WOLF–A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File Systems

Jun Wang and Yiming Hu
Department of Electrical & Computer Engineering and Computer Science
University of Cincinnati
Cincinnati, OH 45221-0030
e-mail: {wangjun, yhu}@ececs.uc.edu

Abstract

This paper presents the design, simulation and performance evaluation of a novel reordering write buffer for Log-structured File Systems (LFS). While LFS provides good write performance for small files, its biggest problem is the high overhead from cleaning. Previous research concentrated on improving the cleaner's efficiency after files are written to the disk. We propose a new method that reduces the amount of work the cleaner has to do before the data reaches the disk. Our design sorts active and inactive data in memory into different segment buffers and then writes them to different disk segments. This approach forces data on the disk into a bimodal distribution. Most data in active segments are quickly invalidated, while inactive segments are mostly intact. Simulation results based on both real-world and synthetic traces show that such a reordering write buffer dramatically reduces the cleaning overhead, slashing the system's overall write cost by up to **53%**.

1 Introduction

Disk I/O is a major performance bottleneck in modern computer systems. The Log-structured File System (LFS) [12, 15, 16] tries to improve the I/O performance by combining small write requests into large logs. While LFS can significantly improve the performance for small-write dominated workloads, it suffers from a major drawback, namely the *garbage collection overhead* or *cleaning overhead*. LFS has to constantly reorganize the data on the disk, through a process called *garbage collection* or *cleaning*, to make space for new data. Previous studies have shown that the garbage collection overhead can considerably reduce the LFS performance under heavy workloads. Seltzer *et al.* [17] pointed out that cleaning overhead reduces LFS performance by more than 33% when the disk is 50% full. Due to this significant problem, LFS has limited success in real-world operating system environments, although it is used internally by several RAID (Redundant Array of Inexpensive Disks) systems [20, 10]. Therefore it is important to reduce the garbage collection overhead in order to improve the performance of these RAID systems and to make LFS more successful in the operating system field.

Several schemes have been proposed [9, 20] to speed up the garbage collection process. These algorithms focus on improving the efficiency of garbage collection *after* data has been written to the disk. In this paper, we propose a novel method that tries to reduce the I/O overhead during the garbage collection, by reorganizing data in two or more segment buffers, *before* data is written to the disk.

1.1 Motivation

Figure 1 shows the typical writing process in an LFS. Data blocks and inode blocks are first assembled in a segment buffer to form a large log. When the segment buffer is full, the entire buffer is written to a disk segment in a single large disk write. If LFS has synchronous operations or if dirty data in the log have not been written for 30 seconds, partially full segments will be written to the disk. When some of the files are updated or deleted later, the previous blocks of that file on the disk are invalidated correspondingly. These invalidated blocks become holes in disk segments and have to be reclaimed by the garbage collection process.

The problem with LFS is that the system does not distinguish *active data* (namely short-lived data) from *inactive data* (namely long-lived data) in the write buffer. Data are simply grouped into a segment buffer randomly, mostly according to their arrival order. The buffer is then

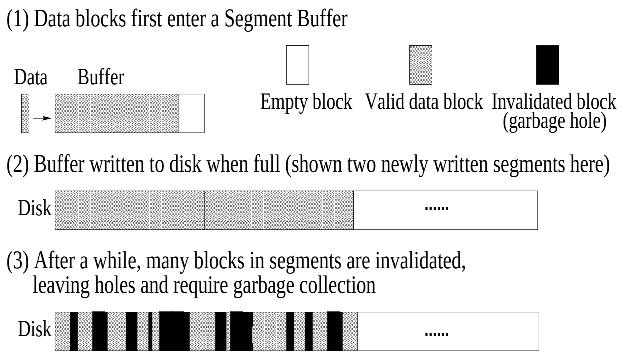
(1) Data blocks first enter a Segment Buffer
Data
Buffer
Empty block
Valid data block
Invalidated block (garbage hole)
(2) Buffer written to disk when full (shown two newly written segments here)
Disk
......
(3) After a while, many blocks in segments are invalidated, leaving holes and require garbage collection
Disk
......

Figure 1: The writing process of LFS

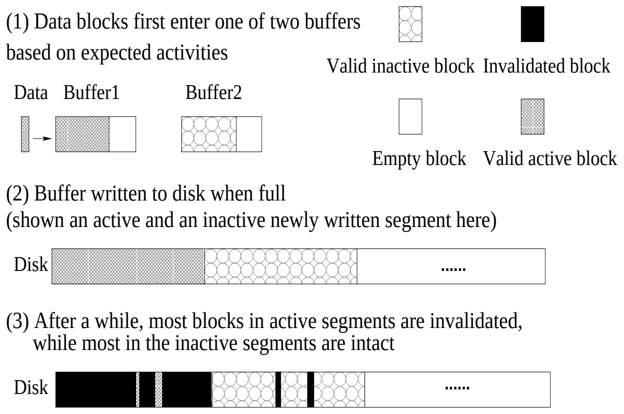
(1) Data blocks first enter one of two buffers based on expected activities
Valid inactive block
Invalidated block
Data
Buffer1
Buffer2
Empty block
Valid active block
(2) Buffer written to disk when full (shown an active and an inactive newly written segment here)
Disk
......
(3) After a while, most blocks in active segments are invalidated, while most in the inactive segments are intact
Disk
......

Figure 2: Our new scheme–WOLF

written to a disk segment when it is full. Within the segment, however, some data are active and will be quickly overwritten (therefore invalidated), while others are inactive and will remain on the disk for a relatively long period. The result is that the garbage collector has to compact the segment to eliminate the holes in order to reclaim the disk space.

1.2 Our New Scheme

Based on this observation, we propose a new method called WOLF (reordering Write buffer Of Log-structured File system) that can dramatically reduce the garbage collection overhead. Instead of using one segment buffer, we use **two or more** segment buffers(here is two), as shown in Figure 2. When write data arrives, the system sorts them into different buffers according to their expected longevity. Active data are grouped into one buffer, while less-active data are grouped into the other buffer. When the buffers are full, two buffers are written into two disk segments using two large disk writes (one write for each buffer).

Because data are sorted into active and inactive segments *before* reaching the disk, garbage collection overhead is drastically reduced. Since active data are grouped together, most of an active segment will be quickly invalidated (sometimes the entire segment will be invalidated, and the segment can be reused right away without garbage collection). On the other hand, very few data blocks in an inactive segment will be invalidated, resulting in few holes. The outcome is that data on the disk have a bimodal distribution, namely segments are either mostly full or mostly empty. Similar to Rosenblum and Ousterhout's analysis [15], this is an ideal situation. In a bimodal distribution, segments tend to be nearly empty or nearly full, but few segments are in between. The cleaner can select many nearly empty segments to clean and compact their data into a small number of segments. The old segments are then freed, resulting in a large number of available empty segments for future use. Furthermore, there is no need to waste time to clean the nearly-full segments.

