Accelerating External Sorting via On-the-fly Data Merge in Active SSDs

Young-Sik Lee[†], Luis Cavazos Quero[§], Youngjae Lee[§], Jin-Soo Kim[§], and Seungryoul Maeng[†]
[†]*KAIST*, [§]*Sungkyunkwan University*

Abstract

The concept of active SSDs (solid state drives) has been introduced in order to cope with the demands required to process the ever-increasing volumes of data. In active SSDs, some of the data-processing tasks are offloaded to SSDs, freeing host system resources and improving overall performance of data analysis.

In this paper, we propose a novel active SSD architecture focused on improving the external sorting algorithm that is used extensively in data-intensive computing. By performing merge operations on-the-fly in active SSDs, our method can remove the extra data transfer and enhance the lifetime of SSDs. Our evaluation results on a real SSD platform indicate that the proposed scheme outperforms the traditional external sorting by up to 39%.

1 Introduction

We are witnessing the proliferation of solid state drives (SSDs) in various storage systems due to their high performance, low power consumption, small form factor, light weight, and shock resistance. In particular, SSDs are being widely adopted for data-intensive computing as the I/O performance becomes critical in processing the ever-increasing volumes of data.

The recent trend is to make SSDs play a more important role in data-intensive computing by revisiting the notion of active disks [6]. The so-called *active SSDs* offload data-processing functions (e.g., min/max, scan, count, histogram, etc.) to SSDs where the data is already stored [7, 8, 12, 13, 17]. The host system reads the results directly from active SSDs without excessive data transfer nor host-side computation, thereby improving performance and saving energy. As the computing capability of SSDs becomes more powerful, this approach will be more promising to enhance the efficiency and scalability of data-intensive computing.

In this paper, we propose a novel active SSD architecture which performs external sorting efficiently, leveraging the characteristics of SSDs. External sorting is one of the core data-processing algorithms in data-intensive computing because of its need to handle large-scale data using very limiting memory. For example, the MapReduce frameworks extensively utilize external sorting to generate intermediate and final outputs from map/reduce tasks [1, 9]. Also, external sorting is a key component for many query processing algorithms in database systems [10].

The key idea behind the proposed architecture is to let the host system avoid computing the final sorted output of external sorting, and storing it in SSDs. Instead, the final output is created in real time from the partially sorted data inside the active SSD, when the host system issues read requests to the result. In other words, the active SSD internally performs data merge operations on-the-fly and transfers the merged data to the host as a result of read requests. What makes this feasible is that (1) SSDs already exploit parallelism across multiple flash channels operating independently, and (2) reads can be done much faster than writes in SSDs.

The benefits of the propose scheme can be summarized as follows. First, it eliminates extra data transfer and host-side merge operations which are otherwise needed to compute and store the final output. This reduces the processing power and memory requirements of the host system. Second, the lifetime of SSDs is enhanced as the expensive writes are replaced with reads, not to mention that performance and energy efficiency are improved. Our experimental results with a prototype on a real SSD platform indicate that the proposed scheme outperforms the traditional external sorting by up to 39%.

2 Background

2.1 Solid State Drives (SSDs)

A typical SSD is composed of a single SSD controller, DRAM, and an array of NAND flash memory chips. Unlike hard disk drives, NAND flash memory has several unique characteristics. First, the previous data should be erased before another data is written into the same area. Second, write operations are much slower than read op-

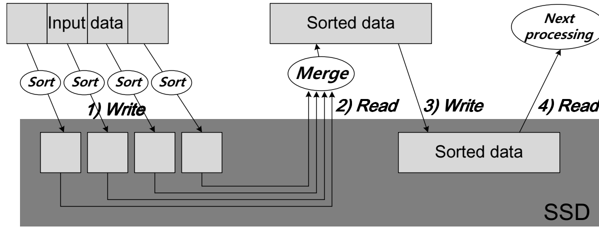

Figure 1: The process of the traditional external sorting.

