Attribute-Based Prediction of File Properties

Daniel Ellard,[*] Michael Mesnier,[‡] Eno Thereska,[†] Gregory R. Ganger,[†] Margo Seltzer[*]

Abstract

We present evidence that attributes that are known to the file system when a file is created, such as its name, permission mode, and owner, are often strongly related to future properties of the file such as its ultimate size, lifespan, and access pattern. More importantly, we show that we can exploit these relationships to automatically generate predictive models for these properties, and that these predictions are sufficiently accurate to enable optimizations.

1 Introduction

In "Hints for Computer System Design," Lampson tells us to "Use hints to speed up normal execution." [14] The file system community has rediscovered this principle a number of times, suggesting that hints about a file's access pattern, size, and lifespan can aid in a variety of ways including improving the file's layout on disk and increasing the effectiveness of prefetching and caching. Unfortunately, earlier hint-based schemes have required the application designer or programmer to supply explicit hints using a process that is both tedious and error-prone, or to use a special compiler that can recognize specific I/O patterns and automatically insert hints. Neither of these schemes have been widely adopted.

In this paper, we show that applications *already* give useful hints to the file system, in the form of file names and other attributes, and that the file system can successfully predict many file properties from these hints.

We begin by presenting statistical evidence from three contemporary NFS traces that many file attributes, such as the file name, user, group, and mode, are strongly related to file properties including file size, lifespan, and access patterns. We then present a method for automatically constructing tree-based predictors for the properties of a file based on these attributes and show that these predictions are accurate. Finally, we discuss uses for such predictions, including an implementation of a system that uses them to improve file layout by anticipating which blocks will be the most frequently accessed and grouping these blocks in a small area on the disk, thereby improving reference locality.

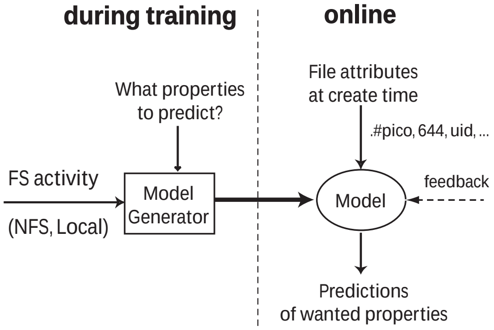


Figure 1: Using file attributes to predict file properties. During the training period a predictor for file properties (*i.e.*, lifespan, size, and access pattern) is constructed from observations of file system activity. The file system can then use this model to predict the properties of newly-created files.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 describes the collection of NFS traces we analyze in this study. Section 4 makes the case for attribute-based predictions by presenting a statistical analysis of the relationship between attributes of files and their properties. Section 5 presents ABLE, a classification-tree-based predictor for several file properties based on their attributes. Section 6 discusses how such models might be used, and demonstrates an example application which increases the locality of reference for on-disk block layout. Section 7 concludes.

2 Related Work

As the gap between I/O and CPU performance has increased many efforts have attempted to address it. An entire industry and research community has emerged to

[*] Harvard University Division of Engineering and Applied Sciences.

[†] Parallel Data Laboratory, Carnegie Mellon University.

[‡] Intel Corporation and Parallel Data Laboratory, Carnegie Mellon University.

attack I/O performance; file systems have been modified, rewritten and rethought in attempts to reduce the number of synchronous disk requests. Significant effort has also been expended to make caches more effective so that the number of disk requests can be reduced. Many powerful heuristics have been discovered, often from the analyses of real workloads, and incorporated into production file systems. All of these endeavors have been productive, but I/O performance is still losing ground to CPU, memory, and network performance, and we have not resolved the I/O crisis to which Patterson refers in the original RAID paper, written more than fifteen years ago [24].

There is extensive ongoing research in the file system and database communities regarding the optimization of various aspects of performance, reliability, and availability of data access. Many heuristics have been developed and incorporated into popular file systems like the Fast File System (FFS) [17]. Many of these heuristics depend on assumptions about workloads and file properties.

One example of a contemporary file system is the Fast File System (FFS) [17], whose basic design is nearly twenty years old and yet continues to be tuned [7]. For example, FFS is optimized to handle small files in a different manner than large files; it attempts to organize small files on disk so that they are near their metadata and other files in the directory, under the assumption that files in the same directory are often accessed together. Some file systems go to more extreme lengths, such as storing the contents of short files in the same disk block as their inode [22] or storing the directory and inode information in the same block [11].

In addition to size, other properties of files, such as whether they are write-mostly or read-mostly, have been found useful to drive various file system policies. For example, the assumption underlying the design of the log-structured file system (LFS) is that write-latency is the bottleneck for file system performance [26]. Hybrid schemes that use LFS to store write-mostly files have also found this approach useful [23]. In contrast, if a file is known to be read-mostly, it may benefit from aggressive replication for increased performance and availability [27].

Unfortunately, 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, computation, 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 applications to supply *hints* to the file system about the files' anticipated access patterns. In some contexts these hints can be extremely successful, especially when used to guide the policies for prefetching and selective caching [5, 25]. The drawback of this approach is that it requires that applications be modified to provide hints. There has been work in having the compiler automatically generate hints, but success in this area has been largely confined to scientific workloads with highly regular access patterns [21], and no file system that uses these ideas has been widely deployed.

In previous work, we noted that for some workloads, applications (and the users of the applications) already provide hints about the future of the files that they create via the names they choose for those files [8]. In this paper we generalize this finding and show that file names, as well as other attributes such as uid and mode, are, in fact, hints that may be useful to the file system.

In addition to static analyses of workloads, there has been research aimed at understanding the dynamic behaviors of files. Previous work has shown that properties of files depend on the applications and users accessing them [2, 9] and because users and applications may change, workloads change as well.

Considerable work has been done in developing and exploiting predictive models for the access patterns of files (or their data blocks) [1, 30, 31]. Most work in this area focuses on rather complex and computationally expensive predictive models. Furthermore, such models are often needed on a file-by-file basis and do not attempt to find relationships or classes among files to generalize [15]. We extend this work by providing a framework for automatically classifying files with similar behaviors.

There also exist systems that use a variety of layout policies that provide non-uniform access characteristics. In the most extreme case, a system like AutoRAID [31] employs several different methods to store blocks with different characteristics. On a more mundane level, the performance of nearly all modern disk drives is highly influenced by the multi-zone effect, which can cause the effective transfer rate for the outer tracks of a disk to be considerably higher than that of the inner [19]. There is ample evidence that adaptive block layout can improve performance; we will demonstrate that we can preemptively determine the layout heuristics to achieve this benefit without having to reorganize files after their initial placement.