Basically, while previous researchers agreed that the cleaner plays one of the most important roles in LFS, their work focused only on *making the cleaner more efficient* **after** data are written onto the disk. We believe that there exists another opportunity to improve the LFS performance. By re-organizing data in RAM **before** they reach the disk, we could also *make the system do less garbage collection work*. Traditional LFS did try to separate active data from inactive data and force a bimodal distribution, but only during the garbage collection period, long *after* files are written to the disk. Our simulation shows that significant performance gain can be obtained by applying our new method.

1.3 File Access Locality

Accurate prediction of which blocks will be invalidated soon is the key to the success of our strategy. We looked at both the temporal and spatial locality of file accessing patterns. File system accesses show strong temporal locality: many files are overwritten again and again in a short period of time. For example, Hartman and Ousterhout [7] pointed out that 36%–63% of data would be overwritten within 30 seconds and 60%–95% within 1000 seconds in the system they measured. In year 2000, Roselli et al. [14] pointed out that file accesses obey a bimodal distribution pattern: some files are written repeatedly without being read; other files are almost exclusively read. Data that have been actively written, should be put into active segments, and others into inactive segments.

File system accesses also show strong spatial locality, as many data blocks are accessed together. For example, data blocks of one file are likely to be changed together. Similarly, when a file block is modified, the inode of the file, together with the data blocks and the inode of the directory containing the file, are also likely to be updated. These blocks should therefore be grouped together in semantics such that when one block is invalidated, all or most other blocks in the same segment will be invalidated also.

1.4 Related Work

Many papers have tried to improve the LFS performance since the publication of Sprite LFS [15]. Seltzer [16] presented an implementation of LFS for BSD. Several new cleaning policies have also been presented [2, 20, 9]. In traditional cleaning policies [15], including *greedy cleaning* and *benefit-to-cost cleaning*, the live blocks in several partially empty segments are combined to produce a new full segment, freeing the old partially empty segments for reuse. These policies perform well when the disk space utilization is low. Wilkes et al. [20] proposed the hole-plugging policy. In their scheme, partially empty segments are freed by writing their live blocks into the holes found in other segments. Despite the higher cost per block, at high disk utilizations, hole-plugging does better than traditional cleaning because it avoids processing so many segments. Recently, Matthews et al. [9] showed how adaptive algorithms can be used to enable LFS to provide high performance across a wider range of workloads. These algorithms, which use hybrid policies of the above two methods, improved write performance by modifying the LFS cleaning policy to adapt to the changes in disk utilization. The system switches to a different method based on the cost-benefit estimates. They also used cached data to lower cleaning costs. Blackwell et al. [2] presented a heuristic cleaning to run without interfering with normal file access. They found that 97% of cleaning on the most heavily loaded system was done in the background. We proposed a scheme called PROFS which incorporates the knowledge of Zone-Bit-Recording into LFS to improve both the read and write performance. It reorganizes data on the disk during LFS garbage collection and system idle period. By putting active data in the faster zones and inactive data in the slower zones, PROFS can achieve much better performance for both reads and writes [19]. Lumb *et al.* applied a new technique called freeblock scheduling to the LFS cleaning process. They claimed an LFS file system could maintain ideal write performance when cleaning overheads would otherwise reduce performance by up to a factor of three [13].

In this paper, our strategy has a distinctive difference compared with above methods: WOLF works with the initial writes in the reordering write buffers which reduce the cleaning overhead **before** writes go to disk. This scheme finds a new “free” time to solve the same garbage collection problem for LFS. WOLF can be easily combined with other strategies to improve LFS performance. More importantly, it helps LFS provide high performance even in heavy loads and full disks.

Several researchers tried to improve the file system performance without using LFS. Ganger and Patt [4] proposed a method called “Soft Updates” that can eliminate the needs of 95% of synchronous writes. File system performance can be significantly improved because most writes become asynchronous and can be cached in RAM. Hu et al. proposed the Disk Caching Disk [8, 11] which can improve the performance of both synchronous and asynchronous writes. WOLF and Soft-Updates are complementary approaches: The latter improves disk scheduling in traditional file systems through aggressive caching, while WOLF addresses what to do in write caching before the data go to media.

The remainder of the paper is organized as follows. Section 2 describes our design of WOLF. Section 3 describes our experimental methodology. Section 4 shows the simulation results and analysis. Section 5 summarizes our new strategy.

2 The Design of WOLF

2.1 Writing

After the file system receives a write request, WOLF decides if the requested data is active or inactive and puts the write data into one of the segment buffers accordingly. (We discuss how to do this in Section 2.2.) Old data in a disk segment will also be invalidated. The request is then considered complete.

When the write buffers are full, all buffers are written to disk segments in large write requests in order to amortize the cost of many small writes. Since WOLF contains several segment buffers and each buffer is written into a different disk segment, several large writes occur during the process (one large write for each buffer).

As in the LFS, WOLF also writes buffers to the disk when one of the following conditions is satisfied, even when the buffers are not full:

- A buffer contains modifications that are more than 30 seconds old.
- A *fsync* or *sync* occurs

Since the LFS uses a single segment buffer, when a buffer write is invoked, only one large write is issued. WOLF maintains two or more segment buffers. To simplify the crash recovery process (discussed in Section 2.3), when WOLF has to write data to the disk, *all* segment buffers in RAM will be written (logged) to the disk at the same time. While the logging process contains several large disk write operations since each segment buffer is written to a different disk segment, WOLF considers the log operation *atomic*. A logging is considered successful only if all segment buffers are successfully written to the disk. The atomic logging feature means that we can view the multiple physical segments of WOLF as a *single virtual segment*.

The atomic writing of multiple segments can easily be achieved with a timestamp. All segments written together will have the same timestamp and the same "number of segments written together" field. During crash recovery, the system searches for the segments with the latest timestamp. If the number of segments with the same latest timestamp matches the "number of segments written together" field, then the system knows that the last log-writing operation was successful.

2.2 Separating Active and Inactive data

One of the important problems in the design of WOLF is how to find an efficient and easy-to-implement method that can separate active data from inactive data and put them into different buffers accordingly.

2.2.1 An Adaptive Grouping Algorithm

We developed a heuristic learning method for WOLF. The tracking process implements a variation of the least-recently used algorithm with frequency information. Our algorithm is similar to virtual memory page-aging techniques.

To capture the temporal locality of file accesses, each block in the segment buffers has a reference count associated with it. This number is incremented when the block is accessed. The count is initialized to zero and is also reset to zero when the file system becomes idle for a certain period. We call this period as *time-bar*. It is initialized to 10 minutes[1]. If the age of this block exceeds current time-bar, WOLF will reset the reference count of this block to zero. WOLF only does this zero clearing in write buffers. The value of the count indicates the active level of the block in *most recent active period*, which starts since the time-bar. The higher the value of the count, the more active a block is. The Time-bar could be adaptively tuned for the various incoming accesses. When the system identifies that there is no significant difference among the blocks' active ratios in the reorder buffers, which means the 90% reference counts of blocks are equal, the time-bar will be doubled. If most blocks have too different active ratios, when only 10% reference counts of blocks are equal, the time-bar will be halved. The Time-bar makes the reordering buffers work heuristically for different workloads. Active data are then put into the active segment buffer, and other data in the inactive buffer.

