Towards Personalized File Systems

Benjamin Rutt
The Ohio State University
Computer and Information Science
2015 Neil Ave, DL 395
Ohio State University
Columbus, Ohio 43210
(614) 292-6139
rutt@cis.ohio-state.edu

Srinivasan Parthasarathy
The Ohio State University
Computer and Information Science
2015 Neil Ave, DL 395
Ohio State University
Columbus, Ohio 43210
(614) 292-2568
srini@cis.ohio-state.edu

ABSTRACT

Filesystems and storage systems are the foundation of the information age. Efficient and flexible access to information stored within such systems is of paramount importance. To address this need we propose VIP-FS, a visual intelligent and personalized file system.

The key component of our system is the ability to intelligently learn about user behavioral trends from their file and operating system activity patterns. The usage data that is tracked can be subsequently queried and visualized. More importantly, this data can also be mined for intelligent system-level qualitative and quantitative performance enhancements including selective file compression, predictive file prefetching, and lastly intrusion detection. We conduct an in-depth performance evaluation to demonstrate the potential benefits of the proposed system.

Keywords

filesystems, prediction, association rule mining

1. INTRODUCTION

Filesystems are extremely active and important; just enter the "server room" of any organization's IT department during a typical workday and you will witness the sounds of large amounts of disk activity. Programs are launched, libraries are loaded, files are read, changes are written, and new files are created. Although these file events cause a change in system state, the events are not typically recorded. However, these events if recorded can provide a complete history of user-specific file access activity. This history on its own and in conjunction with operating system command activity has many potential uses both from a qualitative and quantitative performance perspective.

From the qualitative and user-utility perspective, querying and visualizing file system activity traces can enable users a greater degree of flexibility in terms of understanding their past activity. A user may be interested in visualizing the current state or past activity patterns. In our system concise information visualization is achieved through the use of treemaps [1]. Or beyond visualization the user may be interested in searching for a file whose exact path they may have forgotten (especially true for unorganized directory spaces). Sample queries that may be of interest in this context include: "Give me a list of all of my file accesses during Jan 3rd, 2003"; or "What other files did I read around the same time that I last wrote a file having the name `my-plans.txt`?" Enabling such queries and the visualization of results adds to user-utility.

The main thrust of this paper however are the potential quantitative system-level performance gains from learning about and then leveraging user behavior patterns. To learn user behavior patterns we evaluate the use of novel N-gram based and frequent itemset based approaches. Hybrid combinations of both approaches are also empirically evaluated. We present results on mining these file traces to use as a basis for prefetching and selective compression of files (files that are too large or files that are very infrequently used). We evaluate our results based on an in depth simulator. Our filesystem simulation compares the performance of several novel prefetching schemes (each based on an affiliated model generated by the mining of traces) against a baseline non-prefetching cache. We show that this approach can enable effective user-specific file space compression and file prefetching that outperforms the baseline approaches prevalent in modern day filesystems. Finally, we explore using filesystem and operating system usage patterns to build a better Intrusion Detection System (IDS). We believe that these usage patterns could play a pivotal role in both detecting abnormal user activity and reducing false alarms. We validate our new algorithms on well known datasets.

In summary the main contributions of the proposed Visual Intelligent Personalized File System (VIP-FS) are:

- Efficient methods for collecting and compressing file system activity at run-time.
- Enabling the user to query and visualize file system patterns.
- Mining user-specific file system traces and enabling personalized prefetching and selective compression of the user's file system space. Both of these strategies

SIGKDD 2003 Washington, D.C. USA

ensure that file accesses are more efficient, especially in the case of networked file systems, while conserving space.

- Mining user-specific operating system and filesystem activity and building a signature model of the user. Subsequently if accesses deviate significantly from the model potential intrusions may be signaled.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 details the system architecture for the proposed personalized file system (VIP-FS). Section 4 discusses experimental results that demonstrate the quantitative utility of the proposed system. Section 5 discusses this paper's conclusions and future work.

2. RELATED WORK

Prior work by Kroeger and Long [2] evaluated several approaches to file access prediction, considering the *last-successor* model, the *graph-based* model [3] (which considered the successors of a given file access as an unordered set), and the *finite multi-order context (FMOC)* model. In addition, the authors improved on the *FMOC* model by reducing the dynamic memory occupation to *O(N)*, where *N* is the number of unique files in the system.