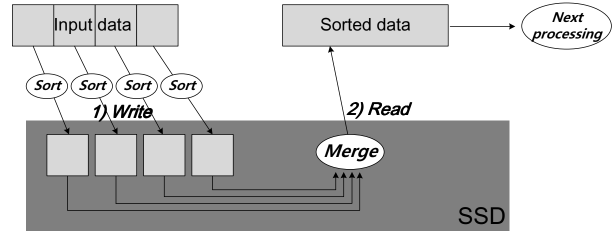

Figure 2: The process of ActiveSort.

erations. Finally, as flash memory cells wear out over time, there is a limitation in the number of erase operations that can be performed.

In order to deal with these characteristics of NAND flash memory, the SSD controller runs sophisticated firmware called *flash translation layer (FTL)*. FTL manages the physical storage space in a log-structured manner and performs *wear-leveling* to prolong the lifetime of SSDs. The functions of FTL are getting complex to match the increasing host interface speed and to exploit parallelism across a number of NAND flash chips connected to multiple channels. Accordingly, the hardware computing resources of SSDs become more powerful; Samsung 840 Evo, one of the latest high-end SSDs, features ARM-based triple cores operating at 400MHz and 1GB of DRAM with eight parallel NAND channels [4].

2.2 External Sorting

External sorting is a type of sorting algorithms to handle large-scale data efficiently which does not fit into memory. As illustrated in Figure 1, the traditional external sorting consists of two phases: *partial sorting* and *merge*.

During the partial sorting phase, the input data is divided into chunks where the size of a single chunk is smaller than the available memory. The data of each chunk is sorted via in-memory sorting algorithms and the sorted data of the chunk is written to the disk (Step 1). In the merge phase, the partially sorted chunks are read from the disk and merged to produce the final sorted output (Step 2). Some of read requests to the partially sorted chunks can be served by page cache. However, since the data size is much larger than the available memory size, the hit ratio of the page cache is very low. The final sorted output is written into the disk (Step 3), and usually sent to the other task later for further data processing (Step 4).

3 Design and Implementation

3.1 External Sorting with Active SSDs

We propose a new external sorting scheme called ActiveSort using on-the-fly data merge in active SSDs. In ActiveSort, the final sorted output is synthesized by merging data on-the-fly from multiple partially sorted chunks, when the host requires the final output. Since the final sorted output is not stored in active SSDs, ActiveSort can remove the read and write operations (Step 2 and 3 in Figure 1) which are needed to merge partially sorted chunks and to store the final output.

Figure 2 outlines the 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 skipped in ActiveSort until the host issues read requests to the final sorted output. When a read request to the final output arrives from the host, ActiveSort performs on-the-fly data merge inside active SSDs and transfers the result as the response of the read request (Step 2).

Compared to the traditional external sorting, ActiveSort requires only half of read and write operations to obtain the final sorted output. Since writes take longer than reads in SSDs, ActiveSort can achieve significant performance gain in spite of the run-time data merge operations during reads. Reducing the amount of writes is also helpful to enhance the lifetime of SSDs.

3.2 On-the-fly Data Merge

The on-the-fly data merge is activated by the read request for the final sorted output from the host. Initially, ActiveSort arranges a chunk buffer in DRAM for each partially sorted chunk and prefetches several records from each chunk to the associated chunk buffer. And then, ActiveSort compares the keys in chunk buffers which come from different chunks. The record with the minimum key value is copied to the output buffer for the read request. These steps are repeated until the output buffer is filled, at which point the read request is returned to the host. When the contents of a chunk buffer runs out, ActiveSort issues another prefetch request to the NAND flash memory to fill 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 are performed on-the-fly. Fortunately, the overhead can be reduced by overlapping key comparison and memory copy with reading the data from NAND

flash memory to chunk buffers. This is possible because SSDs have embedded CPUs for computation while data transfer between DRAM buffers and NAND flash memory is done by another flash memory controller dedicated to each NAND flash channel. In practice, ActiveSort allocates more than one chunk buffer for each partially sorted chunk to maximize the overlap between CPUs and flash memory controllers even when there is a skew in the key values.