If two blocks have the same reference counts, then spatial locality is considered. If the two blocks satisfy one of the following conditions, they will be grouped into the same segment buffer:

- If the two blocks belong to the same file.
- If the two blocks belong to files in the same directory.

If none of the above conditions is true, the blocks are randomly put into buffers.

The overhead of this learning method is low. Most active blocks have no more than a hundred accesses in a short period. Only a small amount of additional bits

[1]For different workloads, this threshold may be different. We choose this value for most workloads. This threshold works well when active data live less than 10 minutes and inactive data lives more than 10 minutes

are needed for each block. *Time-bar* is managed by the reordering buffer manager with little overhead. WOLF only resets the reference count in the reordering buffers.

2.2.2 Data Lifetimes

In order to choose the proper threshold for different workloads, we calculate the byte lifetime by subtracting the byte's deletion time from its creation time. This "deletion-based" method was used by [1] in which all deleted files are tracked. For considering the effects of overwrites, we measured byte lifetime rather than file lifetime. Figure 3 tells the byte lifetime of four real-world workloads in details(these traces will be described in section 3.2.1).

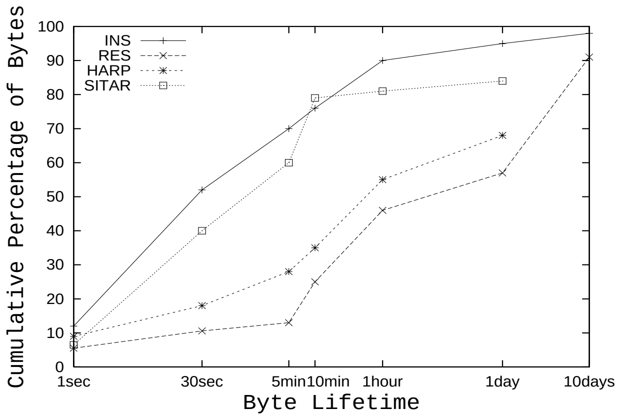

Figure 3: Byte Lifetime of Four Real-world Workloads

From the picture, we can see the active data's lifetimes shows various behaviors in different workloads. More than 70% of the data in INS and Sitar traces have a lifetime less than 10 minutes. Around 35% of the data in RES and Harp traces have a lifetime less than 10 minutes. Since the lifetime of active data varies in different workloads, it is necessary to develop this adaptive grouping algorithm to separate active data and inactive data for different workloads.

2.3 Consistency and Crash Recovery

In additional to LFS' high performance, another important advantage of LFS is fast crash recovery. LFS uses checkpoints and maintains the order of updates in the log format. After a crash, the system only has to *roll forward*, reading each partial segment from the most recent checkpoint to the end of the log in write order, which involves incorporating any modifications that occurred. Thus there is no need to perform a time-consuming job like $fsck$.

In WOLF, data in memory are re-grouped into two or more segment buffers and later written into two or more disk segments. As a result, the original ordering information may be lost. To keep the crash recovery process simple, WOLF employs the following strategies:

1. While data blocks are reordered by WOLF to improve the performance, their original arrival ordering information is kept in a data structure and written to the disk in the summary block together with each segment.
2. While WOLF maintains two or more segment buffers, its atomic logging feature (discussed in Section 2.1) means that these multiple physical buffers can be viewed as a *single virtual segment*.

Since WOLF maintains only a single *virtual segment* which is logged atomically, and the information about original arrival orders of data blocks in the virtual segment is preserved, crash recovery in is nearly as simple as in LFS.

2.4 Reading

WOLF only changes the write cache structures of LFS. The read operations are not affected. As a result, we expect that WOLF has similar read performance as that of LFS when the system is lightly loaded. When the system is heavily loaded, WOLF should have better read performance because of its more efficient garbage collection process that reduces the competition for disk bandwidth.

2.5 Garbage Collection

WOLF does not completely eliminate garbage, therefore garbage collection is still needed. Benefit-to-Cost cleaning algorithm works well in most cases while hole-plugging policy works well when the disk segment utilization is very high. Since previous research shows that a single cleaning algorithm is unlikely to perform equally well for all kinds of workloads, we used an adaptive approach similar to the Matthews' method [9]. This policy automatically selects either the *benefit-to-cost* cleaner or the *hole-plugging* method depending on the cost-benefit estimates.

In WOLF, the cleaner runs when the system is *idle* or disk utilization exceeds a *high water-mark*. In our simulation, the high water-mark is when 80% of the disk is full, and *idle* is defined as the file system has no activities in 5 minutes. The amount of data that the cleaner may process at one time can be varied. In this paper, we allowed the cleaner to process up to 20 MB at one time. To calculate the benefit and overhead of garage collection, we used the following mathematical model. These formula were developed by Matthews *et al.* (See more details in [9]).

The benefit-to-cost ratio is defined as follows:

$$\frac{benefit}{cost} = \frac{(1 - utilization) * age\ of\ segment}{(1 + utilization)}$$

Here utilization represents the ratio of the live bytes to one segment size. Specifically, the cost-benefit values of cleaning and hole-plugging policies are calculated as follows:

$$CostBenefit_{Cleaning} = \frac{TransferTime_{Cleaning}}{SpaceFreed_{Cleaning}}$$

$$CostBenefit_{Plugging} = \frac{TransferTime_{Plugging}}{SpaceFreed_{Plugging}}$$

The adaptive policy always picks up segments with the lower Cost-Benefit estimates to clean. Segments with more garbage (hence very low segment utilization and high benefit-to-cost ratios) will be cleaned first. Older segments will also be cleaned first, as data in younger segments will have a better chance to be invalidated in the future.

Because WOLF's buffer manager separates the active data from inactive data which leads to a bimodal disk segment layout, both the benefit-to-cost and hole-plugging methods can benefit from this nice layout. For benefit-to-cost, since most active segments are mostly garbage (hence very low utilization), their benefit-to-cost ratios are very high. These segments will be cleaned first to yield many blank segments. For hole-plugging, when the adaptive cleaner switches to this method (which will tend to occur in very high disk utilization), cleaner uses the least utilized segments to plug the holes in the most utilized segments. WOLF simply reads the few remaining live bytes from an active disk segment and plug them into the few available slots of an inactive disk segment (very high segment utilization).

3 Experimental Methodology

We used trace-driven simulation experiments to evaluate the effectiveness of our proposed new design. Both real-world and synthetic traces are used during simulation. In order to make our experiments and simulation results more convincing, we use four different real-world traces and four synthetic traces in the comprehensive covering fields.

