External Mergesort for Flash-based Solid State Drives

Joonhee Lee, Hongchan Roh, Sanghyun Park

Abstract— Mergesort is the most widely-known external sorting algorithm, which is used when the data being sorted do not fit into the available main memory. There have been several attempts to improve mergesort by reducing I/O time, since mergesort is I/O intensive. However, these methods assumed that mergesort runs on hard disk drives (HDDs). Flash-based solid state drives (SSDs) are emerging as next generation storage devices and becoming alternatives to HDDs. SSDs outperform HDDs in access latency, because they have no physical arms to move. In addition, SSDs benefit from their inner structure by exploiting internal parallelism, resulting in high I/O bandwidth. Previous methods for improving mergesort focused on reducing random access cost, which is insignificant on SSDs. In this paper we propose an external mergesort algorithm for SSDs called ***FMsort***. FMsort calculates a *block read order* which is the order of blocks needed in the merge phase. With a block read order, a number of blocks required during the merge phase are read into main memory via multiple asynchronous I/Os. Our experiments show that FMsort outperforms other mergesort algorithms, at an invisible cost of calculating a block read order.

Index Terms—Mergesort, External sorting, Flash-based SSDs, DBMS

1 Introduction

In recent years, flash-based solid state drives (SSDs) have emerged as next generation storage devices. SSDs have been adopted in a wide range of areas because of their superior characteristics such as high I/O bandwidth and short access latency. These features are due to the inherent properties of the flash memory technology.

External mergesort is the most common sorting algorithm, which is used for sorting large amounts of data. Generally, a traditional external mergesort reads one block of data at a time during merge phase. After a specific block in the input buffer is emptied, the next block of the corresponding run is loaded. This approach, however, does not utilize the high random I/O bandwidth of SSDs. Thus, there is much more room for improvement by exploiting the internal features of SSDs.

In this paper, we propose an external mergesort for SSDs, called ***FMsort*** (an abbreviation for Flash Mergesort). To the best of our knowledge, FMsort is the first method that focuses on enhancing the merge phase of external mergesort for SSDs. FMsort exploits the short access latency and high random I/O bandwidth of SSDs. To utilize these advantages, a number of blocks are read simultaneously into main memory in the merge phase, in the exact order. Therefore, the order of the blocks needed in the merge phase must be known in advance. This order, called the block read order, is pre-calculated in the run formation phase by sorting on the first tuple's key of each data block.

We conducted an experiment using modified PostgreSQL and data files created by the creation tools of TPC-H [20], and TPC-C [19]. In our experiment, FMsort outperformed traditional mergesort and mergesort with double buffering up to 4.87 times and 4.67 times, respectively.

This paper is organized as follows. Section 2 provides the background and the related work. In section 3, we introduce our new mergesort algorithm for Flash-based SSDs, FMsort. We evaluate FMsort in section 4. Section 5 concludes our work.

- *Joonhee Lee is with the Department of Computer Science, Yonsei University, Seoul, Republic of Korea. E-mail: joonnc@cs.yonsei.ac.kr*
- *Hongchan Roh is with SK Telecom, Bundang, Republic of Korea. E-mail: hongchan.roh@sk.com*
- *Sanghyun Park is with the Department of Computer Science, Yonsei University, Seoul, Republic of Korea. E-mail: sanghyun@cs.yonsei.ac.kr*

2 Background & Related Work

This section provides the prominent features of Flash-based SSDs, two phases of the traditional external mergesort, and previous works to improve the external sorting on flash memory.

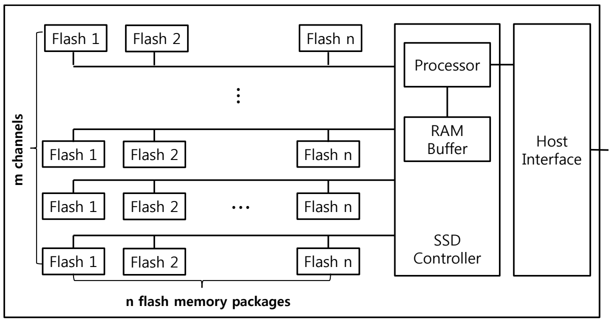


Fig. 1. Internal architecture of Flash-based SSDs

2.1 Flash-based SSD Features

The internal architecture of a SSD is described in Fig. 1. The SSD includes a host interface, a controller, multiple data channels and multiple flash memory chips. Each chunk of *n* flash memory packages are connected in parallel into a channel. *m* channels are connected to a SSD controller, which includes a CPU and a RAM buffer. Due to this architectural design, SSDs have an outstanding feature called internal parallelism [7], which cannot be observed in hard disk drives (HDDs). To take full advantage of the internal parallelism, a number of I/O requests are delivered simultaneously into *m* channels (channel-level parallelism), and each channel spread its I/O over *n* flash memory packages (package-level parallelism). In this way, the maximum bandwidth of a SSD can be utilized, which can theoretically reach $m \cdot n$ times the bandwidth of one flash memory package.

A recent study [16] also suggested a new I/O request interface called *Psync* I/O, which is designed to submit multiple random I/Os at once in order to fully exploit the internal parallelism of SSDs. *Psync* I/O can deliver multiple random I/Os to a SSD with a single I/O request and retrieves the results at once. *Psync* I/O was implemented with libaio API [4].

2.2 Traditional External Mergesort

Traditional external mergesort is composed of the run formation phase, and the merge phase. Although the two phases play different roles in the algorithm, both phases follow the 'load-sort-store' mechanism.

In the run formation phase, a portion of the input data is fully loaded into the input buffer of main memory, sorted with in-memory sorting algorithm, and stored back to the disk as a sorted run (Fig. 2a). This is repeated until all input data is processed. As a result, a number of sorted runs are created. In the merge phase, sorted runs are merged into a single sorted run (result run). As Fig. 2b describes, each sorted run is allocated an input buffer. At first, the first portion of each sorted run is loaded into the corresponding input buffer. Then, tuples in the input buffers are copied into the output buffer one by one, in a sorted order. If a specific input buffer is exhausted during the merge phase, the next portion of the corresponding run is loaded into it. If the output buffer becomes full, it is flushed to disk. In the end, every portion of the sorted runs is processed into the result run.

Depending on the environment, there may not be enough memory to allocate an input buffer to every sorted run. In this case, only some of the sorted runs are merged in the first pass. The result run of the first pass is treated as another sorted run, and it is merged with remaining runs in the following pass. This process is repeated until only one sorted run is left. However, analysis in [15] shows that one pass is enough in practical situations. Therefore, we do not consider multi-pass mergesort in the rest of this paper.

