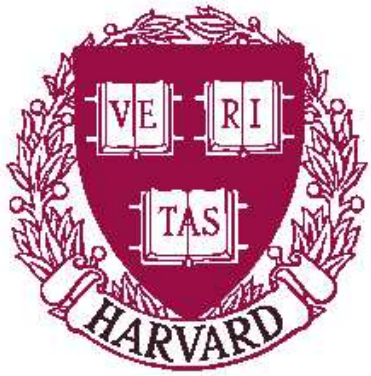
The Utility of File Names

Daniel Ellard, Jonathan Ledlie, Margo Seltzer

TR-05-03



Computer Science Group
Harvard University
Cambridge, Massachusetts

The Utility of File Names

Daniel Ellard, Jonathan Ledlie, Margo Seltzer
Harvard University
{*ellard,jonathan,margo*}*@eecs.harvard.edu*

Abstract

For typical workloads and file naming conventions, the size, lifespan, read/write ratio, and access pattern of nearly all files in a file system are accurately predicted by the name given to the file when it is created. We discuss some name-related properties observed in three contemporary NFS workloads, and present a method for automatically creating name-based models to predict interesting file properties of new files, and analyze the accuracy of these models for our workloads. Finally, we show how these predictions can be used as hints to optimize the strategies used by the file system to manage new files when they are created.

1 Introduction

As CPU and memory bandwidth continue to grow at a much faster rate than disk access speed, disk I/O has become an increasingly important concern to system designers. Although there are constant improvements in disk latency and other aspects of I/O hardware architectures, it is increasingly the case that good I/O performance depends on accurate predictions of future operations that allow the system to anticipate and prepare for requests before they occur. This is particularly true for read requests, where there has been extensive work in prefetching (sometimes called pre-caching), but it is also true that file systems can benefit from knowledge about the size and future access patterns of a file when writing.

A long history of benchmarking has shown the effectiveness of the heuristics made popular by the Fast File System (FFS) [9], including aggressive read-ahead and clustering [17]. FFS also employs the heuristic of grouping files belonging to the same directory in same area on disk, in the expectation that files in the same directory are likely to be accessed together. This idea, along with the concept of *immediate files* (files that are short enough that it makes sense to embed them in the same disk page as their metadata [11]) has been extended to create variants of FFS such as C-FFS [5], which co-locates inode and directory information to improve directory lookups and uses explicit grouping to allow groups of small files in the same directory to be accessed entirely sequentially. These heuristics, while effective in many cases, are limited because they are static and do not adapt to variations in workload that violate the assumptions of their designers.

To overcome this limitation, there has been research in dynamically learning the inter-file access patterns among groups of files, both for general-purpose file systems [6, 8] and for highly tuned special-purpose file systems for applications like web servers (such as the Hummingbird file system [18]).

There has also been considerable work in developing and exploiting predictive models at the disk-block level, instead of the file system level. These can be done at the hardware level (*i.e.,* AutoRAID [20]) or under the control of the operating system [1] or via a hybrid of block and file level optimization [19].

Every widespread heuristic approach suffers from at least one of the following problems: first, if the heuristics are wrong, they may cause performance to degrade, and second, if the heuristics are dynamic, they may take considerable time, com-

putation, and storage space to adapt to the current workload (and if the workload varies over time, the adaptation might never converge).

One partial solution to the problem of inappropriate or incomplete file system heuristics is for the application using the files to supply *hints* to the file system about their anticipated access patterns. These hints can be extremely successful, especially when combined with the techniques of prefetching and selective caching [3, 12]. The drawback with this approach is that it requires that applications be modified to provide hints. There has been work in having the application compiler automatically generate hints, but success in this area has been largely confined to scientific workloads with highly regular access patterns [10].

Based on our analysis of three recent long-term NFS traces, we have discovered that the *names* of files are a powerful predictor of several file properties. In essence, applications (and to a lesser extent, the users of the applications) *already* give useful hints to the file system about each file, in the form of the file name, and this information is available immediately at the time the file is created. Furthermore, it is possible to build name-based predictive models for several file properties that can be used to augment static heuristics for file layout, or to accelerate the process of developing an adaptive model for the usage patterns of the file.

The rest of this paper is organized as follows. In Section 2 we describe how predictive models for file properties can be constructed, and give an analysis of the accuracy of these models for three NFS traces. In Section 3 we briefly describe the NFS traces used in our analysis, and the limitations of our trace-based analysis. An analysis of the relationship between file name features and some properties of the files follows in Section 4. In Section 5 we give an example of a file system optimization for one of our trace workloads, using name-based information. Section 6 discusses our plans for future work and Section 7 concludes.

2 Names-Based Predictions

In an earlier trace study [4], we discovered that for an email-oriented workload, there is a strong relationship between patterns within the names of files and the properties of the files.

It is one thing for a human to be able to perceive patterns in trace data, but it is another thing entirely to be able to automate the process of discovering these patterns and building predictive models that can guide decisions about file system policies. In this section, we show that it is possible to construct accurate predictive models for three distinct workloads.

One of the constraints of the algorithms we use to construct and use our predictive models is that they must be efficient to compute and require low space overhead (both for computing and storing the model). If the model is overly cumbersome to compute or represent, the cost of using it may overshadow any benefit it provides. Our goal, therefore, was to develop an efficient algorithm for building reasonably accurate models, rather than striving to achieve accuracy at any cost.