Advances in artificial intelligence and machine learning have resulted in efficient algorithms for building accurate predictive models that can be used in today's file systems. We leverage this work and utilize a form of classification tree to capture the relationships between file attributes and their behaviors, as further described in Section 5.

The work we present here does not focus on new heuristics or policies for optimizing the file system. Instead it enables a file system to choose the proper policies to apply by predicting whether or not the assumptions on which these policies rely will hold for a particular file.

3 The Traces

To demonstrate that our findings are not confined to a single workload, system, or set of users, we analyze traces taken from three servers:

DEAS03 traces a Network Appliance Filer that serves the home directories for professors, graduate students, and staff of the Harvard University Division of Engineering and Applied Sciences. This trace captures a mix of research and development, administrative, and email traffic. The DEAS03 trace begins at midnight on 2/17/2003 and ends on 3/2/2003.

EECS03 traces a Network Appliance Filer that serves the home directories for some of the professors, graduate students, and staff of the Electrical Engineering and Computer Science department of the Harvard University Division of Engineering and Applied Sciences. This trace captures the canonical engineering workstation workload. The EECS03 trace begins at midnight on 2/17/2003 and ends on 3/2/2003.

CAMPUS traces one of 14 file systems that hold home directories for the Harvard College and Harvard Graduate School of Arts and Sciences (GSAS) students and staff. The CAMPUS workload is almost entirely email. The CAMPUS trace begins at midnight 10/15/2001 and ends on 10/28/2003.

Ideally our analyses would include NFS traces from a variety of workloads including commercial datacenter servers, but despite our diligent efforts we have not been able to acquire any such traces.

The DEAS03 and EECS03 traces are taken from the same systems as the DEAS and EECS traces described in earlier work [9], but are more recent and contain information not available in the earlier traces. The CAMPUS trace is the same trace described in detail in an earlier study [8], although we draw our samples from a longer subset of the trace. All three traces were collected with `nfsdump` [10].

Table 1 gives a summary of the average hourly operation counts and mixes for the workloads captured in the traces. These show that there are differences between these workloads, at least in terms of the operation mix. CAMPUS is dominated by reads and more than 85% of the operations are either reads or writes. DEAS03 has proportionally fewer reads and writes and more metadata requests (`getattr`, `lookup`, and `access`) than CAMPUS, but reads are still the most common operation. On EECS03, meta-data operations comprise the majority of the workload.

Earlier trace studies have shown that hourly operation counts are correlated with the time of day and day of week, and much of the variance in hourly operation count is eliminated by using only the working hours [8]. Table 1 shows that this trend appears in our data as well. Since the "work-week" hours (9am-6pm, Monday through Friday) are both the busiest and most stable subset of the data, we focus on these hours for many of our analyses.

One aspect of these traces that has an impact on our research is that they have been anonymized, using the method described in earlier work [8]. During the anonymization UIDs, GIDs, and host IP numbers are simply remapped to new values, so no information is lost about the relationship between these identifiers and other variables in the data. The anonymization method also preserves some types of information about file and directory names – for example, if two names share the same suffix, then the anonymized forms of these names will also share the same suffix. Unfortunately, some information about file names is lost. A survey of the file names in our own directories leads us to believe that capitalization, use of whitespace, and some forms of punctuation in file names may be useful attributes of file names, but none of this information survives anonymization. As we will show in the remaining sections of this paper, the anonymized names provide enough information to build good models, but we believe that it may be possible to build even more accurate models from unanonymized data.

4 The Case for Attribute-Based Predictions

To explore the associations between the create-time attributes of a file and its longer-term properties, we begin by scanning our traces to extract both the initial at-

All Hours					
Host	`read`	`write`	`lookup`	`getattr`	`access`
DEAS03	48.7% (50.9%)	15.7% (55.3%)	3.4% (161.6%)	29.2% (49.3%)	1.4% (119.5%)
EECS03	24.3% (73.8%)	12.3% (123.8%)	27.0% (69.5%)	3.2% (263.2%)	20.0% (67.7%)
CAMPUS	64.5% (48.2%)	21.3% (58.9%)	5.8% (44.4%)	2.3% (60.7%)	2.9% (51.4%)
Peak Hours (9:00am – 6:00pm Weekdays)					
DEAS03	50.0% (24.3%)	16.8% (28.9%)	3.4% (29.3%)	26.6% (29.4%)	1.3% (44.8%)
EECS03	18.2% (63.5%)	12.3% (86.7%)	27.0% (33.6%)	3.0% (129.9%)	21.5% (39.8%)
CAMPUS	63.5% (8.5%)	22.3% (16.7%)	5.6% (8.1%)	2.4% (32.6%)	3.0% (10.8%)

Table 1: The average percentage of read, write, lookup, getattr, and access operations for a fourteen day trace from each server. The averages are shown for both all hours during the trace and for the peak hours (9:00am – 6:00pm on weekdays). The coefficient of variation for each hourly average is given in parentheses.

tributes of each file (such as those we observe in `create` calls) and the evolution of the file throughout the trace, so we can record information about its eventual size, lifespan, and read/write ratio. From these observations, we are able to measure the statistical association between each attribute and each property. The stronger the association, the greater the ability to predict a property, given the attribute.

Some of these associations are intuitive: files that have the suffix `.gz` tend to be large, files whose names contain the string `lock` tend to be zero-length and live for a short period of time, etc. Other associations are less obvious. Particular users and groups, for example, often have unique lifespan distributions for their files. We find that the mode of a file (*i.e.,* whether the file is readable or writable) often serves as a fingerprint of the environment in which it was created, and can even expose certain idiosyncrasies of the users and their applications. The mode is often a surprisingly good indicator of how a file will be used, but not as one would expect: on the DEAS03 trace, for example, any file with a mode of `777` is likely to live for less than a second and contain zero bytes of data – it is somewhat nonintuitive that a file that is created with a mode that makes it readable and writeable by any user on the system is actually never read or written by anyone. Most of these files are lock files (files that are used as a semaphore for interprocess communication; their existence usually indicates that a process desires exclusive access to a particular file).

In order to capture some of the information expressed by different file-naming conventions (for example, using suffixes to indicate the format of the contents of a file), we decompose file names into name *components*. Each of these components is treated as a separate attribute of the file. We have found it is effective to use a period ('.') to delimit the components. For example, the file name `foo.bar` would have two name components (`foo` and `bar`). To simplify our analysis, we limit a file name to three name components (first, middle, and last). Files with more than three components would have the remainder subsumed by the middle name. For example, the file `foo.bar.gz.tmp` would have a middle name of `bar.gz`. Filenames with fewer than three components will take on NULL component values. There may be other useful features within file names, but we are constrained by the information that remains after the anonymization process.

