Fast Log Writes using Disk Mimic

Ashok Anand & Sayandeep Sen

Computer Sciences Department
University of Wisconsin, Madison

Abstract

We investigate a suite of algorithms for reducing the **Small synchronous disk writes** *overhead, which are an important & frequently occurring workload in journalling filesystems and transactional database systems. The studied algorithms try to optimize the performance by trying to predict the disk head position. We find the current state-of-art algorithms/systems either memory inefficient or bad at predicting the actual disk head position. We carry out an emulation based performance study to understand the trade-offs between memory overhead and accuracy. We then design algorithms with configurable memory and accuracy parameters (interdependent). Finally we design and implement a pseudo driver for Linux based systems, which can be integrated with the existing Linux based journalling filesystems.*

1 Introduction

The disk and storage systems have become the main speed bottleneck of computer systems, and hence optimizing their performance, has been an important area of research. A multitude of techniques and optimizations have been developed to optimize the performance of file-systems. For example, the Berkeley Fast File System (FFS) [4] mandates the placement of files belonging to same directory on same cylinder groups, in disk to reduce reads with spatial locality. The Log-structured File System (LFS) [7] organizes all file system data and metadata in a log, and writes it asynchronously to disk in large segments, improving the small write performance over FFS.

However, one important workload that still incurs a performance penalty is the: small, synchronous disk writes. These workloads are typically generated by transactional applications such as databases and messaging systems, where a synchronous write is used to ensure data persistence. Synchronous writes are also occur in journalling filesystems, such as ext3 [8, 1] etc. The journalling file systems utilize synchronous writes to ensure, a proper ordering of operations on disk, so as to ensure data recovery in case of a crash.

Synchronous writes are hard to optimize as, unlike other disk operations, their cost cannot be reduced by caching. The cost of a synchronous write is generally dominated by the time it takes to position the disk head, especially for the small writes that are common in transactional applications.

To motivate the need for designing efficient optimizations for small write operations, we first present the problem in context of data logging in file systems. To understand the problem of small writes in case of transactional filesystems, we first briefly describe how transactional filesystems work. The key idea behind a journalling filesystem is to write information pertaining to pending updates (ie, updates not actually written on disks) to the log, or journal, on disk. The log can either be stored at a fixed disk location, or it may be stored within a file. The file system meta-data and data are also written to fixed locations on the disk. Forcing journal updates to disk before updating the actual disk locations (which is a time consuming process) leads to efficient crash recovery, as the system can be brought to a consistent state by simply scanning the journal and redoing the incomplete updates.

The relative performance of the journalling mostly depends on the workload. In case a large amount of data is being written back to the disk, journalling filesystems

perform poorly as the data blocks are written twice to the disk. However, data journalling performs relatively well when small, random writes are performed. In this case, journalling filesystems, write the data sequentially to the log (which it does efficiently), while the actual updates to disk locations are done in background and hence do not affect the performance of the system. While, the system saves the penalty of carrying out small random writes to the disk, it still incurs a rotational penalty while performing the sequential writes to the log. The reason behind the aforementioned penalty is the way the journalling filesystems actually carry out the write operation. While updating log, the journalling system divides a write into three parts, a) a descriptor block, b) the actual data and c) the commit block. The filesystem first writes the descriptor block, which has metadata information about the corresponding log entry. Once this block is persistently written to the disk the system then writes the data block synchronously to the disk and finally it writes the commit block to the disk. The performance penalty is incurred because of the fact, that the filesystem mandates that these blocks be placed consecutive to each other on the log and as the disk head moves out by some extent before the filesystem can issue the consequent write request. This causes the disk head to take a full rotation before it can write the next block.

Higher-end systems attempt to overcome the synchronous write problem by using expensive hardware such as NVRAM to buffer the synchronous writes [3]. Unfortunately, low-cost computer systems cannot utilize such modifications because of their excessive costs. Another line of research [6, 2] tries to reduce the cost of small writes by directing the actual writes to those regions of the disk which are about to be written out. These, schemes rely on a predictor of the disk head to direct the writes and incur minimum response delays.

