# Expressing File System Reorganization Policies with the ClassAd Language

Nicholas Coleman and Vinod Yegneswaran
University of Wisconsin, 1210 West Dayton Street, Madison, WI 53703
{ncoleman,vinod}@cs.wisc.edu

December 18, 2000

**Abstract**

In many modern file systems files must be periodically reorganized on disk in order to preserve disk space and provide for efficient I/O operation. Devising adaptive file system reorganization policies is a good first step towards increased flexibility, but does not achieve the goal of separating these policies from the underlying mechanisms of the operating system. The ClassAd language used to express resource allocation policies in the Condor distributed computing system is an excellent candidate for a file system policy description paradigm. A log structured file system (LFS) is an ideal environment for the use of ClassAds for file system reorganization policies. In this paper we explore using ClassAds to express file system reorganization policies in LFS, particularly cleaner policies.

# 1 Introduction

In many modern file systems files must be periodically reorganized on disk in order to preserve disk space and provide for efficient I/O operation. For example, in a log structured file system[9] the log must be periodically cleaned to reclaim unused blocks. Policies for file system reorganization are typically hard wired into an operating system and are geared towards a specific workload. This makes it quite difficult to modify file systems to keep up with changing technology and adapt to the demands of new applications. Matthews et al.[5] any many others have proposed several adaptive algorithms to give operating systems more flexiblity to deal with such situations.

Devising adaptive file system reorganization policies is a good first step towards increased flexibility, but does not achieve the goal of separating these policies from the underlying mechanisms of the operating system. It would be useful to have a framework to facilitate the modification of file system policies without having to vastly restructure the OS. A simple but expressive policy description language would be ideal for this purpose.

The ClassAd language used to express resource allocation policies in the Condor distributed computing system is an excellent candidate for a file system policy description paradigm. A ClassAd contains the characteristics of an entity as well as policies concerning that entity. In the case of matchmaking in Condor, the entities are jobs and machines, and the policy expressions determine which jobs are allocated to which machines.

A log structured file system (LFS) is an ideal environment for the use of ClassAds for file system reorganization policies. Unlike standard Unix file systems, LFS requires periodic reorganization of blocks of data on disk in order to reclaim unused disk blocks in a process called cleaning. In addition to cleaning, other disk reorgnizations may be beneficial in LFS to improve read performance.

In this paper we explore using ClassAds to express file system reorganization policies in LFS, particularly cleaner policies. Section 2 describes related work in adaptive file system reorganization, and policy description languages. Section 3 is an overview of the ClassAd language. In section 4 we discuss Linlog FS[3], an implementation of LFS for Linux which we have chosen to test out the feasibility of our work. We present the details of file system policy ClassAds for LFS and discuss our implementation in section 5. Section 6 and 7 are devoted to future work and conclusions respectively.

## 2 Related Work

A great deal of research has been done to determine ideal file system reorganization policies for log structured file systems. We have used such policies in order to evaluate the expressiveness of the ClassAd language. We have not run across any policy specification languages developed explicitly for expressing file system policies, but there has been research into generic policy expression languages other than ClassAds.

Several performance optimizations for LFS are proposed in [5]. Three of these are aimed at improving write performance and one aimed at read performance. The key write optimization is an adaptive method of choosing cleaning policies. We believe that ClassAds could be used to express the cost-benefit formulas for deciding between cleaning and hole-plugging (an alternative form of LFS cleaning). Our implementation of this policy is discussed later in this paper.

Storage reorganization is also proposed in [2] as a performance enhancenment for LFS. Their solution is similar to the read performance optimization in [5], that is to reorganize data based on access patterns, but they have a much more robust system of maintaining global and block level statistics. It would be possible to use ClassAds to express the block level statistics, and use matchmaking to determine how to reorganize these blocks.

A number of other papers discuss and analyze the overhead of the LFS cleaner. In [7] the author argues that significant gains may be made in effective disk bandwidtch if reorganization overhead is decreased, but does not propose any explicit solutions. A second paper [8] focuses on cleaning the least-utilized segments (a policy partially implemented in the LinLogFS cleaner prototype). Finallly, [1] suggests some simple heuristics to reduce the performance degradation of the cleaner.