2.3 Improving External Sorting for Flash Memory

HDD-based mergesort was studied extensively in previous research. For example, L. Zheng and P. Larson [18]

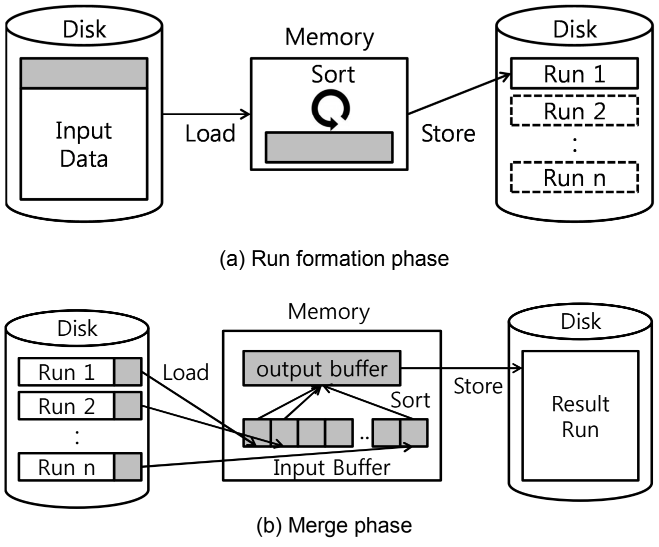


Fig. 2. Traditional external mergesort

introduced few approaches such as interleaved layout and read scheduling. However, the main focus of each algorithm was to reduce seek time of HDDs, which is insignificant in SSDs.

Recently, there have been a number of studies on how to improve the performance of external sorting with flash memory. Graefe composed 3-level memory hierarchy in [11], by placing flash memory between main memory and hard disk. Flash memory plays role as a cache between two devices, and relatively large file can be sorted in small time with a limited amount of memory. In addition, Lee at al. [12] introduced the potential of SSDs as storages for DBMS, and they exemplified external sorting. They insisted that performances of the run formation on both HDD, and SSD are similar because only sequential reads/writes are executed. However, since random read consistently occurs in the merge phase, SSDs are more favorable because of their short access latency. Another study [14] proposed an active SSD architecture to improve the external sorting. By performing the merge phase on-the-fly in active SSDs, they reduced the extra data transfer and enhanced the lifetime of SSDs.

There were some attempts to improve the run formation of mergesort for flash memory. Yang Liu et al. [15] reduced the number of writes during the run formation phase, by utilizing the nature of the 'partially sorted data'. They reduced the number of write operations at the expense of increasing the number of read operations. Andreou et al. [6] proposed FSort, which is an external sorting algorithm for flash-based sensor devices with a small memory footprint. FSort uses a top-down replacement selection algorithm in the run formation phase to prevent multi-pass merge. By reducing the number of runs, Fsort minimizes the expensive write/delete operations of flash memory and saves energy cost. However, [6], [15] do not focus on enhancing the merge phase, which is a more crucial part of the overall performance due to the intensive random reads. On the other hands, Flash MinSort [8], [9] used an index to take advantage of fast random reads in flash memory. Flash MinSort is also designed for embedded devices such as sensor node, which has a very limited amount of memory. Unlike other external sorting algorithms, Flash MinSort does not follow the conventional run formation-merge phase. Instead, Flash MinSort divides the input relation into a number of regions and maintains minimum index that stores the smallest sort key value for each region. On each pass, a number of tuple with a smallest sort key are sent to output and minimum index is updated. Similar to [15], Flash MinSort reduced the number of write operations by increasing the number of random read operations. Finally, [13][1] proposed F-mergesort that utilizes the characteristic of flash-based SSDs by adopting *Psync* I/O. By using *Psync* I/O, F-mergsort loads multiple blocks in the merge phase at a single request. Since this request cannot be divided, sorting process must wait until entire requested I/O is completed, even if the needed data block is already in memory. For this reason, F-mergesort does not outperform mergesort with double buffering.

3 FMSORT

In this section, we propose our new mergesort algorithm for SSDs, FMsort. Similar to the traditional mergesort, FMsort consists of two phases. In the run formation phase, sorted runs are created and a block read order is calculated. In the merge phase, sorted runs are merged into a single sorted run by using the block read order. For simplicity, we assume that data is sorted in an ascending order.

3.1 Run Formation Phase

The run formation phase of FMsort is described in Algorithm 1.[2] Each iteration (line 2-12 of Algorithm 1) produces one sorted run, which is given a run number starting at 1 (Run_Number). In each iteration, a portion of the input data is loaded into the input buffer (line 3), and sorted by key (lines 4). Then, each tuple *t* is copied to the output buffer in a sorted order (lines 6). If the current output buffer block was empty before inserting *t*, *t*.key and a current run number are paired and inserted into the block read order B (line 7-8). Whenever the output buffer becomes full, it is flushed to disk (line 9-10). The iteration terminates when there is no more data to read. After all sorted runs are created, B is sorted by sort key (line 13). Then, sort keys in B are discarded (line 14) because their values are not required in the merge phase. As a result, B only contains a sequence of run numbers sorted by the first tuple's key of each data block. FMsort requires additional memory to store the block read order. However, the extra memory is very small compared to the original data and inversely proportional to the data block size. With an 1GB input data and a block size of 4 KB, block read order only takes 2 MB of main memory (assuming each sort key, and run number takes 4 byte). Moreover, this is halved after sort keys are dropped.

Algorithm 1: run formation phase of FMsort

```
Procedure FMSORT_RUNFORMATION
Input : data to be sorted, sort key
output : sorted runs, block read order B
1:  Run_Number = 1
2:  While there is more data to read do
3:      Read data into the input buffer
4:      Sort the input buffer by sort key
5:      For each tuple t in the input buffer in sorted order
6:          Copy t to the output buffer
7:          If the output buffer block was empty before inserting t
8:              Insert (Run_Number, t.key) into the B
9:          If the output buffer is full
10:             Flush the output buffer
11:     Flush the output buffer
12:     Run_Number ++
13: Sort B by sort key
14: Discard sort keys in B
```

3.2 Merge Phase

Before providing details of the merge phase, we classify the input buffer blocks according to their role. ***Sort block*** participates in a sorting process to find an unprocessed tuple with the smallest key (the smallest tuple). An unprocessed tuple is a tuple which has not been copied to the output buffer yet. Each sort block maintains a pointer that points the smallest tuple in its block. The smallest tuple among these pointed tuples is consistently sent to the output buffer. ***Assist block***, on the other hand, is used for reading a data block on the disk in advance. Data is loaded into an assist block via asynchronous I/O, in order to overlap sorting (CPU processing) and I/O.

Algorithm 2 describes the merge phase of FMsort. In the beginning, first *n* blocks indicated by the block read order are loaded into the *n* sort blocks (line 1 of Algorithm 2). Contrary to the traditional mergesort, *n* blocks may not be from the first block of each run. In FMsort, the order of the block loaded into the input buffer only follows the block read order. Depending on the block read order, some runs may be excluded from initial *n* blocks. This, however, does not affect the order of tuples in the result run (proof is given later in this section). Next we request *l* asynchronous I/Os in a series to read the next *l* blocks into the *l* assist blocks (line 2). Although these *l* I/Os are executed in parallel, we consider one whose data block appear first in the block read order to be the oldest. The data block read with the oldest asynchronous I/O is immediately needed whenever one sort block is exhausted. In addition, we use Async_cnt to store the number of asynchronous I/Os (line 3).