3.3 Prototype Implementation

To evaluate the performance of ActiveSort, we implement a prototype SSD based on the OpenSSD platform [3]. The OpenSSD platform consists of 87.5MHz ARM7TDMI embedded CPU, 64MB DRAM, and four 32GB flash memory modules each connected to the different flash channel. The prototype SSD is connected to the host machine via the SATA2 interface. The on-the-fly data merge has been implemented by modifying a page-mapped FTL.

For fast prototyping, we statically fix the locations of partially sorted chunks and the final sorted output in the prototype SSD's address space and access them directly without passing file system layers. The other information such as key length and record length has been delivered to the prototype SSD using a special sector as in [17]. The contents of each partially sorted chunk are striped across four flash channels to maximize sequential performance.

4 Experiments

4.1 Evaluation Methodology

We compare the performance of ActiveSort with that of two other sorting algorithms: NSORT and QSORT. NSORT [2] is a representative external sorting library which records good performance in the Sort Benchmark contest [5]. QSORT is a famous quick sort implementation for in-memory sorting available in the Linux library.

For all the experiments, we use a Linux machine equipped with a 3.4GHz Intel Core i5 CPU and 16GB of memory running the Ubuntu 12.04. It also has a separate commercial Samsung 840 Pro SSD for its root file system and swap device. All of input, output, and intermediate results are stored in the prototype SSD when we run the three sorting algorithms. For NSORT and QSORT, the prototype SSD acts like a normal SSD with the unmodified page-mapped FTL. For input data, we generate a 2GB data set using the `gensort` program from the Sort Benchmark contest homepage. Each record is set to 4KB in size including a 10-byte key to focus on the I/O traffic. The original record size of the benchmark is 100 bytes.

Table 1: The base performance of the prototype SSD

Operation	Bandwidth (MB/s)
WRITE	61.04
READ	137.07
READ with merge (sorted)	116.38
READ with merge (random)	85.81

In order to evaluate the effect of the available memory on the performance, the main memory size is varied from 1GB to 3GB using the Linux kernel boot option.

4.2 Base Performance

Table 1 shows the bandwidth of sequential read and write operations of the prototype SSD. We have also measured the read bandwidth when we enable the on-the-fly data merge on the already sorted data set (sorted) and on the randomly generated data set (random).

When we fully utilize four NAND flash channels, the read bandwidth achieves 137MB/s which is almost two times higher than the write bandwidth. If we perform the on-the-fly data merge on the sorted data, the read bandwidth drops to 116MB/s due to the overhead of comparing keys and copying records. When the data set is already sorted, the partially sorted chunks are perfectly striped into four NAND channels and ActiveSort can retrieve data sequentially with the full bandwidth provided by four channels. On the other hand, the random data set hurts the interleaving efficiency since ActiveSort sometimes has to read two or more records from the same chunk. Although the on-the-fly data merge lowers the read bandwidth slightly, the resulting bandwidth is still higher than the write bandwidth. Since ActiveSort removes expensive writes during sorting, we can expect improved performance.

4.3 Sort Benchmarks

While running the three sorting algorithms, we measure the elapsed time to compare the performance. The elapsed time includes the time for reading the input data set, performing sorting, and writing the sorted output. Since ActiveSort does not write the final output, we also include the time for reading the final output after the sorting is completed.

Figure 3 compares the elapsed times for each sorting algorithm with various memory size. We can see that QSORT suffers from excessive page swapping when the memory is not enough to contain the input data. With 1GB of memory, the number of swapped-out and swapped-in pages in QSORT is 3.3x and 300x higher than that in ActiveSort, respectively.