Our work differs from Kroeger and Long's in several ways. First, we present an *association mining* model that considers the successors of a given file access as an unordered set. In contrast, Kroeger and Long presented a *graph-based* model that performed poorly while considering successors as an unordered set. Secondly, we build upon the basic *last-successor* model by using a variable number of predecessors and a variable number of successors. Finally, we evaluate performance in terms of filesystem cache hit rates, while Kroeger and Long evaluate performance in terms of whether a predicted file is the next file to be accessed. In a cache, prefetches are useful if they result in a cache hit, *regardless of whether the hit occurs during the very next file access or some number of accesses later.*

The *N-gram* prediction model is commonly used in speech-processing research [4]. An *N-gram* model associates a sequence of events of length *N* with the event that follows. If the same sequence of events of length *N* are witnessed again, a prediction can be made that the same event that followed the last time will occur again.

Su et al. [5] proposed a *N-gram+* model to perform predictions of WWW accesses. An *N-gram+* model is a collection of *N-grams* that work together to make predictions. This scheme gives the *N-gram* with the largest *N* the first chance to make a prediction; if the largest *N-gram* cannot make a prediction, then the second-largest *N-gram* gets a chance, and so on down to the *N-gram* with smallest *N*. In this paper, we evaluate the predictive ability of the *N-gram+* model with respect to file system access patterns and user command patterns. We extend the *N-gram+* model to include a variable number of successors, in what we term the *p-s-gram+* model.

The filesystem access events we care about all originate via independent user activity. As such, we have no control over when these events will happen, and how many of them will happen over time. In other words, the filesystem access events are a good example of a *data stream*. Data streams present unique challenges; an overview of the topic appears in [6]. Some applications that take the filesystem access streams as input do not necessarily require that the events be stored indefinitely. For example, a prefetching filesystem cache may only be interested in recent filesystem events, considering older filesystem events to be less relevant. In fact, a prefetching filesystem cache may wish to expire less relevant records to limit the state occupation. A simulation of such a scheme can be found in section 4.1.4. However, other applications, such as a file access querying system or intrusion detection system may require that each and every file access event is stored permanently. A sample implementation of the persistent storage of these events can be found in section 3.3.

Sequeira and Zaki developed ADMIT [7], an anomaly-based intrusion detection system. It uses command line stream data to both establish a user's normal command line patterns and also to determine when a user's command line patterns deviate from what is normal. We would like to build on this foundation yet also use file access data to distinguish normal users from intruders.

Forrest et. al [8] developed an anomaly-based approach to intrusion detection that measured the deviation of sequences of system calls from established normal sequences for well-known programs. Our approach in Section 4.2 is similar, although we measure the deviation of sequences of UNIX commands by users from established normal patterns.

Classification Based on Associations (CBA) [9] is an algorithm for building an accurate classifier for prediction based on association rules. In our prefetching cache simulation, we apply a similar technique. First, we generate association rules from the set of correlated values (file accesses) found in the training dataset. Next, we rank the rules them by support and confidence. Finally, we form a predictive model based on the highest ranking rules, to predict in real time what file accesses will follow a given set of file accesses.

Treemaps, used in *VIP-View*, were invented by Ben Shneiderman in 1992 and have the ability to display hierarchical data in a finite area [1].

3. ARCHITECTURE

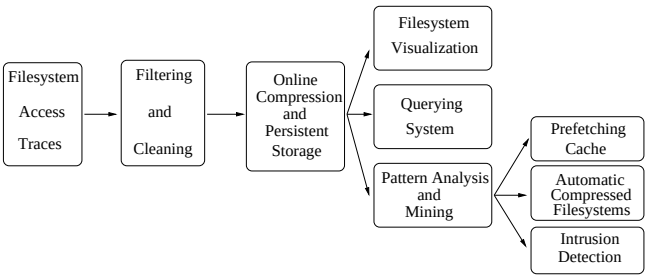

Figure 1: VIP-FS Architectural Overview

3.1 Filesystem Access Traces

In order to apply filesystem access trace data in useful ways, we obviously needed a source of filesystem traces with attributes sufficient to meet our needs. We decided to build our own subsystem to produce these traces.

Our test machine was a Solaris 2.8 system with four 296-MHz CPUs and three Gigabytes of RAM. We designed a program that would start up when a login shell was invoked

by a user and shutdown when that user logged out. This way, each set of filesystem access traces was generated on a per-user basis, resulting in no interference from other users or system processes.