In the remainder of this section we use the chi-square test [16] (pages 687–693) to show that the association between a files attributes and properties is more than a coincidence. We provide statistical evidence that the association is significant and quantify the degree of associativity for each attribute.

4.1 Statistical Evidence of Association

We use a chi-square test (also known as a two-dimensional contingency table) to quantify the association between each attribute and property. The chi-square test of association serves two purposes. First, it provides statistical evidence that associations exist and quantifies the degree of associativity for each attribute. Second, it is one of the mechanisms we use to automatically construct a decision tree that uses the information we extract from the traces to predict the properties of files.

If there is no association, then the probability of a file having a given property is independent of the attribute values. For example, suppose that we find that 50% of the files we observe are write-only. If the write-only property is associated with the file name suffix, then this percentage will be different for different suffixes. For example, we may find that 95% of the `.log` files are write-only. If no association exists, then the expected percentage of write-only files with each extension will not differ from 50% in a statistically significant manner. The diffi-

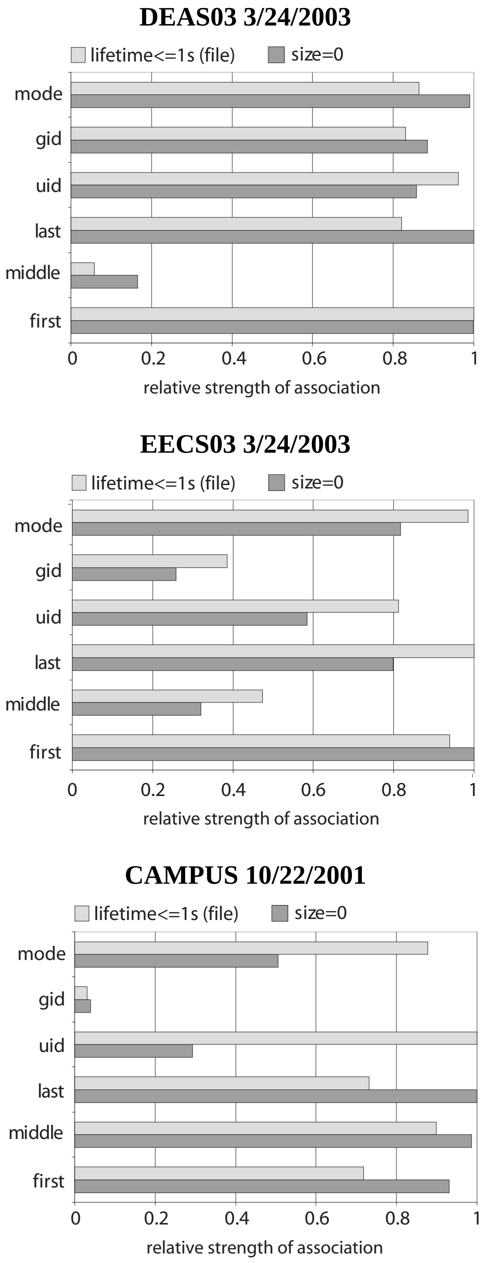


Figure 2: The relative strength of the correlation between the properties "lifetime (lftmd) is one second or shorter" and "size is zero" and several file attributes (as indicated by the chi-squared values) for one day of each trace. The chi-squared values are normalized relative to the attribute with the strongest association. The last, middle, and first attributes refer to components of the file name, as described in Section 5.

culty with such a test is distinguishing natural variation from a statistically significant difference; the chi-squared test is used to detect and quantify such differences.

The sum of squared differences between the expected and observed number of files is our chi-squared statistic, and we calculate this statistic for each combination of attribute and property. In statistical terms, we are trying to disprove the null hypothesis that file attributes are not associated with file properties. A chi-squared statistic of zero indicates that there is no association (*i.e.*, the expected and observed values are the same), while the magnitude of a non-zero statistic indicates the degree of association. This value is then used to calculate a *p-value* which estimates the probability that the difference between the expected and observed values is coincidental.

For all of our tests, we have a high chi-square statistic, and the p-values are very close to zero. Therefore we may, with very high confidence, reject the null hypothesis and claim that attributes are associated with file properties.

The chi-squared test can also be used to rank the attributes by the degree of association. Figure 2 shows how the chi-squared values differ for the size and lifespan properties. There are two important points to take from this figure. First, the attribute association differs across properties for a given trace – for example, in CAMPUS the uid shows a relatively strong association with the lifespan, yet a weak association with the size. The second point is that the relative rankings differ across traces. For example, on CAMPUS the middle component of a file name has strong association with lifespan and size, but the association is much weaker on DEAS03 and EECS03.

Although we show only two properties in these graphs, similarly diverse associations exist for other properties (*e.g.*, directory entry lifespan and read/write ratio). In Section 5 we show how these associations can be dynamically discovered and used to make predictions.

The chi-squared test described in this section is a one-way test for association. This test provides statistical evidence that individual attributes are associated with file properties. It does not, however capture associations between subsets of the attributes and file properties. It also does not provide an easy way to understand exactly what those associations are. One can extend this methodology to use n-way chi-square tests, but the next section discusses a more efficient way for both capturing multi-way associations and extracting those associations efficiently.

5 The ABLE Predictor

The results of the previous section establish that each of a file's attributes (file name, uid, gid, mode) are, to some extent, associated with its long term properties (size, lifespan, and access pattern). This fact suggests that these associations can be used to make predictions on the properties of a file at creation time. The chi-squared results also give us hope that higher order associations (*i.e.*, an association between more than one attribute and a property) may exist, which could result in more accurate predictions.

To investigate the possibility of creating a predictive model from our data, we constructed an *Attribute-Based Learning Environment* (ABLE). ABLE is a learning environment for evaluating the predictive power of file attributes. The input to ABLE is a table of information about files whose attributes and properties we have already observed and a list of properties for which we wish to predict. The output is a statistical analysis of the sample, a chi-squared ranking of each file attribute relative to each property, and a collection of predictive models that can be used to make predictions about new files.

In this paper, we focus on three properties: the file size, the file access pattern (read-only or write-only), and the file lifespan. On UNIX file systems, there are two aspects of file lifespan that are interesting: the first is how long the underlying file container (usually implemented as an `inode`) will live, and the other is how long a particular name of a file will live (because each file may be linked from more than one name). We treat these cases separately and make predictions for each.