ActiveSort shows the steady performance independent

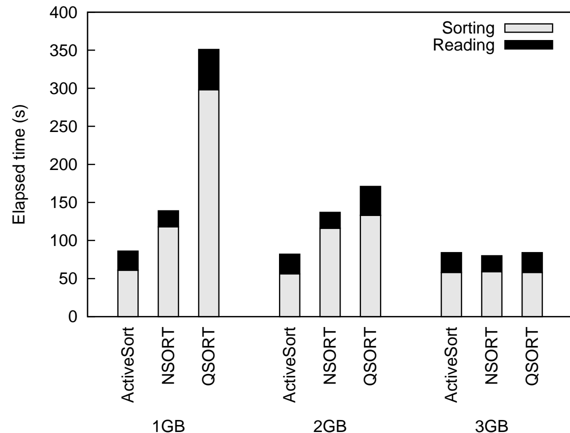

Figure 3: The elapsed times for each sorting algorithm with varying the memory size from 1GB to 3GB.

Table 2: The amount of I/O of ActiveSort and NSORT with 1GB of memory.

Sorting method	READ (MB)	WRITE (MB)
ActiveSort	4100	2048
NSORT	6157	4106

of the available memory size, especially achieving best performance when the memory size is smaller than the data set size. Compared to NSORT, ActiveSort takes slightly longer time to read the sorted result due to on-the-fly data merge, but overall it generates the final sorted result more quickly. This is because ActiveSort reduces the amount of I/O needed to merge intermediate results.

Table 2 shows the amount of I/O received by the prototype SSD while running ActiveSort and NSORT with 1GB of memory. ActiveSort just issues one read for the input data and one read/write for the partially sorted chunks. On the other hand, NSORT generates more reads and writes as the merge phase of NSORT requires data transfer between the main memory and the prototype SSD. From Table 2, it is apparent that ActiveSort can almost double the lifetime of SSDs.

Note that NSORT performs even better than QSORT when the memory size is 3GB. This is because NSORT switches to in-memory sorting when the memory is sufficiently large to maintain all of the input data. In all experiments, ActiveSort and NSORT have about 6% CPU usages because of I/O-bound processing.

5 Discussion

The current implementation of ActiveSort has several limitations as well as rooms for improvements. First, we note that the computing resource of the prototype SSD is much inferior to the latest SSDs described in Section 2.1 in aspects of embedded CPU power, DRAM size, and internal bandwidth. 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. The interleaving efficiency during on-the-fly data merge can be improved with the increased number of flash channels and the use of aggressive prefetching and more sophisticated buffer management techniques.

Second, the current implementation performs the partial sorting phase in the host as in the traditional external sorting. However, if the computing resource is sufficient, the partial sorting phase can be also offloaded into active SSDs further reducing the amount of I/O.

Another possible approach is to use multiple active SSDs to enhance the scalability of data-intensive computing. Similar to the scheme presented in [16], the scalability of external sorting can be improved by partitioning the input data into multiple active SSDs and then performing partial sorting completely in each active SSD. In this way, low-power microservers combined with multiple active SSDs can be a new energy-efficient vehicle for next-generation data-intensive computing.

To enable the on-the-fly data merge, the information on key length, key type, record length, sizes and locations of partially sorted chunks that constitute the final output should be available to active SSDs. Those information can be delivered to active SSDs by defining an additional interface [17] or more easily by using the object-based interface [14].

With the object-based interface, active SSDs can manage the locations of partially sorted chunks by itself. The other information to identify keys and records can be transferred through the object attributes. To support variable-length records, the record header can be used for obtaining the size of each record.

6 Related Work

The concept of the active disk has been extensively studied from the past. There are several researches to offload data-processing functions to hard disk drives (HDDs) for improving performance [6, 11, 15, 16].

