Future Generation Computer Systems 65 (2016) 76–89

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

ActiveSort: Efficient external sorting using active SSDs in the MapReduce framework

Young-Sik Lee [a], Luis Cavazos Quero [b], Sang-Hoon Kim [a], Jin-Soo Kim [b,*], Seungryoul Maeng [a]

[a] School of Computing, Korea Advanced Institute of Science and Technology, 291 Daehak-ro, Yuseong-gu, Daejeon 34141, Republic of Korea

[b] College of Information and Communication Engineering, Sungkyunkwan University, 2066 Seobu-Ro, Jangan-gu, Suwon 16419, Republic of Korea

HIGHLIGHTS

- A mechanism of efficient external sorting by on-the-fly data merge inside SSDs.
- Performance of data merge is improved using parallelism of multi-channel inside SSDs.
- Performance of Hadoop is improved by reducing the amount of I/O using active SSDs.
- Lifetime of SSDs in Hadoop is extended by reducing the amount of write.

ARTICLE INFO

Article history:
Received 1 July 2015
Received in revised form
18 February 2016
Accepted 4 March 2016
Available online 16 March 2016

Keywords:
Data-intensive computing
MapReduce
External sorting
Solid state drives
In-storage processing

ABSTRACT

In the last decades, there has been an explosion in the volume of data to be processed by data-intensive computing applications. As a result, processing I/O operations efficiently has become an important challenge. SSDs (solid state drives) are an effective solution that not only improves the I/O throughput but also reduces the amount of I/O transfer by adopting the concept of active SSDs. Active SSDs offload a part of the data-processing tasks usually performed in the host to the SSD. Offloading data-processing tasks removes extra data transfer and improves the overall data processing performance.

In this work, we propose ActiveSort, a novel mechanism to improve the external sorting algorithm using the concept of active SSDs. External sorting is used extensively in the data-intensive computing frameworks such as Hadoop. By performing merge operations on-the-fly within the SSD, ActiveSort reduces the amount of I/O transfer and improves the performance of external sorting in Hadoop. Our evaluation results on a real SSD platform indicate that the Hadoop applications using ActiveSort outperform the original Hadoop by up to 36.1%. ActiveSort reduces the amount of write by up to 40.4%, thereby improving the lifetime of the SSD.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

Data-intensive computing is a class of applications that process a large amount of data to extract useful information. The amount of data to be processed is continuously increasing at a fast rate in such fields as web search, social networking service, e-commerce, scientific computing, and log processing. As a result, the efficiency of computing data in these data-intensive applications becomes a major concern. Google's MapReduce [1] provides a famous programming model for data-intensive computing, and Apache Hadoop [2] implements this model as an open source project.

Data-intensive computing handles a large amount of data which generates a vast amount of I/O. Thus, I/O performance is crucial to the processing of the ever-increasing volumes of data. Adopting fast storage devices is one of the simplest, yet most effective ways to improve the I/O performance [3,4]. Solid state drives (SSDs) are in the spotlight since they are increasingly being adopted in data-intensive computing because of their advantages over legacy hard disk drives (HDDs) such as high performance, low power consumption, small form factor, light weight, and shock resistance.

One of the recent trends in the use of SSDs in data-intensive computing is to revisit the concept of active disks [5]. Instead of reading and processing data in the host, active disks process data within the storage device. In this spirit, the so-called *active SSDs*

* Corresponding author.
E-mail addresses: yslee@calab.kaist.ac.kr (Y.-S. Lee), luis@skku.edu (L.C. Quero), sanghoon@calab.kaist.ac.kr (S.-H. Kim), jinsookim@skku.edu (J.-S. Kim), maeng@kaist.ac.kr (S. Maeng).

http://dx.doi.org/10.1016/j.future.2016.03.003
0167-739X/© 2016 Elsevier B.V. All rights reserved.

attempt to offload data processing functions (e.g., min/max, scan, count, histogram, etc.) to SSDs where the data is already stored [6–10]. In these active SSDs, the host can read results directly from the active SSDs without incurring excessive data transfer nor host-side computation, achieving better performance and energy saving. Additionally, this approach becomes more promising to improve the efficiency and scalability in data-intensive computing as the computing capability of SSDs is becoming more powerful.

However, the previous active SSD studies are limited in terms of the types of data-processing functions they consider. They primarily consider simple data filtering or aggregation functions, which only output a small amount of result after summarizing a large amount of data. For the other type of data processing function such as external sorting which has the same size of input/output data, another approach is required to get the benefits of active SSDs.

In this work, we propose a novel mechanism called *ActiveSort* that accelerates external sorting using the concept of active SSDs. By adopting the concept of ActiveSort, we improve the MapReduce framework in the aspects of the performance, energy efficiency, and lifetime of SSDs. External sorting is one of the core data-processing algorithms that enables to sort large-scale data using a limited amount of memory. The MapReduce framework extensively utilizes external sorting to generate intermediate and final outputs during the map/reduce phases [1,2]. Additionally, many query processing algorithms in database management systems (DBMSes) perform external sorting as one of their key components [11].

ActiveSort provides an efficient in-storage external sorting mechanism. The key idea behind ActiveSort is to perform the merge operation involved in external sorting within the storage device. In the traditional external sorting approach, the host produces the final sorted results by reading partially sorted data from storage and merging them in the host. In contrast, ActiveSort creates the sorted output on-the-fly within the active SSD when the host fetches the output data via standard read requests. Therefore, ActiveSort can eliminate the unnecessary data transfer incurred by reading the intermediate data and writing back the sorted data to the storage. ActiveSort significantly reduces the I/O burden of the host machine in processing large-scale data. ActiveSort is also designed to utilize the inherent characteristics of SSDs to optimize its performance.

We have integrated ActiveSort into the MapReduce framework. Our ActiveSort prototype has been implemented in a real SSD platform, and Hadoop has been modified to use ActiveSort whenever external sorting is required. While sorting the data, our experimental results indicate that ActiveSort reduces the amount of data written to the SSD by up to 50.0% and improves the elapsed time by up to 53.1% compared to the original external sorting scheme. Consequently, Hadoop using ActiveSort reduces the amount of written data by up to 40.4% and outperforms the original Hadoop by up to 36.1% while running the HiBench workloads. At the same time, Hadoop based on ActiveSort decreases the energy consumption by up to 35.2% compared to the original Hadoop.

The contributions of this work can be summarized as follows. First, we present ActiveSort, a mechanism to reduce the amount of I/O during external sorting by offloading the merge operation to the SSDs. The reduced I/O traffic improves the performance of external sorting and the lifetime of SSDs significantly. Second, we show that ActiveSort is very effective in accelerating the performance of data-intensive applications based on the Hadoop framework, as the Hadoop framework heavily relies on external sorting to manipulate a large amount of data that does not fit into the memory. The fast execution of Hadoop applications enabled by ActiveSort is essential to support near real-time big data processing in the cloud environment. Third, we also demonstrate that ActiveSort is a cost-effective solution to reduce the amount of energy consumption for running Hadoop applications. As Hadoop is generally deployed on very large-scale clusters, this also helps to reduce the operational expenditure (OPEX) in a data center for air conditioning, ventilation, and other maintenance.

The rest of this work is organized as follows. Section 2 gives the background information on SSD, external sorting, and Hadoop. The main concept of ActiveSort and how to integrate it in the Hadoop framework are described in Section 3. The detailed implementation of ActiveSort and the modified Hadoop framework is explained in Section 4. Evaluation results are presented in Section 5. Section 6 overviews the related work. Finally, we conclude the paper in Section 7.

2. Background

2.1. Solid state drives (SSDs)

Fig. 1 illustrates the general architecture of a typical SSD which is composed of an SSD controller, DRAM, and an array of NAND flash memory chips connected to flash controllers by multiple channels. Unlike HDDs, NAND flash memory has several unique characteristics. First, in-place update is not supported, hence previous data should be erased first before any new data is written into the same area. Second, write operations take a much longer time to complete than read operations. Third, there is a limit in the number of erase operations that can be performed on a given memory cell. This is commonly known as *NAND endurance* and impacts the SSD's operational lifetime.

In order to cope with the aforementioned characteristics of NAND flash memory, the SSD controller runs a sophisticated firmware called *flash translation layer (FTL)*. Usually, FTL manages the physical flash memory space in a log-structured manner and keeps track of logical-to-physical address mapping information internally. It also performs *wear-leveling* to prolong the lifetime of the SSD.

The functions performed by the FTL are getting more complex to match the increasing host interface speeds and to exploit increasing parallelism across a number of NAND flash chips connected to multiple channels. Thus, the hardware resources of an SSD are becoming more powerful; for example, Samsung 850 Pro, one of the latest high-end SSDs, features ARM-based triple cores operating at 400 MHz and 1 GB of DRAM with eight parallel NAND channels [12].

2.2. External sorting

External sorting [13] is a type of sorting algorithms that handles large-scale data which does not fit into the memory. As illustrated in Fig. 2, the traditional external sorting consists of two phases: *Partial sorting* and *Merge*.

During the Partial sorting phase, the input data is divided into chunks whose size is smaller than the available memory. The data of each chunk is sorted using an in-memory sorting algorithm and the sorted data of the chunk is written to the storage (Step 1). In the Merge phase, the partially sorted chunks are read from the storage and merged to produce the final sorted data (Step 2). The final sorted data is written into the storage (Step 3), and usually sent to the other task later for subsequent data processing (Step 4).

2.3. Hadoop

Hadoop [2] is an open-source framework for processing a large amount of data in a distributed manner. Hadoop processes the data with a simple programming model called MapReduce [1].