To simplify our evaluation, each property we wish to predict is represented by a Boolean predicate. For example:

$size = 0$
$0 < size \leq 16\text{KB}$
inode lifespan $\leq$ *1 sec*
file name lifespan $\leq$ *1 sec*
read-only
write-only

We believe these properties are representative of properties that a file or storage system designer might use to optimize for different classes of files. For example, if we know that a file will be read-only, then we might choose to replicate it for performance and availability, but this optimization would be inappropriate for files that are written frequently but rarely read. Write-only files might be stored in a partition optimized for writes (*e.g.*, a log-structured file system), and short-lived files could live their brief lives in NVRAM. In Section 6, for example, we show that by identifying small, short-lived files and hot directories, we can use predictions to optimize directory updates in a real file system.

ABLE consists of three steps:

Step 1: Obtaining Training Data. Obtain a sample of files and for each file record its attributes (name, uid, gid, mode) and properties (size, lifespan, and access pattern).

Step 2: Constructing a Predictive Classifier. For each file property, we train a learning algorithm to classify each file in the training data according to that property. The result of this step is a set of predictive models that classifies each file in the training data and can be used to make predictions on newly created files.

Step 3: Validating the Model. Use the model to predict the properties of new files, and then check whether the predictions are accurate.

Each of these steps contains a number of interesting issues. For the first step, we must decide how to obtain representative samples. For the second, we must choose a learning algorithm. For the third, we must choose how to evaluate the success of the predictions. We may consider different types of errors to have different degrees of importance – for example, if the file system treats short-lived files in a special manner, then incorrectly predicting that a file will be short-lived may be worse than incorrectly predicting that a file will be long-lived.

5.1 Obtaining Training Data

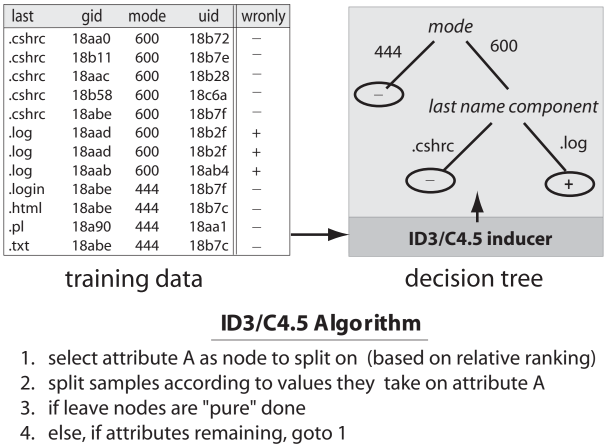
There are two basic ways to obtain a sample of files: from a running system or from traces. ABLE currently uses the latter approach, using the NFS traces described in Section 3.

Determining some of the attributes of a file (gid, uid, mode) is a simple matter of scanning the traces and capturing any command (e.g., `lookup` or `getattr`) that get or set attributes. To capture file names, ABLE simulates each of the directory operations (such as `create`, `symlink`, `link`, and `rename`) in order to infer the file names and connect them to the underlying files.

Table 2 shows samples take from DEAS03. The specific property for this table is the write-only property; each file is classified as write-only or not. For this property, ABLE classifies each file by tracking it through the trace and observing whether it is ever read.

Attributes				Property
last	*gid*	*mode*	*uid*	**wronly**
`.cshrc`	18aa0	600	18b72	NO
`.cshrc`	18b11	600	18b7e	NO
`.cshrc`	18aac	600	18b28	NO
`.cshrc`	18b58	600	18c6a	NO
`.cshrc`	18abe	600	18b7f	NO
`.log`	18aad	600	18b2f	YES
`.log`	18aad	600	18b2f	YES
`.log`	18aab	600	18ab4	YES
`.login`	18abe	444	18b7f	NO
`.html`	18abe	444	18b7c	NO
`.pl`	18a90	444	18aa1	NO
`.txt`	18abe	444	18b7c	NO

Table 2: ABLE training samples obtained from the DEAS03.



last	gid	mode	uid	wronly
.cshrc	18aa0	600	18b72	−
.cshrc	18b11	600	18b7e	−
.cshrc	18aac	600	18b28	−
.cshrc	18b58	600	18c6a	−
.cshrc	18abe	600	18b7f	−
.log	18aad	600	18b2f	+
.log	18aad	600	18b2f	+
.log	18aab	600	18ab4	+
.login	18abe	444	18b7f	−
.html	18abe	444	18b7c	−
.pl	18a90	444	18aa1	−
.txt	18abe	444	18b7c	−

ID3/C4.5 Algorithm

1. select attribute A as node to split on (based on relative ranking)
2. split samples according to values they take on attribute A
3. if leave nodes are "pure" done
4. else, if attributes remaining, goto 1

Figure 3: Constructing a simple decision tree from the training data in Table 2.

5.2 Constructing Prediction Models

There are a variety of learning algorithms to build classifiers for data. In general, these algorithms attempt to *cluster* the set of observations into a group of classes. For example, if we were trying to predict the write-only property, our learning algorithm would attempt to find all write-only files with similar attributes, and place those into a single class (*e.g.*, all files ending in `.log`). If the learning algorithm can successfully determine the classes of files, then it is able to *classify* the files. On new files that do not yet have established properties, the algorithm simply examines the attributes to determine the file's class, and predicts that the properties of the new file will be the properties of its class. Therefore, the first step in making predictions is to select an appropriate learning algorithm to classify the training data. ABLE uses the ID3 algorithm to construct a decision tree [4].

A decision tree is ideal for our purposes for several reasons: First, our attributes and properties are categorical (*i.e.*, file names, uid, gid, and mode are symbols, with no inherent ordering, as are the binary classifications for each property). Second, the computational scalability of a decision tree model makes it well-suited for use in an on-line system. Third, decision trees are humanly readable and allow us to gain some intuition about how the files are being classified. In short, decision trees are easy to train, produce predictions quickly, and the resulting models are easy to interpret.

Given a sample of files, a decision tree learning algorithm attempts to recursively split the samples into clusters. The goal is to create clusters whose files have similar attributes and similar classifications. Figure 3 illustrates the ID3 algorithm used by ABLE to induce a tree from sample data.

The first step in ID3 is to pick the attribute that is most strongly associated with the property we are trying to predict. ABLE uses the same chi-square test described in Section 4 to make this determination.

Once the ID3 algorithm has determined which attribute to select as the root node in the tree, it repeats the algorithm recursively for each of the subtrees. It re-runs the chi-square test for the sub-samples in each subtree, and selects the best attribute that has not already been used in an ancestor of that subtree. The algorithm terminates when all data has been correctly classified, or when all of the attributes have been used.