As SSDs have emerged as an alternative storage device, several studies have been conducted to apply the concept of the active disk to SSDs. Bae et al. [7] have presented the performance model of the active SSD which performs data processing functions for big data mining and analyzed its performance benefits. Kim et al. [13] and Cho et al. [8] have proposed 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. A flash memory controller is a hardware logic of the SSD controller which is responsible for data transfer between

NAND flash memory and DRAM.

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, thereby improving the performance of data processing. In this paper, we propose an active SSD architecture to accelerate external sorting, which is one of the core data-processing algorithms. We have shown that even such applications as external sorting, whose output size is same as input size, can benefit from active SSDs.

Tiwari et al. [17] 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.

7 Conclusion

We propose an active SSD architecture to perform external sorting efficiently, which is one of the core algorithms in data-intensive computing. By performing merge operations on-the-fly inside of SSDs, the proposed scheme can eliminate extra data transfer and improve the lifetime of SSDs. The experimental results show that the active SSDs are a promising approach to enhance the performance in data-intensive computing.

As future work, we plan to integrate the proposed scheme into the Hadoop MapReduce framework. Since external sorting is extensively used in the Hadoop framework, we expect that the proposed scheme can improve the performance and energy efficiency of MapReduce applications.

8 Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIP) (No. 2013R1A2A1A01016441). This work was also supported by the IT R&D program of MKE/KEIT (No.10041244, SmartTV 2.0 Software Platform).

References

[1] Apache hadoop. `http://hadoop.apache.org/`.

[2] Nsort. `http://www.ordinal.com/`.

[3] The openssd project. `http://www.openssd-project.org`.

[4] Samsung SSD 840 evo. `http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/about/SSD840EVO.html`.

[5] Sort benchmark. `http://sortbenchmark.org/`.

[6] ACHARYA, A., UYSAL, M., AND SALTZ, J. Active disks: Programming model, algorithms and evaluation. In *Proc. ASPLOS* (1998).

[7] BAE, D.-H., KIM, J.-H., KIM, S.-W., OH, H., AND PARK, C. Intelligent SSD: A turbo for big data mining. In *Proc. CIKM* (2013).

[8] CHO, S., PARK, C., OH, H., KIM, S., YI, Y., AND GANGER, G. R. Active disk meets flash: A case for intelligent SSDs. In *Proc. ICS* (2013).

[9] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *Proc. OSDI* (2004).

[10] GRAEFE, G. Implementing sorting in database systems. *ACM Computing Surveys 38*, 3 (September 2006), 1–37.

[11] HUSTON, L., SUKTHANKAR, R., WICKREMESINGHE, R., M.SATYANARAYANAN, R.GANGER, G., RIEDEL, E., AND AILAMAKI, A. Diamond: A storage architecture for early discard in interactive search. In *Proc. FAST* (2004).

[12] KANG, Y., KEE, Y.-S., MILLER, E. L., AND PARK, C. Enabling cost-effective data processing with smart SSD. In *Proc. MSST* (2013).

[13] KIM, S., OH, H., PARK, C., CHO, S., AND LEE, S.-W. Fast, energy efficient scan inside flash memory solid-state drives. In *Proc. ADMS* (2011).

[14] LEE, Y.-S., KIM, S.-H., KIM, J.-S., LEE, J., PARK, C., AND MAENG, S. OSSD: A case for object-based solid state drives. In *Proc. MSST* (2013).

[15] RIEDEL, E., FALOUTSOS, C., GIBSON, G. A., AND NAGLE, D. Active disks for large-scale data processing. *Computer 34*, 6 (June 2001), 68–74.

[16] RIEDEL, E., FALOUTSOS, C., AND NAGLE, D. Active disk architecture for database. Tech. Rep. CMU-CS-00-145, Carnegie Mellon University, April 2000.

[17] TIWARI, D., BOBOILA, S., VAZHKUDAI, S. S., KIM, Y., MA, X., DESNOYERS, P. J., AND SOLIHIN, Y. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Proc. FAST* (2013).