3.1 The Simulators

The WOLF simulator contains more than 10,000 lines of C++ code. It consists of an LFS simulator, which acts as a file system, on top of a disk simulator. The disk model is ported from Ganger's disk simulator [5]. Our LFS simulator is developed based on Sprite LFS. We ported the LFS code from the Sprite LFS kernel distribution and implemented a trace-driven class to accept trace files. By changing a configuration file, we can vary important parameters such as the number of segment buffers, the segment size and the read cache size. In the simulator, data is channeled into the log through several write buffers. The write buffers are flushed every 30 seconds of simulated time to capture the impact of partial segment writes. A segment usage table is implemented to maintain the status of disk segments. Meta-data structures including summary block and inode map are also developed. We built a checkpoint data structure to save blocks of inode map and segment usage table periodically.

The disk performance characteristics are set in Disksim's config files. We chose two disks for testing, a small (1 GB capacity) HP2247A disk and a large (9.1 GB) Quantum Atlas10K disk. The small HP2247A was used for Sitar and Harp traces, because the two traces have small data-sets (total data accessed < 1 GB). A small disk is needed in order to observe the garbage collection activities. The large disk was used for all other traces. Using two very different disks also helps us to investigate the impacts of disk features like capacity and speed on WOLF performance. The HP2247A disk' spindle speed is set to 5400 RPM. The read-channel bandwidth is 5 MB/sec. Its average access time is 15 ms. The Quantum Atlas10K has a 10024 RPM spindle speed. Its read-channel bandwidth is 60 MB/sec and average access time is 6 ms.

3.2 Workload Models

The purpose of our experiments is to conduct a comprehensive and unbiased performance study of the proposed scheme and compare the results with that of unmodified LFS. We paid special attention to select the traces. Our main objective was to select traces that match as close to realistic workloads as possible. At the same time, we also wanted to cover as wide a range of environments as possible. The trace files that have been selected in this paper are discussed below.

3.2.1 Real-world Traces

Four real-world file system traces we used in our simulation. We got two sets of real-life traces from two different universities to validate our results. Two of them came from University of California, Berkeley, called **INS** and **RES** [14]. **INS** came from a collection from a group consisting of 20 machines located in labs for undergraduate classes. **RES** was attained from 13 desktop machines of a research group. **INS** and **RES** were recorded over 112 days from September 1996 to December 1996. Both traces came from their clusters running HP-UX 9.05. The other set of two traces, from University of Kentucky, contain all disk activities on two SunOS 4.1 machines during ten days for **Sitar** trace and seven days for **Harp** trace[6]. **Sitar** trace represents an office environment while **Harp** reflects common program development activities. More specifically, **Sitar** trace is a collection of file accesses by graduate students and professors doing work such as emailing, compiling programs, running LaTeX, editing files, and so on. **Harp** trace shows a collaboration of two graduate students working on a single multimedia application. Because **Sitar** and **Harp** have a small amount data, we use the small disk model with these two real-world traces. Notice in the experiments, we expand Sitar and Harp by appending files with same access pattern in original traces but with different file names in order to explore the system behavior under different disk utilizations. For large traces with more than 10GB data traffic, we do not use this procedure.

These real-world traces are described in more detail in Table 1.

3.2.2 Synthetic Traces

While real-world traces give a realistic representation of some real systems, synthetic traces have the advantage of isolating specific behaviors not clearly expressed in recorded traces. We therefore also generated a set of synthetic traces. We varied the trace characteristics as much as possible in order to cover a very wide range of different workloads.

We generated the following four sets of synthetic traces:

1. Uniform Pattern *(Uniform)*

   Each file has equal likelihood of being selected.

2. Hot-and-cold Pattern *(Hot-cold)*

   Files are divided into two groups. One group contains 10% of files; it is called *hot* group because its files are visited 90% of the time. The other group is called *cold*; it contains 90% of the files but they are visited only 10% of the time. Within groups each file is equally likely to be visited. This access pattern models a simple form of locality.

3. Ephemeral Small File Regime *(Small Files)*

   This suite contains small files and tries to model the behavior of systems such as the electronic mail or the network news systems. The sizes of files are limited from 1 KB to 1 MB. They are frequently created, deleted and updated. The data lifetime of this suite is the shortest one in this paper (90% of byte lifetimes are less than 5 minutes).

4. Transaction Processing Suite *(TPC-D)*

   This trace consists of a typical TPC-D benchmark which accesses twenty large size database files from 512 MB to 10 GB. The database files consist of the different number of records ranged from 2,000,000 to 40,000,000. Each record is set to 100 bytes. Most transaction operations are queries and updates in this benchmark. The I/O access pattern is random writes followed by sequential reads. Random updates are applied to the active portion of the database. And then sometime later, large sweeping queries read relations sequentially [18]. This represents the typical I/O behavior of a decision support database. In this trace, we use sequential file reads to simulate 17 SQL queries for business questions. As for implementing TPC-D update functions, we generate random writes to represent following categories: updating 0.1% of data per query, inserting new sales data with 0.1% of table size and deleting old sales data of 0.1% of table size.

Features	INS	RES	SITAR	HARP	Uniform	Hot-cold	TPC-D	SmallFS
Data read(MB)	94619	52743	213	520	8000	8000	8000	8000
Data write(MB)	16804	14105	183	249	8000	8000	4000	8000
Read:Write ratio	5.6	3.7	1.16	2.08	1.0	1.0	2.0	1.0
Reads(thousands)	71849	9433	57	26	800	800	800	4000
Writes(thousands)	4650	2216	49	12	800	800	400	4000
File Size($<$ 16KB)	82.8%	63%	80%	73	80%	80%	0%	95 %
File Size(16KB–1MB)	16.9%	36.5%	19.96%	26.98%	19.95%	19.95%	0%	5%
File Size(1MB+)	0.2 %	0.5 %	0.04 %	0.02 %	0.05%	0.05%	100%	0%

Table 1: Four real-world traces and Four synthetic traces

The other information of these four synthetic traces can be seen in Table 1.

4 Simulation Results and Performance Analysis

In order to understand the insight effect of WOLF, we compare our design with the most recent LFS using adaptive method which is the baseline system. The reason is that we want to explore the effect with our reordering write buffers rather than the adaptive cleaning policy. Therefore, two compared systems use the *same* adaptive garbage cleaning strategy. In experiments, system automatically selects either *benefit-to-cost* or the *hole-plugging* depending on the cost-benefit estimates. WOLF separates active data from cold data to generate active/inactive segments in initial writes. The different disk layouts in two systems lead to different performance.

In our experiments of this paper, we set several default parameters unless specified: a 64 MB read cache, each disk segment is 256 KB and each segment buffer is 256 KB.

4.1 Overall Write Cost

Write cost is the metric traditionally used in evaluating LFS write performance. It only considers the effect of the number of segments. Matthews et al. pointed out segment size also plays a larger role in the write performance. They described a way to quantify this trade-off between amortizing disk access time across larger transfer units and reducing cleaner overhead: *Overall Write Cost*, which captures both the overhead of cleaning as well as the bandwidth degradation caused by seek and rotational latency of log writes [9].