In each iteration, the smallest tuple in the sort blocks is copied to the output buffer (line 5). In order to retrieve the smallest tuple, any in-memory comparison algorithm can be used. We used a tree of losers, which finds the smallest element with log (n) comparisons by utilizing the previous match result. If the output buffer becomes full,

[1] This paper is an improved version of the paper [13]. Due to the changes made to the algorithm, we rewrote the paper including the new algorithm explanation, new figures and new experimental results.

[2] Run generation is not the focus of this paper. However, we describe our own implementation details for run generation to describe the process of generating the block read order.

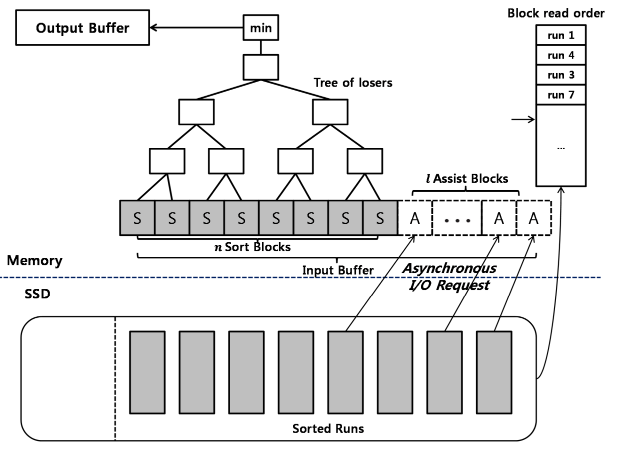
Algorithm 2: Merge phase of FMsort

```
Procedure FMSORT_MERGE
Input : n sorted runs, l assist blocks, block read order B
Output : single sorted run
1:  Read the first n blocks (indicated by B) into the n sort
     blocks
2:  Read next l blocks (indicated by B) into the l assist
     blocks, using separate asynchronous I/Os
3:  Async_cnt = l
4:  While there are unprocessed tuples in the sort blocks
5:     Copy the smallest unprocessed tuple in the sort blocks
        to the output buffer
6:     Flush the output buffer whenever it becomes full
7:     If a certain sort block is exhausted and Async_cnt > 0
8:        If the oldest asynchronous I/O is not complete
9:           Wait until it is completed
10:       The assist block with the oldest asynchronous I/O
           becomes a sort block
11:       Async_cnt--
12:       If there are more blocks to read
13:          The exhausted sort block becomes an assist block
              and the next block (indicated by B) is read into it
              via asynchronous I/O
14:          Async_cnt++
15:  Flush the output buffer
```

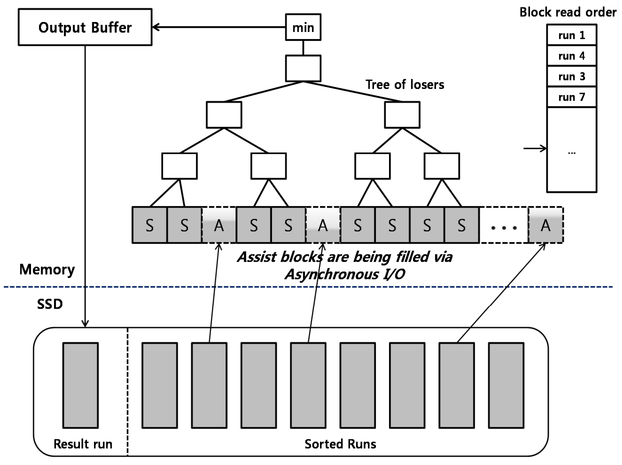
we flush it to disk (line 6). If a specific sort block is exhausted and at least one asynchronous I/O exists (line 7), we check if the oldest asynchronous I/O has been completed (line 8). If not, the entire sorting process is blocked and waits until it finishes the reading (line 9). Then, the assist block with the oldest asynchronous I/O becomes the sort block (line 10), and Async_cnt is decreased by 1 (line 11). If there are more blocks to read, we use the exhausted sort block as an assist block to read the next data block into it via asynchronous I/O (line 12-13). In this case, Async_cnt is increased by 1 (line 14). After all tuples are processed, we flush the output buffer for the last time (line 15).

Fig. 3a presents the memory status after loading the first *n* blocks into the input buffer. S and A denote the sort block and the assist block respectively. A number of asynchronous I/Os are executed to fill the assist blocks, while the smallest tuple in the sort blocks is constantly copied to the output buffer. During the merge phase, an exhausted sort block is immediately reused as an assist block for reading the data block on the disk. Throughout the merge phase, each input buffer block repeatedly changes its role between a sort block, and an assist block (Fig. 3b).

Figure 4 illustrates an example of FMsort after 4 sorted runs are generated. Each run is composed of 3 blocks, and each block contains 3 tuples. We assume that there are 6 available memory blocks (M1 to M6). After allocating one memory block per run (M1 to M4), remaining 2 memory blocks are used as assist blocks(M5, and M6). At the initial stage, 4 data blocks (Block 4, 1, 7, and 5) are loaded and become sort blocks. While these blocks are being sorted, next 2 blocks (Block 8 and 10) are separately loaded and become assist blocks. After value 12 is sent to the output, sort block M2 is exhausted. Since the next block we need (block 8) is pre-loaded in M5, M5 becomes a sort



(a) After loading first n blocks



(b) During sort process

Fig. 3. Memory status of the merge phase

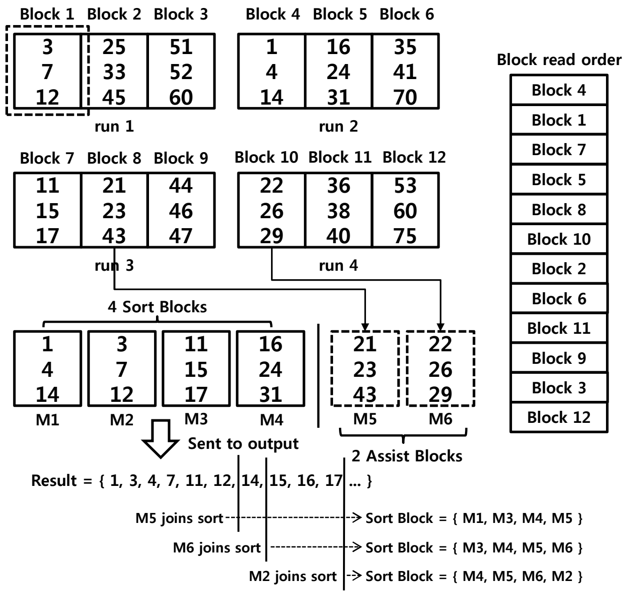


Fig. 4. Merge phase of FMsort with 4 runs and 2 assist blocks

block and participates in sorting. M2, on the contrary, becomes an assist block and loads the next block (Block 2) from disk. In similar way, M6 becomes a sort block after value 14 is sent to the output, and M1 is used to load Block 6. This role-switching between a sort block (M3) and an assist block (M2 containing Block 2) happens again after value 17 is sent to the output.