In the situation illustrated in Figure 3, ID3 would terminate the algorithm in the left subtree after splitting on only the mode attribute because all data in the left subtree has been correctly classified (*i.e.*, any file with mode 444 is classified as not write-only in our sample). In the right subtree, an additional split is necessary and the next attribute that would be selected is the last component of the file name. After this split, ID3 terminates the right subtree because all data has been classified.

Although this simple example results in a tree that perfectly classifies the data, in reality there may not be enough attributes to perfectly split the data (*i.e.*, all files in a leaf node have the same classification). More importantly, splitting on too many attributes may cause *overfitting* of the data, which leads to trees that match only the training data and do not generalize well to unseen files [4, 13, 20]. For example, imagine that all of the files in the training sample had a suffix of either `.hot` or `cold`. If a new file with a suffix of `.warm` appeared in the test data, the model would not be able to classify it at

	DEAS03		EECS03		CAMPUS	
Predicate	**ABLE**	**MODE**	**ABLE**	**MODE**	**ABLE**	**MODE**
size=0	98.97%	58.59%	97.06%	66.58%	98.57%	94.05%
0<size≤16KB	95.42%	63.00%	89.96%	57.69%	98.83%	95.00%
lftmd≤1s (file)	88.16%	53.28%	93.60%	58.80%	72.95%	66.00%
lftmd≤1s (direntry)	96.96%	63.74%	91.90%	52.80%	77.66%	75.49%
wronly	91.17%	51.76%	83.56%	46.98%	81.83%	82.85%
rdonly	75.55%	48.79%	71.79%	49.63%	81.24%	46.60%

Table 3: A comparison of the accuracy of the ABLE and MODE predictors for several properties for the three traces. MODE always predicts the value that occurred most frequently in the training sample, without considering any attributes of the new file.

all, even if the new file shared many other attributes with `.hot` or `.cold` files. To avoid this problem, ABLE instructs the ID3 algorithm to continue expanding the tree until all attributes are exhausted (or the data perfectly classified) and then ABLE prunes bottom leaves of the tree to eliminate potentially unnecessary or overly specific. This is one of many pruning methods commonly used to favor a smaller tree to a larger one [4](pages 279–293), in the hope that a smaller tree generalizes better on future samples. Note that building the tree top-down and selecting the most strongly associated attributes first guarantees that only the least associated attributes will be pruned in this process.

5.3 Validating the Model

At this point, we have used our training data to induce a decision tree model that can classify the data. The result is a model that can be used to classify new files (*i.e.,* predict their properties). For example, if a new file were to be created with mode 600 and name `foo.log`, the model will predict that the file will be write-only. For our simple example, we only have one rule: a file is write-only only if its mode is 600 and its last name is `.log`. In general, a rule is a conjunction of all attributes on a path to a positively classified leaf node.

For each of the binary predicates, we induce a decision tree from a sample of files seen during the peak hours (9am-6pm) on a Monday from the trace (10/22/2001 for CAMPUS and 3/24/2003 for EECS03 and DEAS03). We then make predictions about the files created during the peak hours on the following day. The decision to train on the peak hours of Monday and test on the peak hours of Tuesday is not completely arbitrary; as shown in Section 3, the peak hours are the most active hours of the day. The resulting size of the training and testing samples are approximately 40,000 files for DEAS03, 35,000 for CAMPUS, and 15,000 for EECS03.

For comparison purposes, we compare against a simple model named MODE that always predicts the mode of a property, which is defined as the value of the property that occurs most frequently in the training data. For example, if most of the files created on Monday were write-only, then the MODE predictor would predict that *every* file created on Tuesday would be write-only, without considering any of the file attributes. Because all our properties are binary, each prediction is either correct or incorrect and the predication accuracy is simply the ratio of correct predictions to the sample size.

Table 3 shows the prediction accuracies on Tuesday, for each of DEAS03, EECS03 and CAMPUS. In nearly all cases, ABLE more accurately predicts the properties of files, and in some cases, nearly doubles the accuracy relative to probability-based guessing (MODE). However, there are some cases, specifically on the CAMPUS trace, where the workload is so uniform that MODE does almost as well.

5.4 MABLE and NABLE

ABLE's decision trees successfully exploit the statistical association between file attributes and properties and can be used to produce accurate predictions about future file system activity. We were also curious about which attributes make the largest contribution. The chi-squared analysis in Section 4 established that many of the attributes had strong associations, but this is not enough to determine whether or not multi-way attribute associations would have much effect on prediction accuracy.

The easiest way to measure the effects of additional attributes is to compare the ABLE trees (induced using all available attributes) against a set of constrained trees (induced with a limited set attributes).

If multi-way associations exist between the attributes, then we can empirically measure their effect by comparing prediction accuracies. To this end, we construct two new sets of constrained decision trees, and compare these against the ABLE (unconstrained) decision trees.

MABLE: trees induced with only the inode attributes (mode, uid, gid).

NABLE: trees induced with only file names.

Figure 4 compares the predication accuracies for ABLE, MABLE and NABLE. For the purpose of clarity, this figure only shows the accuracy for three of our binary properties (size, write-only, and file name lifespan); the results for our other properties are similar. The figure shows that ABLE usually outperforms both MABLE and NABLE. This tells that some multi-way associations exist between the file name attributes and other attributes that allow us to make more accurate predictions when all are considered. An example of a multi-way association would be that the lifespan of a file depends on both the file name and the user who created the file.

However, the CAMPUS and EECS03 results tell us that, in some situations, ABLE does worse than MABLE or NABLE. In these traces, some multi-way associations existed on Monday that did not generalize to new files on Tuesday. This is a common problem of over-fitting the data with too many attributes, although the differences are not severe in our evaluation.

There are two important points to take away from our analysis of MABLE and NABLE. First, more attributes are not always better. We can fall into a trap known as the *curse of dimensionality* in which each attribute adds a new dimension to the sample space [6]. Unless we see a sufficient number of files, our decision trees may get clouded by transient multi-way associations that do not apply in the long run. Second, NABLE and MABLE offer predictions roughly equivalent to ABLE. This is somewhat surprising, particularly in the case of MABLE, because it means that we can make accurate predictions even if we do not consider file names at all.

Given enough training data, ABLE always outperforms MABLE and NABLE. For the results presented in the paper, ABLE required an extra week of training to detect the false attribute associations, due in part to the small number of attributes. We anticipate that more training will be required for systems with larger attribute spaces, such as object-based storage with extended attributes [18] and non-UNIX file systems such as CIFS or NTFS [29]. Furthermore, irrelevant attributes may need to be pre-filtered before induction of the decision tree [6] to prevent over-fitting. The automation of ABLE's training policies, including attribute filtering, is an area for future work.