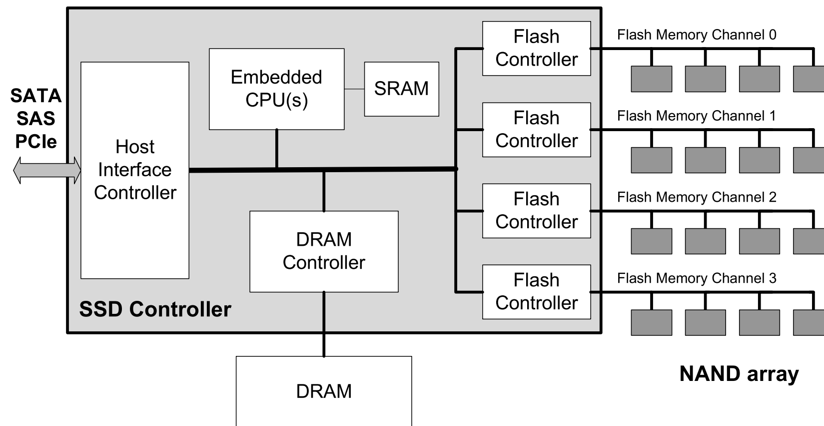


Fig. 1. The architecture of SSD.

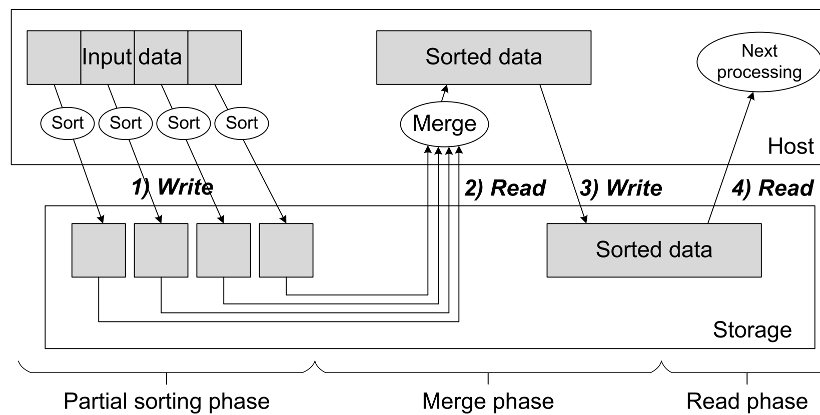


Fig. 2. The process of the traditional external sorting.

The MapReduce programming model consists of *map* and *reduce* functions that process data using structured key/value pairs. After parsing input data into key/value pairs, the map function takes the input key/value pairs and creates new intermediate key/value pairs. The reduce function performs an operation on each group of intermediate records with the same key and generates output key/value pairs.

In Hadoop,[1] tasks which perform the map/reduce functions are scheduled and assigned to several computing nodes. While an application is running on Hadoop, input and output data are usually stored in distributed file systems such as HDFS (Hadoop Distributed File System). The intermediate data produced by map/reduce tasks is stored in the local storage of the computing node running the task.

Fig. 3 depicts the detailed execution flow of a Hadoop application. First, the input data on HDFS is divided into *splits*, which are contiguous portions of the input data. In the *Map phase*, each map task applies the map function to the split and stores the intermediate data as several temporary files in the local disk. The map task partitions the temporary data using a user-defined partition function, sorts the data, and writes to the disk. Each partition is assigned to one reduce task, so that the number of reduce tasks is equal to the number of partitions. Next, in the *Shuffle phase*, the partition including the intermediate data is transferred to appropriate reduce tasks. All the intermediate data with the same key must be processed by the same reduce task. The intermediate data from several map tasks is sorted in the *Sort phase* to get the records with the same key easily. After the reduce function is applied in the *Reduce phase*, the reduce task stores the final output in HDFS.

2.4. Motivation

External sorting is a core function to execute MapReduce applications in Hadoop. In the map task, the split, which is the input data of a map task, can be larger than the available memory size. In this case, the intermediate results produced by applying the map function to the input data are sorted and then stored in the local disk as *spill files*. This procedure corresponds to the Partial sorting phase in the external sorting. After processing all the input data of the map task, the spill files are merged to pass the sorted data to the reduce tasks. This step corresponds to the Merge phase in the external sorting. The reduce task also executes external sorting to group the records by key in the reduce function. However for the reduce task, only the Merge phase of the external sorting is required since the intermediate data is already sorted by the map task.

As mentioned in Section 2.2, the external sorting produces massive I/O requests to generate the sorted output because it stores the intermediate data into the disk due to the limitation in the available memory size. Since the external sorting is extensively used in Hadoop, the large amount of I/O requests from the external sorting can overload the storage device. Thus, the efficiency of external sorting is critical to the overall performance of Hadoop applications.

[1] This description is based on the Hadoop version 1.1.2.

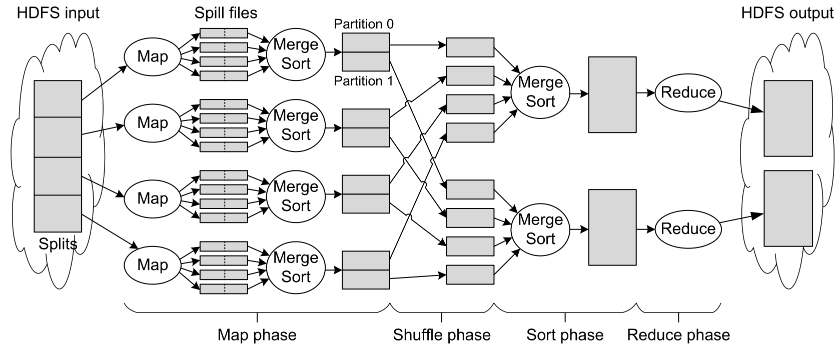


Fig. 3. The execution flow of Hadoop.

3. Design

3.1. ActiveSort

We propose a new external sorting mechanism called ActiveSort which performs data merge inside active SSDs. ActiveSort follows the conventional wisdom of moving computation where the data resides. In ActiveSort, the sorted output is synthesized when the host requires it by merging the partially sorted chunks stored in SSDs. Since the final sorted output is generated on-the-fly, the host does not have to read the partially sorted chunks and then write the sorted output to the storage (cf. Step 2 and 3 in Fig. 2).

Fig. 4 illustrates the overall process of ActiveSort. In ActiveSort, the Partial sorting phase is the same as that of the traditional external sorting (Step 1). However, the Merge phase is postponed until the host issues read requests to the final sorted output. When a read request to the final output arrives from the host, ActiveSort merges data on-the-fly inside the active SSD and transfers the result as the response of the read request (Step 2).

Algorithm 1 Host-side pseudo-code of ActiveSort

Input: unsorted data I
Output: sorted data O
1: $S \leftarrow \{S_0, S_1, \cdots, S_{k-1} \mid S_i = split(I)\}$; ▷ $|S_i| <$ the available memory size
2: **for all** S_i in S **do**
3: $C_i \leftarrow sort(S_i)$; ▷ in-memory sort
4: $write(C_i)$;
5: **end for** ▷ Partial sorting phase
6: **while** $r \leftarrow read() \neq \emptyset$ **do** ▷ issue read requests to the SSD
7: $O \leftarrow O + \{r\}$;
8: **end while**

The pseudo-codes of ActiveSort executed in the host and the SSD are presented in Algorithm 1 and 2, respectively. In the host-side (Algorithm 1), ActiveSort has a separate Partial sorting phase for each chunk (lines 2–5) and then generates the requests to the sorted output to the SSD (line 6). These read requests activate the on-the-fly data merge inside the SSD.

Before executing the merge operation, the SSD-side of ActiveSort (Algorithm 2) prepares it by reading the first records of all the chunks (lines 2–4). When the requests to the sorted data arrive at the SSD (line 5), ActiveSort finds the record whose key is the smallest (line 9) and sends the record to the output (line 10) as the response of the read request. If there is a next record to process, ActiveSort reads the record (lines 12–14) and finds the record with the minimum key again. Note that while reading the records from the chunks and sending the results to the host, ActiveSort can use multiple internal buffers to increase the I/O bandwidth. ActiveSort can also use the min-heap algorithm [14] to find the minimum key efficiently among the chunks.

Algorithm 2 SSD-side pseudo-code of ActiveSort

Input: $C = \{C_i \mid 0 \le i < k, k = \#\text{ of chunks}\}$,
partially sorted chunk $C_i = \{r_{i,j} \mid 0 \le j < n_i, n_i = \#\text{ of records in } C_i\}$
Output: record by key order
1: $S \leftarrow \emptyset$; ▷ S holds candidate records to be merged
2: **for** $i \leftarrow 0$ to $k-1$ **do**
3: $S \leftarrow S + \{r_{i,0}\}$;
4: **end for**
5: **while** read requests to the sorted data arrive **do**
6: **if** $S = \emptyset$ **then**
7: send_to_host($\emptyset$);
8: **end if**
9: $r_{i,j} \leftarrow minimum_key(S)$; ▷ find a record with minimum key in S
10: send_to_host($r_{i,j}$);
11: $S \leftarrow S - \{r_{i,j}\}$;
12: **if** $j < (n_i - 1)$ **then**
13: $S \leftarrow S + \{r_{i,j+1}\}$; ▷ add the next record in C_i
14: **end if**
15: **end while**

As shown in Fig. 4, ActiveSort requires only half of read and write operations to obtain the final sorted output compared to the traditional external sorting. Although the read performance of ActiveSort is slightly degraded due to the run-time data merge operation, reducing the amount of I/O compensates the overhead of the on-the-fly merge performed in the SSD. In addition, reducing the amount of data written to the SSD helps to improve the lifetime of SSDs significantly.

By offloading the merge operation to SSDs, ActiveSort has the following advantages. First, ActiveSort regards the SSD as an additional computation entity, so that it enables concurrent computation in the SSD and in the host. Second, the sorted results can be delivered to the other devices without involving the host CPUs if the devices support DMA. Last, ActiveSort processes I/Os efficiently by exploiting the internal architecture of SSDs, which is difficult to access from the host.

3.2. Hadoop with ActiveSort

As we mentioned in Section 2.3, Hadoop relies on external sorting extensively to deal with a large amount of data that does not fit into the limited memory. This subsection describes how to integrate ActiveSort with Hadoop.