FMsort utilizes internal parallelism (high random I/O bandwidth) of SSDs by reading multiple blocks in parallel in the merge phase.[3] This approach is not suitable for HDDs because these blocks are not from the same run. Since HDD must move its disk arm to a specific position to read, reading non-contiguous blocks simultaneously will incur a significant amount of seek time.

In the rest of this section, we prove that FMsort produces a sorted run. The terms used for the proof are defined in Table 1.

Lemma 1. *Suppose that R is not sorted. Then, there must have been an instance during the merge phase where $m > d$.*

Proof. During the merge phase, m is copied to the output buffer. Once m is placed in the output buffer, its position in R is fixed. Therefore, if R is not to be sorted, there must have been an unprocessed tuple on the disk, which is smaller than m of the moment. Since d is the smallest unprocessed tuple on the disk, $m > d$ must have been satisfied.

Theorem 1. *(Feasibility of FMsort) FMsort produces a single sorted run with each data block having been read only once, provided that the number of input buffer blocks is at least equal to the number of runs.*

Proof. We give a proof when the number of sorted runs and the number of input buffer blocks is equal. Since the number of sort blocks in FMsort equals the number of runs, all input buffer blocks are used as sort blocks. We cannot overlap CPU processing and I/O because there are no assist blocks. Whenever a certain sort block is exhausted, the sort process is blocked and the next data block on the disk is loaded to the exhausted sort block. However, the order of the data block loaded to the input buffer always follows the block read order. Therefore, a result run is not affected by the number of assist blocks. When there are more input buffer blocks available, extra buffer blocks are used as assist blocks.

The block read order only contains a sequence of run numbers. The total number of entries in the block read order is the total number of data blocks of n sorted runs. Given a specific run number, FMsort reads one block of the corresponding run, starting from a position which has not been read yet. Therefore, every block of the sorted runs is read only once.

Next, we prove that R is sorted. Based on Lemma 1, $m \leq d$ must be always satisfied if R is to be sorted. First, we classify the state of the merge phase in Table 2, and prove that $m \leq d$ holds for every instances of each state.

[3] Basically, FMSort's asynchronous I/O mechanism with block read order increases the probability of reading multiple blocks simultaneously. However, if the number of assist blocks is not enough, then FMSort can be blocked and "reading one by one" can happen.

TABLE 1
TERMS USED IN THE PROOF

n	Number of runs
t	Unprocessed tuple in the sort blocks
m	Smallest unprocessed tuple in the sort blocks
d	Smallest tuple on the remaining blocks on disk
S	Sort block held in memory
$S.f$	First tuple of the sort block S
R	Result run produced by FMsort
$a > b$	Tuple a's sort key is greater than tuple b's

TABLE 2
STATE OF THE MERGE PHASE

State	Description
1	There are remaining blocks on the disk to be read
1-a	Sort blocks contain data block from every n sorted run (1 to 1 matching)
1-b	At least one run exists whose data block is not in the sort blocks
2	All data blocks on the disk have been read

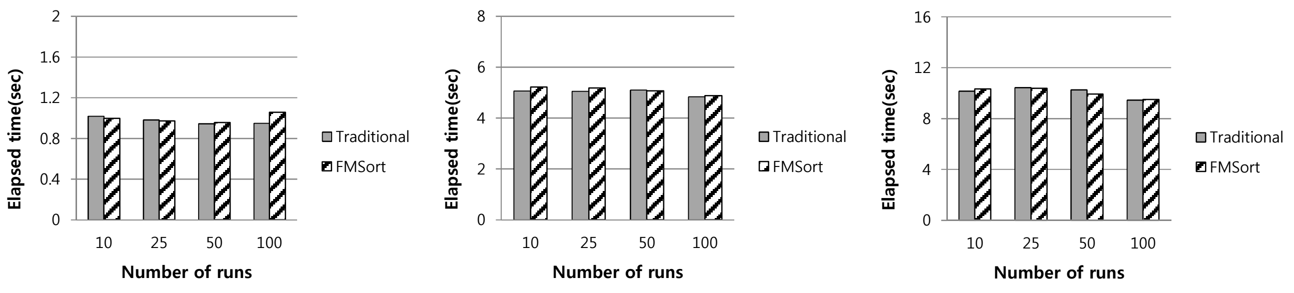
**State 1-a, and State 1-b are sub-state of the State 1.*

The merge phase is basically in State 1 most of the time. Depending on the data in the sort blocks, we further classify State 1 into State 1-a, and State 1-b. State 1-a is exactly equal to the traditional mergesort. Let D be a run which contains d. Then there is a sort block S^* which contains a data block from D. Since D is sorted, $(\forall t \in S^*) \leq d$. And by the definition of m, $(\forall t \in S^*) \geq m$. Therefore, $m \leq d$. In State 1-b, at least one run is missing in the sort blocks. Missing runs cannot participate in a sorting process because their portions are not in the sort blocks. Assume that $m > d$. By the definition of m, $\forall t \geq m$. By the assumption, it follows that, $\forall t > d$. Recall that there are n sort blocks, and at least one run is not participating in a sorting. By the pigeonhole principle, at least two sort blocks contain a data block from the same run. Therefore, at least one sort block S' satisfies $S'.f > d$. This contradicts the block read order, because d is also the first tuple in the data block on disk. Therefore, if the assumption is to be true, the block containing d should have been loaded instead of S'. Therefore, $m \leq d$. (proof by contradiction)

The merge phase falls into State 2 after all blocks on the disk are read. Since there are no more tuples on the disk, only tuples in the sort blocks are of concern. Thus, tuples cannot be produced in an incorrect order. Since $m > d$ is never satisfied during the merge phase, R is sorted.

4 EXPERIMENTAL RESULTS

We evaluate FMsort by comparing its performance with other mergesort variants. Traditional mergesort does not overlap CPU processing and I/O. After a certain input buffer block is exhausted, the next block of the corre-



(a) Data A (b) Data B (c) Data C

Fig. 5. Elapsed time of the run formation phase on IODrive with a block size of 4 KB

sponding run is loaded. Thus, the sorting process is blocked until loading is complete. On the other hand, mergesort with double buffering uses half of its input buffer blocks for reading the next block of each run. If there are *n* sorted runs, *n* reads are executed in parallel. FMsort uses the number of assist blocks as an input parameter, which represents a parallel I/O level. Other methods introduced in section 2.3 were excluded in our experiments for following reasons. [6], [15] focused on improving the run formation phase, while FMsort focuses on enhancing the merge phase. Therefore, FMsort and [6], [15] are orthogonal to each other. [14] used active SSD to merge the runs on-the-fly, and no changes were made to the algorithm. Finally, Flash MinSort [8], [9] is designed to sort small data with small memory sizes where external mergesort is not executable, and performs best when sort keys are clustered. In [8], their experiments were conducted with the sensor node having 4KB SRAM, 2MB serial flash chip, and a data size of 1.6MB. Under our environment, performance of Flash MinSort was even worse than that of the traditional mergesort, because of the excessive read and CPU costs. Since running environment of Flash MinSort and FMsort are very different, we excluded Flash MinSort from the experiment.