Our program was invoked via shell startup files and ran in the background, in order to avoid disrupting the user. Our users consisted of graduate students who simply performed their normal work on the system for several weeks while the tracing program was enabled. For example, their normal work consisted of checking email, editing files, browsing web sites, compiling programs, etc.

In order to capture all of the file accesses made by a user, we captured any file accesses made by the shell process and any of its subprocesses. We used the system call tracer `truss` to accomplish this, recording each `open()` system call. On a similar test machine, Griffioen and Appleton [3] showed that "there is relatively little time between the moment when a file is opened and the moment when the first read occurs". Therefore, we consider `open()` to be the system call most representative of a single file access.

The attributes we considered meaningful for each file access included:

- the absolute path name of the file accessed
- the date and time of the access, measured in seconds
- the access mode (read, write, or read+write)
- the absolute path name of the application accessing the file
- the process ID of the process that accessed the file

3.2 Filtering and Cleaning

The accesses output by the access tracer program was then fed into a filtering and cleaning subsystem, which eliminated files considered useless for analysis. We filtered out accesses for temporary files, such as any files residing in `/tmp`. We also filtered out cache files created automatically by an application, such as files that are a member of the `netscape` web browser's own per-user disk cache.

3.3 Online Compression and Persistent Storage

The accesses remaining after the filtering and cleaning process were then passed to a compression and persistent storage subsystem. For each distinct absolute filename encountered by the system, a new file ID (FID) was generated as a 4-byte integer. The association between the FID and the absolute filename was saved in persistent storage. From this point on, the filename and application attribute of a file access could be stored as a FID. A total of 17 uncompressed bytes were needed to describe a file access, including 4 bytes each for the accessed file FID, date, application FID, and process ID, and one byte for the access mode. Next, a queue of multiple file accesses were compressed as a unit using the zlib library (`http://www.zlib.org`). Finally, the compressed data was written to disk at regular intervals. Once the data was committed to disk, it immediately became available for reading, and could serve as input to the applications discussed below. *Each file access compressed down to around 3.65 bytes, resulting in an average compression ratio of 4.65.* For example, two weeks worth of accesses (roughly 20,000 unique file accesses) required only about 70KB of storage.

3.4 Visualization (VIP-View)

The primary goal of *VIP-View* is to visually convey information about a user's filesystem that is not normally provided through standard tools such as `ls` or a graphical directory explorer. By making use of the file access histories discussed above, *VIP-View* has more information to work with than these standard tools and can provide an enriched view.

Pictures of *VIP-View* in action can be seen in Figure 2. *VIP-View* displays the contents of a user's entire home directory on a single screen. Top-level files and directories appear as horizontal rows. Files or directories nested under any of the top-level directories are then displayed in vertical columns within the containing directory's row. This alternating of horizontal and vertical cuts allows presentation of the entire directory structure within a finite area.

As well as revealing the directory structure, the horizontal and vertical cuts reveal the size of the filesystem components. In Figure 2, larger pieces are used to indicate larger files and directories, while smaller pieces indicate smaller files and directories.

In addition, each filesystem object is colored to communicate another attribute about the object. In the leftmost image of Figure 2, darker colors are used to display files that have more recently been accessed, and lighter colors are used to display files that have less recently been accessed.

Other information about files that can be conveyed using coloring include:

- the last application that accessed a particular file
- how recently a file has been read or written
- what user and group owns a particular file
- what files have world-readable permissions

VIP-View provides several advantages over standard filesystem information tools. First, users can see at a glance which files are most active, and where these most active files are being stored in the directory structure. A user might want to move more active files higher up in the directory tree, since deeply nested files take longer to access, starting from the top level directory. (Users frequently find themselves at the top level directory, for example at login time). Second, users can easily spot what they consider to be filesystem anomalies. For example, if permission coloring is turned on, a file with unintended world-readable permissions can easily be spotted, and a corrective action taken. An example of this permission coloring can be found in the rightmost image of Figure 2.

3.5 Querying (VIP-Query)

Another application of the file access histories stored above is a query tool called *VIP-Query*. This end-user application serves as a direct interface between users and their past file accesses. Figure 3 shows an example of *VIP-Query* in action.