3.2.1. ActiveSort in map and reduce tasks

Hadoop can benefit from active SSDs by replacing the external sorting with ActiveSort in map and reduce tasks. Fig. 5 outlines how ActiveSort can be used in Hadoop to improve its performance. In the map task, spill files are written into a local disk after applying the map and sort functions. This step is similar to the original Hadoop process. However, the Merge phase is skipped until the read request for the output of the map task arrives. When the

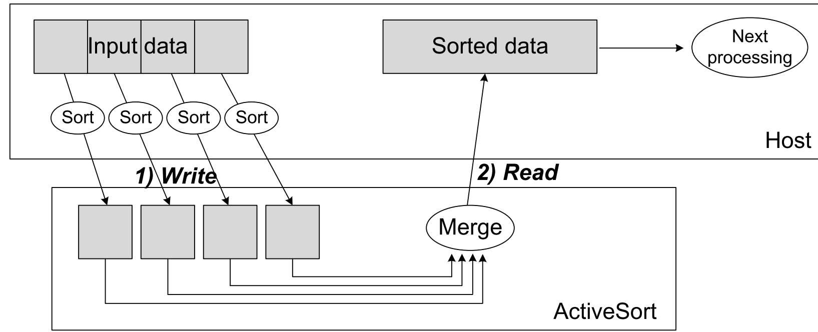


Fig. 4. The process of ActiveSort.

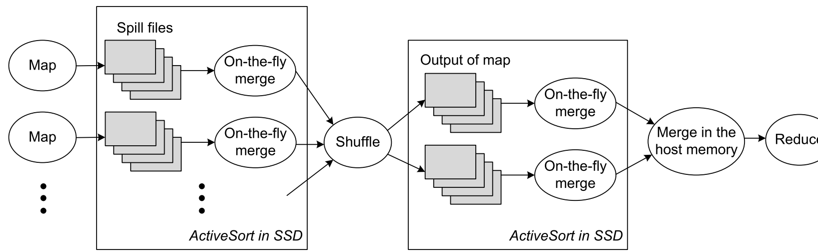


Fig. 5. The execution flow of Hadoop with ActiveSort.

request comes, ActiveSort merges data internally and responds to the request with the result. Compared to the original Hadoop, ActiveSort can reduce the amount of I/O to manipulate the sorted output by generating the output on-the-fly inside SSDs.

Applying ActiveSort to the reduce task is more complicated. In the original Hadoop, the output files of map tasks are read into the host memory for the reduce task. Usually, the reduce task generates the input data for the reduce function by merging the entries with the same key. However, if there are too many files, all the input data cannot be stored in the host memory. Thus, if the number of files to merge is large, Hadoop uses multi-pass merge sort which merges a number of files into a single large file in each pass. This multi-pass merge sort incurs massive I/Os because the intermediate data of each pass is stored in the disk.

ActiveSort can reduce the amount of I/Os efficiently when the reduce task uses the multi-pass merge sort. When the multi-pass merge sort requests the data to merge, ActiveSort produces the data by performing the merge operation internally. Since ActiveSort combines several pre-sorted files into fewer files inside the SSD, it decreases the number of files to merge gracefully. This not only reduces the number of steps in the multi-pass merge sort, but also reduces the amount of I/Os.

Algorithm 3 describes the simplified steps of map and reduce task with ActiveSort. Most of steps remain same as in the original Hadoop. ActiveSort replaces the merge operations and removes extra read/write operations in the map and reduce task (lines 11, 21). ActiveSort in the map task starts by the request from the shuffle phase in the reduce task (line 18). In the reduce task, ActiveSort is activated by the request from the reduce function (line 22) if it requires the multi-pass merge sort.

Algorithm 3 The steps of map and reduce task with ActiveSort

1: **procedure** MapTask(*split*) ▷ *split* = input split of map task
2: $size_s \leftarrow$ size of a spill buffer;
3: $i \leftarrow 0$;
4: **while** $B \leftarrow read_HDFS(split, size_s) \neq EOF$ **do**
5: $B' \leftarrow map(B)$;
6: $S_i \leftarrow sort(B')$;
7: $write(S_i)$;
8: $i \leftarrow i + 1$;
9: **end while**
10: **if** access to the output **then**
11: $M \leftarrow run_ActiveSort(S_0, S_1, \cdots, S_{i-1})$;
12: $send(M)$; ▷ send the result M to the reduce task
13: **end if**
14: **end procedure**
15: **procedure** ReduceTask( )
16: $j \leftarrow 0$;
17: **for all** *map tasks* **do**
18: $M_j \leftarrow get_result()$; ▷ shuffle, activate ActiveSort in the map task
19: $j \leftarrow j + 1$;
20: **end for**
21: $R \leftarrow merge(M_0, M_1, \cdots, M_{j-1})$; ▷ use ActiveSort
22: $R' \leftarrow reduce(R)$;
23: $write_HDFS(R')$;
24: **end procedure**

3.2.2. Interface between ActiveSort and Hadoop

To enable the on-the-fly data merge inside SSDs, ActiveSort needs to know the information about the metadata such as key length, key type, record length, total sizes and locations of the input/output files. The identifier of map/reduce tasks for input/output files are also required since ActiveSort needs to distinguish them to execute multiple merge operations. These information can be transferred to the SSD by defining a new interface [10] or by using the object-based interface [15]. With the object-based interface, ActiveSort can acquire the necessary information easily because the object-based interface allows to describe objects (or files) with one or more *attributes*. For this, the SSD requires additional modules to parse command, store objects, and index objects, which entails a considerable amount of engineering effort.

Instead, we used a predefined LBA (logical block address) to transfer such information. Before running the data merge operation, Hadoop writes the required information to the designated LBA and ActiveSort extracts it from the contents of the LBA. It is difficult to transfer the physical locations of a file with one simple write command since the amount of information can be enormous. Therefore, we located each input/output file at the fixed locations in the file system for fast prototyping. The detailed mechanism is described in Section 4.2.3.

3.2.3. Other design issues

When integrating ActiveSort with Hadoop, one important consideration is to guarantee that the input data is written completely to the SSD before executing the data merge in ActiveSort. Since the data merge is performed inside of the SSD, the entire input data must be available before the data merge operation is started. This requires performing an `fsync()` operation on intermediate files whenever we initiate the ActiveSort mechanism. However, while running the Hadoop application using large data sets, we have observed that ActiveSort spends very short time in flushing the data to the SSD. This is because the dirty data in the page cache is frequently written back into the disk due to the limited memory size. In most cases, the input data was already available in the disk before starting the on-the-fly data merge.

Another consideration is the record size. As the record size becomes smaller, the performance of ActiveSort gets worse due to the computation overhead (such as key comparison) during the data merge operation. In practice, the computation overhead plays an important role in the overall performance, since the computing power of the embedded CPU and the memory bandwidth in the SSD is limited. For example, when the record size is 128 bytes, the throughput of ActiveSort is reduced by 84.9% in our prototype SSD compared to the case when using the record size of 32 kB (cf. Section 5.2).

To overcome this limitation, we apply ActiveSort to Hadoop selectively depending on the record size. In the map task, ActiveSort is enabled if the record size of its input split is large. In the reduce task, only the part of intermediate data whose record size exceeds a certain threshold is merged by ActiveSort. For the data with the small record size, Hadoop uses the original mechanism to merge the records in the host.

It is not obvious to adopt ActiveSort in the Hadoop environment where the data is encrypted or compressed. In this case, ActiveSort cannot obtain the key data to use during merge operation. One simple solution is to apply encryption or compression only to the value data. Hadoop already provides this mechanism in the `SequenceFile` format to store binary key/value data. Another approach is to use the order-preserving encryption [16] for the records, as ActiveSort requires just the ordering information among the keys, not the contents.

ActiveSort does not require any modification for the fault tolerance in Hadoop because ActiveSort only manipulates the intermediate data in the nodes, which is locally stored and can be generated again upon the fault. If a node running a map task has a failure, the map task can be restarted on a different node and generates the spill files to merge again in the relocated node. Similarly, a failed reduce task will be also relocated to other node and initialize ActiveSort by collecting the results of map tasks again before starting the merge operation.

4. Implementation

In this section, we present the implementation details of ActiveSort including the mechanism to exploit the internal parallelism in SSDs. We also describe the changes in the Hadoop framework required to use ActiveSort. ActiveSort can be integrated easily into Hadoop because ActiveSort just replaces the external sorting phase without altering the existing execution flow of Hadoop. However, ActiveSort requires an interface to deliver the data-specific metadata information to the SSD to enable the on-the-fly data merge inside the SSD.

Fig. 6 depicts the overall architecture of the ActiveSort-enabled Hadoop framework. The framework consists of ActiveSort front-end module, ActiveSort file manager, and ActiveSort itself. The ActiveSort front-end module sends the information such as key size, key type, record length, and size of input/output data to the SSD, which are required to perform ActiveSort. It also creates an interface file which is the access point to get the sorted results in Hadoop. The interface file is allocated logically, but the contents are served by the on-the-fly data merge process when Hadoop requests the sorted results. The ActiveSort file manager in the file system handles the block locations of input and output files, and notifies ActiveSort of the information.

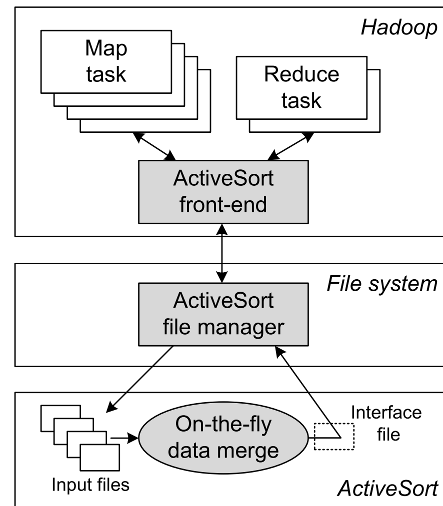


Fig. 6. The overall architecture of the Hadoop framework based on ActiveSort.