Our experiments were conducted under a single-user environment. Traditional mergesort uses a small fraction of the available SSD bandwidth, because only one block is read at a time. On contrary, FMsort and double buffering can utilize more bandwidth by reading multiple blocks simultaneously. The essence of these algorithms is independent of a running environment. Therefore, experimental results of this section may also be applied to a multi-user environment, provided that there is enough available bandwidth for the latter two algorithms.

TABLE 3
DATA FILES USED IN EXPERIMENT

	Data size	Number of tuple	Table name	Sort key
Data A	220 MB	1,500,000	ORDERS (TPC-H)	ORDER-DER-DATE
Data B	1.1 GB	7,500,000		
Data C	2.2 GB	14,999,980		
Data D	930 MB	9,000,000	ORDER_LINE (TPC-C)	OL_I_ID

TABLE 4
SPECIFICATION DATA FOR THE FLASH-BASED SSDS

	Micron P300 [5]	Fusion-io IODrive [3]
NAND Type	SLC	SLC
Interface	SATA 6 Gb/s	PCI-E x4
Capacity	100 GB	80 GB
Read Bandwidth	360 MB/s	700 MB/s
Write Bandwidth	270 MB/s	550 MB/s

4.1 Settings

To implement above methods, we modified buffer manager of PostgreSQL. We used table files created by the PostgreSQL with the database creation tool of TPC-H (DBGen [2]), and TPC-C (BenchmarkSQL [1]). Table 3 presents 4 data files that were used in our experiments. We conducted the experiments on a Linux machine with an 8-core CPU, and a 16GB main memory. We adopted direct I/Os to bypass the OS cache, since data in the OS cache can be reused without I/O operation. Finally, we used two commercial SSDs for our experiments that are described in Table 4.

4.2 Run Formation

To estimate the overhead for calculating block read order, we compared run formation of FMsort to that of a traditional mergesort. With 3 data files, we varied the number of runs created.

We conducted this experiment on an IODrive with a fixed block size of 4 KB. Calculating block read order requires CPU work, thus I/O time had to be fast enough to verify the extra overhead. Furthermore, we chose block size to be 4 KB because a larger block size will decrease the overhead. The result is shown in Fig. 5. Even with this configuration, the extra overhead for calculating block read order was almost invisible. Therefore, we concluded that calculating block read order results in insignificant overhead in the run formation phase.

4.3 Merge

Since FMsort focuses on improving the merge phase, we compared the merge phase of the three algorithms. We

TABLE 5
PARAMETERS AND BLOCK SIZE CALCULATION

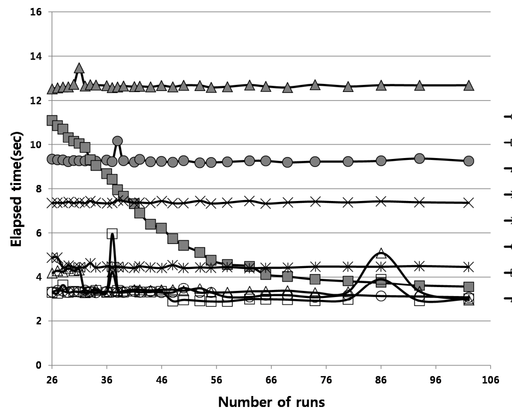
Number of runs			*n*
Number of assist blocks (parallel I/O level of FMsort)			*l*
	Number of input buffer blocks	block size	Total size of input buffer
Traditional mergesort	*n*	***s* (base block size)**	
Double buffering	**2*n***	**2/*s***	***Ns***
FMsort	***n*+*l***	***ns*/(*n*+*l*)**	

varied the number of runs created in the run formation phase, and also varied the block size of the input buffer used in traditional mergesort (*base block size*). Once these parameters are determined, we adjusted the block size of the other algorithms to guarantee that all three algorithms use the same amount of memory for the input buffer. This is depicted in Table 5. Traditional mergesort uses the largest block size, but does not overlap CPU processing and I/O. Block size of the double buffering strategy is halved, but *n* reads are overlapped. The block size of the FMsort is between that of the double buffering and that of the traditional mergesort, depending on the number of assist blocks (*l*) used.

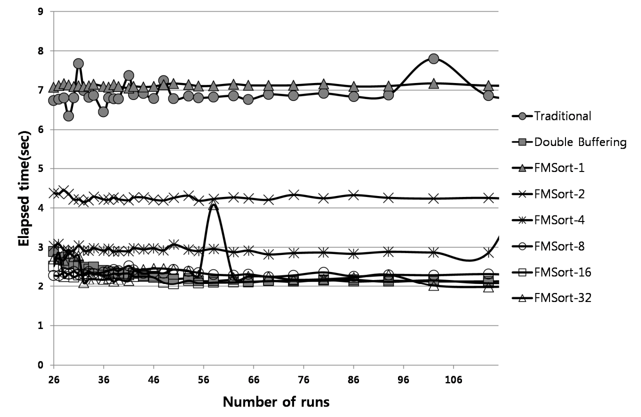
Fig. 6 and Fig. 7 illustrate the elapsed time of the merge phase for the two different Flash-based SSDs with the base block sizes of 4 KB, and 8 KB. For FMsort, the number of assist blocks was doubled from 1 to 32 (FMsort-1 to FMsort-32). The traditional mergesort showed nearly constant performance regardless of the number of runs. Without overlapping CPU processing and I/O, the amount of time to read mainly depends on the total data size and the block size. Since I/O bandwidth of SSD is greater with a larger I/O chunk, traditional mergesort showed performance improvement when the base block size became larger. This can be verified by comparing two subfigures of Fig. 6 and Fig. 7 horizontally. With a small number of runs, double buffering performed even worse than the traditional mergesort in most cases. Performance of double buffering improved gradually as the number of runs increased, which converged after the number of runs reached a certain point. This converging point appeared earlier in the large block size, clearly shown in subfigures c, and d of Fig. 6 and Fig. 7. Finally, FMsort showed stable performance regardless of the number of runs. FMsort-1 did not show outstanding performance because only one read operation was performed at a time. By increasing the number of parallel reads, FMsort was able to use more of the SSD's bandwidth. As a result, the performance of FMsort improved dramatically as the number of assist blocks (*l*) doubled. Improvement was not significant after *l* reached 8, because smaller block size countervailed the parallel I/O level.

