# Self-Sorting SSD: Producing Sorted Data Inside Active SSDs

Luis Cavazos Quero
luis@skku.edu
College of Info. & Comm. Engineering
Sungkyunkwan University
Suwon, South Korea

Young-Sik Lee
yslee@calab.kaist.ac.kr
Computer Science Dept.
KAIST
Daejeon, South Korea

Jin-Soo Kim
jinsookim@skku.edu
College of Info. & Comm. Engineering
Sungkyunkwan University
Suwon, South Korea

***Abstract*—Nowadays solid state drives (SSDs) are gaining popularity and are replacing magnetic hard disk drives (HDDs) in enterprise storage systems. As a result, extracting the maximum performance from SSDs is becoming crucial to deal with the increasing storage volume and performance needs. Active disks were introduced as a way to offload data-processing tasks from the host into disks freeing system resources and achieving better performance.**

**In this work, we present an active SSD architecture called Self-Sorting SSD that targets to offload sorting operations which are commonly used in data-intensive and database environments and that require heavy data transfer. Processing sorting operations directly on the SSD reduces data transfer from/to the storage devices, increasing system performance and the lifetime of SSDs. Experiments on a real SSD platform reveal that our proposed architecture outperforms traditional external merge sort by up to 60.75%, reduces energy consumption by up to 58.86%, and eliminates all the data transfer overhead to compute sorted results.**

## I. Introduction

Solid State Drives (SSDs) have become a common device in computer systems due to superior characteristics that makes them more attractive when compared to traditional magnetic hard disk drives (HDDs). Some of the characteristics are: high sequential and random performance, low power consumption, small and lightweight form factor, and shock resistance. These advantages result in an increasing adoption in data-intensive computing and database systems where I/O performance is critical. However, the ever increasing volume and performance needs demand new methods to extract more performance out of SSDs.

The concept of active disks [1] was introduced in order to increase performance by using the hardware resources already available in disks. The main idea is simple: by processing data within the disk, expensive data transfer to the host can be avoided resulting in increased performance and lower power consumption. As processing capabilities of SSDs become more powerful, we believe that there are new opportunities to further improve performance by expanding the type of operations that can be performed within the disk. Sorting operations, which involve expensive I/Os, are used extensively in data intensive computing and database systems [2] making it a good candidate for our scheme.

978-1-4673-7619-8/15/$31.00 

In this paper we propose an active SSD architecture called Self-Sorting SSD which enables to completely offload sorting operations into an SSD. The basic idea is to build an index within the SSD. Once the index is built, sorted output is available as we fetch records by traversing the index. Since a full index is already available, it is also possible to serve other functions directly such as selection, range queries, min, max, etc. The index can be produced by two different mechanisms: The first one is called *sort-on-write* and it produces the index by parsing the input data when it is written to the SSD. The second mechanism is called *sort-on-command* and it generates the index on-demand by issuing a special command to the Self-Sorting SSD.

Having two indexing mechanisms adds flexibility in the ways Self-Sorting SSD can be employed. For example, recent NoSQL systems such as Couchbase Server supports several *views* on the stored documents using secondary indexes. Secondary indexes allow users to decide which fields within the documents they want to query at view definition time [3]. In this case, a primary index for new documents can be built at writing time by employing the sort-on-write mechanism in Self-Sorting SSDs. Later, when a secondary index is needed, it can be built on-demand by using the sort-on-command mechanism.

The main benefit of the proposed architecture is the total elimination of data transfer overhead to compute sorted results, which reduces the elapsed time and energy consumption, and improves the lifetime of SSDs. Our evaluation results using the Jasmine OpenSSD platform show that our scheme outperforms the traditional external merge sort scheme by up to 60.75% and ActiveSort[4] , a novel active SSD architecture that accelerates external sorting using on-the-fly merge by 27.35%. Additionally our scheme reduces up to 58.86% power consumption.