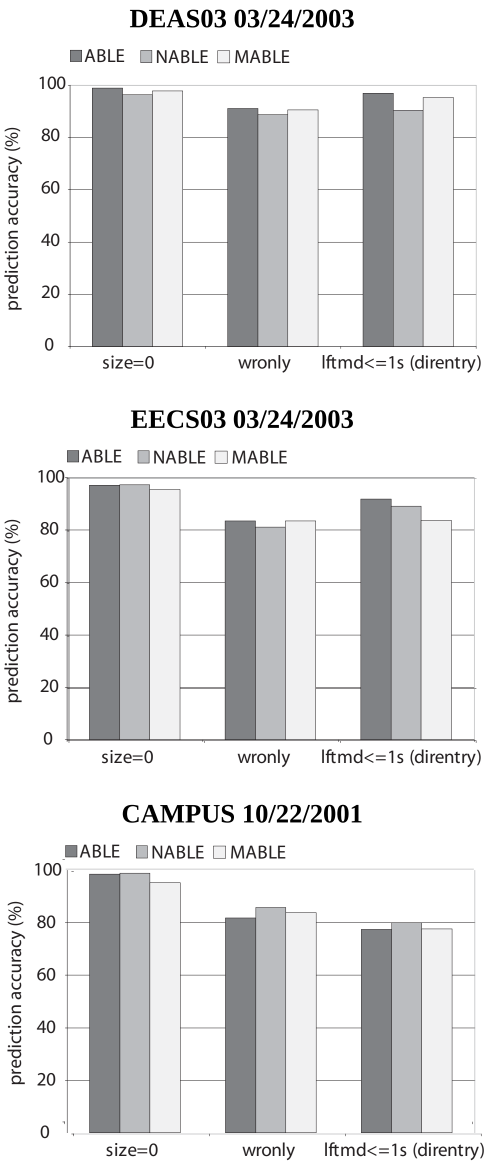


Figure 4: Comparing the prediction accuracy of ABLE, NABLE, and MABLE for the properties *size=0*, *write-only*, and *lifetime* $\leq$ *1 second*. Prediction accuracy is measured as percentage correct.

- NABLE predicts “write-mostly" if
  first=cache & last=gif *[5742/94.0%]*
- MABLE predicts “size=0” if
  mode=777 *[4535/99.8%]*
- ABLE predicts “deleted within 1 sec” if
  first = 01eb & last = 0004 & mode = 777 &
  uid = 18abe *[1148/99.7%]*

Figure 5: Example rules for DEAS03 discovered by NABLE, MABLE, and ABLE. The number of files that match the attributes and the observed probability that these files have the given property are shown on the right. For example, NABLE predicts that names whose name begins with `cache` and end in `.gif` will be “write-mostly”. This prediction is based on observations of 5742 files, 94.0% of which have the “write-only” property.

5.5 Properties of Our Models

In our experience, a model that predicts well for one day will continue to perform well for at least the next several days or weeks [9]. However, workloads evolve over time, and therefore our models will eventually decrease in accuracy. We are exploring ways to automatically detect when new models are necessary. Fortunately, building a new model is an inexpensive process (requiring approximately ten minutes of processing on a modest Pentium-4 to build a new model from scratch for the peak hours of the heaviest workloads in our collection), so one possible approach is simply to build new models at regular intervals, whether or not the current models have shown any sign of degradation.

In general our decision trees yield roughly a 150:1 ratio of files to rules. Rules can be easily inspected after the fact to determine interesting patterns of usage (which is how we discovered the associations originally). On DEAS03, for example, the 45K sample files on which we induced a decision tree with only 300 rules (*i.e.*, a decision tree with 300 leafs). This means that the resulting model only requires only a few kilobytes to store.

6 Using the Predictions

Now that we have the ability to make predictions about the future properties of a file based on its attributes when it is created, the question remains what benefit we can reap from this foresight.

One type of application that we believe can benefit from our predictions is file cache management policy. When choosing a buffer to evict, it would be helpful to have an accurate prediction of whether or not that buffer would be accessed in the near future (or at all). For the DEAS03 workload, for example, we can identify write-only files with a high degree of accuracy, and we know that we can immediately evict the buffers created by writing these files. Similarly, in a disconnected environment, knowing which files are read-only can help select files to hoard.

Pre-fetching can also benefit from predictions; if we can identify files that are highly likely to be read sequentially from beginning to end (perhaps on a user-by-user basis), then we can begin pre-fetching blocks for that file as soon as a client opens it. If cache space is plentiful, it might make sense to do aggressive pre-fetching for every file opened for reading, but if cache space is at a premium, it is valuable to know which files will benefit the most from this treatment.

Our predictions may also be helpful in optimizing file layout – if we can predict how large a file will be, and what access patterns the file will experience, then we can pre-allocate space on the disk in order to optimally accommodate these properties (instead of adapting to these properties as they become evident). For example, yFS uses three different block allocation and layout policies for different kinds of files and migrates files from one policy to another as they grow or their access patterns change [32]. Given accurate predictions, we can begin with the correct policy instead of discovering it later.

Another application of ABLE is to guide adaptive as well as pro-active techniques – we can use its models to predict not only what the future holds for new files, but also for existing files. In this paper we focus primarily on the prediction of the properties of new files, because this is a capability we have not had before. Nevertheless it is important to recognize that the ABLE models can be used for adaptation as well.

The rest of this section discusses the use of name-based hints to cluster active directory blocks and inodes into a designated “hot” area of the disk. By placing this hot area in high-speed media (e.g., NVRAM) or placing it in the middle of the disk, we should reduce the overall disk access time. We use as our evaluation metric the degree to which we induce a hot spot on the designated area of the file system. We discuss how to benchmark the resulting system, and measure its performance on our three traces.

6.1 Benchmarking Attribute-Based Systems

One of the difficulties of measuring the utility of attribute-based hints in the context of real file systems is finding a suitable benchmark. Synthetic workload generators typically create files in a predictable and unrealistic manner – they make little or no attempt to use realistic file names or mimic the diverse behaviors of different users. If we train our models on data gathered when these benchmarks are running then our predictions will probably be unrealistically accurate, but if we train on a workload that does not include the benchmarks, then our predictions for the files created by the benchmark will be uncharacteristically bad.

Our solution to this problem is to construct a benchmark directly from traces of the target workload, thereby ensuring that the associations between file names, modes, and uids during the trace will resemble those present in the actual workload. This leads immediately to a new problem – in order to replay the traces, we need a real file system on which to play them. The usual solution to this problem is to recreate the traced file system from a snapshot of its metadata taken at a known time, and then begin replaying from that time [28]. This method works well when snapshots are available, and when a suitable device is available on which to reconstruct. Unfortunately we have neither – there are no publicly-available snapshots of the systems from which the traces were taken, and even if there were, reconstructing them would require at least 500GB of disk space and many hours of set-up time per test.