In this paper we used this new metric – overall write cost to evaluate WOLF performance. The following formula are adapted from [9]:

First, two terms, *write cost* and *Transfer Inefficiency* ($Ineff_{Xfer}$) are defined:

$$WriteCost = \frac{Segments\,Transferred_{Total}}{Segments\,Transferred_{NewData}}$$

$$= \frac{SegsW_{NewData} + SegsR_{Clean} + SegsW_{Clean}}{SegsW_{NewData}}$$

Here $SegsW_{NewData}$ is the total number of segments written to the disk caused by new data. $SegsR_{Clean}$ and $SegsW_{Clean}$ are the total numbers of segments read and written by the cleaner, respectively. This term describes the overhead of cleaning process.

$$Ineff_{Xfer} = AccessTime \times \frac{DiskBandwidth}{SegmentSize} + 1$$

$Ineff_{Xfer}$ measures the bandwidth degradation caused by seek and rotational delays of log writes. AccessTime represents the average disk access time.

And finally,

$$Overall\ Write\ Cost = WriteCost \times Ineff_{Xfer}$$

4.1.1 Performance under Different Workloads

In order to understand how the WOLF and LFS perform under different workloads, results for the four synthetic traces and four real-world traces are compared in Figure 4.

It is clear from the figure that the WOLF significantly reduces the overall write cost compared to the LFS. The new design reduces the overall write cost by up to 53%. The overall write cost is most reduced when the disk space utilization is high. When the disk becomes more full, the garbage collection is more important. WOLF plays the more important role in reducing garbage on the disk.

Although the eight traces have different characteristics, we can see that the performance of WOLF is not sensitive to the variation in workloads. This derives from our heuristic reorganizing algorithm. On the other hand, LFS performs especially poor for the TPC-D workload because of its random updating behavior. This is not a surprise. Similar behavior was observed by Seltzer and Smith in [17]. WOLF, on the other hand, significantly reduces the garbage collection overhead so it still performs well under TPC-D.

4.1.2 Effects of the Number of Segment Buffers with Real-world Traces

Figure 5 shows the results of the overall write cost versus disk utilization for the four real-world traces. We varied the number of segment buffers of WOLF from 2 to 4. We also varied the segment buffer size of the LFS from 256 KB to 1024 KB.

Increasing the number of segment buffers in WOLF would slightly reduce the overall write cost but does not have a significant impact on the overall performance.

The reason we studied LFS with different segment buffer sizes, is to show that the performance gain of WOLF is not due to the increased buffer numbers (hence the increased total buffer size). The separated active/inactive data layout on disk segments contributes to the performance improvement. In fact, for LFS, increasing the segment buffer sizes may actually increase the overall write cost. This observation is consistent with previous studies [9, 15]

Note that because WOLF uses more segment buffers than the LFS does, data may stay in RAM longer. However, this does not poses a reliability problem. As discussed before, in WOLF, if the segment buffers contain data older than 30 seconds, they will be flushed to the disk, just as LFS.

4.1.3 Effects of Segment Sizes with Real-world Traces

The size of the disk segment is also a substantial factor on the performance of both WOLF and LFS. If the size of the disk segment is too large, it would be a little difficult to find enough active data to fill one segment and enough inactive data for another segment. The result will be active data and inactive data are mixed together in a large segment, resulting in poor garbage collection performance. The limited disk bandwidth will also have a negative impact on the overall write cost when the segment buffer size exceeds a threshold. On the contrary, if the segment size is too small, the original benefit of LFS, namely taking the advantage of large disk transfer, is lost.

Figure 6 shows the simulation results with the overall write cost versus the sizes of segment buffers. We can see that for both WOLF and LFS, a segment between 256-1024 KB is good for these kind of workloads.

4.1.4 Segment Utilization Distribution

In order to provide insights into understanding why WOLF significantly outperforms the LFS, we also compared the segment utilization distributions of WOLF and LFS. Segment utilization is calculated by the total live bytes in the segment divided by the size of this segment.

Figure 7 shows the distribution of segment utilizations under the four real-world traces. We can see the obvious bimodal segment distribution in WOLF when compared to the LFS. Results for other workloads are similar. The nice bimodal distribution is the key to the performance advantage of WOLF over the LFS.

4.2 Read/Write Latency

In previous discussion, we used *overall write cost* as the performance metric. Overall write cost is a direct measurement of system efficiency. We have shown that WOLF performs encouragingly better than LFS, as the former has much smaller overall write cost than the latter.

However, end-users would be more interested in user-measurable metrics such as the access latencies [3]. Overall write cost quantifies the additional I/O overhead when LFS does the garbage cleaning. The LFS performance is very sensitive to this overhead. To see whether the low overall write cost in WOLF can be translated to low access latencies, we also measured the average file read/write response times in the file system level. We collected the total file read/write latencies and divided the total number of file reads/writes requests. All these results include the cleaning overhead. The results are presented in this subsection.

4.2.1 Write Latencies

Figure 8(a) shows the file write performance of LFS and WOLF under eight traces. Figure 8(b) plots the performance improvement of WOLF over LFS. We can see that WOLF significantly enhances the write performance by 27–35.5%, in terms of improved response times. The lower overall write cost in WOLF directly leads to a smaller write response time. The Hot-cold trace achieves the best improvement because of its good active behavior.

4.2.2 Read Performance

Figure 9(a) shows the file read performance of LFS and WOLF under eight traces. Figure 9(b) plots the performance improvement of WOLF over LFS. The results show that, for most traces, the read performance of WOLF is at least comparable to that of LFS. This is expected, as WOLF does not directly affect the read operations of LFS. Although WOLF changes the physical layout on disk for LFS, WOLF's grouping algorithm includes the similar policy which is used in locality-grouping rules of regular LFS, such that files in same directory are put in same segment and *etc.*, WOLF does not have much impact on the read performance when the load is light. When the load is heavy, we may see a little better read performance of WOLF than that of LFS because WOLF reduces the cleaning overhead so that WOLF ameliorates the competition of disk bandwidth. RES and TPC-D got little loss for their more random reads because random reads have poor spatial locality which results in much longer disk seeks and rotations during garbage collection.

4.3 Implication of Different Disk Models

From the results of sections 4.1 and 4.2, we can see WOLF achieves significant performance gains for both the small/slow and the large/fast disk models. The results suggest that the disk characteristics do not have a direct impact on WOLF. While the absolute performance parameters may vary on different disk models, the overall trend is clear: WOLF can markedly reduce garbage collection overhead under many different workloads on different disk models.

5 Conclusion and Future Work

We have proposed a novel reordering write buffer design called WOLF for the Log-structured File System. WOLF improves the disk layout by reordering the write data in segment buffers *before* writing data to the disk. By utilizing an adaptive algorithm that separates active data from inactive data, and taking advantages of file temporal and spatial localities, the reordering buffer forces actively-accessed data blocks into one hot segment and inactive data into another cold segment. Since most of the blocks in active segments will be quickly invalidated while most blocks in inactive segments will be left intact, data on the disk form a good bimodal distribution. This bimodal distribution significantly reduces the garbage collection overhead.