2.1 File Name Features

The features we use to build our predictive models for file properties are based entirely on file names. We have also considered using some of the additional information that is readily available to the file system when the file is created, including the effective user ID of the user creating the file, the flags used by `open` (or `creat`) when the file is created, properties of the directory within which the file is created, and the time of day, but we have not discovered a way to exploit this information without constructing a substantially larger and more computationally complex model.

The feature used by our algorithm for classifying file names is the set of *components* of the file names. The components are used to construct simple regular expressions that are tested against future file names. Our initial procedure for discovering the components of a file name was very simple:

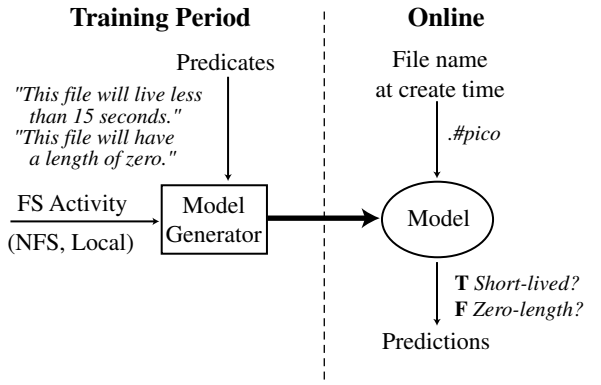


Figure 1: **Generating and using a model.** The model generator takes user-generated predicates and traces of file system activity as input and creates a model. Models take the names of files at creation time and predict whether each of the predicates will be satisfied by the corresponding files. Currently the training is done offline. Example predicates are shown in italics.

---

1. Let S be the file name. Let $E(s)$ be a function to "escape" a string s, protecting any characters in the string that would be interpreted as regular expression operators.

2. Add $E(S)$ with a `^` prepended and `$` appended to the set of components.

3. Prepend `^` and append `$` to S, and then split the resulting string at each period, giving $s_0 \cdots s_n$. For each s_i, add $E(s_i)$ to the set of components.

This extremely simple partitioning worked quite well for predicting the properties of many kinds of files, because many applications construct file names by pasting together components with periods. Unfortunately, some applications do not use periods, but instead build file names by concatenating a random (or opaque) string to a fixed template. For example, the Netscape browser names all of its cached files `cache`*X*`.`*suf* (where *X* is an opaque identifier, and *suf* is the suffix of the original file), and the `pine` composer `pico` creates temporary files that begin with the string `#pico`. For the `cache`*X* files, we can still learn useful information from the suffix, but we do not benefit from learning that they are part of a browser cache, and hence tend to have much different lifetime characteristics than "ordinary" files with the same suffix. The `#pico` files have distinct and interesting properties of their own, but models constructed from these components entirely overlook them. To address this limitation, we add the following step to our component-generating procedure:

- If the file name S contains at least 5 characters, let p be the first 5 characters of S. Add $E(p)$, with a `^` prepended to the set of components.

For example, the file name "prediction.tex" has the components "`^prediction\.tex$`", "`^prediction`", "`tex$`", and "`^predi`".

A diagram of our system is given in Figure 1, and our algorithm for building the predictive model used in the model generator is given in Figure 2. This algorithm produces a model that uses a file name to classify a file as matching one of a set of predicates such as "this file will have a length of zero" or "this file will have a length of more than zero bytes, but less than one megabyte."

Note that this algorithm only properly handles sets of logically disjoint predicates. If more than one predicate is true for a particular file instance, the resulting model will be weakened. For example, the predicates "this file will have a length of zero" and "this file will have a length of less than 8k" are not logically disjoint, because any file that satisfies the first will also satisfy the second. Similarly, the predicates "this file will have a length of zero" and "this file will live for less than 15 seconds" are not logically disjoint, because it is possible for a file to satisfy both of them. If the set of predicates is not logically disjoint, then it is better to partition the set into logically disjoint subsets and construct separate models for each subset.

Also note that the predicates are chosen statically. We envision that file system designers will choose predicates based on properties of the file system to help them to identify files with properties that they can exploit. For example, different block layout strategies might be used for files that are predicted to be very large and accessed sequentially, or very short and rewritten many times.

Figure 2: **An algorithm for building a predictive name-based model.**

- Let $C = \{c_0 \cdots c_n\}$ be a set of disjoint predicates about file attributes.
- Let B be a set of tuples $\langle c, s, n \rangle$, where $c \in C$, s is a pattern, and n is a count. B is initially empty. B is used to store the number of times a file with a name matching pattern s has satisfied the predicate c.
- Let N be a set of tuples $\langle s, n \rangle$, where s is a string, and n is a count. N is initially empty. N is used to store the number of times each string has been observed.