## II. Background

### A. SSDs

SSDs are composed of an array of NAND flash memory chips, DRAM, and an SSD controller. While the flash memory

chips serve as permanent storage, the available DRAM is used to maintain management data structures to service data from flash memory. The SSD controller orchestrates the exchange of data. Internally the SSD controller is composed of one or more embedded processors, a series of flash memory controllers which handle the low-level operations required to drive the flash memory chips, a DRAM controller, a host interface controller that serves as the interface with the host and implements one of the standard interface protocols such as SATA, SAS or PCIe. One of the principal functions of the controller is to present the SSD as block device to the host and hide some of the complexities of flash memory. These complexities include:

- Erase before write: Only previously erased pages can be written.
- Erase and write granularity: Erase operations are performed on batches of pages called blocks, while write operations are performed in single page granularity. Erase operations take longer than write operations to complete.
- Cell wear: Flash cells have a limited number of erase operations.

In order to hide these peculiarities, the SSD controller implements an abstraction layer called flash translation layer (FTL). The FTL separates the logical block device seen by the host from the physical flash media by translating logical block address (LBA) requests into physical page requests that are serviced internally by the SSD controller. This address translation information is stored inside the DRAM in the form of translation tables. The logical-physical separation not only helps hiding the complexities of flash memory but also enables a wide range of options to optimize SSDs performance without having to modify anything in the host.

### B. External Merge Sort

External sort refers to those algorithms in which the data to be sorted does not fit into the main memory of a system and external memory must be used to handle the sorting process. It is widely used in almost all large-scale sorting applications [5]. External merge sort is a specific algorithm that solves the problem by sorting large data sets in two or more phases. The first phase creates subsets of sorted data (commonly called runs) by using an in-memory sorting algorithm and writing the partial sorted results as temporal data stored on external memory. Once all runs have been processed the second phase takes place by merging each run into a single array of data. The merging process can be done in several passes; however, usually a one pass merge phase is preferred. The reason is that for every pass the corresponding data has to be read and written at least one time from/to disk, thus generating expensive I/O operations.

## III. Design and Implementation

### A. Overall Architecture of Self-Sorting SSD

We present Self-Sorting SSD which provides a new external sorting scheme by exploiting the internal hardware infrastructure of SSDs. The hardware architecture of Self-Sorting SSD is the common architecture found in any SSD as described in II-A. SSDs employ the FTL abstraction layer to serve data requests and hide the storage management tasks from the host. In order to process the sort operations internally the standard software architecture of the SSDs must be modified. In addition to the standard address translation algorithms found in SSDs, we have implemented and indexing algorithm based on the B+-tree data structure that will be used to produce sorted data. This data structure is maintained in the available DRAM of the drive. The B+-tree data structure is used to maintain all the index information. Once an index is built it is possible to use it to redirect requests and deliver the data in different ways other than how it was originally written.

### B. External Sorting with Self-Sorting SSD

From the host perspective the external sorting operation using Self-Sorting SSD has two phases. The first one is the *generate tree* phase which can be performed in two different ways depending on the application.

- *sort-on-write*: The generation of the sorted index is performed simultaneously with the write operation of the input data. As data is being written into the SSD, for each record, the key information is parsed and inserted into the B+-tree data structure.
- *sort-on-command*: The host explicitly sends a *generate* command to the SSD when needed. Once the command is received the SSD will internally read the unsorted input data, parse the key information, and generate the index by inserting the key into the B+-tree.

The second phase is the *active read* phase which involves reading the sorted data back to the host. From the host perspective this only requires generating read requests for the output data, which is nothing different from the traditional approach. Inside the SSD the sorted index previously generated using the B+-tree will be used to redirect the host requests to read the sorted output. The redirection method is explained in III-D.

### C. Building Indexes

In order to build an index for our Self-Sorting SSDs, we make use of the B+-tree data structure. The B+-tree is a structure commonly used to retrieve information. The information is stored in the form of key-record pairs, and the keys can be used to retrieve the record data. This kind of tree has two types of nodes: internal nodes which store keys but no record information and are used for indexing, and leaf nodes which are nodes at the bottom of the tree containing both key and record information. Leaf nodes are linked left-to-right which produce a sequential ordered set making it ideal for our scheme.