4.1. ActiveSort

4.1.1. Prototype SSD

To implement ActiveSort, we have developed a prototype SSD based on the Jasmine OpenSSD platform [17]. It consists of an 87.5 MHz ARM7TDMI embedded CPU, 64 MB DRAM, and four 32 GB NAND flash memory modules with each connected to a different flash channel. The NAND flash memory module is configured in such a way that the size of a flash page is 32 kB and an erase block occupies 4 MB. The prototype SSD is connected to the host via the SATA2 interface.

The Jasmine platform is an open-source SSD platform which enables us to customize its firmware. We have implemented a page-mapped FTL [18] to service normal read/write requests. The proposed on-the-fly data merge scheme is implemented in the read path of the FTL.

4.1.2. On-the-fly data merge

The on-the-fly data merge is activated by read requests for the sorted output from the host. Initially, ActiveSort arranges a chunk buffer in the internal DRAM for each partially sorted chunk and prefetches several records from each chunk to the associated chunk buffer. Then, ActiveSort compares the keys in the chunk buffers which come from different chunks. The record with the minimum key value is copied to the output buffer that handles the read request. These steps are repeated until the output buffer is filled, at which point the read result is returned to the host. After processing one flash page in a chunk buffer, ActiveSort tries to prefetch subsequent records from the NAND flash memory to the chunk buffer.

The overhead of comparing keys and the additional memory copy between the chunk buffer and the output buffer is critical to the performance of ActiveSort because they should be performed in the read path. Fortunately, the overhead can be reduced by overlapping the key comparison and memory copy operation with reading the data from NAND flash memory to chunk buffers. This is possible because SSDs have embedded CPUs for running the

FTL while another flash memory controller dedicated to each flash channel transfers data between DRAM buffers and NAND flash memory. In practice, ActiveSort allocates more than one flash page for a chunk buffer to maximize the overlap between computation on CPUs and data transfer by flash memory controllers.

4.1.3. Exploiting multi-channel parallelism

We can utilize the internal parallelism inherent in SSDs by activating multiple flash memory chips simultaneously to maximize the performance of ActiveSort. While writing the partially sorted chunks in ActiveSort, it is easy to increase the parallelism by interleaving the data across the available flash memory channels since each chunk is written sequentially. However, a more sophisticated mechanism is required to read the data as there can be a severe channel conflict during the data merge operation for the following reasons; (1) Although the records in a single chunk are interleaved across flash memory channels, one or more chunks can start from the same channel. In this case, reading the first record from those chunks should be serialized. (2) Even if the chunks are fully interleaved, the channel conflict can occur if the key value is skewed. Thus, when reading data from NAND flash memory, it is highly likely that the target channel is busy servicing the previously issued request from the other chunk buffer.

ActiveSort tries to maximize the utilization of channels by avoiding the waiting time for the completion of previous requests. When one flash page in a chunk buffer is copied to the output buffer entirely, ActiveSort prepares to issue the next read request. At this point, ActiveSort checks the state of the target channel for the next read request. Since ActiveSort already knows the location of each chunk, it is possible to check whether the target channel is busy or not before issuing the read request. If the target channel is busy, ActiveSort just skips issuing the request without waiting for the completion of the current operation in progress. And then, ActiveSort finds other pages to read whose target channel is idle.

Fig. 7 illustrates an example, assuming there are four chunks interleaved across four channels. At some point during the data merge operation, chunk buffers have empty pages to fill if the speed of data processing is faster than that of data fetching from NAND flash memory. When ActiveSort tries to fill the last page of chunk b, b_8, we can see that the target channel 1 is already occupied by the read request for the page a_7. In this case, ActiveSort skips the processing of chunk b and moves to the other chunks, generating the read requests for a_8 (in channel 2), c_6 (in channel 3), and c_7 (in channel 4).

In the implementation of our prototype SSD, we allocated 8 flash pages for each chunk buffer. Since the number of channels is four, ActiveSort uses four pages for computation and another four pages for data transfer from NAND flash memory. It is useless to allocate more than 8 flash pages for each chunk buffer because we cannot issue more than four requests at once.

4.2. Hadoop with ActiveSort

4.2.1. Hadoop support in the prototype SSD

While running the Hadoop application, multiple reduce tasks simultaneously access the files containing the results of map tasks. Thus, ActiveSort needs to reserve multiple sets of chunk buffers to service concurrent on-the-fly data merge on multiple sets of input/output files. In the prototype SSD, ActiveSort requires only 32 flash pages (1 MB) to get one sorted result if the number of files to merge is four since the size of one chunk buffer is 8 flash pages. Although the space of DRAM is limited in the prototype SSD, ActiveSort can support a sufficient number of concurrent on-the-fly data merge operations. In addition, we can restrict the number of concurrent execution of data merge operations by limiting the

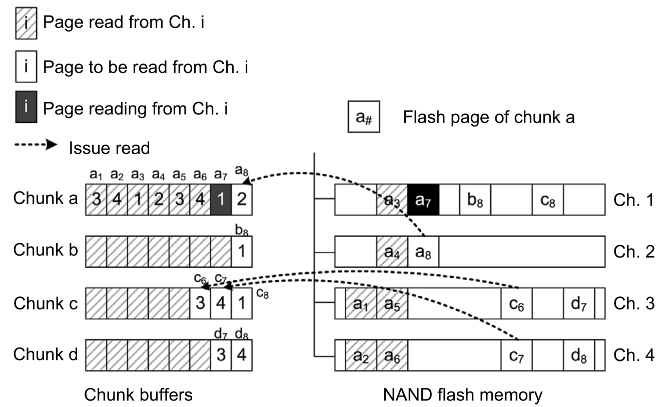


Fig. 7. The status of chunk buffers and the layout of NAND flash memory at one point during the data merge operation.

number of threads that fetch the map output by adjusting the configurable parameter in Hadoop.

We fixed the sizes of keys and records in ActiveSort for ease of prototyping, although Hadoop applications use variable-length keys and records. Inside the SSD, finding the actual location of the variable-sized record is not difficult because Hadoop leaves the length information in front of each record while writing the data. However, supporting the variable-sized records on a real SSD platform is not obvious for handling buffer management for the record spanning several flash pages, memory copy smaller than the unit of internal `memcpy()`, etc. We believe that we can support variable-sized records with low overhead if we implement ActiveSort on the latest commercial SSDs which have more powerful computing resources.

4.2.2. ActiveSort front-end in Hadoop

We have newly implemented the front-end module of ActiveSort which replaces the `Merger` class that performs merge operation in the original Hadoop. The front-end module sends the information needed by ActiveSort using a write operation on the predefined LBA as described in Section 3.2.2.

The front-end module also creates the interface file to access the sorted results in Hadoop. The interface file is created via the `fallocate()` system call which creates an inode and allocates data blocks in the file system without file contents. The file size of the interface file is set to the sum of input files to merge. When Hadoop accesses the interface file, the file system issues read requests to the disk and ActiveSort returns the data generated by the data merge operation in the SSD.

4.2.3. File manager for ActiveSort

We also developed the *File Manager* to manage the allocation of data blocks in the file system and notifies the prototype SSD of its locations. The File Manager uses the predefined location for the input and output files according to the file name and the index number of the map/reduce tasks used in Hadoop. ActiveSort extracts the input data and recognizes the access to the interface file based on the LBA specified by the File Manager.

We have modified the ext4 file system to manage the allocation of the files. When a file is created, the `ext4_create()` function allocates an inode in the ext4 file system. In this function, the File Manager checks the file name and identifies the type of the current file. The index number of map/reduce task is delivered via the `ioctl()` system call from the front-end module in Hadoop. Then, the File Manager sets the target location to allocate data blocks for the file at the predefined LBA. When the ext4 file system places the data blocks on the LBA space in the `ext4_ext_map_blocks()` function, it uses the target location to map the data blocks in the LBA specified by the File Manager.

5. Evaluation

5.1. Methodology

We conducted several experiments to evaluate ActiveSort. Throughout the experiments, we used Dell PowerEdge R420 servers equipped with two Quad-core Intel Xeon E5 2.4 GHz CPUs and 8 GB of memory, running Debian 7.0 with the Linux kernel 3.2.0. Each server has a Western Digital 500 GB HDD for its root file system and our prototype SSD based on the Jasmine OpenSSD platform. For Hadoop applications, we use the HDD as the storage for HDFS and the prototype SSD as a local disk for the intermediate files of map and reduce tasks. Both HDD and the prototype SSD are connected to the host via the SATA2 interface. For the experiments on multiple nodes, we use 8 servers which are connected via the Gigabit Ethernet. We also measured the amount of energy consumed by the server using the Yokogawa WT210 digital power meter. The power meter samples the power consumption in every 0.25 s and provides the sum of the power over the running time.

5.1.1. Sort benchmark

We use the well-known sort benchmark [19] to compare the performance of ActiveSort with that of two other sorting algorithms: Merge sort and Quick sort. Merge sort is implemented using the traditional external merge sorting scheme [13] and Quick sort is the famous in-memory sorting implementation provided by the Linux library. All files are stored in the ext4 file system on top of the prototype SSD. The prototype SSD acts like a normal SSD with the un-modified page-mapped FTL while running Merge sort and Quick sort.

For all experiments with the sort benchmark, we generate a 2 GB data set using the `gensort` program from the Sort Benchmark contest site [19]. Each record is set to 4 kB in size including a 10-byte key generated randomly. The input data is divided into four chunks by the Partial sorting phase in ActiveSort and Merge sort. We executed the benchmark with and without the `DIRECT_IO` flag, bypassing and using the kernel page cache, respectively. In order to evaluate the effect of the available memory size on the performance, the size of the main memory is varied from 1 to 8 GB using the Linux kernel boot option. Note that, when the memory is small, Quick sort generates a huge amount of swap traffic to sort data larger than the memory size. Thus, for this experiment only, we use a Samsung 840 Pro SSD as the swap device to mitigate the overhead of swap.

5.1.2. Hadoop benchmark

We use Hadoop applications to evaluate ActiveSort-enabled Hadoop on the real machines with the prototype SSD implementing ActiveSort. ActiveSort has a benefit when Hadoop generates a large amount of intermediate files since ActiveSort can reduce the amount of I/Os to access those files. Although the number of intermediate files can be minimized by tuning Hadoop parameters, it is difficult to eliminate all the files if the data set size is larger than the available memory size.