While much has been written about reorgnanization policies for LFS, it does not appear that anyone has proposed using a policy language to facilitate implementation of such policies. However, general purpose policy specification languages such as PDL (Policy Description Language)[4] have been proposed. PDL was highly influenced by planning systems in artificial intelligence. Policies are made up of *event-condition-action* rules which outine the actions a system must take under specified conditions. This framework is very similar to our use of ClassAds for event based policies, but considerably more complex.

## 3 The ClassAd Language

The ClassAd language was developed for Condor as a means of representing resources, and expressing resource allocation policies. Condor uses the ClassAd language to facilitate the process of matching jobs submitted by users to compatible machines. Because Condor is used in heterogenous computing environments, it is difficult to express compatibility in a single centralized policy. Instead, Condor uses classified advertisements, or ClassAds to delegate the authoring of resource policies to the owner of the resource in question. Each resource (e.g. a job or a machine) has associated with it a ClassAd which consists of a list of characteristics of the resource, as well as policy expressions relating to that resource.

A single ClassAd is a collection of pairs of attributes and expressions. Each attribute corresponds to either a characteristic of the resource being represented or a policy which the resource owner wishes to have enforced. In the case of resource characteristics, the values are typically constants in the form of strings, numbers, boolean values or timestamps. The values for the policy attributes are typically mathematical or boolean expressions whose variables may be attributes of the same ClassAd, or of other ClassAds. An example of a machine ClassAd is shown in Figure 1. An example of a job ClassAd is shown in Figure 2.

```
[
  Type         = "Machine";
  Activity     = "Idle";
  KeybrdIdle   = '00:23:12'; // h:m:s
  Disk         = 323.4M;     // mbytes
  Memory       = 256M;        // mbytes
  State        = "Unclaimed";
  LoadAvg      = 0.042969;
  Mips         = 104;
  Arch         = "INTEL";
  OpSys        = "LINUX";
  KFlops       = 21893;
  Name         = "foo.cs.wisc.edu";
  Subnet       = "128.105.175";
  Start          = other.Type == "Job"
             && other.Owner != "riffraff"
             && LoadAvg < 0.3
             && KeybrdIdle>'00:15'
  Rank           = DayTime() >= '9:00' &&
             DayTime() <= '17:00' ?
             1/other.ImageSize : 0;
  Requirements= Start;
]
```

Figure 1: ClassAd describing a Machine in Condor

The process of allocating jobs to machines in Condor is called matchmaking. Both job and machine ClassAds contain expressions assigned to the attributes Requirements and Rank. A job's

```
[
  Type          = "Job";
  QDate         = 'Thu Dec 14 10:53:31
                    2000 (CST) -06:00';
  CompletionDate = undefined;
  Owner         = "ncoleman";
  Cmd           = "run_sim";
  WantRemoteSyscalls = true;
  WantCheckpoint = true;
  Iwd           = "/usr/ncoleman/sim2";
  Args          = "-Q 17 3200 10";
  Memory        = 31m;
  Rank          = KFlops/1E3 +
                   other.Memory/32;
  Requirements = other.Type == "Machine"
                   && other.Arch=="INTEL"
                   && other.OpSys=="LINUX"
                   && other.Memory >=128
]
```

Figure 2: ClassAd describing a Job in Condor

*Requirements* expression defines constraints on what machines it may be matched with and its *Rank* expression is used to select preferred machines. The corresponding expressions for a machine are used similarly to express constraints and preferences for jobs. A job matches with a machine if each ClassAd satisfies the other's constraints. In the event that a job matches with more than one machine, the *Rank* expressions in both ClassAds are used to narrow the field of candidate machines.

ClassAds are used to express other policies as well. Each machine has a *Start* expression which must evaluate to **true** before a job may begin running. The *Start* expression is typically identical to the *Requirements* expression in a machine (a machine would not accept a job it had no intention of running), but the state of the machine may have changed since the matchmaking process concluded. This is a much simpler application of the ClassAd framework, but it allows for generic policies to be expressed in terms of the state of the machine. We will revisit this type of policy expression in our discussion of file system policy ClassAds.

ClassAds have been used very successfully in Condor for expressing the policies described above. For a more detailed examination of how ClassAds are used in Condor, interested readers are encouraged to reference [6].