VIP-Query lists each file access event recorded by the system in tabular format. The tool can sort accesses and provides searching capabilities, enabling users to locate any file access by providing a regular expression that matches file accesses for one of the chosen attributes.

VIP-Query shows an audit trail of file events, which can be useful in deriving lost filenames. For example, a user may have remembered accessing a file that has since been

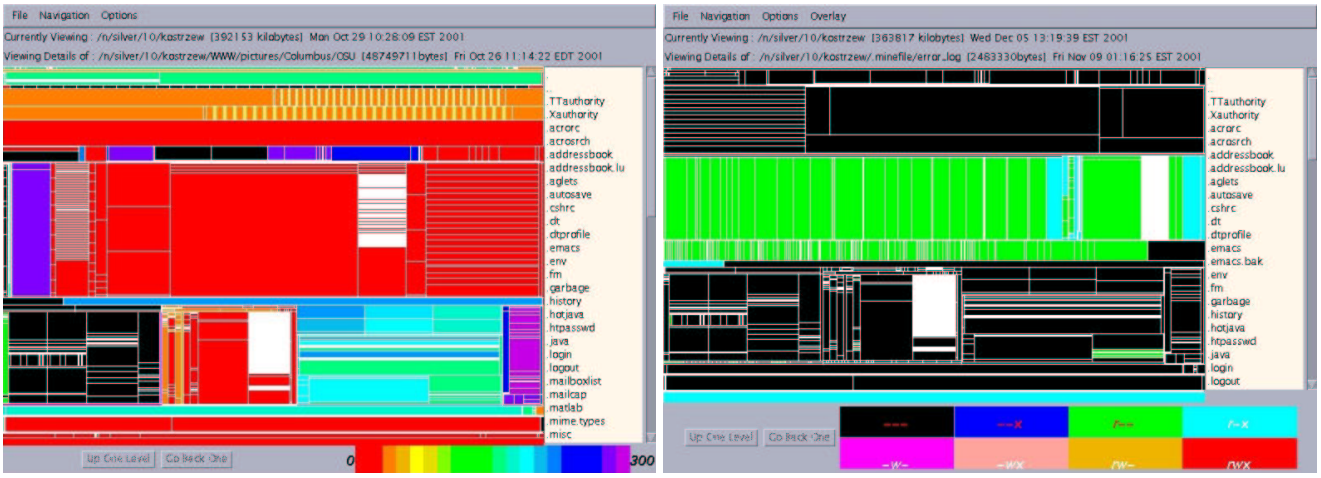

Figure 2: VIP-View

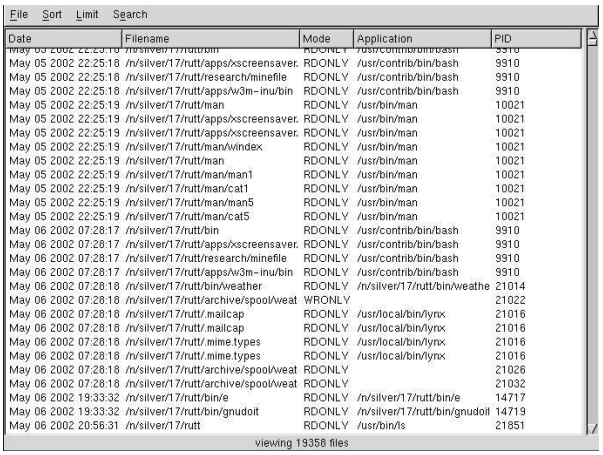

Figure 3: VIP-Query

deleted around the same time that user accessed a file named `my-plans.txt`. In this case, *VIP-Query* could help the user iterate through historical accesses for the file `my-plans.txt` and look for files accessed around the same time. This way, the user could locate the access record for the lost file, and recover the file using a web search based on the discovered filename. In other words, *VIP-Query* reveals associations between file accesses.

VIP-Query can also indicate what files are opened by specific applications. This can help users configure their applications. Even without reading an application's documentation, a user could determine exactly which files an application accesses. This can aid in the configuration of that application, since the user can then examine those files for configuration possibilities.

3.6 Pattern Analysis and Mining

While *VIP-View* and *VIP-Query* were able to use the file access histories directly, some applications require that more sophisticated analysis be performed. For example, prefetching, automatic compression, and intrusion detection systems all need to know what groups of files were accessed at the same instant. In other words, the usage patterns need to be extracted from the raw data.