We use Hadoop version 1.1.2 with the configuration parameters shown in Table 1. The block size of HDFS is set to 500 MB. Usually a map task uses one HDFS block as its input split. The spill buffer size (`io.sort.mb`) is set to 141 MB and the threshold to flush the spill buffer to make a spill file (`io.sort.spill.percent`) is set to 0.9. As a result, each map task handles 500 MB of input data and generates four spill files as the partially sorted chunks during the external sorting phase. The number of files to merge (`io.sort.factor`) is set to 10 to perform multi-pass merge sort in the reduce task.

The maximum number of simultaneous map and reduce tasks is set to four so that both map and reduce tasks can run concurrently on the evaluation platform which has 8 CPU cores. The heap size of the map and reduce task is set to 1 GB to reserve sufficient memory space for each task. The number of threads for the HTTP service is set to three to limit the number of threads to fetch the output of map task simultaneously as described in Section 4.2.1. For ease of analysis, we turn off the Hadoop functionalities such as replication, speculative execution, and compression on HDFS and intermediate data, which are not related to the proposed scheme.

Table 1
The configuration parameters of Hadoop.

Parameters	Value
io.sort.mb	141
io.sort.spill.percent	0.9
io.sort.factor	10
tasktracker.http.threads	3
mapred.tasktracker.map.tasks.maximum	4
mapred.tasktracker.reduce.tasks.maximum	4
mapred.map.tasks.speculative.execution	false
mapred.reduce.tasks.speculative.execution	false
mapred.child.java.opts	-Xmx1024m
dfs.replication	1
dfs.block.size	500 MB

Table 2
The characteristics of HiBench workloads.

Workload	Key size (bytes)	Avg. record size (bytes)	# of records	Total input size (MB)	Total output size (MB)
Sort	10	4096	1,958,864	7697	7,697
TeraSort	10	4096	1,958,864	7641	7,641
Hive-join	10	1645.8	4,840,000	7623	11,777
Nutch	10	2185.5	1,450,000	7632	2,841

We use the HiBench-2.2 benchmark suite [20] to evaluate several workloads on the Hadoop environment. Among the workloads, we select four representative workloads which generate a large amount of intermediate data: Sort, TeraSort, Hive-join, and Nutch. Table 2 summarizes the characteristics of HiBench workloads used in this paper.

Both Sort and TeraSort workloads sort the input data. The difference is that TeraSort partitions the input data using a trie-based data structure before running the map function [21]. Therefore, TeraSort can generate the fully sorted data without merging a number of results from multiple reduce tasks. The key size of Sort and TeraSort workloads are 10 bytes and the record size is 4 kB.

Hive-join executes the sort-join algorithm for two tables using the Hive library [22]. We create one large table with 10-byte keys and 4-kB records. The other table is small with 10-byte keys and variable-sized records whose average size is 143 bytes. For Hive-join, only the map tasks for the large table use ActiveSort. The other map tasks for the small table does not make any intermediate files due to the small size of the input split.

Finally, Nutch indexes web pages and tests the indexing subsystem in *Nutch*, a popular open source search engine [23]. The input data created by the generation tool in HiBench consists of web pages, the status of crawling, hyperlink, and the metadata of web pages. In Nutch, we set the size of hashed URL to 10 bytes as the key and the size of a web page to 4 kB. The map tasks which handle the text of web pages utilize ActiveSort. The other map tasks use the `Merger` class in the original Hadoop because the record size is small.

While running the HiBench workloads, we perform the experiments with three different configurations to investigate the effect of ActiveSort in map and reduce tasks. The *Original* configuration is the traditional Hadoop without ActiveSort. The intermediate files are stored in the prototype SSD using an un-modified page-mapping FTL. Next, *ActiveSort+map* is the

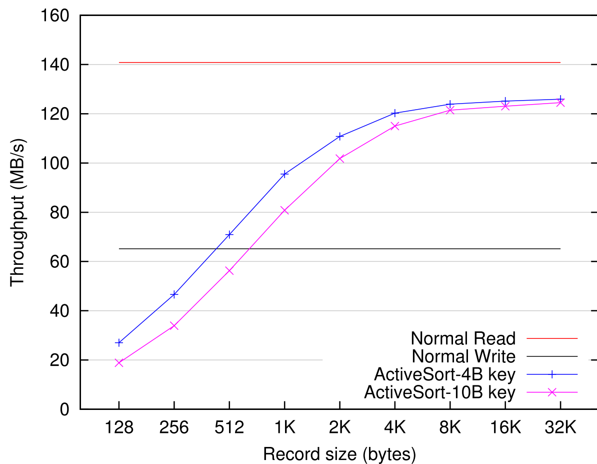


Fig. 8. The throughput of ActiveSort varying record size.

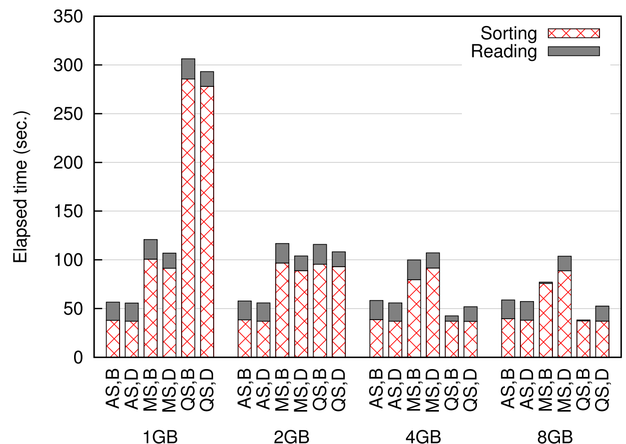


Fig. 9. The results of sort benchmark varying the memory size. *AS* is ActiveSort, *MS* is Merge sort, and *QS* is Quick sort. *B* is buffered mode using the page cache and *D* is `DIRECT_IO` mode to access data.

configuration where ActiveSort is enabled only in the map tasks while the reduce tasks use the original merge sort. Finally, in the *ActiveSort+all* configuration, both map and reduce tasks use ActiveSort to get the sorted results.

5.2. Base performance of ActiveSort

Fig. 8 shows the bandwidth of read operations to get the sorted data using the data merge operation inside the SSD. We measured the results while running the sort benchmark with `DIRECT_IO` flag to access files. We change the key size (4 bytes or 10 bytes) and the record size in the sort benchmark to evaluate the performance of ActiveSort with various configurations. The minimum record size is 128 bytes which is the minimum size of `memcpy()` operation that the Jasmine platform supports. The maximum record size is 32 kB which is the flash page size of the platform. We also present the raw bandwidth of the normal read and write operations in Fig. 8 for comparison.

When we use 32 kB record and 4-byte key, the bandwidth of read with data merge achieves 125.9 MB/s which is slightly lower than the normal read bandwidth (140.8 MB/s) and almost two times higher than the normal write bandwidth (65.1 MB/s). Although the implementation of ActiveSort requires more optimizations, the performance drop from the raw read bandwidth is low because of exploiting multi-channel parallelism and overlapping computation with the access to NAND flash memory. However, as the record size is reduced and the key size is increased, the overhead of computation affects the performance significantly since it takes more time to process the on-the-fly data merge. Thus, it is required to enable ActiveSort selectively based on the record size as mentioned in Section 3.2.3.

5.3. Results of the sort benchmark

Fig. 9 compares the elapsed time of each sorting algorithm with various memory sizes while running the sort benchmark. We break down the elapsed time to *Sorting* and *Reading*. *Sorting* includes the time for generating the input data, performing sorting, and writing the sorted output. Since the data merge occurs when reading the output in ActiveSort, we also include the time for reading the final output after the sorting is completed, which is denoted as *Reading*.

ActiveSort shows a steady performance independent of the available memory size. Interestingly, ActiveSort achieves the best performance when the memory size is smaller than the data set size. Compared to Merge sort, ActiveSort takes slightly longer time to read the sorted result since the result is produced on-the-fly. However, ActiveSort outperforms Merge sort by reducing the amount of I/O needed to merge intermediate results. As a result, ActiveSort reduces the elapsed time by 53.1% compared to Merge sort when the memory size is 1 GB.

Table 3
The amount of I/O of ActiveSort and Merge sort with 1 GB of memory and buffered mode to access data.

Method	Read (MB)	Write (MB)
ActiveSort	2048.3	2048.4
Merge sort	4085.9	4096.7

When the available memory is sufficient while performing Merge sort and Quick sort, the elapsed time for reading results is very short under the buffered mode due to the page cache. On the other hand, ActiveSort cannot take the advantage of the page cache since the sorted results are generated within the SSD when processing the read request. Nevertheless, ActiveSort has better performance than Merge sort even when the memory is sufficient. This is because Merge sort still needs to produce a large amount of writes to store intermediate data and final output even though the read request is removed by the page cache. ActiveSort does not store final output, and the sorted result is generated by reading from NAND flash memory and merging the results internally which is still faster than the write bandwidth of the disk.

In Fig. 9, we can see that Quick sort suffers from excessive page swapping when the memory is not enough to contain the input data. Many I/O requests from the page swapping degrades the performance of the benchmark. With 1 GB of memory, Quick sort swaps out 3.5 GB and swaps in 514 MB while ActiveSort and Merge sort do not swap at all.

Table 3 shows the amount of I/O handled by the prototype SSD while running ActiveSort and Merge sort with 1 GB of memory using the buffered mode to access data. ActiveSort transfers the same amount of read and write data, which are equal to the size of the data set. In contrast, Merge sort generates larger amounts of read and write operations than ActiveSort since the Merge phase transfers data between the main memory and the disk. Table 3 implies that ActiveSort can almost double the lifetime of SSDs by reducing the amount of writes by 50%.