In this paper we concentrate on making the small synchronous disk write operations efficient for log writing. Our solution approach is along the the lines of [6, 2]. However, existing solutions either do not predict the disk head position well or predict disk head position well with high memory overhead. Our contribution is that, we predict the disk head position *reasonably* well with reduced memory overhead. Another contribution arises from the level of actual implementation of the work. We have implemented the entire predictor as a kernel device driver, which gives us a better interface to integrate it with a filesystem.

The rest of the paper is organized as follows, In the Section 2 we briefly present the background work that has been done in the area of such disk head prediction. In Section 3 we give an overview of the algorithms for disk head prediction designed by us. In Section 5 we give an overview of the device driver architecture that we put in place to leverage the efficient disk head prediction algorithms. We finally conclude in Section 6, after pointing the future research questions to be answered. we present two such approaches in the next section.

2 Background

2.1 Key Terms

Before describing the algorithms, we define a few terms. The first term is the **response time** of a disk which is defined as the time between issuance of a disk write request and its completion. The **skip distance** is defined as the number of sectors to be skipped by the disk head, the main intent of all the algorithms to predict the skip distance as accurately (below current disk head position) as possible so as to ensure minimal response time. The third definition is of **write size** which is defined as the amount of data to be written on the disk. Lastly we define **think time** as the time interval between issuance of consecutive write requests.

Next we present the *Disk Mimic Algorithm*.

2.2 Disk Mimic Algorithm

The Disk Mimic Algorithm [5], tries to predict the exact position of the disk head by first modeling the disk's behavior (in terms of various parameters to be described next), and then referring to decide the exact skips for future references. We briefly describe the Disk Mimic algorithm next.

The Disk Mimic algorithm works in two phases, in the first phase it creates an exhaustive profile of the disk behavior, it does so by issuing write requests of varying sizes, the requests are sent with varying the think times between them and by varying the skip distances. Also, the size of the requests themselves are varied. The algorithm

records those skip sizes which give the minimal response time for a given think time and write size. In order to gain confidence of the measured value each experiment is repeated and the average response time is calculated.

In the second phase, the algorithm uses the previously constructed table to predict the disk head position, by returning a skip size based on the size of the write and the duration after which it is issued (think time). This data is stored in a table format. The table is indexed by the current think time and the request size and returns the skip distance. Storing the necessary information in table format leads to efficient lookups that can be performed at runtime. However, the table can have high memory footprint. Another data structure maintained by the disk mimic is a free list of memory blocks. This list is consulted to check whether the blocks requested by the diskmimic predictor are actually free, in case they are not, the diskmimic algorithm mandates that the write take place at blocks which are further down the track, where sufficient free memory is available.

We show the characteristic behavior of a disk in Fig 1, it shows that the skip distance necessary to achieve minimal response time increases monotonically with think time. Which essentially points to the rotation of the disk head in during the think time. Also after approximately 8 msec the pattern repeats itself this happens due to the fact that the disk has completes a full rotation in 8 msec. We describe the Weblogic Disk Prediction mechanism [2] next.

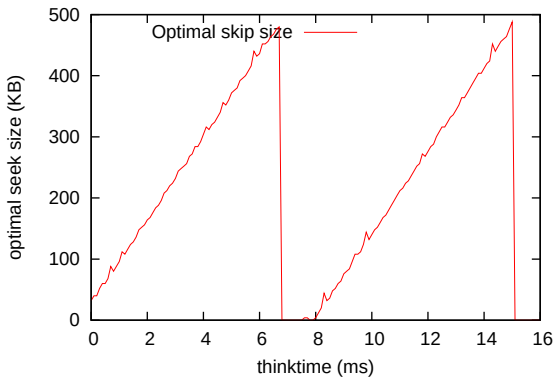


Figure 1: Optimal skip versus think time

2.3 Weblogic Disk Head Prediction Algorithm

The Disk Head predictor algorithm described in [2], predicts the disk head using a simple formula,

$$\delta B = (C + \delta T) * L$$

Where, δB is the amount of skip, δT is the think time. and, C and L are factors in the linear model. The values of C and L are assigned according to the following logic. C is the elapsed time since the issuance of a write and the start of actual transfer to the disk. C also accounts for the command overhead of the disk. Hence, the duration $(C + \delta T)$ represents the amount of time that elapses between the end of one data transfer and the beginning of the next. L is the block speed of the disk, i.e., the number of blocks that pass under the disk head per unit of time. The product of these quantities gives the incremental position of the disk head at the start of the data transfer of the next write. The quantity δT is directly measurable after each write, and the authors approximate the quantity C, by the minimum observed response time. The block speed of the disk is affected by the track being used: tracks towards the outside of the platter have more blocks and thus a higher speed. So, this algorithm periodically adjusts the value of L after each write to the disk.