1. For each file observed during the training period:
   (a) Let $S = \{s_0 \cdots s_m\}$ be the set of components of the file name (as defined in Section 2.1).
   (b) Let $M = \{c_i \in C\}$ be the subset of predicates in C that are satisfied by the file. (If the predicates are logically disjoint, then $|M| \leq 1$.)
   (c) For each $s \in S, m \in M$: if there exists a tuple $\langle m, s, n \rangle \in B$, then replace it with $\langle m, s, n+1 \rangle$. Otherwise, add $\langle m, s, 1 \rangle$ to B.
2. For each tuple $t = \langle c, s, n \rangle \in B$, if $n <$ **mincount** then discard t.
   The default **mincount** is 5.
3. For each $t = \langle c_t, s_t, n_t \rangle \in B$, find the corresponding $\langle s_S, n_S \rangle \in S$. If $n_t / n_S <$ **minfrac**, then discard t.
   The default **minfrac** is 0.8.

The final set B defines the predictive model. For each new file name, if there exists any $\langle c, s, n \rangle \in B$ such that the s matches the file name, then we predict that c is true for the corresponding file. Note that it is possible for the model to predict that more than one c will be true. In this case we predict that the rule with the highest n is most likely.

2.2 Computation Costs

Our model generator has two modes. In the first mode, it collects data about every file accessed during the training file. In the second mode, it only collects information about files that are created during the training period.

Our algorithm is quite simple, both in terms of implementation and computational complexity.

If only considering the files for files created during the training period, the algorithm requires less than 25 seconds on a PIII-800MHz to do the processing for a full day of trace data from our busiest system (approximately 60,000 files created per day). Building a model for all of the files observed during the training period requires approximately 40 seconds on a PIII-800MHz for the same day. In both of the cases, most of the time is spent parsing the log files (which are stored as text and could be represented in a more efficient manner); after parsing the files, the additional cost is approximately five seconds per predicate, depending on the complexity of the predicate.

The drawback of this algorithm is that it requires a noticeable amount of space to store the information observed during the training period; for a busy day on our busiest server, the log can consume nearly 40 megabytes.

The largest inefficiency of our current implementation is that it gathers all of its information about each file directly from our NFS traces, which in an uncompressed form can exceed 15G per server per

day. Simply scanning through this volume of data to extract the appropriate 40MB of relevant information is a time-consuming process, but one that is not inherent in our algorithm. We believe, that gathering this information at the file system interface (or from reduced NFS traces that only include the information we need, and not a full transcript of every NFS call and response) would greatly reduce this overhead, both in terms of storage space needed and computational overhead.

2.3 The Accuracy of the Models

To test the effectiveness of our algorithm, we generated and tested models for a variety of predicates from three NFS different workloads. (The three workloads are described in Section 3.) In Table 1 we show the accuracy of the models produced for each of the three systems for four simple predicates. The models were generated during the first day of each test, and then used to predict the properties of the files created during the next seven days.

The δ-err columns of this table are the most interesting, because they express the accuracy of the model in comparison to simple guessing based on overall probabilities. For example, on a typical day we can achieve 95% accuracy by simply guessing that *all* of the files created on CAMPUS will be zero-length (or by guessing that *none* of the files created on EECS will be zero-length). This estimate is accurate enough to be useful by itself, but by considering names we can trim the error rate even further (to nearly perfect for typical days on CAMPUS, and a respectable 98% on EECS). For other predicates, such as *Small File* on EECS, we can typically increase the accuracy from approximately 50% to more than 80% by considering names.

Our models appear to be accurate as long as the workload remains similar. For workloads like CAMPUS, they converge after only a few hours of training data. We have run our models for as long as 12 consecutive days with no perceptible loss in overall accuracy. However, there are situations where the models are no better (and sometimes considerably worse) than simple guessing. For example, the *Small File* and *Write-Only* predictions are much worse for CAMPUS on 10/29/01. From our examination of the traces and the output of our testing system (which records the file names for which the predictions were incorrect, for future training), the reason is quite clear: on that day, one user untar'd the source code for a large software package (but did not compile the package), thus creating several thousand small files that were written but never read. Small write-only files are rare on CAMPUS, and the naming convention used for the source files for this particular package did not match any files seen during the training day.

Our models suffer from the same flaw as any other predictive model: they can only be as good as their training. If behavior of the system changes rapidly, then our model will degrade in accuracy.

2.4 Discussion

Our primary focus has been on predicting the behavior of future files (especially short-lived files) based on the properties of files created during the training period ("active" files), and not on properties of files that were created before the training period. On CAMPUS, for example, nearly all of the read and write traffic is to mailbox files, and these files are essentially immortal [4]. By default, our model generator do not attempt to "learn" anything about files like mailboxes, because it is primarily interested in the properties of newly-created files. We have experimented with extending our training to make note of the properties of the older files in the system as well, but this requires more processing and slightly decreases the accuracy of the short-term predictions. We do not believe that this is an critical problem, however, since the problem of dynamically adapting to the access patterns of long-lived files is relatively well-studied. We feel it is more important to fine-tune our name-based predictions for the behavior of a file while it is young (and use other methods to deal with the file as it ages), if a trade-off must be made.