An important consideration is that the amount of information that can be stored by a node is variable, in other words a node can direct to multiple nodes and store multiple keys. The amount of information that a node can store is known as *fanout*. By selecting different fanout values the shape of the tree can be modified and tuned for specific purposes. In our implementation, we chose a fanout of 128 which produces

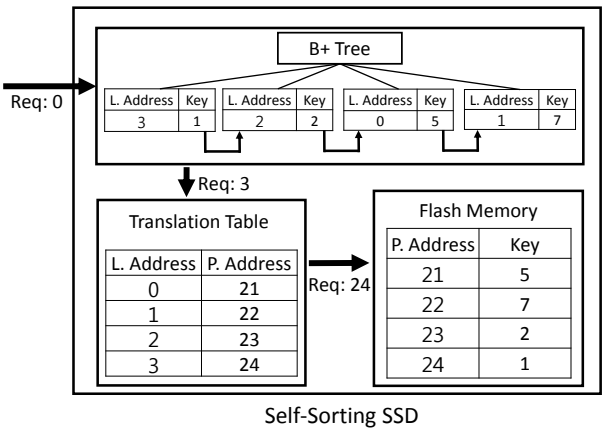


Fig. 1. An example of redirection process.

a node size of 2,060 bytes. This value was selected since it produces the most compact tree without compromising B+-tree insertion performance. Also the overhead of the node data structures is reduced as the fanout value increases. B+-trees are well suited for both random and sequential processing as described by Comer in [6].

The B+-tree data structure resides in the DRAM memory, however in order to guarantee durability it is possible to flush this information into non-volatile flash memory. Even on power shortages the index information can be retrieved in a similar fashion as in $\mu$-Tree [7] which employs a similar index structure tailored to the characteristics of flash memory.

### D. Reading Sorted Results from Self-Sorting SSD

The B+-tree data structure is used to redirect read requests from the host when sorted data is requested. The tree itself does not possess any logical-physical translation information, instead the leaf nodes only contain the logical address and the key of each record stored in flash. An example of the redirection process can be seen in Fig. 1. In this example we assume the index has been previously generated by any of the two mechanisms in the *generate tree* phase. The input data has been written to the active Self-Sorting SSD at logical address 0. The translation table shows the corresponding physical location starting from physical address 21, and the corresponding record is in flash memory. In the *active read* phase, the host issues a read request for the sorted data at the same location of the original data (logical address 0 (Req: 0)). The B+-tree handles the read request. Since the request is for the first record, the tree will take the first logical address on its left most node and change the logical address from the request to the one associated with the key (Req: 3). Then the physical address is obtained from the translation table (Req: 24) and the data is served to the host. The following requests are served by visiting the next entries in the B+-tree until all the requested data is served. We found this scheme to be beneficial for the following reasons:

- Any modification to the input records on flash (as long as the corresponding key is not modified) can be serviced by the standard FTL allowing high read and write performance.
- Using a redirection approach enables to maintain the original input data and the sorted data for the space footprint of only the original data.

### E. Self-Sorting SSD Interface

Before generating indexes the Self-Sorting SSD must have the format information of the data. Required information such as record size, key size, and key offset location within the record must be passed by the application on the host to the SSD to be used as indexing parameters. A simple approach in which required information is written by the host to a pre-defined LBA as a normal write request was implemented. When the firmware detects a request targeting the pre-defined LBA it uses the data in the request to configure the indexing parameters. Other functions such as enabling the *sort-on-write* scheme, the *generate* command required by the *sort-on-command* scheme, enabling reading the sorted results from the Self-Sorting SSD and generating secondary indexes are implemented in a similar fashion.