2.4 Comparison

The two approaches to measure the disk performance described above are inherently different from one another. While diskmimic keeps a big table, which completely characterizes the disk performance, the Weblogic predictor keeps a simple equation to represent the disk head rotational position. The apparent benefits of the Weblogic model over the diskmimic model is its small memory overhead.

To compare the performance characteristics of the two predictor algorithms, we took the following approach. We characterized the complete performance of a disk by running the first phase of disk mimic algorithm. Then, we initialize the C parameter of Weblogic predictor with the smallest response time observed. The authors haven't clearly specified how they measure and update L periodically in [2]. Hence, we calculated the parameter L based on the skip distances and think times in Figure 1. We tested the performance by querying the Weblogic predic-

tor algorithm for various values of think times and a fixed write size of 4 KB. We compared its response time with the response time observed using skip size recommended by the disk mimic algorithm.

We have plotted these results in Figure 2. As can be seen from the figure the diskmimic algorithm gives better performance (smaller response time) than the Weblogic predictor in all the cases. These results can be attributed, to the oversimplifying assumptions of the Weblogic Algorithm. The Weblogic predictor takes the minimum observed response time to be a good estimator of the parameter C. While, this assumption in itself is a plausible conservative estimate, for a given write instruction the associated overhead due to the software paths are quite variable and in case they are actually higher than C and minimum response time (its estimator), the disk head wold actually have moved out of range, and hence would suffer the penalty of an extra write. Similarly as pointed out in [2] itself estimating the value of L is extremely hard and in case of miscalculation, the disk head may persistently be forced to suffer the overhead of an extra rotation. As the diskmimic algorithm stores the information pertaining to the disk's response time behavior at a desired granularity it is resistant to this kind of errors. However, disk mimic algorithm requires offline characterization of each disk, but the characterization is done once only.

To summarize, the diskmimic algorithm predicts the disk head much more accurately than the Weblogic algorithm but occupies a far more amount of space. In the next section we would describe our algorithm which tries to achieve the performance characteristics similar to the diskmimic algorithm algorithm but occupies a far lesser amount of memory.

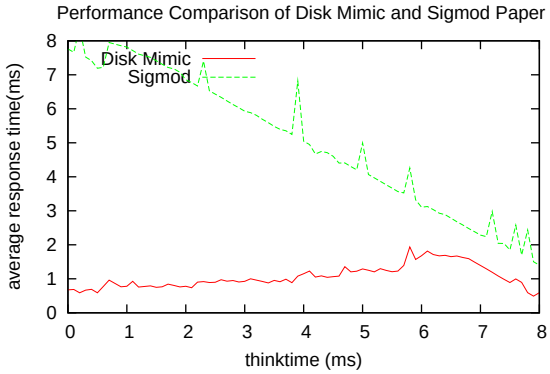


Figure 2: Performance comparison of diskmimic and Weblogic disk head predictors

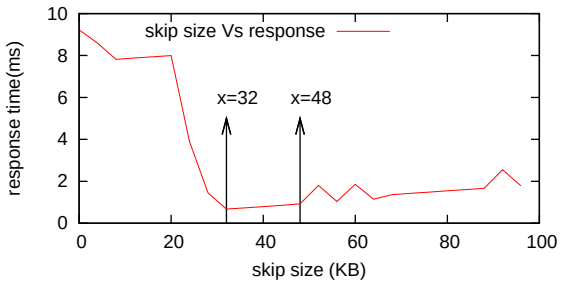


Figure 3: Response time variation with skip size, for 0 ms think time

3 Configurable Compact DiskMimic Algorithms

We design a compact algorithm with aim of not only reducing kernel memory footprint but also keeping the response time with in a particular limit(tolerance) of optimal response time. This tolerance is configurable, and also determines the reduction of memory foot print that we can achieve by the algorithm.