5.4. Results of the Hadoop benchmark

5.4.1. HiBench on single node

Fig. 10 illustrates the elapsed times to execute HiBench workloads including the reading of input data from HDFS, the running of map/reduce tasks, and the writing of result data to

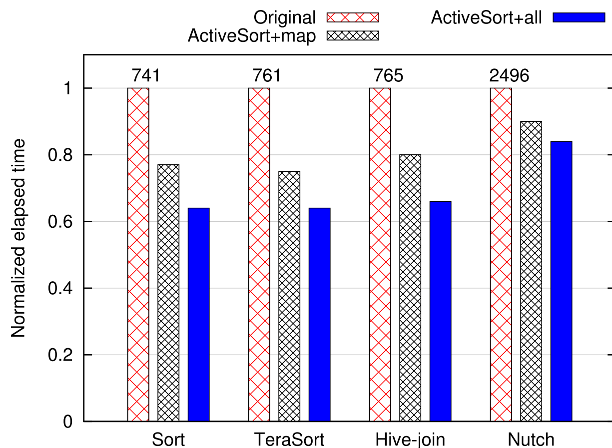


Fig. 10. The elapsed time of HiBench workloads. The results are normalized to that of the original Hadoop. The total elapsed time of the original Hadoop is shown at the top of each graph in second.

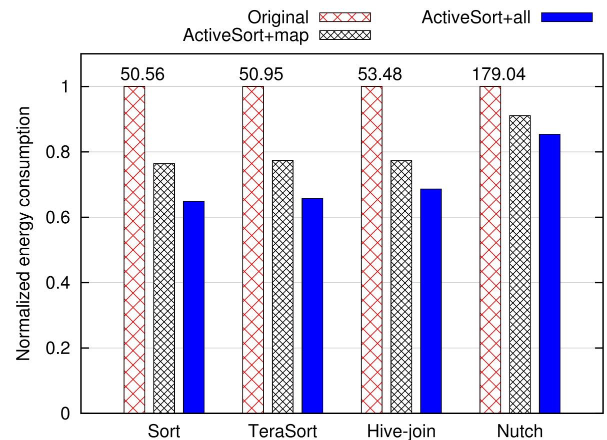


Fig. 12. The energy consumption results of HiBench workloads. The results are normalized to the result of the original Hadoop. The energy consumption under the original Hadoop is shown at the top of each graph in Kilojoules.

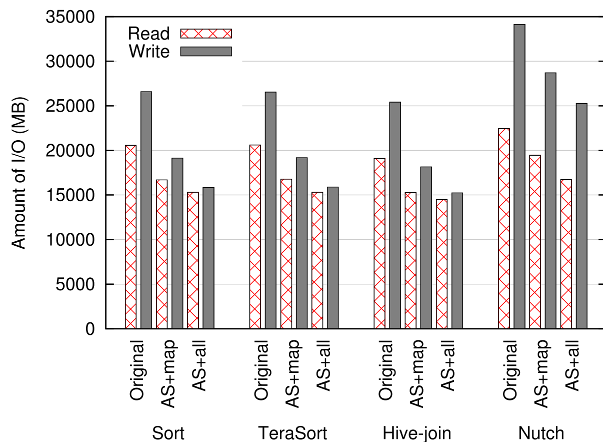


Fig. 11. The amount of I/O of HiBench workloads. *AS* is the abbreviation of ActiveSort.

HDFS. The results are normalized to the result of the Hadoop without ActiveSort for each workload.

For all workloads, ActiveSort improves the performance of Hadoop applications. Compared to the original Hadoop, ActiveSort reduces the elapsed time by up to 36.1%. While ActiveSort decreases the elapsed time of the sort benchmark by up to 50% (cf. Section 5.3), the improvement in the Hadoop applications is slightly lower than this, mainly due to the time to get and put the data from/to HDFS.

The performance improvement using ActiveSort in the reduce task is smaller than that in the map task. In the reduce task, the amount of intermediate data for the merge operation is less than that of the map task. For example, in the Sort workload, the reduce task generates 3.4 GB of the intermediate data whereas the map task does 8 GB. Thus, the amount of I/O decreased by ActiveSort in the reduce task becomes smaller than that in the map task.

For the Nutch workload, the performance improvement is smaller than the other cases. Nutch has a very complex reduce function and it spends long time to process the reduce task. Thus, the advantage of ActiveSort has a less impact on the overall performance compared to the other workloads. With the ActiveSort + all configuration, Nutch takes 2095 s while Sort takes 474 s.

Fig. 11 depicts the amount of I/O received by the prototype SSD while running the workloads. As seen in the figure, ActiveSort can reduce the amount of I/Os using the on-the-fly data merge inside the SSD. The page cache fails to absorb read/write operations for the data because the data size is larger than the available memory size. As a result, ActiveSort can reduce the amount of writes by up to 40.4% compared to the original Hadoop. Accordingly, ActiveSort in Hadoop also extends the lifetime of SSDs by reducing the amount of write.

In Nutch, there is still a large amount of write operations compared to the other workloads due to the small record size. Regarding the amount of data, only 74.6% of the input data has the record size large enough to be processed by ActiveSort. For the remaining data, the `Merger` class in the original Hadoop has better performance than ActiveSort. However, the `Merger` class still requires a large amount of I/Os.

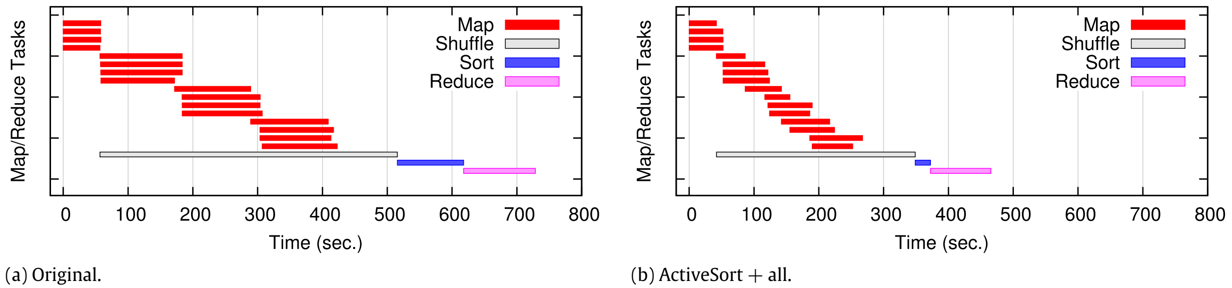
Fig. 12 shows the energy consumption while running the HiBench workloads on a single node. Since energy is a product of power and time, the trend of the energy consumption results is similar to the results of the elapsed time shown in Fig. 10. We can see that ActiveSort achieves energy saving by up to 35.2%. Eliminating extra data transfers by ActiveSort shortens the elapsed time of Hadoop applications, which also improves the overall energy consumption. In practice, ActiveSort requires more power on average than normal I/O because ActiveSort utilizes more computing resources in SSDs. However, the power overhead of ActiveSort is negligible, less than 1.8% in our experiments.

Fig. 13 illustrates the execution charts before and after ActiveSort is applied to the Sort workload. Each bar shows the start time and end time of the corresponding phase including Map, Shuffle, Sort, and Reduce while running the workload. In the comparison of two execution charts, the elapsed times of Map and Sort phase are gracefully reduced after adopting ActiveSort. Especially, the Sort phase is completed in a very short time during the reduce task since it does not include the time to merge data. The actual data merge occurs during the Reduce phase to produce the input data of the reduce function.

Fig. 14 depicts the execution charts of the Nutch workload. Similar to the Sort workload, ActiveSort reduces the elapsed times of the Map and Sort phases. However, the Reduce phase takes long time due to the complex reduce function. Thus, the benefit of ActiveSort is reduced compared to the other workloads.

5.4.2. *Sort workload varying the memory size*

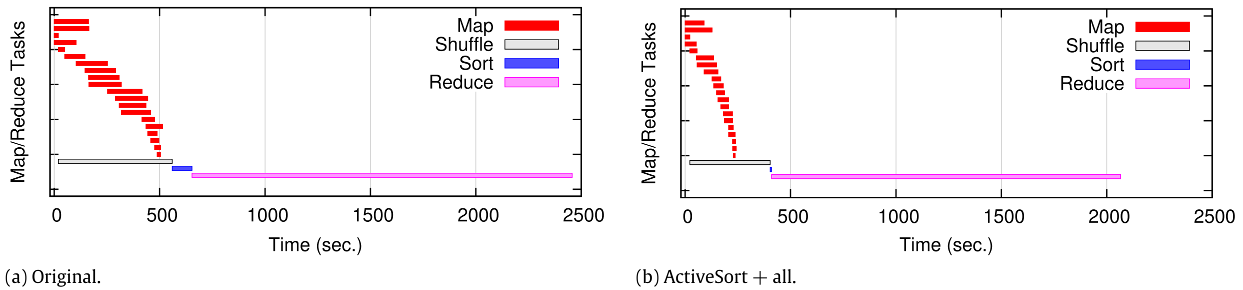
To examine the benefit of ActiveSort on various memory size, we run the Sort workload with 2 GB data set. The configuration is the same as described in Section 5.1.2 except for the total input size. When applying ActiveSort in Hadoop, only the map task utilizes ActiveSort to get the sorted results. In the reduce task, the merge operation does not incur any disk I/O on the local disk



(a) Original.

(b) ActiveSort + all.

Fig. 13. Execution charts of Sort workload.



(a) Original.

(b) ActiveSort + all.

Fig. 14. Execution charts of Nutch workload.

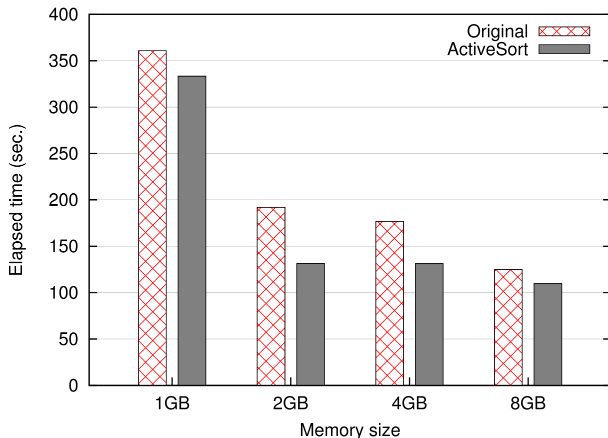


Fig. 15. The elapsed time of Sort workload with 2 GB data set, varying the memory size.