The implementation of the *generate* command required by the *sort-on-command* scheme over a write request has a special consideration. Generating long indexes might take a long time which can trigger a time-out on the host side while waiting for the SSD's response. This issue is handled by breaking down the indexing process in smaller chunks. By iterating the *generate* command, large indexes can be generated without timeouts.

### F. Prototype Implementation

In order to study the advantages of Self-Sorting SSD we have implemented our firmware in the Jasmine OpenSSD platform [8] which consists of an SSD controller integrated by a 87.5MHz ARM7TDMI CPU, 64 MB DRAM, and four 32 GB flash modules connected to its own channel. The flash chips have been configured in single plane mode such that the physical page size is 16KB wide. A page-level mapping FTL is used to handle the standard disk requests. Accesses to the SSD are performed using DIRECT IO to avoid page caching effects. Any required information such as record or key size can be communicated to the SSD as explained in III-E and similar to [9].

## IV. Experiments

### A. Evaluation Methodology

In this section we evaluate the performance of the four schemes described in section IV-C. Each sort scheme will involve sorting the same random input data. The experiments were performed in a 3.4GHz Intel Core i5 powered machine with 16GB of main memory running Ubuntu 12.04. Since external sort is used for data sets that are larger than the free memory size the amount of main memory available to the operating system has been reduced to 3GB using boot commands. The input data consists of 16KB long records that

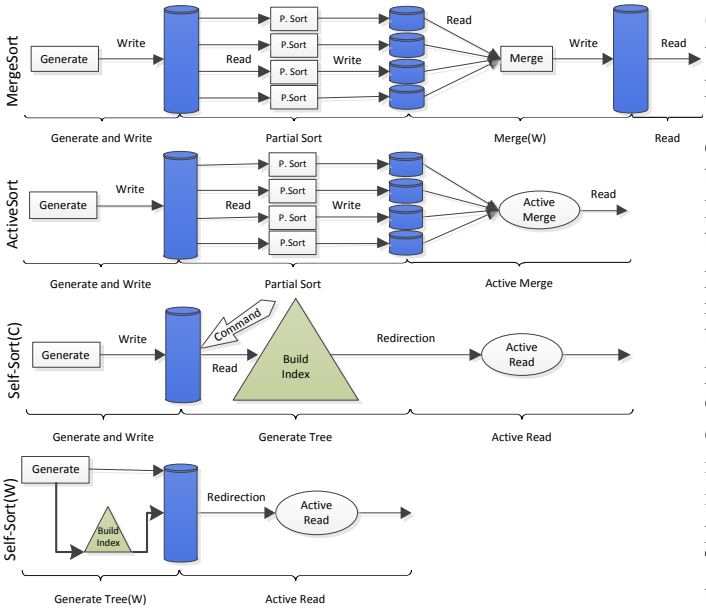


Fig. 2. Various sorting schemes and their phases. Each arrow arriving or leaving a cylinder represents an access to storage. The triangles represents the use of B+-tree for indexing inside of active SSDs.

include a 10-byte key. The total input data size is set to 8GB (524,288 records) due to the limited memory of the Jasmine OpenSSD platform. The standard read and write operations of the four sorting schemes are served by the same page-mapping FTL thus allowing us to compare the *active* operations of the ActiveSort and Self-Sorting SSD algorithms side by side.

### B. B+-Tree Performance

The fanout controls the shape of the B+Tree. Higher values create short and wide trees, while low values create tall and narrow trees. Trees of different shapes have different properties that affect the performance of the Self-Sorting SSD. The B+-Tree in the Self-Sorting SSD was evaluated using different fanout values. The results indicate that as the fanout value increases, the time required to complete the key insertion into the tree increases. The reason is that the number of keys that must be compared in each node while traversing the tree in order to find the insertion point increases. Using lower fanout values improve the insertion performance, however this comes at the cost of an increasing memory footprint of the B+-Tree caused by an increasing number of required nodes and larger overhead of the node data structures. A fanout value of 128 provides the most compact B+-Tree without significantly affecting insertion performance. Using the 128 fanout configuration the required memory footprint to handle the 8GB (524,288 records) experiments is 12,103 KB.