The key idea behind our approach is that we rely on the non-optimal skip data points in skip size-response time plane, rather than approximating the optimal skip size by line interpolation. For a think time, we consider all those skip sizes, whose response times are in the tolerance limit of the minimum response times. For example, as shown in the Figure: 3 , 32KB gives the minimum response time. But for tolerance limit of 0.5 ms, we consider the skip sizes in the range of 32-48 KB. Such set of skip sizes is called as *candidate-set* of a think time. Next we present our compact algorithms based on this idea.

- **Basic Compact** We select a skip size for a range of think times, such that the skip size is present in the candidate-set of each think time in the range(inclusive). By enforcing the condition that the selected size should be in each think time's candidate set, we are ensuring that the response time is

in the limit; while, we are also reducing memory footprint by storing just ¡begin-think-time,end-think-time,skip-size¿ for a range of think times. For a small tolerance, the range of think times would be small, and hence, the compression benefit is going to be small. Similarly, we expect more compression for slightly larger tolerance.

- **Line Compact** We make further improvements on this approach. Instead of selecting a skip-size for a range of think times, we choose one line equation to represent set of skip sizes for range of think times. We ensure that, the skip size represent by the line equation is in the candidate set of think time. Thus, this algorithm further improves the compression, while having response time with in tolerance limit. The selection of line equation is non trivial. We apply following greedy approach for the set of think times.

algorithm **Line Selection**
$L_{candidate} \leftarrow \phi$
Init $\leftarrow 0$,
For all data points i : (think time $T_i, Skip(T_i)$)
$if L_{candidate} = \phi,$
$L_{candidate} \leftarrow allPairOfLines(Init, i)$
$else,$
$L_{newCandidate} \leftarrow extendLines(i)$
$if L_{newCandidate} = \phi$
$equation((T_{init}, T_{i-1})) \leftarrow L$
$whereL \in L_{candidate}$
$Init \leftarrow i, L_{candidate} \leftarrow \phi$
$else$
$L_{candidate} \leftarrow L_{newCandidate}$
$end - For$
end-algorithm

Figure 4: Line Selection Algorithm

The basic idea behind the greedy approach is to begin with candidate set of lines, and extend the line-equations to new data points as long as there is at least one extensible line equation in the candidate line set. A line equation is said to be extensible for a data point, if it gives a skip size for corresponding think time, that belongs to its candidate set. So the algorithm tries to greedily extend the lines as long as they can be extended. When there is no extensible line equation for new data point, we select line equation for previous data point, and use it to describe the range of think times up to previous data point. The candidate set of lines is initialized as set of all possible pair of lines between two consecutive data points, i.e. for think time T_i and T_{i+1}, the lines are initialized as all possible pair of lines between set $Skip(T_i)$ abd $Skip(T_{i+1})$, where $Skip(T_j)$ denotes the candidate-set corresponding to think time T_i. In the pseudo code for the algorithm, the function $extendLines(i)$ correspond to extending lines for ith data point and the function $allPairofLines(Init, i)$ initializes the candidate set with all possible line equations between data points $Init$ and i.By this approach, we end up partitioning data points to different sets, where each set of data points is represented by one line equation. The amount of compression benefit is determined by the size of each of these sets.

- **Plane Compact** So far, we have considered only skip sizes and think times for compression. We extend this to further include request size. So in this approach, we will consider plane equations to represent skip sizes, for set of request sizes and think times. This approach should improve the compression further.

3.1 Configuration Policies

So far we have described the policy of configuration as to keep the average response time with in the tolerance limit of optimal response time. However, there could be policies as follows :-

- **Aggressive Servicing**: In this policy, we consider those skip sizes who have the minimal service time, but don't necessarily have minimum average response time. With such skip sizes, the disk head could be right on the edge of writing and small difference in the workload can cause the disk to have an extra rotation. This policy has a configurable parameter *majority* which is the probability of service time being very close to minimal service time. With

$(1 - majority)$ probability, the disk may have extra rotation.