# 4 Linux Log Structured File System (Linlog FS)

Linlog FS is an implementation of a log structured filesystem for Linux by Christian Czezatke at the Technical University of Vienna. Linlog has several unique features which made it seem like a very attractive platform to test out our ClassAd policy engine. First of all, Linlog FS can be easily implemented as linux kernel module. A linux implementation of course has all the advantages of availability, familiarity and open source. More importantly it came packaged with a very simple user level cleaner that did not have any mature policies. The minimal cleaner implementation seemed to rewrite all live data to the end of the log and free segments, without giving any consideration to segment utilization. This cleaner in many ways seemed ideal for our purposes and could greatly benefit from a relatively simple ClassAd based policy checker.

## 4.1 File System Organization

Linlog FS, like other log structured filesystems is tuned for improved write performance on small files. There are two critical characteristics of Linlog FS that directly contribute to this enhancement. A log structured file system accumulates the blocks to be written in a large chunks or write clusters and performs a single large sequential write into a segment, thereby amortizing the seek and rotational penalties over several writes. A single file write on a traditional file system takes two seeks, one to update the inode and another seek to write the data. Linlog FS tries to optimize by storing some of the meta data information along with write clusters at the tail of the log.

## 4.2 Linlog FS Data Structures

The three essential meta data structures in linlog are the dtfs superblock, traditional superblock and the checkpoint area. These are the only meta data that are located at fixed segments in the filesystem. In the following dtfs and Linlog FS might be used interchangeably as although dtfs has been officially renamed to Linlog FS, dtfs nomenclature still persists in most of the source code.

The dtfs superblock holds basic filesystem geometry information such as block size, the total number of inodes allocated, and the number of blocks per segment. It holds a reference to one or more dtfs filesystem descriptors that contain pointers to their respective checkpoint areas and traditional filesystem superblocks. The superblock contains more than one filesystem descriptor because Linlog FS supports more than one filesystem within a dtfs partition. These higher level file systems that reside on top of dtfs are known as file system personalities. Currently only an adaptation of ext2 called dext2 has been implemented.

Each filesystem has two checkpoint areas that are updated alternately for recovery in case of failures during updates to the checkpoint area. A checkpoint area is composed of checkpoint entries that represent a consistent state of the file system at a certain point in time. The checkpoint entries point to checkpoint blocks that are located, usually in the last block of the log segments.

The Linlog file system is divided into a fixed number of segments. Each segment is itself divided into a certain number of blocks. The number of blocks in a segment is usually specified at file system creation time and defaults to 128 blocks of 4KB. The in-memory segment usage bitmap (stored in /proc) keeps track of which segments are in use and provides for fast access and updates.

Furthermore, the Linlog implementation contains three important metadata files, which allow for easy access to essential meta data information through normal file system operations. The .ifile stores all the inodes (with the exception of .ifile itself). The .iusage file contains the inode utilization information and is used to detect free inodes quickly. The .atime file stores the access time information for all the inodes in the filesystem. Like any other file in Linlog FS these files are written to a log, rather than residing on a fixed area of the disk.

### 4.3 The Linlog FS Cleaner: MrClean

As mentioned earlier, Linlog Filesystem is equipped with a minimal cleaner that evolved from an Advanced Operating Systems class project by Trey Cain and Anthony Sargent here at the University of Wisconsin Madison. To focus on the positives, most of the user level cleaner code was reasonably simple, well written and bug free. The following pseudocode describes the basic cleaner algorithm:

- Read Superblock, initialize layout descriptor
- Read the Segment Usage Table, to determine segment utilization and build a list of dirty segments.
- For each dirty segment and for all partial segments within each segment
  - read check point block associated with the segment to determine block
  - determine liveness of the block by reading checkpoint and comparing the backpointer to the inode from the block
  - if Live Block, add to Live block list for the segment.
- For each dirty segment,
  - For each live block in the segment
    - invoke ioctl call on each live block to free, refile the block
  - update Segment Usage Table to free the old segment

Our experience with the cleaner has convinced us that most of this user level code is in fact working as expected. The user level system correctly recognizes the dirty segments, builds the dirty-segment-list and live-block-list and seemingly invokes the ioctl kernel call to refile the blocks with the correct parameters. However, we soon realized that the kernel support for the cleaner was the weak link (supposedly written by Christian). Some of the obviously trivial bugs that kept appearing in the kernel that we managed to fix leads us to believe that the integrated cleaner (with kernel support) has never really been tested. After delving deeper into the kernel code, and fixing a few more segfaults and trivial mistakes, we have managed to reach the apparent conclusion that Linlog does refile the dirty blocks and release the old ones, but fails to update the metadata corresponding to these blocks correctly.