### C. Sorting Schemes Evaluated

In this section, we compare MergeSort, ActiveSort, and our Self-Sorting SSD to sort data. Fig. 2 illustrates the different data transfer operations involved in the sorting process of each scheme. All the schemes begin with a *Generate and Write* (Gen. and W.) phase, where the input data is generated and then written into the storage. We explain the details of each phase for the different schemes below:

*1) MergeSort:* The MergeSort scheme uses the traditional external merge sort algorithm to sort the input data. Fig. 2 breaks down the phases followed to produce the sorted output. MergeSort reduces the host main memory requirement by breaking down the sort process into two parts. In the first part, presorted chunks of data called runs are generated by reading chunks of data into the host, sorting and writing back the partial sorted results to disks. This is an iterative process that should be repeated to process the whole input data. The number of iterations required is equal to the ratio of the original data size and the chunk size processed in each iteration. This process is represented as a *partial sort* phase in Fig. 2. The second part involves merging each of the runs previously produced into one complete stream of sorted data. Traditionally this merged stream is written back to storage. For this reason we have depicted this phase as *merge (W)* denoting that the output of the stream is written to disk.

*2) ActiveSort:* ActiveSort uses on-the-fly data merge to accelerate external sorting[4]. This process is divided into two parts. The first is similar to that of MergeSort in the sense that runs of presorted data are generated by the host (*partial sort* phase). However the second process called *active merge* employs on-the-fly data merge to generate the final sorted stream. Active merge works by writing the sorted runs from the *partial sort* phase into special LBA regions. Once the output is requested the active SSD will internally merge the partial sorted data by comparing the record key, and serve it to the host. By transforming what would be write operations on the *merge (W)* phase into read operations, it is possible to reduce the time to service the requests since read operations are processed faster than write operations and will eventually reduce erase operations increasing the SSD's lifetime.

*3) Self-Sorting SSD:* The Self-Sorting SSD schemes take a more radical approach, they completely eliminate the sorting processes into the SSD. This eliminates the overhead of read and write operations completely. Since there are two mechanisms to generate indexes, there are two Self-Sorting schemes: Self-Sort(C) refers to the sort scheme that uses the *sort-on-command* mechanism to generate the index, while Self-Sort(W) refers to the scheme that uses the *sort-on-write* mechanism.

Self-Sort (C) has three phases. The first one, *generate and write*, is same as in the previous schemes. The second phase named *generate tree* is activated on demand by the host when required. To activate the indexing process in the second phase the host must send a *generate* command to the active disk. Once the command is received the Self-Sorting SSD will read the unsorted input data internally and generate the index. The final phase (*active read*) just involves the host issuing standard read requests. The index will handle the requests and provide the sorted data.

In Self-Sort (W), the sorting process is divided into two phases: the *generate* tree phase which triggers the internal

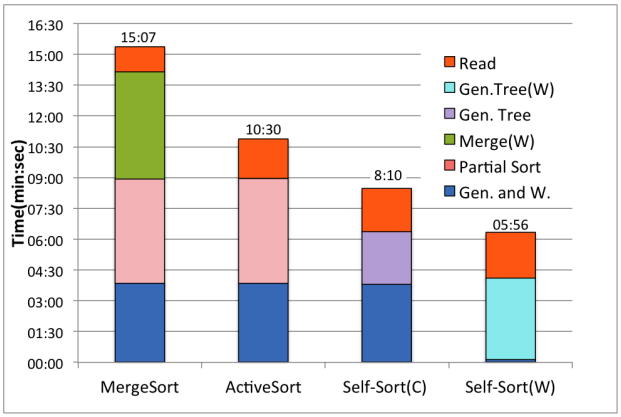


Fig. 3. The total elapsed time of each scheme. The final read phase of each scheme (Read, Active Merge, and Active Read) has been grouped into the Read category.