- **Defensive Servicing**: In this policy, those skip sizes are considered whose maximum service times are bounded by a configurable parameter.

- **Average-limit Servicing**: In this policy, we consider those skip sizes whose average response times are in the tolerance limit of average response time. It has a configurable parameter *tolerance*, as described earlier. This is different from defensive servicing policy, as it considers average response time instead of maximal response time.

In this paper, we discuss about the average-limit servicing policy only.

4 Evaluation

In this section, we study the characteristics of compact disk-mimic algorithms. First, we compare our approach with other approaches.Then we study the variation of compression benefits with tolerance.

4.1 Experimental Setup

In this section we present a thorough evaluation of our algorithm. We first experiment with log skipping in emulated environment. The emulator is a user level program that issues synchronous write requests to a raw disk partition after the recommended skip distances by predictor algorithms. We measure the service time (response time) for the synchronous write requests.

For these sets of experiments, we vary the request size from 4 KB to 200 KB in 4KB increments and think time from 0 to 9 ms in $100\mu s$ increments. We calculate the average service time over 25 different samples for each combination of request size and think time. Our experiments are run using a 80 GB SATA disk (HDS728080PLA380 PF2O). The system had a 1 GHz AMD Opteron processor, and a Linux 2.6.9 kernel installed. We had set the size of the log to 500 MB.

4.2 Performance Comparison

First, we compare the our algorithm with two naive approaches, *Line fitting* and *Coarse Segments*, as follows.

- **Line Fitting:** In this approach, we use line equations to fit the plot of optimal skip size versus think time. For example, Fig 1shows the variation of optimal skip distance versus think time for 4KB request size writes. The curve is piecewise linear, and repeats itself after every 8.3 ms, which is the disk rotation period. We use distinct line equations to predict the disk behavior while the think times lie in each linear segment. For the above disk behavioral pattern, we use two line equations, one line equation for think times in range from 0 ms to 7.4 ms and the other one from 7.5 ms to 8 ms.

- **Coarse Segments**: In this approach we select a coarse set of points, in the plot of optimal skip size versus think time. We find skip size for intermediate points, as a line interpolation of two immediate neighbor points. The number of such data points is a configurable parameter, which determines its memory footprint.

  We have studied the performance of the above mentioned two algorithms by comparing their performance with diskmimic algorithm. We present our results and observations below. We find that both the approaches have lesser memory footprint compared to diskmimic. In the experiment setting for coarse segments, the memory footprint was 1/3 of the total memory footprint due to disk mimic. But these naive approaches do not perform well for some think times, as shown in the Figure 5. The request size for this set of experiments was 4 KB. We observed that in some cases, their average response time is quite high as compared to optimal response time. In order to quantify the effectiveness of these algorithms, we define a parameter called as **outlier fraction** for a tolerance, which is the fraction of data points whose response time is not with in a specific bound of optimal response time. We find that the outlier fraction in Line Fitting is 0.48 for tolerance of 0.5 ms, which is quite high. The outlier fraction for Coarse Segments is relatively less and around 0.20 . Another metrics,

to quantify is the **mean loss** from optimal response time. We present the results in Table 6.

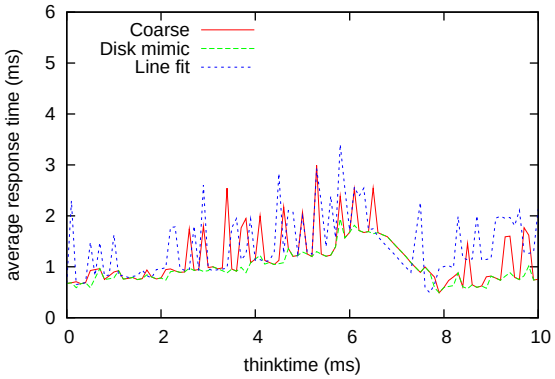


Figure 5: Performance comparison of Naive Algorithms

Next, we present the performance characteristics of compact disk mimic. Our compact disk mimic predictor ensures that response time due to skip distance is always with in the limit of tolerance of minimum average response time. So, we expected that there won't be any outliers.