	Zero-Length		Lock File		Small File		Write Only	
	DEAS		**DEAS**		**DEAS**		**DEAS**	
	$p = 63.50\%$		$p = 63.08\%$		$p = 30.62\%$		$p = 27.77\%$	
Date	**Correct**	**δ−err**	**Correct**	**δ−err**	**Correct**	**δ−err**	**Correct**	**δ−err**
10/21/02	98.79%	96.68%	98.95%	97.16%	82.54%	42.98%	85.70%	48.51%
10/22/02	96.75%	91.10%	98.44%	95.77%	83.80%	47.09%	85.91%	49.26%
10/23/02	95.66%	88.11%	95.76%	88.52%	80.42%	36.05%	80.62%	30.21%
10/24/02	97.24%	92.44%	97.41%	92.98%	81.87%	40.79%	83.41%	40.26%
10/25/02	97.68%	93.64%	98.01%	94.61%	82.86%	44.02%	83.33%	39.97%
10/26/02	97.74%	93.81%	98.66%	96.37%	77.18%	25.47%	77.71%	19.73%
10/27/02	98.72%	96.49%	98.90%	97.02%	81.41%	39.29%	82.06%	35.40%
10/28/02	97.71%	93.73%	97.78%	93.99%	77.99%	28.12%	78.71%	23.33%
	CAMPUS		**CAMPUS**		**CAMPUS**		**CAMPUS**	
	$p = 95.19\%$		$p = 78.17\%$		$p = 4.39\%$		$p = 0.24\%$	
Date	**Correct**	**δ−err**	**Correct**	**δ−err**	**Correct**	**δ−err**	**Correct**	**δ−err**
10/22/01	99.84%	96.67%	94.95%	76.87%	99.49%	88.38%	99.76%	0.00%
10/23/01	99.71%	93.97%	95.04%	77.28%	98.98%	76.77%	99.76%	0.00%
10/24/01	99.80%	95.84%	94.92%	76.73%	99.02%	77.68%	99.77%	4.17%
10/25/01	99.83%	96.47%	95.21%	78.06%	99.27%	83.37%	99.75%	-4.17%
10/26/01	99.87%	97.30%	95.81%	80.81%	98.78%	72.21%	99.86%	41.67%
10/27/01	99.93%	98.54%	98.34%	92.40%	99.54%	89.52%	99.94%	75.00%
10/28/01	99.90%	97.92%	97.73%	89.60%	99.27%	83.37%	99.89%	54.17%
10/29/01	98.46%	67.98%	95.04%	77.28%	89.29%	-143.96%	99.13%	-262.50%
	EECS		**EECS**		**EECS**		**EECS**	
	$p = 5.04\%$		$p = 4.59\%$		$p = 49.95\%$		$p = 57.75\%$	
Date	**Correct**	**δ−err**	**Correct**	**δ−err**	**Correct**	**δ−err**	**Correct**	**δ−err**
10/22/01	98.86%	77.38%	99.29%	84.53%	89.30%	78.58%	74.58%	39.83%
10/23/01	99.37%	87.50%	99.65%	92.37%	91.78%	83.54%	67.85%	23.91%
10/24/01	98.11%	62.50%	98.80%	73.86%	81.29%	62.54%	81.06%	55.17%
10/25/01	97.77%	55.75%	98.67%	71.02%	81.68%	63.32%	80.73%	54.39%
10/26/01	98.70%	74.21%	99.11%	80.61%	85.46%	70.89%	77.12%	45.85%
10/27/01	99.04%	80.95%	99.45%	88.02%	65.23%	30.39%	72.42%	34.72%
10/28/01	97.23%	45.04%	98.27%	62.31%	65.70%	31.33%	77.84%	47.55%
10/29/01	97.97%	59.72%	98.68%	71.24%	83.87%	67.71%	91.11%	78.96%

Table 1: **Accuracy of several predicates for each of the files created on each day.** The "Zero-Length" predicate is that the file will never contain any data. The "Lock File" predicate is "this file will be zero-length and also live less than 5 seconds." The "Small File" predicate is "this file will be written to, but will not grow to more than 16k." The "Write Only" predicate is that data will be written to the file but the file will never be read (at least during the 24-hour testing period). The predictive model is created by using data from the first day only, and then held fixed for the rest of the test. The δ-err is the improvement in accuracy provided by the model, compared with simply guessing the value of each predicate for each file based on the observed probability for that predicate across all files created on the day that the model was built (given as the p for each model).

3 Workloads

To investigate the accuracy of name-based predictions, we analyzed traces from three systems, which are referred to in this paper as CAMPUS, EECS, and DEAS. The CAMPUS and EECS traces have been described and analyzed extensively in previous work [4], but the DEAS traces are new[1].

CAMPUS The CAMPUS system is the primary email system for the Harvard College campus and the Harvard Graduate School of Arts and Sciences (GSAS). At the time the trace was taken, user accounts for the entire college and GSAS (including students and administration) were spread over 14 NFS file systems. The traces analyzed here are taken from one of these file systems.

EECS The EECS system is a Network Appliance filer that hosts the home directories for several research groups in the departments of electrical engineering and computer science at Harvard University.

This file server sees a predominantly engineering workload without email or WWW server traffic. The mail spool for EECS users and the main EECS web site are stored on different servers (for which we do not have traces). We do observe traffic caused by users filing their mail in archives in their home directories, and caused by the department web server accessing the home pages of individual users, but this is a relatively small fraction of the workload.

DEAS The DEAS system is a Network Appliance filer that hosts the home directories for members of the Division of Engineering and Applied Sciences (DEAS) at Harvard. This system also serves the mail spool for DEAS users and the web pages for the central DEAS web site.