TABLE I
BANDWIDTH (MB/S) OF IMPORTANT I/O OPERATIONS. PS INDICATES THE USE OF PRE-SORTED INPUT DATA.

| Scheme | Write | Read | Active Op. | Merge (W) | Generate Tree |
|---|---|---|---|---|---|
| MergeSort | 38.60 | 111.73 | | 27.53 | |
| ActiveSort | 35.55 | 111.09 | 74.62 | | |
| ActiveSort-PS | 35.54 | 111.14 | 112.02 | | |
| Self-Sort (C) | 36.78 | | 63.99 | | 53.54 |
| Self-Sort (C)-PS | 36.14 | | 113.03 | | 51.26 |
| Self-Sort (W) | | | 64.21 | | 36.20 |
| Self-Sort (W)-PS | | | 112.46 | | 35.25 |

indexing process on the Self-Sorting SSD. To indicate that the index generation of the *sort-on-write* mechanism takes place during write we have marked it with (W). Since Self-Sort(W) indexes on write operations we have grouped the *gen. and w.* and *generate tree* phases together. The next *active read* phase involves the host issuing read requests for the sorted data which are served by the Self-Sorting SSD. As seen in Fig. 2 the creation of partial sorted data has been completely removed from the process and instead replaced with the *generate tree* phase that only involves data read and processing but no write.

### D. Performance

Fig. 3 compares the elapsed time of each sort scheme to sort 8GB of input data. Complementing the elapsed information is Table I which shows the bandwidth seen at each phase involving data transfers to/from the SSD storage. Additionally, to compare the overhead of performing data computation within the SSD, presorted data tests results are shown as (PS) next to the scheme name. Using a presorted data set maximizes the internal parallelism enabling us to extract the overhead of data computation. The following observations for each scheme must be noted:

MergeSort: Although read and write operations can be performed at top speed, the need to write the partial runs increases the elapsed time. Moreover seeing the extremely low performance in the *merge (W)* phase suggests that offloading this phase is a good opportunity to increase performance.

ActiveSort: The time for *generate and write* and *partial sort* are similar to that of MergeSort. However, by transforming the expensive write operations of MergeSort into read operations using *active merge* it is possible to achieve better overall performance. As seen in the ActiveSort(PS) algorithm of Table I, the overheads of performing merge operations (key comparison and memory copy) within the SSD are completely hidden. This is possible since the input data is previously sorted so the partially sorted chunks are accessed sequentially and are perfectly striped across the SSD's flash channels fully utilizing the SSD internal parallelism. When using the random input data, the requests cannot be striped across the channels and the parallelism is reduced resulting in a decreased performance of 74.62 MB/s. Conveniently, this performance is still better than the average write performance (36.98 Mb/s) thus explaining the performance gain.

Self-Sort(C): Following the same approach as the ActiveSort scheme, the Self-Sorting SSD seeks to replace write operations with read operations. However, while ActiveSort replaces only the write operations in the *merge (W)* phase, Self-Sorting SSD replaces all the write operations across all the sort process. To do so, it uses the *generate tree* and *active read* phases both of which are read based. The overhead of sorting within the SSD is expected to be larger than that of merging. Table I shows that the *generate tree* and *active read* from Self-Sort SSD have lower performance than the *active merge* phase of ActiveSort. However, the performance of those two phases are still better than that of the combined read-write operations they seek to replace. When using the presorted input data (PS) to verify the overhead of this scheme, we also find that increasing parallelism could potentially hide completely the cost of performing sorting operations within the SSD.

Self-Sort(W): This scheme moves the index generation when the input data is being written. From Table I, the *generate tree* operation is composed of three parts: the generation of the input data, the writing process of the data, and the indexing of the data. This operation has a bandwidth of 36.20 MB/s which is slightly lower than the average write bandwidth of 38.09 MB/s which is the standard write operation bandwidth of the Jasmine OpenSSD platform. The bandwidth while reading using the internal index is 64.21 MB/s which is similar to that of Self-Sort(C). Self-Sort(C) reduces time by generating the index simultaneously with the write operation avoiding executing an internal read of the data input that is required by the Self-Sort(C) scheme. By employing *sort-on-write*, Self-Sort(W) reduces the duration by 60.75% compared to MergeSort and by 27.35% when compared with ActiveSort.