Predictor Algorithms	Outlier Fraction	Mean Loss
Disk Mimic	0	0
Line Fitter	0.48	0.57
Coarse Segments	0.2	0.23

Figure 6: Table of comparison metrics

Our expectation matched with our observation that our predictor performed very close to disk mimic's performance. The graph shown in Figure 7 represents the performance of disk mimic and our compact disk mimic algorithm for 4KB request size writes. Compact disk mimic had memory footprint benefit of factor of 3 over disk mimic.

4.3 Effect of Tolerance variation on amount of compression

We also studied the effect of variation of tolerance on the amount of compression. We present the results

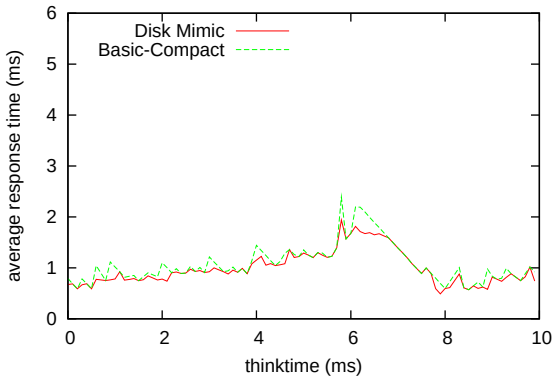


Figure 7: Performance Comparison of Disk Mimic and Compact Disk Mimic

for the same in Figure 8 . We find that as tolerance is increased, a skip distance is expected to be acceptable for greater range of think times, which further implies that there will be further reduction in footprint. As can be seen fro Figure 8 for a tolerance limit of 1.5 ms, we could get footprint reduction by factor of 6.

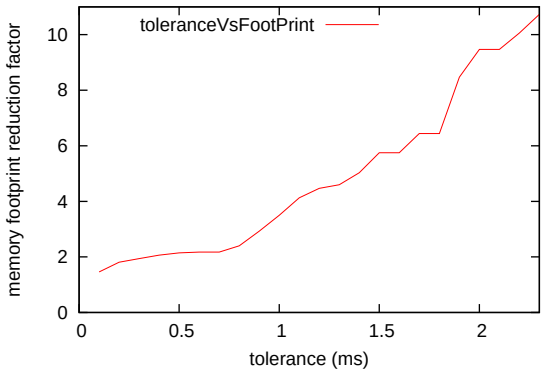


Figure 8: Variation of footprint reduction with tolerance

We further study the performance loss due to increase in tolerance. We measure this as mean loss from optimal response time. The graph shown in Figure 9 depicts this loss. We found that even for tolerance limit of 1.5 ms, the mean loss was 0.4 ms.

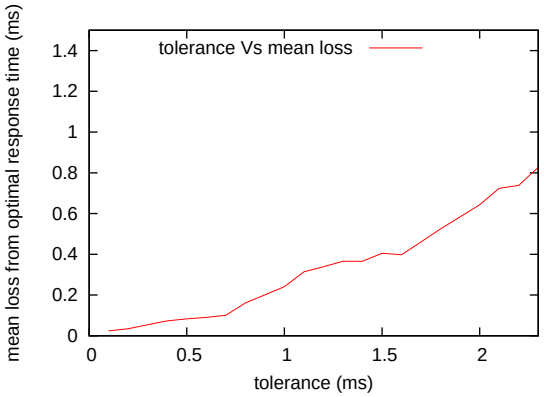
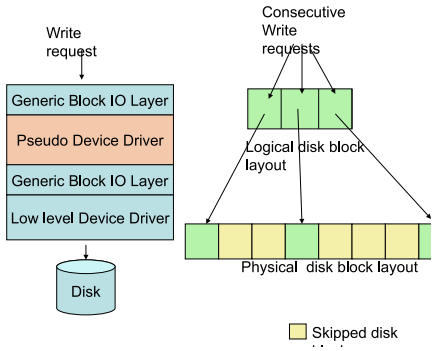