Because WOLF works before initial writes go to disk, it can be integrated with other strategies smoothly to improve LFS performance. By reducing cleaning overhead, WOLF ameliorates the competition of disk bandwidth. Simulation experiments based on a wide range of real-world and synthetic workloads show that our strategy can reduce the overall write cost by 53% and improve write response time by 35.5%. The read performance is generally better than or comparable to the LFS. Our scheme still guarantees fast crash recovery, a key advantage of LFS.

We believe that our method can significantly improve the performance of those IO systems (such as some RAIDs) that use the LFS technology. It may also increase the chance of LFS success in the OS environments like Linux. Moreover, since logging is a commonly used technology to improve the I/O performance, we believe that our new scheme will have a broad impact on high performance I/O systems as well. We also plan to apply this technique to other general file systems like FFS in the future.

Acknowledgments

This work is supported in part by the National Science Foundation Career Award CCR-9984852, and an Ohio Board of Regents Computer Science Collaboration Grant. We would like to thank our shepherd Richard Golding for his many valuable suggestions on the final version of this paper. We would also like to thank the anonymous reviewers for their useful comments.

References

[1] BAKER, M. G., HARTMAN, J. H., KUPFER, M. D., SHIRRIFF, K. W., AND OUSTERHOUT, J. K. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Oct. 1991), Association for Computing Machinery SIGOPS, pp. 198–212.

[2] BLACKWELL, T., HARRIS, J., AND SELTZER, M. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the 1995 USENIX Technical Conference: January 16–20, 1995, New Orleans, Louisiana, USA* (Berkeley, CA, USA, Jan. 1995), USENIX Association, Ed., USENIX, pp. 277–288.

[3] ENDO, Y., WANG, Z., CHEN, J. B., AND SELTZER, M. Using latency to evaluate interactive system performance. In *Proceedings of the 1996 Symposium on Operating System Design and Implementation* (Seattle, WA, Oct. 1996).

[4] GANGER, G. R., AND PATT, Y. N. Metadata update performance in file systems. In *USENIX Symposium on Operating System Design and Implementation (OSDI)* (Nov. 1994), pp. 49–60.

[5] GANGER, G. R., AND PATT, Y. N. Using system-level models to evaluate I/O subsystem designs. *IEEE Transactions on Computers 47*, 6 (June 1998), 667–678.

[6] GRIFFIOEN, J., AND APPLETON, R. The design, implementation, and evaluation of a predictive caching file system. Tech. Rep. CS-264-96, Department of Computer Science, University of Kentucky, June 1996.

[7] HARTMAN, J. H., AND OUSTERHOUT, J. K. letter to the editor. *Operating Systems Review 27*, 1 (1993), 7–9.

[8] HU, Y., AND YANG, Q. DCD—disk caching disk: A new approach for boosting I/O performance. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA'96)* (Philadelphia, Pennsylvania, May 1996), pp. 169–178.

[9] MATTHEWS, J. N., ROSELLI, D., COSTELLO, A. M., WANG, R. Y., AND ANDERSON, T. E. Improving the performance of log-structured file systems with adaptive methods. In *Sixteenth ACM Symposium on Operating System Principles (SOSP '97)* (1997).

[10] MENON, J. A performance comparison of RAID-5 and log-structured arrays. In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing* (Aug. 1995), pp. 167–178.

[11] NIGHTINGALE, T., HU, Y., AND YANG, Q. The design and implementation of DCD device driver for UNIX. In *Proceedings of the 1999 USENIX Technical Conference* (Monterey, CA, Jan. 1999), pp. 295–308.

[12] OUSTERHOUT, J., AND DOUGLIS, F. Beating the I/O bottleneck: A case for log-structured file systems. Tech. Rep. UCB/CSD 88/467, Computer Science Division, Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720, Oct. 1988.

[13] R.LUMB, C., SCHINDLER, J., GANGER, G. R., AND NAGLE, D. F. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Proceedings of the 2000 Conference on Operating System Design and Implementation (OSDI)* (San Diego, Oct. 2000).

[14] ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Conference* (June 2000).

[15] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 26 – 52.

[16] SELTZER, M., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. An implementation of a log-structured file system for UNIX. In *Proceedings of Winter 1993 USENIX* (San Diego, CA, Jan. 1993), pp. 307–326.

[17] SELTZER, M., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: A performance comparison. In *Proceedings of 1995 USENIX* (New Orleans, LA, Jan. 1995), pp. 249–264.

[18] TRANSACTION PROCESSING PERFORMANCE COUNCIL. TPC benchmark D standard specification, April 1995. Waterside Associates, Fremont, CA.

[19] WANG, J., AND HU, Y. Profs-performance-oriented data reorganization for log-structured file system on multi-zone disks. In *Proceedings of the 9th Int'l Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems(MASCOTS'2001)* (2001).

[20] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRaid hierarchical storage system. *ACM Transactions on Computer Systems 14*, 1 (Feb. 1996), 108–136.

