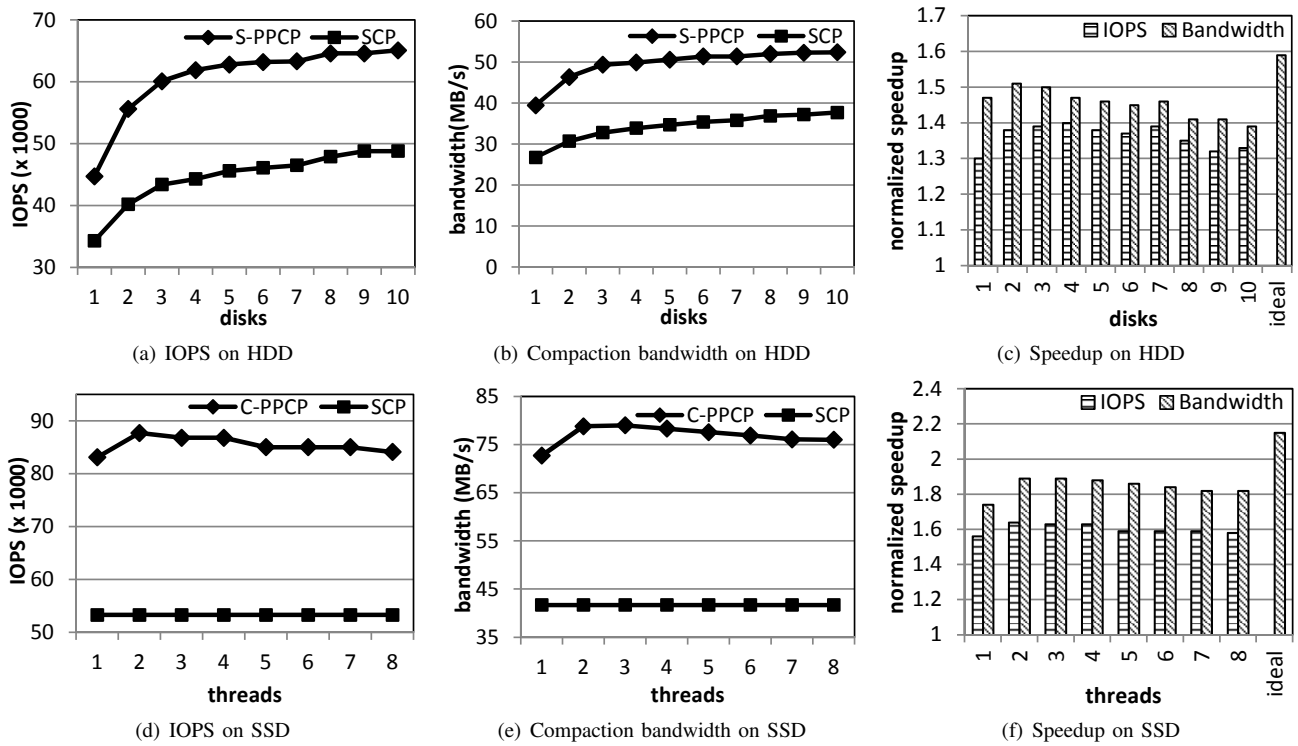
In theory, the pipelined compaction procedure is orthogonal to existing studies.

VI. Conclusion

Background compactions play an important role on the performance of LSM-tree based storage systems. The existing studies on background compactions focus on decreasing compaction frequency, reducing I/Os, or confining compactions on hot key-ranges to improve compaction performance. However, they ignore computation time and don't exploit the parallelism of I/O resource and computation resource. In practice, the compaction may be CPU-bound and the computation takes more than 60% of the total compaction time in storage systems using faster storage devices like flash-based SSDs. In this paper, we analyze the compaction procedure and recognize the performance bottleneck. Then we propose the pipelined compaction procedure. For one compaction, we divide it into multiple sub-tasks and exploit the parallelism of I/O resource and CPU resource to parallelize multiple sub-tasks. Furthermore, we improve the pipelined compaction procedure with multiple storage devices or CPUs to alleviate the performance bottleneck. Theoretical analysis proves that the pipelined compaction procedure can improve the compaction bandwidth. Extensive experimental results show that the (parallel) pipelined compaction procedure improves the compaction performance significantly.

VII. Acknowledgements

We thank the anonymous reviewers for their helpful feedback. This work is supported in part by National Basic Research Program of China under grant No.2011CB302502, National Science Foundation of China under grants No.61379042, No. 61303056, No. 60925009



(a) IOPS on HDD (b) Compaction bandwidth on HDD (c) Speedup on HDD

(d) IOPS on SSD (e) Compaction bandwidth on SSD (f) Speedup on SSD

Fig. 12. The performance of Sequential Compaction Procedure and Parallel Pipelined Compaction Procedure on HDD and SSD.

and No. 61221062, and Huawei Research Program YB2013090048.

REFERENCES

[1] Russell Sears and Raghu Ramakrishnan. *bLSM: A General Purpose Log Sturctured Merge Tree*. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes and Robert E. Gruber. *Bigtable: A Distributed Storage System for Structured Data*. In ACM Trans. Comput. Syst. 26(2): 1-26.

[3] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons and Todd C. Mowry. *Improving Hash Join Performance through Prefetching*. In *ACM Trans. Database Syst.*, 2007.

[4] Parrick O'Neil, Edward Cheng, Dieter Gawlick Elizabeth O'Neil. *The Log-Structured Merge-Tree (LSM-Tree)*. In Acta Informatica, 1996.

[5] Pradeep Shetty, Richard Spillane, and el at. *Building Workload-Independent Storage with VT-Trees*. In 11th USENIX Conference on File and Storage Technologies, 2013.

[6] 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 nineteenth annual ACM symposium on Parallel algorithms and architectures, 2007.

[7] Richard P. Spillane, Pradeep J. Shetty, and etc. *An Efficient Multi-Tier Tablet Server Storage Architecture*. In Proceedings of the 2nd ACM Symposium on Cloud Computing, 2011.

[8] Yinan Li, Bingsheng He, and etc. *Tree indexing on Solid State Drives*. In Proc. VLDB Endow., 2010.

[9] HBase main page. http://hbase.apache.org/.

[10] LevelDB main page. https://code.google.com/p/leveldb/.

[11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels. *Dynamo: amazon's highly available key-value store*. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, 2007.

[12] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, Ramana Yerneni. *PNUTS: Yahoo!'s hosted data serving platform*. In Proc. VLDB Endow., 2008.

[13] Michael A. Olson, Keith Bostic, and Margo Seltzer. *Berkeley DB*. In USENIX ATC, 1999.

[14] Redis. *http://redis.io/*.

[15] Memcached. *http://memcached.org/*.

[16] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, Mike Paleczny. *Workload Analysis of a Large-Scale Key-Value Store*. In SIGMETRICS, 2012.

[17] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, Venkateshwaran Venkataramani. *Scaling Memcache at Facebook*. In 10th USENIX Symposium on Networked Systems Design and Implementation, 2013.

[18] Avinash Lakshman, Prashant Malik. *Cassandra: a decentralized structured storage system*. In SIGOPS Oper. Syst. Rev., 2010.

[19] Christopher Jermaine, Edward Omiecinski, Wai Gen Yee. *The partitioned exponential file for database storage management*. In The VLDB Journal Volume 16 Issue 4, Pages 417-437 , 2007.

[20] B. Chazellel and L. J. Guibas. *Fractional cascding: I. a data structuring technique*. Algorithmica, 1(2), 1986.

[21] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. *An Analysis of Facebook Photo Caching*. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13).