The workload of this system can be described as a combination of the CAMPUS and EECS workloads, since it contains elements of both research and development workloads and the email-oriented workloads. It also sees some traffic due to web service.

[1] Anonymized forms of all of all three traces are freely available. Contact `ellard@eecs.harvard.edu` for more information.

	DEAS 10/21-10/27 (2002)	**CAMPUS** 10/22-10/28 (2001)	**EECS** 10/22-10/28 (2001)
Total ops	211308494	187974468	29550778
`getattr`	41.09%	2.18%	21.91%
`lookup`	3.40%	5.70%	41.52%
`access`	18.24%	2.81%	6.16%
`read`	24.63%	64.82%	9.83%
`write`	9.87%	21.49%	15.43%
Read (MB)	833135.01	845123.07	32498.44
Write (MB)	242376.70	313987.75	61488.73
R/W Ratio	3.43	2.69	0.53

Table 2: **Aggregate statistics for the analysis period**

A summary of the average hourly activity for all of the hours during the test week as well as just the peak usage hours (9:00am-6:00pm Monday through Friday) is given in Table 3. As in our previous trace study [4], we give the statistics for the peak usage hours as well as the entire analysis period, because this gives a more accurate representation of the way the systems behave when they are under load. Note that we are using slightly different subsets of the data for EECS and CAMPUS than the ones analyzed in our earlier work, because for our analyses it makes more sense to use a week starting and ending on Monday (rather than Sunday).

3.1 Discussion

From the aggregate and hourly statistics, it is clear that these workloads differ. The DEAS and CAMPUS workloads are heavily read-oriented, (with a read/write ratio of almost 3) although the DEAS workload also includes many requests for metadata (`getattr` and `access`). On EECS, in contrast, the most frequent operation is `lookup`, and writes outnumber reads by reads by almost 2 to 1.

	All Hours		
	DEAS	CAMPUS	EECS
Total Ops (thousands)	1258 (81%)	1119 (49%)	176 (92%)
Data Read (MB)	4959 (62%)	5030 (44%)	193 (182%)
Read Ops (thousands)	310 (60%)	725 (48%)	17.3 (122%)
Data Written (MB)	1443 (90%)	1869 (57%)	366 (254%)
Write Ops (thousands)	1241 (103%)	240 (57%)	27.1 (211%)
	Peak Hours Only		
	DEAS	CAMPUS	EECS
Total Ops (thousands)	1477 (34%)	1699 (8%)	267 (69%)
Data Read (MB)	7559 (26%)	7153 (7%)	268 (146%)
Read Ops (thousands)	467 (27%)	1088 (7%)	29.2 (77%)
Data Written (MB)	2533 (32%)	2934 (12%)	438 (228%)
Write Ops (thousands)	210 (44%)	377 (12%)	34.1 (159%)

Table 3: **Average Hourly Activity.** The *All Hours* columns are for the entire week of 10/21-10/27/2002 for DEAS and 10/22-10/28/2001 for EECS and CAMPUS. The peak hours are the hours 9:00am - 6:00pm, Monday through Friday of these respective weeks. The numbers in parentheses are the standard deviations of the hourly averages, expressed as a percentage of the mean.

4 Name-Based Analysis

4.1 CAMPUS and DEAS

On CAMPUS we can predict the size, lifespan, and access patterns of most files extremely well simply by examining the last component of the pathname. Nearly all of the files accessed on CAMPUS fall into one of the four categories: lock files, dot files (application configuration files, which on UNIX are typically located in the user's home directory, and prefixed with a period), temporary files created by the editors used to create mail messages, and mailboxes. The size, lifespan, and access patterns are predicted strongly for each of these categories.

In our earlier study, we observed that zero-length lock files used by mail programs make up approximately 96% of the files that were both created and deleted, and 99.9% of these lock files lived less than 0.40 seconds. Temporary files created by the mail composer accounted for 2.5% of the files created each day; 45% of these live less than 1 minute, and 98% are less than 8K in length, and 99.9% are smaller than 40K. The dot files are also small, although there are some multi-block dot files (for example, the primary mail client configuration file, `.pinerc`, that varies in size between 11K and 26K). The mailbox files (including both the primary `inbox` and files used to archive mail) are considerably larger than any other commonly-accessed file and are rarely deleted. Because the CAMPUS workload is entirely dominated by email, it is not surprising that the workload can be characterized in terms of file names.

Table 4 shows the distributions of file access count, I/O cost, and boxplots for the size and lifespan of the common suffixes or prefixes of file names observed on CAMPUS during a single day, sorted by I/O cost. Our metric for estimating the total *cost* of the operations on a file is the number of accesses to the file, plus $0.125\times$ the number of 8k blocks read or written. The scale factor of 0.125 is intended to be an approximation of the additional I/O that each block requires (assuming a reasonably contiguous disk layout). This is a very crude metric, but it does give some feel for the amount of file system activity associated with the file.

Table 4 also shows the same information for DEAS.