To solve this problem, we have developed a new method of performing a snapshot-less trace replay that uses the trace itself to reconstruct the subset of the file system necessary to replay a given section of the trace. We call these *sub-snapshots*. In essence, our method is to replay the trace several times, inferring knowledge about the underlying file system by observing how it is used.

The first pass reconstructs as much as it can of the file system hierarchy, primarily by observing the parameters and responses from `lookup`, `getattr`, `create`, `mkdir`, `rename`, `remove`, and `link` calls. The idea of discovering the file system hierarchy by snooping NFS calls is not new and has been in widespread use since the technique was described by Blaze [3]. Unfortunately, as other researchers have noted, this method is imperfect – some of the information may be absent from the trace because of missed packets or because it is cached on the client during the trace period and thus never visible in the trace. To compensate for this missing data, we keep track of each file or directory that is accessed during the trace, but whose metadata we cannot infer. When the first pass is finished, we may either fill in the missing values with reasonable defaults or discard the incomplete items.

Because we are using attribute-based models, we cannot simply invent file attributes and hope that they will work. However, there is a danger that if we discard all the objects for which we have incomplete information, we may lose a significant portion of workload. For the experiment described in this section, we use only name attributes. After examining the traces we cannot find names for fewer than than 5% of the files mentioned in the workload (and typically much less). Therefore we believe that discarding these "anonymous files" does not alter the workload to an important degree.

Files or directories for which we cannot infer the parent are attached to the root directory, because from our own experiments we have found that this is the directory most likely to be cached on the client. For example, we rarely see lookups for `/home/`*`username`*, because home directories are frequently accessed and rarely invalidated.

The output of the first pass is a table of pathnames of each file and directory observed in the trace along with a unique identifier for each object, and the size, mode, and other relevant information necessary to reconstruct the object. The purpose of the new identifier is to provide a convenient substitute for the file handle that is independent of the actual implementation of the file system. (File handles usually encode the mount point and inode numbers, and we cannot ensure that we will get the same values when we reconstruct the file system.)

The second pass through the trace replaces all of the file handles in the trace with the unique identifiers created in the first pass, and removes references to files for which no information could be inferred.

Based on the table created after the first pass, we then create a file system that matches the rewritten trace, and replay the new trace on that file system. The result is both realistic and repeatable.

Using this method, we constructed several sub-snapshots for each workload. A typical hour of activity on these systems accesses files containing only five to ten GB of data (although there are hours when many directories are scanned, resulting in enormous and unwieldy sub-snapshots). One of the challenges with DEAS03 and EECS03 is that there are apparently some jobs that periodically scan large parts of the directory hierarchy, checking the modification time of each file. Since most of these files are never actually read or written, we could modify our sub-snapshot builder to recognize this and treat these files differently (only creating a short or empty file, instead of a file the same size as the original). This would permit us to create sub-snapshots for a much larger fraction of the underlying file system.

6.2 Increasing Locality of Reference

As an example application, we explore the use of attribute-based hints to control the locality of block reference by anticipating which blocks are likely to be hot and grouping them in the same cylinder.

We use two methods to identify hot data blocks. The first method, which we call *HotName*, automatically classifies as hot any file that we predict will be short-lived and/or zero-length. For this type of file, the overhead of creating and maintaining the inode and name the file (*i.e.*, the directory entry for the file) can be a large fraction of the cost incurred by the file, and therefore there may be benefit to reducing this overhead. The second method, which we call *HotDir*, predicts which directories are most likely to contain files that have the HotName property. Since these directories are where the names for the HotName files will be entered, there may be benefit from identifying them as well.

The model that we use for HotDir is constructed via a method similar to ABLE, but unfortunately in our prototype requires some external logic because ABLE is focused on files and does not currently gather as much information about directories. In general, the HotDir rules are that directories identified as home directories, mail spool directories, and directories named `Cache` are classified as hot directories. (ABLE is capable of identifying the `mail` and `Cache` directories as interesting, but does not currently have a "is-home-directory" attribute.)

To test the effect of HotDir and HotName, we have modified the FreeBSD implementation of FFS so that it uses a simplified predictor (similar in nature to the ABLE predictor, but employing only name attributes, and re-coded to live in the kernel environment) to predict whether each new directory has the HotDir property and whether each new file has the HotName property. If so, it attempts to allocate blocks for that file or directory in a designated area of the disk. Our goal is to measure the increase in the number of accesses to this area of the disk when we use policies guided by HotDir and HotName.

We use two systems as our testbed. Both have a 1 GHz Pentium III processor, 1 GB of RAM, and run FreeBSD 4.8p3. Our experiments use the FreeBSD implementation of FFS with 16KB blocks and soft-updates enabled [12]. We have instrumented the device driver for the disk so that it keeps a count of how many reads and writes are done on each 16KB logical disk block.

Heuristic	**Ops**	**Reads**	**Writes**
	DEAS03		
Perfect	26.17%	0.85%	42.28%
HotDir	0.57%	0.22%	0.76%
HotFile	0.59%	0.00%	0.95%
HotDir+HotFile	1.10%	0.22%	1.60%
	EECS03		
Perfect	23.89%	8.96%	41.61%
HotDir	3.09%	1.11%	4.61%
HotFile	2.82%	0.00%	5.00%
HotDir+HotFile	5.95%	1.15%	9.65%
	CAMPUS		
Perfect	3.90%	0.76%	11.28%
HotDir	1.43%	0.58%	3.36%
HotFile	1.13%	0.00%	3.70%
HotDir+HotFile	2.60%	0.57%	7.23%

Table 4: Average percentage of the total ops, reads, and writes that fall in the 4MB target region of the disk for each of the heuristics on DEAS03, EECS03, and CAMPUS. The "Perfect" heuristic shows the maximum percentage attainable by an algorithm with perfect knowledge. The working set for these runs varies from 5-10GB.

6.3 Results

To test our heuristics, we ran a series of one-hour trace replays for the hours noon-5pm for several days on each of our traces. The models are trained on a Monday (3/24/03 for DEAS03 and EECS03, 10/22/01 for CAMPUS), and the replays are constructed from the following Tuesday through Thursday. Each hour-long replay begins with 15 minutes to warm the cache. Then the block counters are reset, and the test begins in earnest and runs for 45 minutes of replay time.

We designate a 4MB region as the target area for hot objects. Our evaluation examines the distribution of actual accesses to the disk and compares the percentage that go to the target area to the theoretically maximum number of accesses that would go to the hottest 4MB region given perfect knowledge (*i.e.*, if the hottest 256 16KB blocks on the disk were allocated in the target region).