In order to reval the performance gap between FMsort and double buffering, we additionally measured the total amount of blocked time on each algorithm. This is presented in Fig. 8. Double buffering consumed a considerable amount of time waiting for an I/O completion (read) when the number of runs was small. On the other hand, blocked time of FMsort was nearly constant and less sensitive to the number of runs. As the number of assist blocks increased, the blocked time of FMsort decreased. The total blocked time of FMsort did not decrease significantly after FMsort-8, which also matches the result in Fig. 6 and Fig. 7. Furthermore, the shape of the graphs in Fig. 8 is nearly identical to that of the corresponding graphs in Fig. 6 and Fig. 7. Therefore, both algorithms are bounded by the amount of blocked time, which makes difference in overall performance. Since the maximum number of the assist buffer is 32 (FMsort-32), parallel I/O level of the double buffering is greater than that of FMsort in most cases. Thus, the parallel I/O level is not a primary factor of the performance improvement. Rather, it is the order of blocks loaded into the input buffer. In double buffering, each block from every *n* run is being loaded because double buffering does not know which block in the input buffer will be emptied next. Some blocks may wait a long time until the previous block from the same run is emptied. This prevents a more urgent block on the disk from being fetched, causing the sorting process to be blocked. For instance, consider the situation where 3 consecutive blocks (*q*, *r*, *s*) from the same run are needed continuously during the merge phase. While block *q* is being sorted, block *r* is loaded into the second buffer. However, block *s* must wait until block *q* is exhausted and block *r* joins sort. Therefore, block *s* might not be completely loaded when block *r* is exhausted. This is clearly shown when the number of runs is small, because the average time it takes for one sort block to be exhausted is also small. Thus, there is a large possibility that a needed data block is not completely loaded. On the other hand, FMsort knows the exact order of blocks needed by referencing the block read order. FMsort can prefetch the next *l* blocks that will be needed immediately, even if all of them belong to the same run. Thus, blocks are used in their requested order and blocked time is minimized. For this reason, FMsort outperforms double buffering.

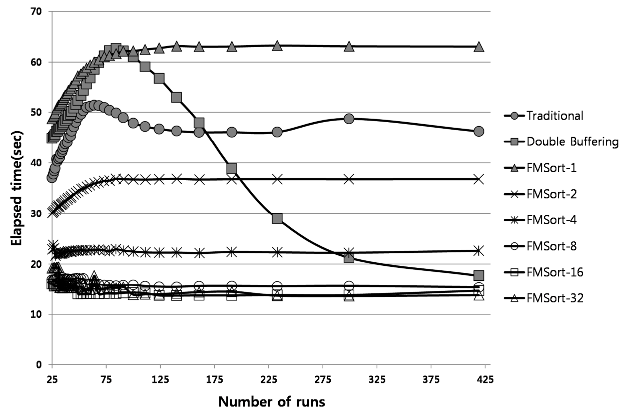
We conducted additional experiment to ensure that above characteristic of mergesort is relation-independent. Fig. 9 presents the elapsed time of the merge phase with Data D, which has a different data schema compared to previous ones. Even with the different relation, the results showed a very similar trend.



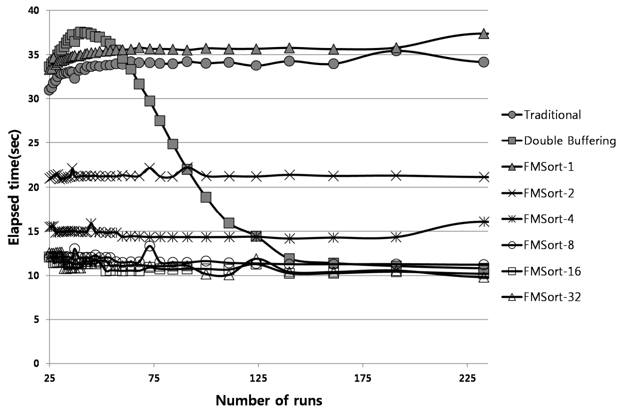
(a) Data A with 4 KB base block size



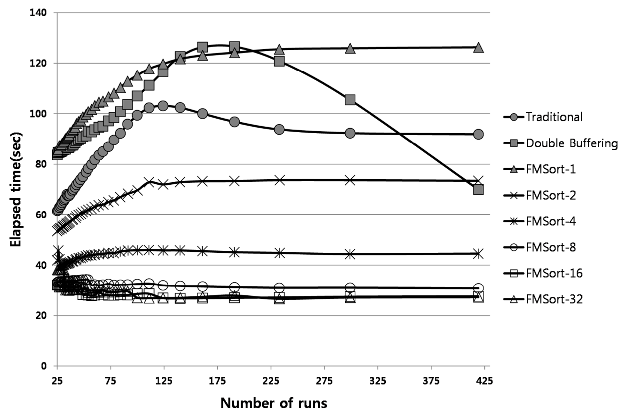
(b) Data A with 8 KB base block size



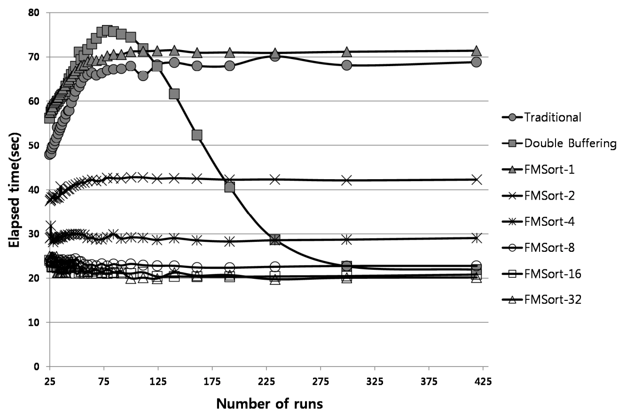
(c) Data B with 4 KB base block size



(d) Data B with 8 KB base block size



(e) Data C with 4 KB base block size



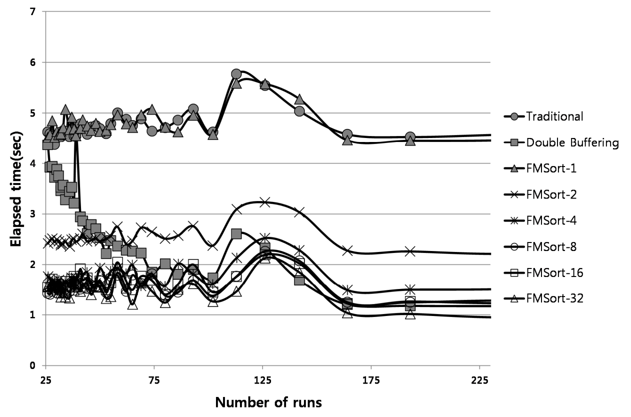
(f) Data C with 8 KB base block size

Fig. 6. Elapsed time of the merge phase on P300

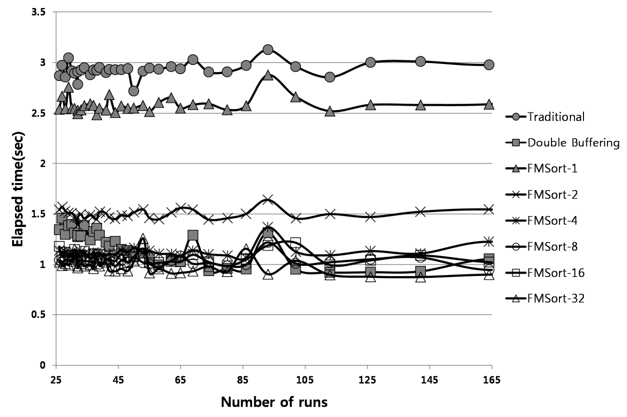
Overall, all three algorithms showed similar tendencies under different conditions. Traditional mergesort showed the worst performance among the three algorithms, because it does not overlap CPU processing and I/O. Performance of double buffering was very sensitive to the number of runs. Finally, FMsort showed stable performance regardless of the number of runs. Furthermore, FMsort with more than 8 assist blocks showed the best performance under every configuration. On the p300, FMsort-32 outperformed traditional mergesort and double buffering by up to 3.86 times and 4.62 times, respectively. FMsort outperformed by up to 4.87 times, and 4.67 times, respectively, on IODrive.