DEAS

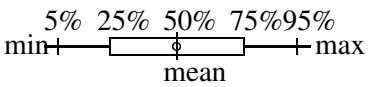
String	Count	Cost	Size	Lifespan
*.lock	35315	35315		
*.HOST	31900	31924		
sent-mail	57	21481		(NA)
mbox	37	18396		(NA)
*.so	450	17144		
*.pac	11	15912		(NA)
*.odb	95	12907		
*.gif	9008	11771		
*.stt	150	9791		
*.abq	11	8755		(NA)
INBOX	12	5138		
*.dat	340	4794		
*.inp	97	4656		
*.res	106	4492		
*.pdf	472	3232		
*.jpg	1563	2876		
*.html	1492	2124		
*.db	97	2037		(NA)
*.ps	352	1761		
*.x	16	1755		(NA)
*.o	448	1567		
*.prt	95	1550		
*.m	1287	1351		
*.mp3	29	1164		(NA)
*.sel	11	1150		(NA)

CAMPUS

String	Count	Cost	Size	Lifespan
*.inbox	1222	2011892		(NA)
*.lock	76027	76029		
*.HOST	65383	65383		
sent-mail	553	22651		(NA)
Sent Items	37	14046		(NA)
#pico*	2746	4987		
SENT-MAIL-*	592	2068		
INBOX	12	1731		(NA)
*.doc	1459	1693		
*.history	789	1572		(NA)
*.pinerc	704	1365		
*.login	767	1148		(NA)
*.cshrc	765	1141		(NA)
*.aliases	745	1117		(NA)
*.logout	633	907		(NA)
postponed-msgs	259	710		
saved-messages	160	608		(NA)
*.jpg	306	575		
*.letter	500	541		
*.addressbook	413	482		
*.mp3	18	436		(NA)
*.lu	253	360		
friends	15	284		(NA)
*.pdf	198	257		
*.newsrc	134	206		

Table 4: **Per-file statistics for simple file name prefix/suffix strings on DEAS and CAMPUS, sorted by file cost, for files accessed on 10/21/2002 (on DEAS) and 10/22/2001 (for CAMPUS).**

In this table, the **String** is either a prefix or suffix of a file name. The special string HOST is substituted when the file prefix or suffix is the name of the originating host. The **Count** is the number of times a file name with a matching prefix or suffix was accessed during a 24-hour period. The **Cost** is an estimate of the amount of I/O activity actually generated by these accesses. The **Size** column shows the file size distribution, shown as a log-scale boxplot (ranging from 0 to 2^{32} bytes). The **Lifespan** column shows a linear-scale boxplot of the lifespan for the files created within this 24-hour period (ranging from 0 to 24 hours). If fewer than 10 files with the same prefix/suffix were created during the sample period, a (NA) is shown instead of a boxplot.
For our boxplots, the horizontal lines extend to the minimum and maximum values in the sample. The vertical lines of the box plot represent, from left to right, 5%, 25% (lower quartile), 50% (median), 75% (upper quartile), and 95% boundaries. The mean value is shown with a small circle. Thus, for a uniform distribution, the boxplot would look like:



EECS

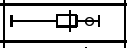
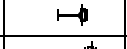
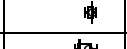
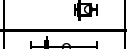
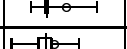
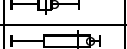
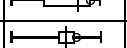
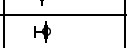
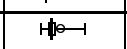
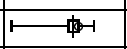
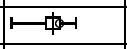
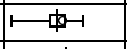
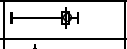
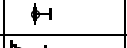
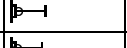
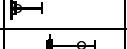
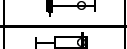
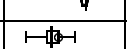
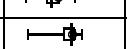
String	Count	Cost	Size	Lifespan
*.3DIG	712	153219		
perfdb0	73	39235		
perfdb1	73	18477		
RMAIL	38	7804		
core	17	7288		(NA)
*.gz	4976	6126		(NA)
*.gif	5045	5459		
mbox	11	5333		(NA)
*.ps	1145	5224		
Applet_B	5009	5129		
Applet_C	4087	4130		
3DIG	2820	4118		
*.pdf	343	4092		
*.html	2104	3002		
*.jpg	1909	2824		
*.o	1537	2625		
Applet_A	2489	2491		
*.HOST	2230	2326		
*.lock	2223	2304		(NA)
10	21	1769		
*.save	21	1525		
*.prev	12	1515		
*.c	1363	1459		
*.db	66	1421		(NA)
*.so	106	1404		

Table 5: **Per-file statistics for simple file name prefix/suffix strings on EECS, sorted by file cost, for files accessed on 10/22/2001.** *Please refer to Table 4 for a description of each column.*

4.2 EECS

User activity on EECS is a union of several kinds of activities, and the observed workload is more complex than CAMPUS although less rich than DEAS. However, our preliminary analyses show that for most files on EECS, the pathname of a file is also a strong predictor of file properties.

Table 5 illustrates the distributions of file access count, cost, size and lifespan by file name.

5 Using the Predictions

In order to test our theory that name-based information can enhance file system performance, we have constructed an experiment that shows how this might be accomplished.

We speculate that it is possible to improve the performance of an email workload by treating the lock files and short-lived files differently than ordinary files. Using our named-based predictive models, we have shown that we can do an excellent job of identifying these files at the time they are created, but it remains to be seen whether this information is actually useful. To test the assumption that there is some benefit to handling small, short-lived files differently from other files, we have devised an experiment where we can perfectly predict which files will be small and short-lived.