As shown in Table 4, both heuristics improve locality compared to the default layout policy, and using both heuristics is an improvement over using either one alone. Write locality is increased more than read locality; this is not surprising because directory contents are read-cached. Using both HotDir and HotName, we manage to increase the number of accesses to two-thirds of that of the hottest possible region on CAMPUS, and on EECS03 nearly 6% of all the disk accesses during the trace are now confined to the target area. These percentages may seem small, but keep in mind that we are focusing only on small files and directories, and normal file traffic is the dominant cause of disk accesses in these workloads.

7 Conclusions

We have shown that the attributes of a file are strong hints of how that file will be used. Furthermore, we have exploited these hints to make accurate predictions about the longer-term properties of files, including the size, read/write ratio, and lifespan. Overall, file names provide the strongest hints, but using additional attributes can improve prediction accuracy. In some cases, accurate predictions are possible without considering names at all. Using traces from three NFS environments, we have demonstrated how classification trees can predict file and directory properties, and that these predictions can be used within an existing file system.

Our results are encouraging. Contemporary file systems use hard-coded policies and heuristics based on general assumptions about their workloads. Even the most advanced file systems do no more than adapt to violations of these assumptions. We have demonstrated how to construct a learning environment that can discover patterns in the workload and predict the properties of new files. These predictions enable optimization through dynamic policy selection – instead of reacting to the properties of new files, the file system can anticipate these properties. Although we only provide one example file system optimization (clustering of hot directory data), this proof-of-concept demonstrates the potential for the system-wide deployment of predictive models.

ABLE is a first step towards a self-tuning file system or storage device. Future work involves automation of the entire ABLE process, including sample collection, attribute selection, and model building. Furthermore, since changes in the workload will cause the accuracy of our models to degrade over time, we plan to automate the process of detecting when models are failing (or are simply suboptimal) and retraining. When cataclysmic changes in the workload occur (*e.g.,* tax season in an accounting firm, or September on a college campus), we must learn to detect that such an event has occurred and switch to a new (or cached) set of models. We also plan to explore mechanisms to include the cost of different types of mispredictions in our training in order to minimize the anticipated total cost of errors, rather than simply trying to minimize the number of errors.

In addition to caching and on-disk layout optimization, we envision a much larger class of applications that will benefit from dynamic policy selection. Attribute-based classification of system failures and break-ins (or anomaly detection) is a natural adjunct to this work (*e.g.,* “has this file been compromised?”). Moreover, through the same clustering techniques implemented by our decision trees, we feel that semantic clustering can be useful for locating information (*e.g.,* “are these files related?”). Both of these are areas of future work.

Acknowledgments

Daniel Ellard and Margo Seltzer were sponsored in part by IBM. The CMU researchers thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. Their work is partially funded by the National Science Foundation, via grants #CCR-0326453 and #CCR-0113660.

References

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

[2] J. Michael Bennett, Michael A. Bauer, and David Kinchlea. Characteristics of Files in NFS Environments. In *Proceedings of ACM SIGSMALL Symposium on Small Systems/PC.*, pages 18–25, Toronto, Ontario, Canada, 1991.

[3] Matthew A. Blaze. NFS Tracing by Passive Network Monitoring. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 333–343, San Fransisco, CA, January 1992.

[4] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Chapman and Hall, 1984.

[5] 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.

[6] Rich Caruana and Dayne Freitag. Greedy Attribute Selection. In *International Conference on Machine Learning*, pages 28–36, 1994.

[7] Ian Dowse and David Malone. Recent Filesystem Optimisations on FreeBSD. In *Proceedings of the*

USENIX Annual Technical Conference (FREENIX Track), pages 245–258, June 2002.

[8] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST'03)*, pages 203–216, San Francisco, CA, March 2003.

[9] Daniel Ellard, Jonathan Ledlie, and Margo Seltzer. The Utility of File Names. Technical Report TR-05-03, Harvard University Division of Engineering and Applied Sciences, 2003.

[10] Daniel Ellard and Margo Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *Proceedings of the Seventeenth Annual Large Installation System Administration Conference (LISA'03)*, pages 73–85, San Diego, CA, October 2003.

[11] 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.

[12] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft Updates: a Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems*, 18(2):127–153, 2000.

[13] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Spring-Verlag, 2001.

[14] Butler W. Lampson. Hints for Computer System Design. In *ACM Operating Systems Review*, volume 15(5), pages 33–48, October 1983.

[15] Tara M. Madhyastha and Daniel A. Reed. Input/Output Access Pattern Classification Using Hidden markov Models. In *Proceedings of IOPAF*, pages 57–67, San Jose, CA, December 1997.

[16] James T. McClave, Frank H. Dietrich II, and Terry Sincich. *Statistics*. Prentice Hall, 1997.

[17] 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.

[18] Michael P. Mesnier, Gregory R. Ganger, and Erik Riedel. Object-Based Storage. *ACM Communications Magazine*, 41(8):84–90, 2003.

[19] Rodney Van Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the Usenix Technical Conference*, January 1997.

[20] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[21] 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.

[22] Sape Mullender and Andrew Tanenbaum. Immediate Files. In *Software – Practice and Experience*, number 4 in 14, April 1984.

[23] Keith Muller and Joseph Pasquale. A High Performance Multi-Structured File System Design. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP-91)*, pages 56–67, Asilomar, Pacific Grove, CA, October 1991.

[24] David A. Patterson, Garth Gibson, and Randy H. Katz. Case for Redundant Arrays of Inexpensive Disks (RAID). In *In Proceedings of the ACM Conference on Management of Data (SIGMOD)*, Chicago, IL, June 1988.

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

[26] 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.

[27] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming Aggressive Replication in the Pangaea Wide-Area File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI).*, Boston, MA, December 2002.

[28] Keith A. Smith and Margo I. Seltzer. File System Aging - Increasing the Relevance of File System Benchmarks. In *Proceedings of SIGMETRICS 1997: Measurement and Modeling of Computer Systems*, pages 203–213, Seattle, WA, June 1997.

[29] David A. Solomon and Mark E. Russinovich. *Inside Microsoft Windows 2000, Third Edition*. Microsoft Press, 2000.

[30] Carl Hudson Staelin. High Performance File System Design. Technical Report TR-347-91, Princeton University, 1991.

[31] 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.

[32] Zhihui Zhang and Kanad Ghose. yFS: A Journaling File System Design for Handling Large Data Sets with Reduced Seeking. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST'03)*, pages 59–72, San Francisco, CA, March 2003.