Figure 9: Variation of mean performance loss with tolerance

5 Implementation

We have implemented the pseudo device driver for a Linux 2.6.9 kernel. The pseudo device driver sits between the generic I/O layer of the kernel. It receives I/O requests to the disk, performs some mapping on them and then passes it on to another lower level device driver to carry out the actual I/O over the disk. The mapping operation mentioned above involves, querying the disk head prediction algorithm to obtain the skip size, and updating a mapping data structure, that keeps track of the actual logical to physical mapping of the block. The functionality of the Pseudo Device driver is shown in Figure 10. The major components of the device driver are the following

- **Mapper** The mapper consists of an mapping from logical file specified disk blocks to physical disk blocks.
- **Predictor** This component when queries with a given write request returns the predicted physical address which would result in fast response time
- **Encapsulator** This module takes a given chunk of data and writes out a header to it, the header contains information pertaining to the actual logical identity of the block, along with a version field. This header is written to assist in crash recovery.
- **Free List** This list maintains the ids of all free blocks on the disk, and is implemented as a bitmap



heightheight

Figure 10: Pseudo Device Driver Architecture

The Pseudo device driver, sits on a disk drive with size much greater than what it exposes to the filesystem sitting on top of it. This give the device driver the flexibility to write anywhere inside the disk, and not worry about efficient space utilization. On receiving a write request, the mapper, predictor and free list are consulted to determine a suitable physical disk block where the write could be done. After this the Encapsulator adds an appropriate header to the data block and then the data block is passed on to the underlying low level device driver, with a request to write to to the predictor selected location.

6 Conclusion and Future Work

In this work we have investigated various methods to make small synchronous writes to disk faster. Small synchronous disk writes are an important workload, a significant amount of journalling workload falls under this category. The performance penalty of small synchronous writes occurs due the extra disk latency involved in writing data reliably in parts on consecutive blocks, which incurs the added cost of extra disk head rotation.

In this work we have tried to solve the problem by developing disk head predictors, and use its output to save the write penalties associated with ordinary writes. The work involved two major parts a) development of a disk head predictor algorithm, and b) design and implementation of a pseudo device driver architecture which can take benefit of the predictor algorithm and make log writing fast.

The key lessons learned from the study is the realization that disk head prediction is possible with reasonable amount of accuracy. We have also realized that the degree of desired accuracy is essentially a trade-off with the size of memory that one wants to devote for the predictor. We have also designed and implemented a device driver architecture to integrate the disk head predictor with an actual filesystem.

Our immediate future work involves, understanding and characterizing the interdependency between disk head prediction accuracy and storage requirement thoroughly. We want to understand the maximum reduction of memory footprint that can be achieved without compromising the performance within tolerance limit. Also, we need to solve the glitches present in our device driver implementation, integrate it with a real journalling system and check its performance. Over a longer duration we would like to investigate the performance of the designed system with a transactional database system.

7 Acknowledgments

We would like to thank Florentina Popovici, Lakshmi Bairavasundaram, Haryadi Gunawi, Nitin Agrawal and Vijayan Prabhakran for their help and suggestions for problems faced in pseudo device driver implementation. We would like to thank Prof. Remzi for his various suggestions and ideas during the project.

References

[1] S. Best. Jfs log: how the journaled file system performs logging. In *ALS'00: Proceedings of the 4th conference on 4th Annual Linux Showcase & Conference, Atlanta*, pages 9–9, Berkeley, CA, USA, 2000. USENIX Association.

[2] B. Gallagher, D. Jacobs, and A. Langen. A High-Performance, Transactional Filestore for Application Servers. In *(SIGMOD '05)*, Baltimore,MaryLand, June 2005.

[3] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[4] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for unix. volume 2, pages 181–197, New York, NY, USA, 1984. ACM.

[5] F. I. Popovici. *Data-Driven Models in Storage System Design*. PhD thesis, UW- Madison, 2007.

[6] F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Robust, Portable I/O Scheduling with the Disk Mimic. In *Proceedings of the USENIX 2003 Annual Technical Conference (USENIX '03)*, San Antonio,Texas, June 2003.

[7] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems*, pages 26–52. ACM, 1992.

[8] S. C. Tweedie. Journaling the linux ext2fs file system. In *In The Fourth Annual Linux Expo,*, Durham, N. Carolina, 1998.