### E. Energy

Power measurements were taken during the execution of the four sort schemes using a Yokogawa WT210 digital power meter configured to integrate the registered power over the elapsed time of the scheme. The configured sample rate on the power meter was 0.25 seconds. Energy consumption results are shown in Fig. 4. Since energy is a product of power and time, the energy consumption follows a similar performance

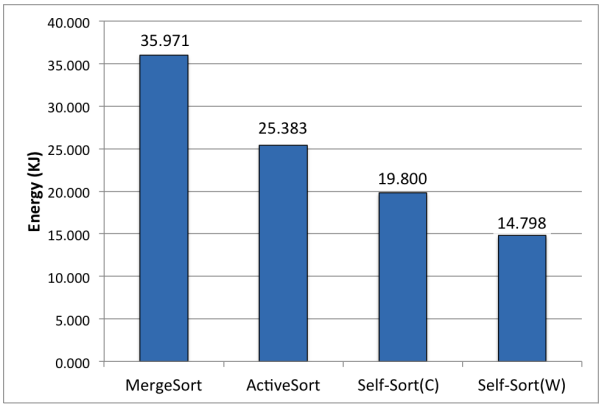


Fig. 4. Energy Consumption Results

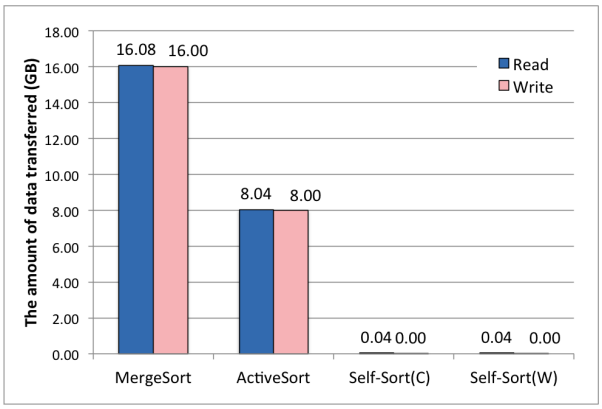


Fig. 5. I/O Operations Overhead

behavior to the elapsed time results. Using the Self-Sorting SSD with the *sort-on-command* mechanism yields energy savings of 46.44% while using the *sort-on-command* yields 58.86% savings in comparison with MergeSort.

### F. I/O Operations Overhead

The amount of data overhead was logged during the evaluations to study the advantages of the Self-Sorting SSD scheme. A request that is neither part of the initial unsorted data writing process nor of the sorted output reading process is considered as an overhead. It was found that when comparing the MergeSort and ActiveSort schemes, the later one reduced the amount of write I/O by almost 50% potentially doubling the lifetime of the SSDs. Moreover, in the case of Self-Sorting SSD scheme write operations required to sort the data have been completely removed.

## V. Discussion

The current SSD prototype platform runs on a single core embedded processor at lower frequencies compared to those found in new multi-core embedded processors based SSDs that are commercially available nowadays. Higher clock speeds and multi-core technologies will further speed up the processing capabilities. Increasing internal parallelism could potentially increase the complexity and amount of operations that could be hidden when performing active operations.

An alternative to the external merge sort algorithm is an index sort algorithm executed in the host. In this algorithm, the host parses records as they are about to be written into the SSD and generates an index in memory using the record keys. Then the host uses the index to read the data from disk in an ordered fashion. This algorithm can avoid the write overhead of the external merge sort algorithm. However, we believe there are still some benefits of using Self-Sort SSD. The first is freeing the host system resources by generating the index within the SSD. Second, performance scalability: by adding more Self-Sorting SSDs the indexing performance increases linearly. Third, in order to read the sorted data the host index sort algorithm needs to generate one I/O request per record. This can become a performance issue since the number of I/O requests will be affected by the record size. Short records particularly will have more overhead. The Self-Sorting SSD avoids this problem.