## 5 LFS Policy ClassAds

As file system reorganization is such an integral part of a properly functioning log structured file system, we have chosen LFS as a platform for exploring the use of the ClassAd language to express file system reorganization policies. We have focused our research on cleaner related policies, as we have a user level cleaner to work with in Linlog FS. Our choice of Linlog FS has been something of a double edged sword. On the one hand, it has been useful to have the source for a user level cleaner which can be easily analyzed and modified. Oh the other hand, the insufficient support for cleaner operations in Linlog FS means that our testing could not be as robust as we would have hoped. In this section we will describe several cleaner policies and show how to use ClassAds to express these policies. We will then discuss the desgin and implementation of our own ClassAd policy system for the Linlog FS cleaner. Finally we will breifly discuss our evaluation of this system.

## 5.1 Cleaner Policies

The cleaner policies we have identified are analagous to the resource allocation policies described above. Previously we made the distinction between matchmaking based policies and event based policies used in Condor. We shall give examples of both types of policies in the context of an LFS cleaner.

### 5.1.1 Event Based Policies

The most straight forward policy decision made with respect to cleaning in LFS is deciding when it is appropriate to begin the cleaning process. Factors which must be taken into account include how recently the last cleaning took place and how effective the cleaning was. We express these factors explicitly by having the cleaner output a log every time it completes cleaning. An example of such a log is given in Figure 3. A simple policy might state that cleaning should only occur if no cleaning has occurred in the last two minutes and more than five segments were cleaned. Such a policy may be expressed by the policy administrator (henceforce referred to as the user) quite easily with the ClassAd in Figure 4. When the decision whether or not to run the cleaner must occur the attributes from the cleaner log are inserted into the user policy ClassAd, and the *StartCleaning* expression is evaluated. If the expression evaluates to **true**, cleaning may begin.

```
[
  CandidateSegmentsRead = 10;
  SegmentsCleaned = 4;
  TransferTimeBlock = 15;
  BlockSize = 4096;
  LiveBlocks = 1003;
  TransferTimeSeg = 20;
  BlocksPerSeg = 128;
  EmptyBlocks = 277;
  TimeOfCleaning = 'Mon Dec 7 13:04:22 2000 (CST) -06:00';
]
```

Figure 3: ClassAd describing an LFS cleaner log

```
[
  StartCleaning = (CurrentTime() - TimeOfCleaning) > '00:02'
                  && SegmentsCleaned > 5;
]
```

Figure 4: ClassAd describing a policy for deciding when to begin cleaning

A more complex example of a cleaning policy is choosing between traditional cleaning and hole-plugging. Hole-plugging is a variation on the normal LFS cleaning algorithm in which live logical blocks from one segement are written into unused physical blocks in another segment, instead of being written to the log. This means of cleaning is often preferable to traditional cleaning during high disk utilization. Matthews et al. describe an algorithm to determine the relative cost benefit of traditional cleaning and hole plugging based on past cleaning performance[5]. We can express these formulas with ClassAd expressions quite easily and effectively. Figure 5 shows such a ClassAd. Like the policy which decides when to begin cleaning, this ClassAd must have all of the attributes from the cleaner log inserted before the *StartHolePlugging* expression is evaluated.

```
[
  SegmentSize = BlockSize * BlocksPerSeg;
  TransferTimeCleaning = (CandidateSegmentsRead +
    LiveBlocks/BlocksPerSeg) * TransferTimeSeg;
  SpaceFreedCleaning = EmptyBlocks * BlockSize;
  CostBenefitCleaning = Real(TransferTimeCleaning)/Real(SpaceFreedCleaning);
  TransferTimeHP = (CandidateSegmentsRead * TransferTimeSeg) +
    (LiveBlocks * TransferTimeBlock);
  SpaceFreedHP = CandidateSegmentsRead * SegmentSize;
  CostBenefitHP = Real(TransferTimeHP)/Real(SpaceFreedHP);
  StartHolePlugging = CostBenefitHP > CostBenefitCleaning;
]
```

Figure 5: ClassAd describing a policy for deciding between traditional cleaning and hole-plugging

#### 5.1.2 A Matchmaking Based Policy

In our implementation we have focussed on a framework for evaluating event based policies. However, there are opportunities for using matchmaking based policies for file system reorganization as well. A good example of a matchmaking based policy for cleaning is choosing which segments to clean. A segment may be represented by a ClassAd, much like a machine or job in condor. The attributes in such a ClassAd might include utilization, segment number (indicating it's current location on disk), and how recently blocks in the segment were read and written to.