5 Conclusion

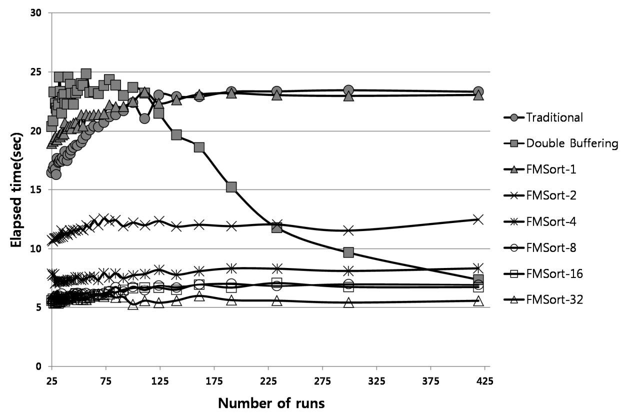
In this paper, we proposed an external mergesort for SSDs called FMsort. FMsort utilizes the internal parallelism of SSDs by reading multiple blocks simultaneously. In order to read blocks in the exact order, the block read order is calculated in the run formation phase. The idea is to use extra time to calculate block read order in the run



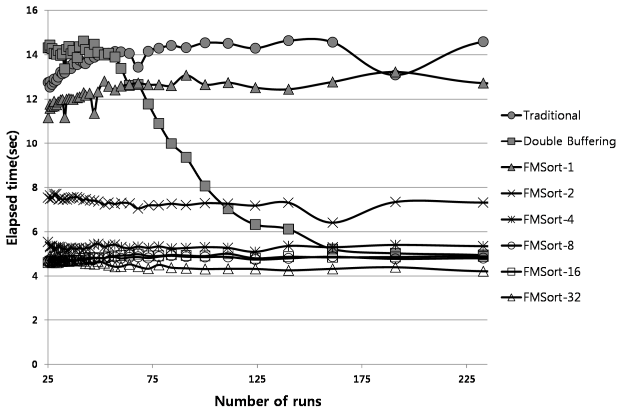
(a) Data A with 4 KB base block size



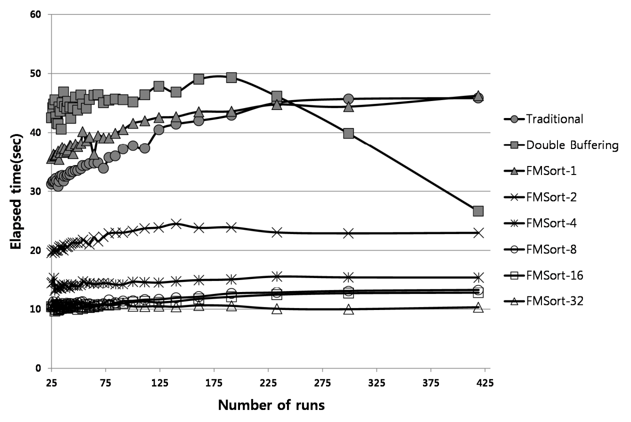
(b) Data A with 8 KB base block size



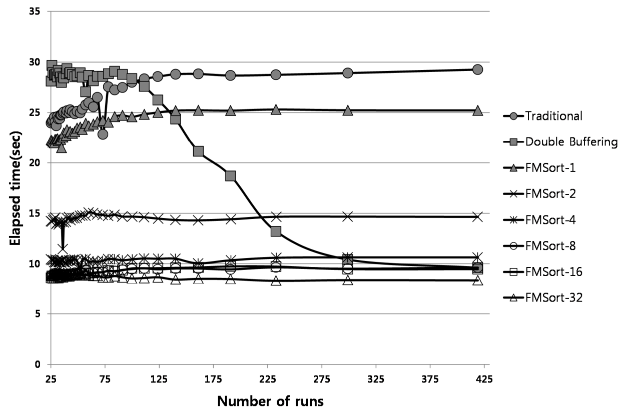
(c) Data B with 4 KB base block size



(d) Data B with 8 KB base block size



(e) Data C with 4 KB base block size



(f) Data C with 8 KB base block size

Fig. 7. Elapsed time of the merge phase on IODrive

formation phase, and save more time in the merge phase.

In our experiment, extra overhead for calculating block read order was almost negligible. In the merge phase, FMsort outperformed traditional mergesort and double buffering up to 4.87 times and 4.67 times, respectively. In addition, FMsort proved to have stable performance with varying configurations. The performance gap between FMsort and double buffering was greater with a larger data size, a smaller block size, and a smaller number of runs. Therefore, we concluded that FMsort is favorable when the amount of data to be sorted is large and the block size is small.

DBMS uses external mergesort in various operations, such as sort-merge-join, duplicate elimination, and set operations [10], [17]. On DBMS, a number of operations are executed in a pipelined way and thus only a limited amount of memory is allowed for each pipelined operation associated with sorting large amounts of data. Since FMsort shows outstanding performance with a small block size and a small number of runs, FMsort can be a stable choice for the external sort algorithm of DBMSs.

To the best of our knowledge, FMsort is the first approach to optimize the merge phase of an external mergesort by exploiting the features of SSDs.

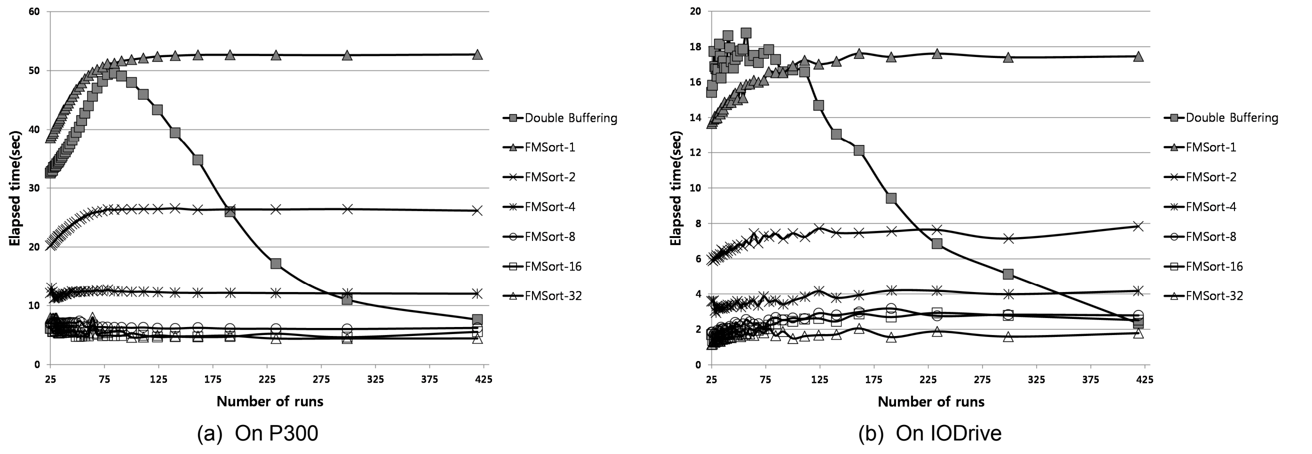


Fig. 8. Total amount of blocked time during merge phase with Data B and a base block size of 4 KB