We modified Postmark, a mail-oriented benchmark [7], to differentiate between lock files, small mail composer files, and mailboxes. Mailboxes are always treated as ordinary files, but our modified benchmark allows us to treat the lock files and composer files differently. In this section, we discuss how the benchmark was modified and provide a discussion of how a file system could perform if it were aware *a priori* of the characteristics of files at the times of their creation. The experiment shows what could be gained through an accurate name-based predictor for lock files and message files.

5.1 Experimental Setup: Embedding an LFS

Because zero-length files contain no data blocks, manipulating them is purely a meta-data problem. Solving these problems has been addressed by techniques such as soft updates and journaling [16]. However, these techniques require disk head movement and incur a rotational delay. We believe that a better approach would be to use a purely log-structured approach, such as LFS [13], to store these files. For short-lived files in particular, LFS is particularly advantageous because the overhead of creating the files is low, and the cleaning process is efficient because we expect nearly all of the files in a segment will die before their segment is cleaned.

For a single-spindle solution, the best solution would be to combine the layout policies of FFS for ordinary files with a log-based scheme like LFS for the short-lived files. Building a new file system to combine these would be a major undertaking, but we can approximate the behavior of such a system with a relatively simple experiment.

Our methodology is to use the default FFS for ordinary files, and LFS for the files we anticipate will be short-lived. We construct an ordinary FFS file system and an LFS file system on a single disk in such a way that the space used by the FFS and LFS file systems are interleaved. This is done by initially partitioning the disk into three partitions: two large partitions, with a smaller partition between them. The two large partitions are then glued together to form one virtual partition, using `ccd`[2], and a FFS is built on top of this partition. An LFS is built on top of the middle partition. This arrangement is illustrated in Figure 3.

The purpose of embedding the LFS within the space used by the FFS is to minimize the penalty of seeking from one file system to the other when we switch from accessing ordinary files to creating and deleting lock files. By locating the LFS within the region of the disk used by the FFS, we more closely mimic the actual behavior we would observe if the LFS and FFS actually shared the same cylinders of the disk.

5.2 Postmark

Postmark is an industry-standard synthetic benchmark for measuring electronic mail, netnews, and web-based commerce. It performs create/delete and read or append transactions on a user-specified workload. It decides what size to make new files and how much to append to them by choosing uniformly at random between smallest and largest possible file sizes. This simple knob does not accurately depict CAMPUS accesses: while most bytes accessed do come from files larger than several kilobytes, the vast majority of accesses in the mail workload were to small files. Instead, accesses

[2] `ccd` is the *Concatenated Disk Driver*, a utility provided with several UNIX distributions.

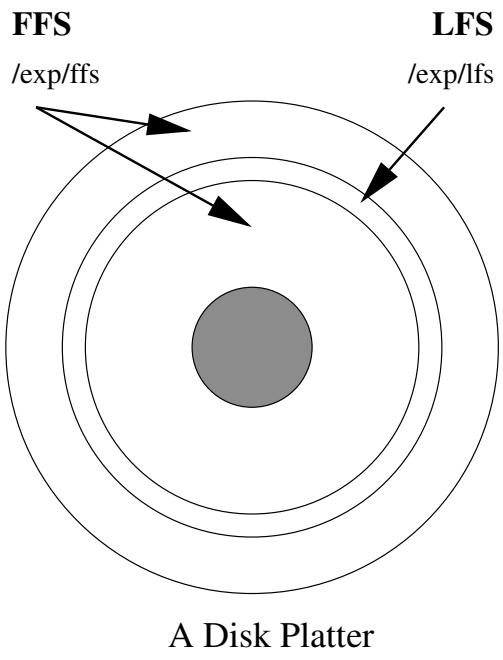


Figure 3: An FFS and LFS sharing the same disk drive, with the LFS "embedded" within the FFS.

tended to fall into three categories: zero-length lock files, generally short dot and composer files, and large mailboxes.

We modified Postmark in three ways to support our experiments:

1. Instead of a uniform distribution, we modified the file creation size to provide a "stepped" distribution, which follows a simplification of our observations of CAMPUS: 96% of all files are 0 byte lock files, 3% are between 1 byte and 16 KB, and 1% are between 16 KB byte and 1 MB.

2. Based on an input parameter, we modified Postmark to put either no files, lock files, or lock and small files into the LFS partition (/exp/lfs). Any files that were not put in LFS were put into FFS.

3. The lock files and small files were not appended to once created.

We used Schoder's NetBSD LFS implementation [15] and the default FFS. *Gerbil*'s LFS partition is

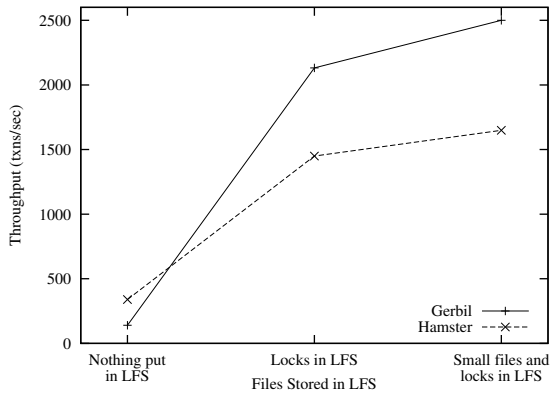


Figure 4: Modified Postmark. Number of transactions per second shown with no files stored in LFS, just locks, and locks and small files. LFS is functioning as a poor-man's circular buffer.