### 5.2 Design and Implementation

Our design of the ClassAd Policy Engine had the following major components:

1. *Cleaner* - We modified the cleaner to record important statistics on each run and write the result in the form of a ClassAd. The user level cleaner could be easily modified to keep track of certain important parameters and write these data in the format of a ClassAd file. The set of attributes recorded by the cleaner depends to a certain extent on the characteristics of the associated policy. For example one of the policies associated with a cleaner is deciding between cleaning and hole-plugging. The other policies could include: deciding when to run the cleaner, deciding where to start looking for free inode etc. Our cleaner implementation records statistics for deciding between cleaning and hole-plugging.

2. *User Policy File* - This is a file which contains a ClassAd created by the user which defines all necessary policy expressions. The user must define a set list of policies (StartHolePlugging, StartCleaning, etc.) in terms of system provided attributes. In a production system the names and uses of all of the attributes would be sufficiently documented, and sample policy files would be provided.

3. *Policy Checker* - The Policy Checker is implemented using the standard ClassAd libraries and hence is written in C++. The job of the Policy Checker is to simply read the User Policy ClassAd as well as the Cleaner ClassAd and generate appropriate policy decisions, usually in the form of a yes/no answer. Our current implementation of the policy checker supports the StartHolePlugging policy.

4. *File System Monitor* - This is a daemon that is wakes up periodically to consult the policy checker do determine what actions to take with regard to the cleaner. We have not implemented this component at this time, for now the policy checker must be run by the user.

## 5.3 Evaluation

Due to the limitations of the current implementation of Linlog FS, particularly the insufficient support for the cleaner, our options for evaluating our code were significantly restricted. We have tried to demonstrate that the ClassAd mechanism is lightweight enough to be used in conjunction with file system reorganization.

The experiments were run on a Pentium III machine runing Red Hat Linux 6.2. We set up one partition with ext2 and a second partition with Linlog FS. All of our code was compiled and run on the ext2 partition. In order to time the performance of the individual parts of our code we inserted gettimeofday() calls, and used the results to calculate how long each section took in microseconds.

In the case of the modified cleaner, we measured the time to build a data structure of blocks to be cleaned, the time complete the ioctl call which does the actual block relocation, and the time to write a ClassAd to the cleaner log. We took measurements for different cleaner loads to demonstrate that the time taken to write the classad is independent of the time taken to clean. The job of the cleaner could be divided into three main phases, the aggregation phase, the refiling phase and the logging phase. During the aggregation phase, the main job of the cleaner is build the list of live blocks and time here is dominated by the multiple kernel calls (once for each block) to read the segusage table from memory. During the refiling phase, the cleaner repeatedly invokes the ioctl syscall to refile each block and finally during the logging phase it writes out the classad. We noticed that time involved in the repeated syscalls to read the segusage table was pretty negligible (400-700$\mu$s) compared to overall cleaning time. Also, as we expected, the refiling phase cleary dominated the bulk of the cleaner's running time and was several orders of magnitude greater than the time to write out the ClassAds. Our measurements for the cleaner are in Table 1.