## VI. Related Work

Active disks were introduced to offload data processing functions to hard disk drives in [1][10][11][12][13] mainly focusing in filter and aggregation functions.

Recently, researchers have applied the same concept to SSDs. Some approaches use custom hardware such as Kim et al. in [14] where porting the scan function used in data-intensive applications to SSDs is explored, and obtains better results that those obtained previously with hard disk drives by employing a special purpose computing module inside the SSD controller. In [15], Cho et al. proposes a model of active disk using an SSD with an added reconfigurable stream processor per flash memory channel at marginal cost to speedup data-intensive applications.

Previous research to improve the performance of external merge sort using SSDs include: Park and Shim [16], who introduce the flash-aware external sorting algorithm which utilizes multiple reads instead of heavy writes to improve sort performance. Liu et al. [17] propose using natural occurring runs whose range of values do not overlap each other to accelerate sorting. Compared to these work, we can completely remove I/O operations during external merge sort by building indexes inside of SSDs.

## VII. Conclusion

In this paper, we have evaluated the Self-Sorting SSD architecture which completely offloads sorting operations into SSDs by creating indexes at write time or on demand as required. The evaluation results show that the proposed scheme can completely eliminate write operations from the external sorting process, improving performance and the lifetime of SSDs.

## Acknowledgment

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

## REFERENCES


[1] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[2] G. Graefe, "Implementing sorting in database systems," *ACM Computing Surveys (CSUR)*, vol. 38, no. 3, 2006.

[3] "Couchbase Server 3.0," Couchbase, 2014.

[4] Y.-S. Lee, L. Cavazos Quero, Y. Lee, J.-S. Kim, and S. Maeng, "Accelerating external sorting via on-the-fly data merge in active SSDs," in *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[5] P.-A. Larson and G. Graefe, "Memory management during run generation in external sorting," in *Proceedings of the ACM SIGMOD International Conference*, 1998.

[6] D. Comer, "The ubiquitous b-tree," *Computing Surveys*, vol. 11, no. 2, pp. 121–137, June 1979.

[7] D. Kang, D. Jung, J.-U. Kang, and J.-S. Kim, "$\mu$-tree: an ordered index structure for NAND flash memory," in *Proceedings of the International Conference on Embedded Software (EMSOFT)*, 2007, pp. 144–153.

[8] "The OpenSSD Project," http://www.openssd-project.org/.

[9] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. J. Desnoyers, and Y. Solihin, "Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[10] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki, "Diamond: A storage architecture for early discard in interactive search," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2004.

[11] E. Riedel, G. Gibson, and C. Faloutsos, "Active disks for large-scale data mining and multimedia," in *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1998.

[12] E. Riedel, C. Faloutsos, G. Gibson, and D. Nagle, "Active disks for large-scale data processing," *Computer*, vol. 34, no. 6, Jun. 2001.

[13] E. Riedel, C. Faloutsos, and D. Nagle, "Active disk architecture for databases," Carnegie Mellon University, Tech. Rep., 2000.

[14] S. Kim, H. Oh, C. Park, S. Cho, and S.-W. Lee, "Fast, energy efficient scan inside flash memory solid-state drives," in *Proceeedings of the International Workshop on Accelerating Data Management Systems (ADMS)*, 2011.

[15] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger, "Active disk meets flash: A case for intelligent SSDs," in *Proceedings of the International Conference on Supercomputing (ICS)*, 2013.

[16] H. Park and K. Shim, "Fast: Flash-aware external sorting for mobile database systems," *Journal of Systems and Software*, vol. 82, no. 8, pp. 1298–1312, 2009.

[17] Y. Liu, Z. He, Y.-P. P. Chen, and T. Nguyen, "External sorting on flash memory via natural page run generation," *The Computer Journal*, vol. 54, no. 11, pp. 1882–1990, 2011.