160 MB and *hamster*'s is 128 MB. Both FFS partitions were 6 GB.

The experiment was performed on two machines with similar configurations: 600Mhz Pentium III processor, 128MB of RAM, and NetBSD 1.6 OS. The differences in hardware configuration are that one machine, *gerbil*, has an IDE DISK (IBM-DPTA-372050) whereas the other machine, *hamster*, here has an older SCSI disk.

We configured Postmark to simulate 50 concurrent mail users with the following parameters: 50 subdirectories (one per user), reads/writes occur in 8k blocks, the maximum number of files was 500, and reads were biased over appends by $7:3$, similar to the read/write ratio seen with CAMPUS. Figure 4 shows the average of five runs of 30,000 transactions; variance was less than 10% of the average.

This experiment shows that using LFS to store the lock files and message files can provide an order of magnitude increase in the number of transactions compared to the same configuration using only FFS. Of course, this only holds when we can accurately predict the future access patterns of the files, and avoid putting long-lived files into the LFS partition. While the particular performance gain and its peak between 64b and 256b are an artifact of our synthetic workload, the experiment does provide evidence that there is a potential for significant performance gains if properties such as file size and lifespan can be correctly predicted and handled accordingly.

6 Future Work

All of our trace data is from NFS-based systems; it remains an open question whether we can build useful name-based models to predict the properties of files on non-UNIX and/or non-NFS workloads.

We leave two aspects of our experimentation for future work. First, we would like to run a similar test to our Postmark test on the more tune-able Fstress synthetic workload generator [2]. It includes eleven sample workloads, one of which is a mail workload based on Saito's analysis of mail file size as a Zipf distribution with α set to 1.3, an average message size of 4.7 KB, and a long-tail Pareto distribution to 1MB [14]. Like the uniform Postmark distribution, this distribution would need to be augmented with more zero-length lock files than would appear with a Zipf distribution.

The second additional aspect of our experimentation we would like to add is a "copy" thread that runs immediately in front of the LFS-cleaner thread. This thread would examine a block in the circular buffer (LFS in our experiments) and copy over to the standard file system (FFS) any files that had not been deleted and mark the contents in the buffer deleted. The cleaner would then always see empty blocks. This would serve the purpose of fixing our guesses when we were wrong. Anything that lived longer than one sweep through the buffer should exist in the standard file system and this mechanism serves to put it there.

These two steps would provide more information on the performance of the circular buffer in a mail-oriented workload. The next logical step is a kernel-level file system that transparently incorporates a name-based predictor, a circular buffer, a migration thread to copy incorrect guesses to long-term storage, and the long-term storage itself.

7 Conclusions

We have shown that there is a strong relationship between file names and the properties of the files on three distinct workloads. We have shown that models that discover and use these relationships to predict the properties of new files created on these systems are efficient to construct and evaluate. We have also demonstrated that our models are highly accurate when predicting several properties of interest to file system designers, including file size and lifespan.

References

[1] Sedat Akyurek and Kenneth Salem. Adaptive block rearrangement. *Computer Systems*, 13(2):89–121, 1995.

[2] Darrell Anderson and Jeff Chase. Fstress: A Flexible Network File Service Benchmark. Technical Report TR-2001-2002, Duke University, May 2002.

[3] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.

[4] Dan Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS tracing of an email and a research workload. In *Proceedings of the 2nd USENIX FAST Conference*, 2003.

[5] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *USENIX Annual Technical Conference*, pages 1–17, 1997.

[6] J. Griffioen and R. Appleton. Improving file system performance via predictive caching, September 1995.

[7] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance, October 1997.

[8] Thomas M. Kroeger and Darrell Long. The case for efficient file access pattern modelling. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, AZ, March 1999. IEEE.

[9] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[10] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 3–17. USENIX Association, 1996.

[11] S. Mullender and A. Tanenbaum. Immediate files. In *Software – Practice and Experience*, number 4 in 14, Apr 1984.

[12] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *ACM SOSP Proceedings*, 1995.

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

[14] Yasushi Saito, Brian N. Bershad, and Henry M. Levy. Manageability, availability and performance in porcupine: A highly scalable, cluster-based mail service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP-99)*, pages 1–15, 1999.

[15] Konrad Schroder. LFS NetBSD implementation. `http://www.hhhh.org/perseant/lfs.html`.

[16] Margo Seltzer, Greg Ganger, M. Kirk McKusick, Keith Smith, Craig Soules, and Christopher Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *USENIX Annual Technical Conference*, June 2000.

[17] Margo Seltzer and Keith Smith. File system logging versus clustering: A performance comparison. In *USENIX Annual Technical Conference*, June 1995.

[18] Elizabeth Shriver, Eran Gabber, Lan Huang, and Christopher A. Stein. Storage management for web proxies. In *USENIX Annual Technical Conference*, pages 203–216, June 2001.

[19] Carl Hudson Staelin. High performance file system design. Technical Report TR-347-91, Princeton University, 1991.

[20] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 90–106. IEEE Computer Society Press and Wiley, 2001.