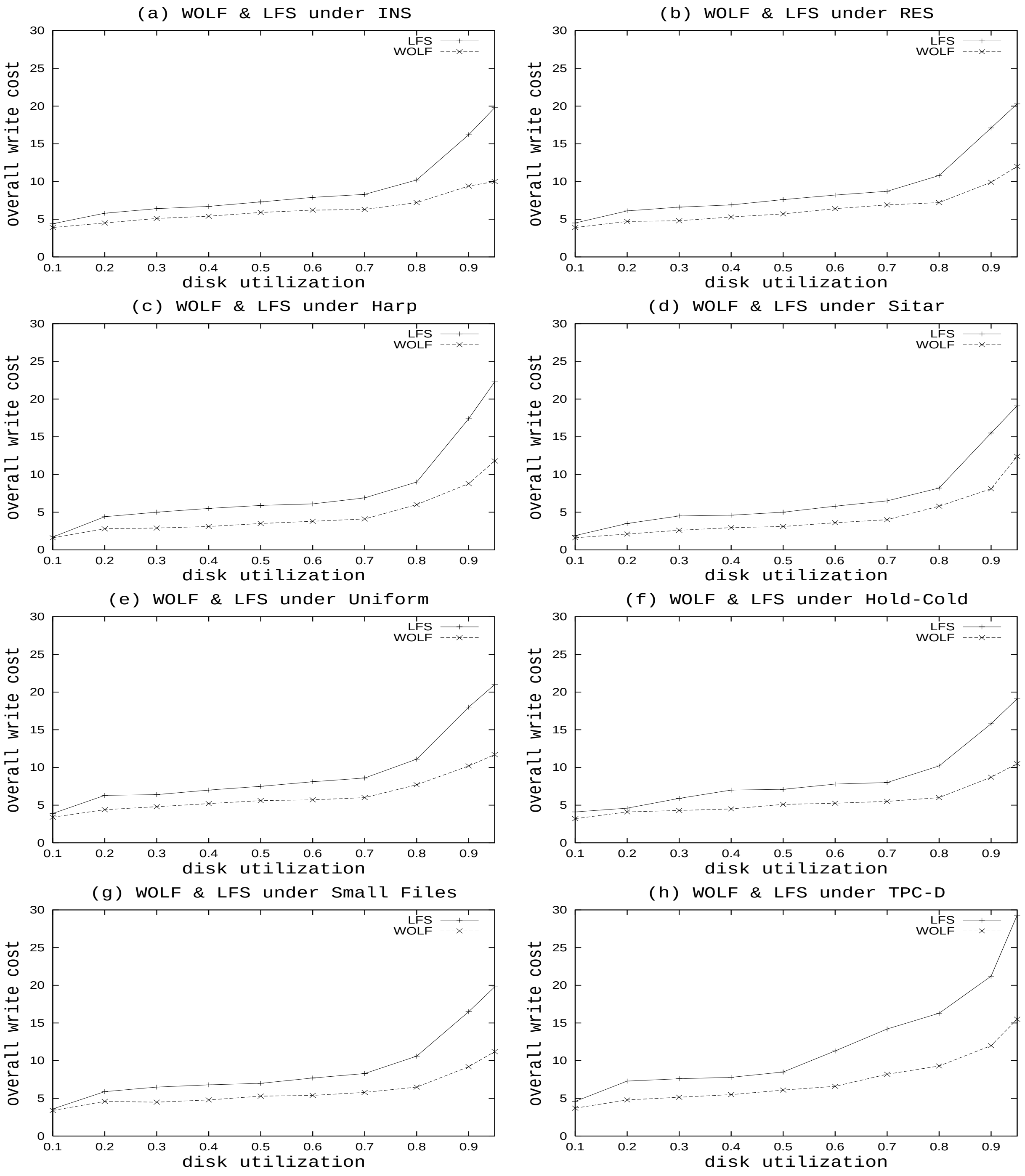

Figure 4: Overall Write-cost versus Disk Utilization under different workloads. *WOLF with 2 segment buffers.*

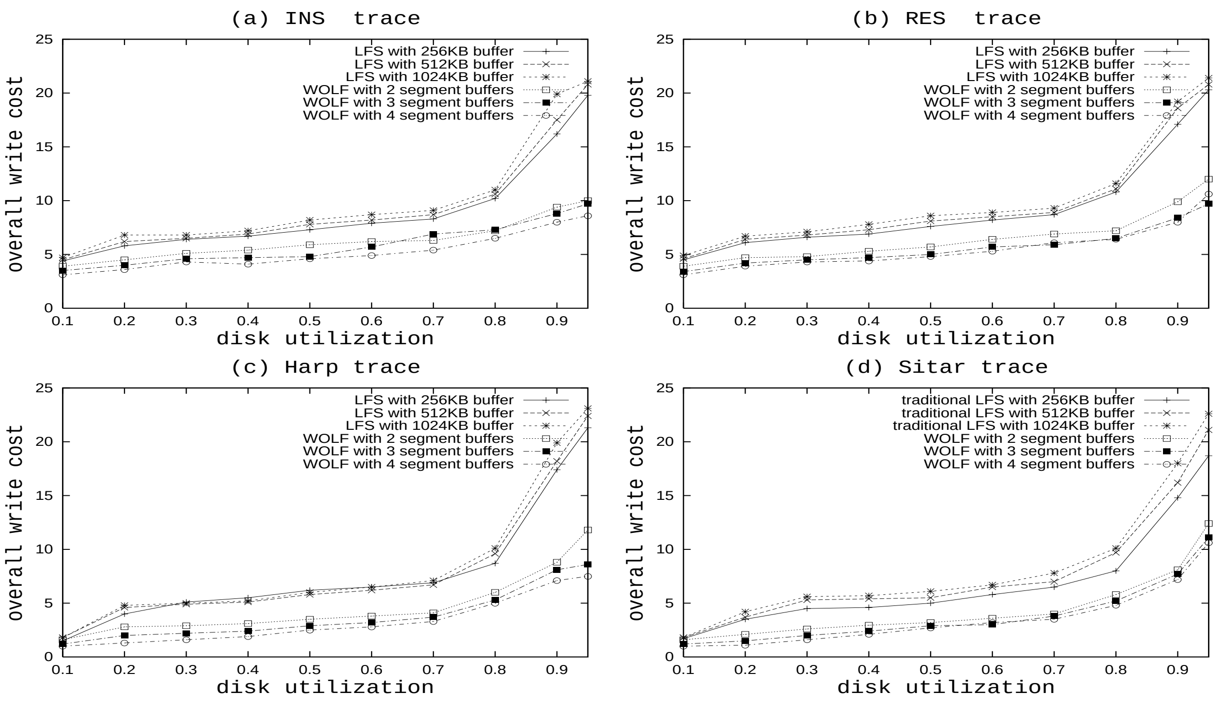

Figure 5: Overall write cost versus Disk Utilization.