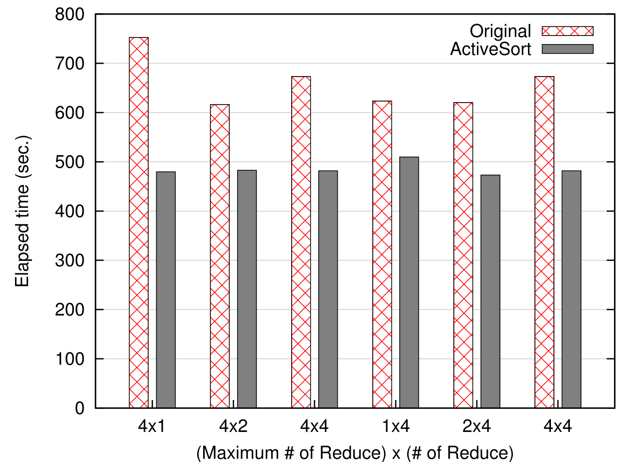


Fig. 16. The elapsed time of TeraSort workload varying the number of concurrent reduce tasks.

because the number of files to merge is small enough to execute the merge operation in the host memory.

Fig. 15 shows the elapsed time of the Sort workload varying the available memory size. In this experiment, ActiveSort also outperforms the original Hadoop and shows the steady performance when the memory size is larger than 2 GB. The elapsed time of the original Hadoop is comparable to that of ActiveSort with the sufficient memory size because the page cache can absorb the I/O requests for the intermediate data. However, as the available memory size is reduced, the performance of the original Hadoop drops significantly.

When the available memory size is 1 GB, the benchmark takes very long time to complete the experiment. After starting the Hadoop platform in the machine with 1 GB memory, only 550 MB of free memory remains since the Hadoop platform has a large memory footprint. The free memory is not sufficient to execute four map tasks concurrently, because each task requires 180 MB to store the spill buffer, Java code, and heap of Java program. Thus, the page swapping degrades the performance of the benchmark while running four map tasks.

5.4.3. TeraSort workload varying the number of reduce tasks

We investigate the impact of ActiveSort while changing the number of concurrent reduce tasks. If many reduce tasks can be executed concurrently, the performance of Hadoop applications can be improved due to the high utilization of CPUs. However, the increased consumption of memory and I/O resources disrupts the performance improvement. Thus, the results varying the number of reduce tasks are needed to estimate the performance and decide the configuration values of Hadoop.

Fig. 16 depicts the elapsed time of the TeraSort workload on a single node varying the maximum number of reduce tasks and the number of reduce tasks. ActiveSort has steady performance over all configurations. This is because the reduce task with ActiveSort becomes I/O-bound job by offloading the merge operation into the SSD. For the case of the original Hadoop, it shows the best performance when the number of concurrent reduce tasks is two due to the increased concurrency. However, many random I/Os incurred by multiple threads degrade the performance when the number of concurrent reduce tasks is four.

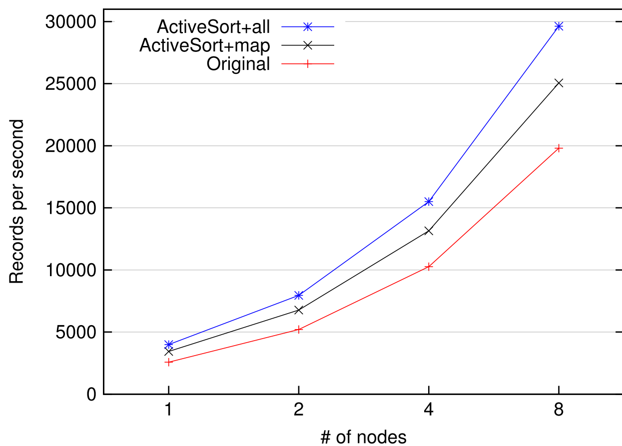


Fig. 17. Processed record per second of TeraSort workload on multiple nodes.

5.4.4. TeraSort workload on multiple nodes

We also perform the experiments using multiple nodes to examine the behavior of ActiveSort in accordance with the scalability of Hadoop. For this experiment, we use TeraSort with the increased data set size according to the number of nodes. For example, when we use 8 nodes, the size of the data set is increased to about 64 GB so that the input size per node remains the same as before. In this case, the total number of map and reduce tasks is also increased to 128 and 8, respectively. The other configuration parameters are the same as in Section 5.1.2. We use the number of processed records per second as a performance metric because the size of data set and the number of map/reduce tasks are increased according to the number of nodes.

Fig. 17 illustrates the results of the TeraSort workload on a various number of nodes. For all the number of nodes evaluated, the processed records per second using ActiveSort is much higher than that of the original Hadoop since ActiveSort improves the performance of Hadoop by reducing the amount of I/Os. In addition, ActiveSort supports the scalability of Hadoop well when using multiple number of nodes.

The detailed execution charts using 8 nodes are depicted in Fig. 18. ActiveSort is efficient to decrease the elapsed times of the Map and Sort phases even if the number of map/reduce tasks are large. As a result, the number of processed records per second in Hadoop is increased by 1.5 times in all experiments using multiple nodes.

6. Related work

For processing big data in the cloud, several approaches are proposed. Apache Spark [24] and Microsoft's Dryad [25] provide the big-data processing framework as Apache Hadoop [2]. NoSQL technology [26] introduces new methods to store data. There are also researches about building services [27] and performance profiling [28,29]. In this work, we focus on the I/O optimization of the MapReduce framework by using the concept of active SSDs.

The concept of the active disk has been extensively studied in the past. The research involves the offloading of data-processing functions to hard disk drives (HDDs) to improve performance [5,30–32]. As SSD has been emerged as an alternative storage device, several studies have been conducted to apply the concept of the active disk to SSD. Bae et al. [6] present the performance model of the active SSD which processes data for big data mining and analyze its performance benefits. Kim et al. [9] and Cho et al. [7] propose an active SSD architecture which executes several data processing functions not only on the embedded CPU(s) of the SSD controller but also on flash memory controllers. Kang et al. [8] also introduce an active SSD model including an interface to transfer a tasklet into the SSD, which is the unit of application task. In addition, they show the benefit of the proposed scheme using Hadoop and a real SATA-based SSD.

Tiwari et al. [10] have studied an approach to utilize active SSDs in the high performance computing (HPC) environment on large-scale supercomputers. They analyze the energy and performance models of active SSDs and discuss how to utilize multiple SSDs on supercomputers. They present that adopting active SSDs is a promising approach for improving both performance and energy efficiency.

The aforementioned studies focus on data processing functions, whose results are much smaller than the input data. In this case, the data transfer between the host system and SSDs is dramatically reduced, hence improving the performance of data processing is rather straightforward. In this paper, we propose a new mechanism to accelerate external sorting using active SSDs, that is one of the core data-processing algorithms in Hadoop. We have shown that even such applications as external sorting, whose output size is same as the input size, can benefit from active SSDs.

Many researchers have tried to improve the performance of sorting in NAND flash memory by exploiting the characteristic that read operations are faster than write operations. Park and Shim [33] utilize the max heap to retain the minimum values while processing the merge phase. Cossentine et al. [34] manage the minimum value of regions each of which is the split of input data to generate the sorted result. However, although these approaches reduce the amount of write, they require multiple reads to get the sorted result.

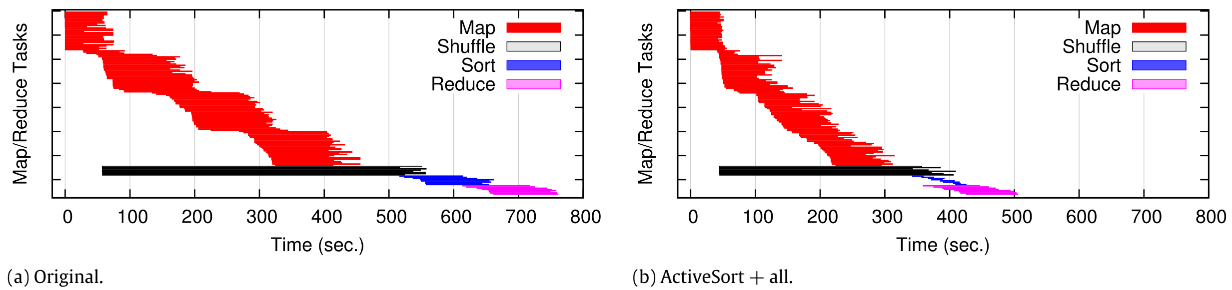
Cavazos et al. [35] build a B+-tree index of keys from the input data, and fetch the sorted result by traversing the keys in the leaf node of the B+-tree index. Wu and Huang [36] make a secondary data structure which contains only keys and extract the sorted result after sorting the secondary data structure. ActiveSort can also use these sorting schemes when making the sorted results on-the-fly.

Rasmussen et al. [37] propose an efficient MapReduce framework in the aspect of I/O. To avoid spilling data on the disk, they manage data in all the steps of MapReduce using careful memory management. They divide the input data into small size chunks to enable in-memory sorting during the reduce phase. On the other hand, ActiveSort does not require the massive change of Hadoop and careful tuning of configuration because ActiveSort just replaces the existing external sorting in Hadoop.

The main idea of ActiveSort is introduced in our previous work [38]. Compared to the previous work, we propose a new enhanced ActiveSort that further improves performance by exploiting multi-channel parallelism. We also implement a mechanism to integrate ActiveSort into the Hadoop framework, and present the benefits of ActiveSort in Hadoop with the extensive experiments.