For the policy checker we measured the time to read the two ClassAd files and the time to build and evaluate the complete policy ClassAd. For uncached files, reading the files took up the bulk of the overall time as might be expected. However, once the files were cached in memory, building and evaluating the classad dominated the computation time. This raises some questions about using ClassAds for evaluating policies in more high performance environments such as CPU scheduling. Our measurements for the policy checker are in Table 2.

| MBs cleaned | total time to clean(ms) | time to write ClassAd($\mu$s) |
|---|---|---|
| 5 | 700 | 157 |
| 10 | 14000 | 169 |
| 20 | 44000 | 168 |

Table 1: Performance measurements for modified cleaner

| Run type | time to read ClassAds($\mu$s) | time to evaluate($\mu$s) |
|---|---|---|
| Uncached files | 5336 | 1930 |
| Cached files | 86 | 1461 |

Table 2: Performance measurements for policy checker

# 6 Future Work

We have chosen for our implementation to focus on cleaning policies for a log structured file system. As mentioned previously, file system reorganization may also be done in LFS (or for that matter any file system) in order to optimize read performance. ClassAds could be used to express policies for which logical blocks ought to be grouped together, based on data access patterns.

ClassAds are also ideal for describing distributed file system policies. On of the reasons ClassAds are ideal for use for Condor's resource allocation policies, is that they enable policy decisions for a particular resource to be specified by the owner of the resource. Just as jobs are assigned ClassAds in Condor, files may be assigned ClassAds in a distributed file system. The file ClassAds might contain policies for replication or caching.

Finally, ClassAds may be used for policies outside the spectrum of file systems. Job scheduling and memory management policies could certainly be expressed using ClassAds; however, ClassAds are not indended for high performance situations. While it may be inappropriate to use ClassAds for fine grained policy decisions such as which job to pull off of the ready queue, they might be effective for specifying adaptive job scheduling policies based on the characteristics of the system workload. Adaptive memory managment policies could be designed in much the same way.

A more interesting application of ClassAds outside of file systems is in the field of security. ClassAd expressions are already used in an ad hoc way to enforce security in Condor. Note in Figure 1 how the machine's *Start* expression (and by indirection its *Requirements* expression) prevents jobs owned by user "riffraff" to be run on this particular machine. This is a fairly trivial example of a security policy, but the ClassAd language is certainly robust enough to specify more complex policies.

# 7 Conclusions

We have proposed using the ClassAd language to express policies for file system reorganization. It is our claim that the existence of a general purpose policy language would enable users to build more complex and effective file systems more easily. We have shown that ClassAds are expressive enough to implement very complex policies, yet simple enough to be comprehensible to the administrator of an operating system. Finally we have shown that for file system reorganization, exemplified in the LFS cleaner, ClassAds are lightweight enough so as not to adversely affect performance.

Indeed we had some modest success in the course of implementing our project, but perhaps more can be learned from our failures. Although it soon became apparent that Linlog FS may not be stable enough to be used as a production level file system, and the cleaner was not fully implemented, we believed we might be able to get a reasonably functional cleaner to test out our ClassAd policies. It is now very clear that there is great deal of work to be done, particularly in the area of support for the cleaner, before Linlog FS can be truly useful for the purposes of operating systems research.

Maybe we were a little too ambitious and put too much of our efforts into trying to get our infrastructure to work, which inevitably limited the scope of what we could realistically accomplish in the time we had. At the very least, we have learned a great deal about policies in file systems, adaptive file system reorganization, and the structure and implementation of a log structured file system. What's more, our preliminary results have reaffirmed our original belief, that ClassAds are indeed a feasible and powerful mechanism to express complex policies and have the potential to improve file system performance and flexibility without incurrning significant overhead.

# References


[1] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the 1995 Winter USENIX Conference*, January 1995.

[2] C. Chee, H. Lu, H. Tang, and C. Ramaomoorthy. Adaptive prefetching and storage reorganization in a log-structured storage system. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):824–838, September 1998.

[3] Christian Czezatke. dtfs—a log-structured filesystem for linux. Diplomarbeit, Technische Universität Wien, Austria, 1998.

[4] J. Lobo, R. Bhatia, and S. Naqvi. A policy description language. In *Proceedings of American Association for Artificial Intelligence*, Orlando,Florida, July 1999.

[5] J. Matthews, D. Roselli, A. Costello, R. Wang, and T. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, October 1997.

[6] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high-throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, July 1998.

[7] John T. Robinson. Analysis of steady-state segment storage utilizations in a log-structured file system with least-utilized segment cleaning. *Operating Systems Review*, 30(4):29–32, October 1996.

[8] John T. Robinson and Peter A. Franaszek. Analysis of reorganization overhead in log-structured file systems. pages 102–110, February 1994.

[9] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transanctions on Computer Systems*, 10(1):26–52, February 1992.