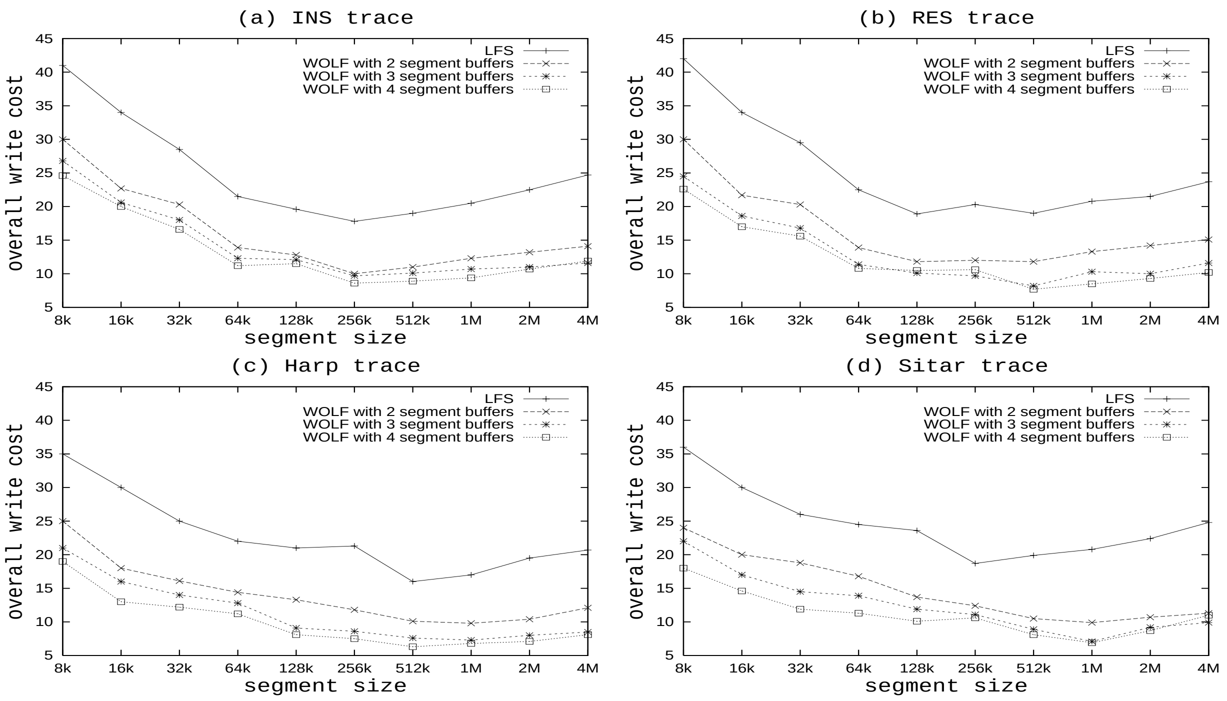

Figure 6: Overall write cost versus Segment Sizes. *Disk utilization is 95%.*

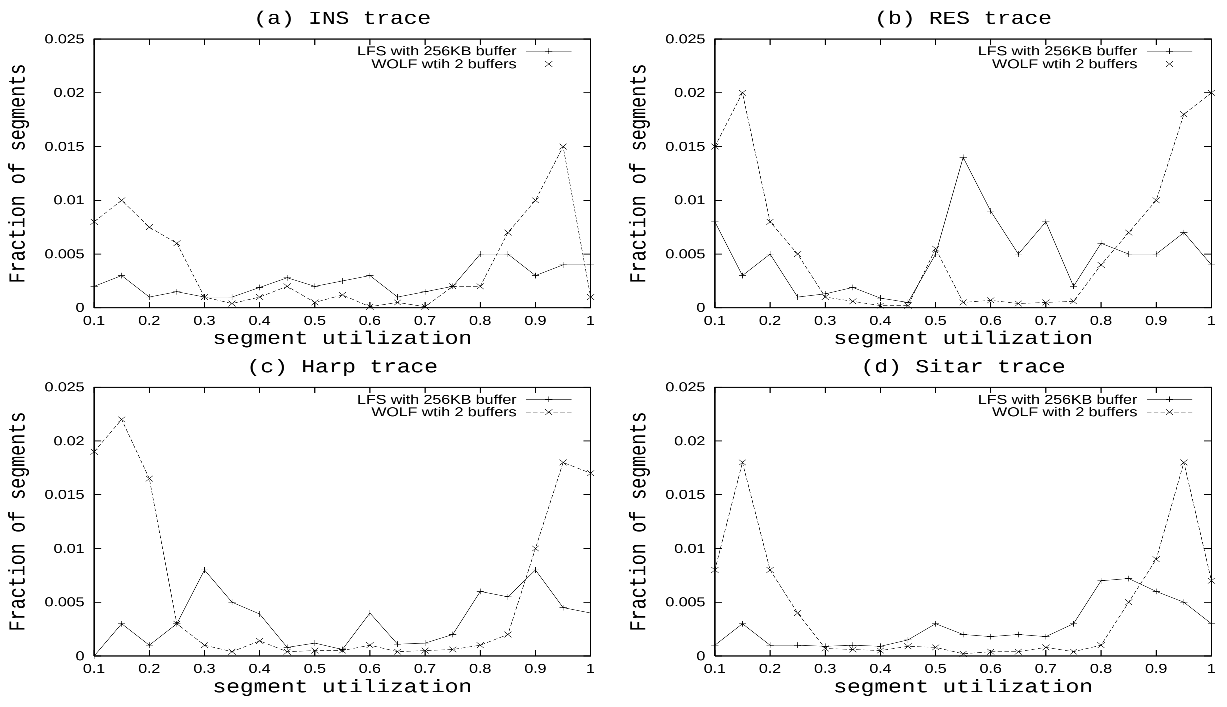

Figure 7: Segment Utilization versus Fraction of Segments. *Disk utilization is 80%.*

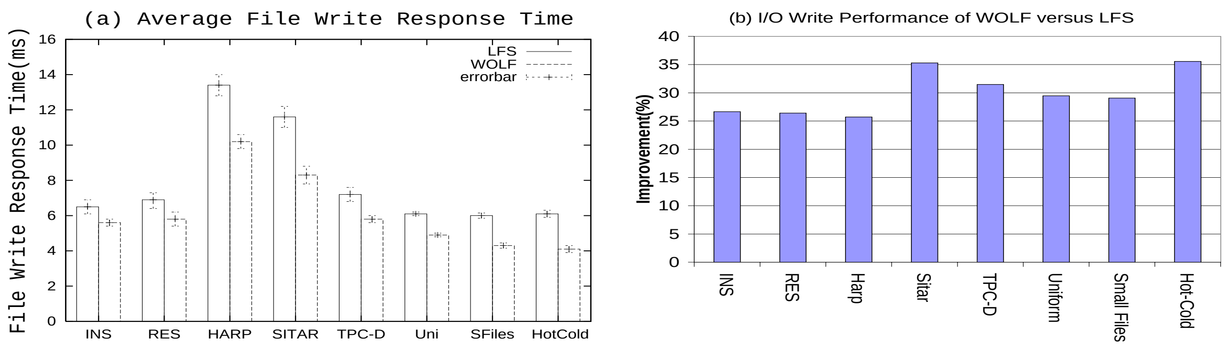

Figure 8: Average File Write Response Time. Errorbar shows the standard deviation. *Disk utilization is 90%.*

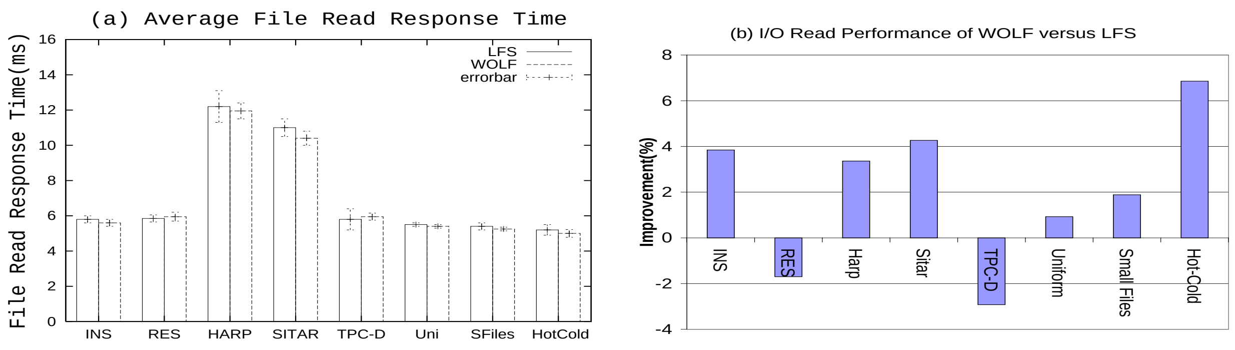

Figure 9: Average File Read Response Time. Errorbar shows the standard deviation. *Disk utilization is 90%.*