7. Conclusion

We propose a novel mechanism called ActiveSort which performs external sorting efficiently using the concept of active SSDs. We also present a mechanism to enable the integration of ActiveSort in Hadoop, which utilizes external sorting as one of the core algorithms. By performing the merge operation on-the-fly, ActiveSort can eliminate extra data transfer during external sorting and improve the lifetime of the SSD as well as the performance. To reduce the overhead of ActiveSort, we exploit the parallelism of multi-channel architecture by extracting data from NAND flash memory based on the status of the channel. Several experimental results with real prototype SSDs show that



(a) Original. (b) ActiveSort + all.

Fig. 18. Execution charts of TeraSort workload with 8 nodes.

ActiveSort is a promising approach to improve the performance and energy consumption of the Hadoop application.

The current prototype implementation of ActiveSort has limitations such as the low performance on small-sized records and using fixed sizes for keys and records. These drawbacks come from the much inferior computing resources of our prototype SSD compared to the latest SSDs in aspects of embedded CPU power and the speed of DRAM access. We expect that the overhead of comparing keys and the additional memory copy can be reduced as the number of CPU cores and its processing power increase.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIP) (No. 2013R1A2A1A01016441).

References

[1] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, in: Proc. of the 6th Symp. on Operating Syst. Design and Implementation, OSDI, 2004.
[2] Apache hadoop. http://hadoop.apache.org/.
[3] S. Moon, J. Lee, Y.S. Kee, Introducing SSDs to the hadoop mapreduce framework, in: Proc. of the 7th IEEE Int. Conf. on Cloud Computing, CLOUD, 2014. http://dx.doi.org/10.1109/CLOUD.2014.45.
[4] K. Kambatla, Y. Chen, The truth about mapreduce performance on SSDs, in: Proc. of the 28th USENIX Conf. on Large Installation System Administration, LISA, 2014.
[5] A. Acharya, M. Uysal, J. Saltz, Active disks: Programming model, algorithms and evaluation, in: Proc. of the 8th Int. Conf. on Architectural Support for Programming Languages and Operating Syst., ASPLOS, 1998. http://dx.doi.org/10.1145/291069.291026.
[6] D.-H. Bae, J.-H. Kim, S.-W. Kim, H. Oh, C. Park, Intelligent SSD: A turbo for big data mining, in: Proc. of the 22nd ACM Int. Conf. on Inform. & Knowledge Manage., CIKM, 2013. http://dx.doi.org/10.1145/2505515.2507847.
[7] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, G.R. Ganger, Active disk meets flash: A case for intelligent SSDs, in: Proc. of the 27th ACM Int. Conf. on Supercomputing, ICS, 2013. http://dx.doi.org/10.1145/2464996.2465003.
[8] Y. Kang, Y.-S. Kee, E.L. Miller, C. Park, Enabling cost-effective data processing with smart SSD, in: Proc. of the 29th IEEE Symp. on Mass Storage Syst. and Technologies, MSST, 2013. http://dx.doi.org/10.1109/MSST.2013.6558444.
[9] S. Kim, H. Oh, C. Park, S. Cho, S.-W. Lee, Fast, energy efficient scan inside flash memory solid-state drives, in: Proc. of the 2nd Int. Workshop on Accelerating Data Manage. Syst. Using Modern Processor and Storage Architectures, ADMS, 2011.
[10] D. Tiwari, S. Boboila, S.S. Vazhkudai, Y. Kim, X. Ma, P.J. Desnoyers, Y. Solihin, Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines, in: Proc. of the 11th USENIX Conf. on File and Storage Technologies, FAST, 2013.
[11] G. Graefe, Implementing sorting in database systems, ACM Comput. Surv. 38 (3) (2006) 1–37. http://dx.doi.org/10.1145/1132960.1132964.
[12] Samsung SSD 850 pro. http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/ssd850pro/overview.html.
[13] D. Knuth, The Art of Computer Programming, second ed., vol. 3, Addison-Wesley, 1998.
[14] J. Williams, Algorithm 232: Heapsort, Commun. ACM 7 (6) (1964) 347–348.
[15] Y.-S. Lee, S.-H. Kim, J.-S. Kim, J. Lee, C. Park, S. Maeng, OSSD: A case for object-based solid state drives, in: Proc. of the 29th IEEE Symp. on Mass Storage Syst. and Technologies, MSST, 2013. http://dx.doi.org/10.1109/MSST.2013.6558448.
[16] R. Agrawal, J. Kiernan, R. Srikant, Y. Xu, Order preserving encryption for numeric data, in: Proc. of the ACM Int. Conf. on Management of Data, SIGMOD, 2004, pp. 563–574. http://dx.doi.org/10.1145/1007568.1007632.
[17] The openssd project. http://www.openssd-project.org.
[18] M. Wu, W. Zwaenepoel, eNVy: a non-volatile, main memory storage system, in: Proc. of the 6th Int. Conf. on Architectural Support for Programming Languages and Operating Syst., ASPLOS, 1994. http://dx.doi.org/10.1145/195473.195506.
[19] Sort benchmark. http://sortbenchmark.org/.
[20] S. Huang, J. Huang, J. Dai, T. Xie, B. Huang, The HiBench benchmark suite: Characterization of the mapreduce-based data analysis, in: Proc. of the 26th IEEE Int. Conf. on Data Engineering Workshops, ICDEW, 2010, pp. 41–51. http://dx.doi.org/10.1109/ICDEW.2010.5452747.
[21] O. O'Malley, Terabyte sort on apache hadoop, 2008. http://sortbenchmark.org/YahooHadoop.pdf.
[22] Apache hive. https://hive.apache.org.
[23] Apache nutch. http://nutch.apache.org.
[24] Apache spark. http://spark.apache.org.
[25] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, ACM SIGOPS Oper. Syst. Rev. 41 (3) (2007) 59–72. http://dx.doi.org/10.1145/1272998.1273005.
[26] J. Han, E. Haihong, G. Le, J. Du, Survey on nosql database, in: Proc. of the 6th Int. Conf. on Pervasive Computing and Applications, ICPCA, 2011. http://dx.doi.org/10.1109/ICPCA.2011.6106531.
[27] V. Chang, A cybernetics social cloud, J. Syst. Softw. (2015) http://dx.doi.org/10.1016/j.jss.2015.12.031.
[28] H. Herodotou, S. Babu, Profiling, what-if analysis, and cost-based optimization of mapreduce programs, Proc. VLDB Endow. 4 (11) (2011) 1111–1122.
[29] V. Chang, G. Wills, A model to compare cloud and non-cloud storage of big data, Future Gener. Comput. Syst. 57 (2016) 56–76. http://dx.doi.org/10.1016/j.future.2015.10.003.
[30] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G.R. Ganger, E. Riedel, A. Ailamaki, Diamond: A storage architecture for early discard in interactive search, in: Proc. of the 3rd USENIX Conf. on File and Storage Technologies, FAST, 2004.
[31] E. Riedel, C. Faloutsos, G.A. Gibson, D. Nagle, Active disks for large-scale data processing, Computer 34 (6) (2001) 68–74. http://dx.doi.org/10.1109/2.928624.
[32] E. Riedel, C. Faloutsos, D. Nagle, Active disk architecture for database, Tech. Rep. CMU-CS-00-145, Carnegie Mellon University, 2000.
[33] H. Park, K. Shim, FAST: Flash-aware external sorting for mobile database systems, J. Syst. Softw. 82 (8) (2009) 1298–1312. http://dx.doi.org/10.1016/j.jss.2009.02.028.
[34] T. Cossentine, R. Lawrence, Efficient external sorting on flash memory embedded devices, Int. J. Database Manag. Syst. 5(1). http://dx.doi.org/10.5121/ijdms.2013.5101.
[35] L.C. Quero, Y.-S. Lee, J.-S. Kim, Self-sorting SSD: Producing sorted data inside active SSDs, in: Proc. of the 31st IEEE Symp. on Mass Storage Syst. and Technologies, MSST, 2015.
[36] C.-H. Wu, K.-Y. Huang, Data sorting in flash memory, Trans. Storage 11 (2) (2015) 7:1–7:25. http://dx.doi.org/10.1145/2665067.
[37] A. Rasmussen, V.T. Lam, M. Conley, G. Porter, R. Kapoor, A. Vahdat, Themis: an i/o-efficient mapreduce, in: Proc. of the 3rd ACM Symp. on Cloud Computing, SoCC, 2012, pp. 13:1–13:14. http://dx.doi.org/10.1145/2391229.2391242.
[38] Y.-S. Lee, L.C. Quero, Y. Lee, J.-S. Kim, S. Maeng, Accelerating external sorting via on-the-fly data merge in active SSDs, in: Proc. of the 6th USENIX Conf. on Hot Topics in Storage and File Syst., HotStorage, 2014.

Young-Sik Lee received the B.S. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2006. He is currently working toward his Ph.D. degree in the School of Computing at KAIST. His research interests include flash memory, storage system, operating systems, and big data processing.

Luis Cavazos Quero received the B.S degree in Electronic and Computer Engineering from the Monterrey Institute of Technology (ITESM) in 2010, and the M.S. degree in Electrical and Computer Engineering at Sungkyunkwan University in 2015. He is pursuing a Ph.D. degree at Sungkyunkwan University. His research interests include the design and development of data processing algorithms and architectures tailored for flash storage systems.

Sang-Hoon Kim received the B.S. degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2002. He is currently a Ph.D. candidate in the School of Computing, KAIST. His research interests include memory systems, embedded systems, and operating systems.

Jin-Soo Kim received his B.S., M.S., and Ph.D. degrees in Computer Engineering from Seoul National University, Republic of Korea, in 1991, 1993, and 1999, respectively. He is currently a professor at Sungkyunkwan University. Before joining Sungkyunkwan University, he was an associate professor at Korea Advanced Institute of Science and Technology (KAIST) from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002 as a senior member of the research staff, and with the IBM T.J. Watson Research Center as an academic visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems.

Seungryoul Maeng received his B.S. degree in Electronics Engineering from Seoul National University (SNU), Republic of Korea, in 1977, and M.S. and Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), in 1979 and 1984, respectively. Since 1984, he has been a faculty member of the Computer Science Department at KAIST. His research interests include micro-architecture, parallel processing, cluster computing, and embedded systems.