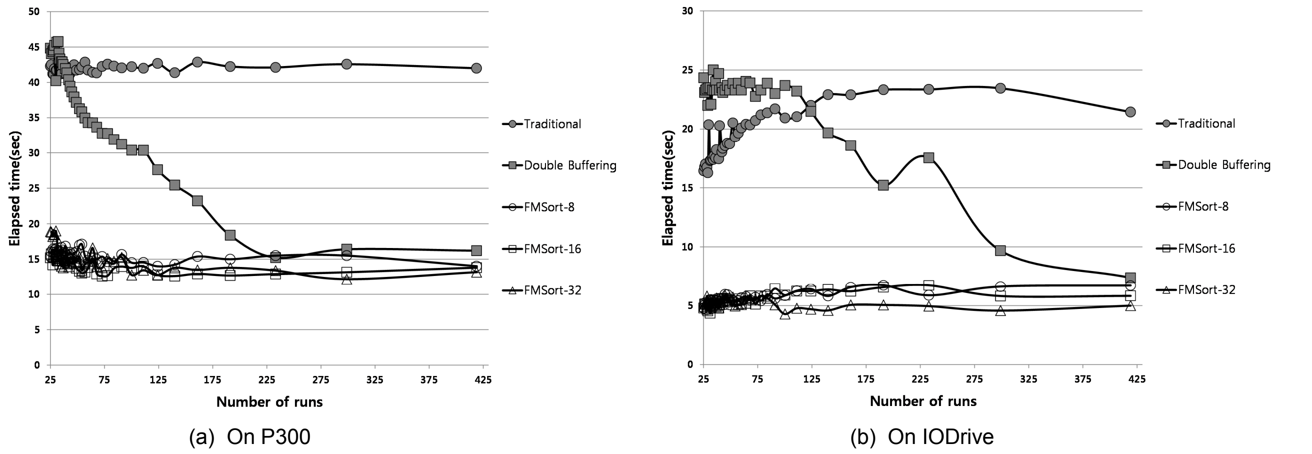


Fig. 9. Elapsed time of the merge phase with Data D and a base block size of 4 KB

Acknowledgment

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (NRF-2015R1A2A1A05001845). The first two authors contributed equally to this work. The corresponding author is Sanghyun Park.

References

[1] BenchmarkSQL. http://sourceforge.net/projects/benchmarksql.
[2] DBGen. http://www.tpc.org/tpch/spec/tpch_2_16_1.zip, 2014.
[3] Fusion-Io, ioDrive. http://www.fusionio.com/load/-media-/1ufytn/docsLibrary/FIO_DS_ioDrive.pdf, 2014.
[4] Libaio. http://lse.sourceforge.net/io/aio.html, 2014.
[5] Micron, P300. http://www.micron.com/~/media/Documents/Products/Data%20Sheet/SSD/p300_2_5.pdf, 2014.
[6] P. Andreou, O. Spanos, D. Zeinalipour-Yazti, G. Samaras, and P. Chrysanthis, "FSort: External Sorting on Flash-based Sensor Devices," *DMSN '09*, 2009.
[7] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on,*. IEEE, 2011, pp. 266-277.
[8] T. Cossentine and R. Lawrence, "EFFICIENT EXTERNAL SORTING ON FLASH MEMORY EMBEDDED DEVICES," *International Journal of Database Management Systems 5.1*, 2013.
[9] T. Cossentine and R. Lawrence, "Fast Sorting on Flash Memory Sensor Nodes," *IDEAS '10*, 2010.
[10] R. Elmasri, and S. B.Navathe, "Fundamentals of DATABASE SYSTEMS 4th edition," 2003, pp. 496.
[11] G. Graefe, "Sorting in a Memory Hierarchy with Flash Memory," *Datenbank Spektrum*, 2011.
[12] S. Lee, B. Moon, C. Park, J. Kim, S. Kim, "A Case for Flash Memory SSD in Enterprise Database Applications," *SIGMOD '08*, 2008.
[13] J. Lee, H. Roh, W. Jung, and S. Park, "External Mergesort for FlashSSDs", *Journal of KIISE: Databases, Volume 41, Number 1, February 2014*, 2014.
[14] Y.-S. Lee, L.C. Quero, Y. Lee, J.-S. Kim, and S. Maeng, "Accelerating External Sorting via On-the-fly Data Merge in Active SSDs," *HotStorage*, 2014.
[15] Y. Liu, Z. He, Y. P. Chen and T. Nguyen, "External Sorting on Flash Memory Via Natural Page Run Generation," *The Computer Journal Advance Access published June 2, 2011*, 2011.
[16] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee, "B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives," *Proc. VLDB Endow*, 2011.
[17] A. Silberschatz, H. F.Korth, and S. Sudarshan, "Database System Concepts sixth edition," McGraw-Hill, 2011, pp. 546-567.
[18] L. Zheng and P. Larson, "Speeding up external mergesort", *IEEE Transactions on Knowledge and Data Engineering, VOL. 8, NO. 2*, 1996.
[19] Transaction Processing Performance Council, TPC BENCHMARK™ C Standard Specification Revision 5.11, 2010.

[20] Transaction Processing Performance Council, TPC BENCHMARK™ H Standard Specification Revision 2.16.0, 2013.

Joonhee Lee received the BS degree in 2013, MS degree in 2015, from Yonsei University, Seoul, Republic of Korea. His current research interests include distributed processing system, system design, database system, and flash memory.

Hongchan Roh received the BS degree in 2006, MS degree in 2008, PhD degree in 2014, from Yonsei University, Seoul, Korea. He is currently a research fellow in the SK Telecom. His current research interests include network processor, database system, flash memory, SSD.

Sanghyun Park received the BS and MS degrees in computer engineering from Seoul National University in 1989 and 1991, respectively. He received PhD degree in the Department of Computer Science from University of California at Los Angeles (UCLA) in 2001. He is now a Professor in Department of Computer Science, Yonsei University, Seoul, Korea. His current research interests include database, data mining, bioinformatics, and flash memory.