There are several models presented for usage pattern analysis, including the *1-gram*, *2-gram+*, *p-s-gram+* and *association rule mining* models. Following the discussion of the prediction models, three applications of the prediction models will be discussed, including a *prefetching cache*, *automatic compressed filesystems* and *intrusion detection*.

3.6.1 1-gram

As mentioned in Section 2, an *N-gram* associates a sequence of events of length N with the event that follows. The *1-gram* approach is an instance of the *N-gram* model where $N = 1$, and all events are file accesses. It has also been described as a *last-successor* model [2], because it records the last successor of each file access. As an example, if the file accesses `[A B C A Z Y]` were made previously, the *1-gram* would have built the following prediction table:

key	value
A	Z
B	C
C	A
Z	Y

It is no accident that the above table looks like a dictionary data structure; the *1-gram* is indeed implemented using a dictionary data structure. In particular, we use a *linear hashtable* [10], which supports constant-time operations (insert, delete and lookup), minimizes hash collisions, and grows incrementally as necessary. In fact, all of the models presented in this paper can be implemented in a similar fashion. All of the usage data has the form `(key,value)`. The only variation is that there are sometimes multiple keys or multiple values that combine to form one hashtable key or one hashtable value, respectively. Our hashtable implementation does not distinguish between multiple keys and a single key; rather, all operations take a buffer representing the key as input and compute the hash bucket from this buffer.

When used in prediction, the *1-gram* simply predicts that the same file to succeed a given file the last time around will also succeed the file the next time around. For example, if the file access `Z` is given as input to the above *1-gram*, it would output a prediction of `Y`, since the last time around,

Z was immediately followed by Y. If in reality, the next file access ends up being Q, the Y value in the above table would then be replaced with Q.

3.6.2 2-gram+

The *2-gram+* approach is an *N-gram+* where the maximum N is 2. It subsumes both a 2-gram and a 1-gram and maintains tables for both. If the file accesses [A B C A Z Y] were made previously, the *2-gram+* would have built the following prediction tables:

1-gram

key	value
A	Z
B	C
C	A
Z	Y

2-gram

key	value
AB	C
BC	A
CA	Z
AZ	Y

When used in prediction, the 2-gram is checked first for a prediction, and if the 2-gram had nothing to predict, the 1-gram is checked for a prediction. For example, consider the case where the above tables had been built and the input ZB was given to the *2-gram+*. Since there is no entry in the 2-gram matching ZB, the 1-gram table would be probed for the key B. In this case, there is a key matching B, so C would be output as a prediction.

3.6.3 p-s-gram+

The *p-s-gram+* model generalizes the *N-gram+* model. Rather than predicting only one successor file, it can be configured to predict any number of successors (s). The number of predecessors *(p)* is configurable as usual. The system supports fall back for p so that if the predictor with the largest predecessor count cannot cannot make a prediction, the predictor with second-largest predecessor count is consulted, and so on. (The system could also support fall back for s, but that is not as meaningful since a given s could automatically subsume all lesser values of s by ignoring one or more of the successors). As an example, if the file accesses [A B C A Z Y] were made previously, a *2-3-gram+* would build the following prediction tables:

2-3-gram

key	value
AB	C A Z
BC	A Z Y

1-3-gram

key	value
A	B C A
B	C A Z
C	A Z Y

When used in prediction, the *p-s-gram+* performs exactly like the *N-gram+*, except that multi-file predictions can be made. For example, if the above tables had been built and the input BC was given to the *p-s-gram+*, the output would be AZY, meaning that files A, Z and Y are all predicted to be accessed soon.

Cache simulations revealed that the 1-predecessor and 5-successor combination performed as good as any of the *p-s-gram+* models tested, so the *1-5-gram+* was chosen to represent the *p-s-gram+* model for the cache experiments in this paper. Section 4.1.5 discusses this choice in more detail.

3.6.4 Association Rule Mining

The *Association Rule Mining* (*ARM*) approach builds association rules [11] out of the file accesses that have been seen. The *Apriori* [12] algorithm is applied to the training data to yield a set of association rules that can be used during testing. This approach is unique among the prefetching approaches simulated in this paper since it does not consider the order of file accesses. Rather, it produces rules pertaining to all files accessed within the same *window size* (configured to 5 accesses). For example, if the file accesses [A B C A Z Y] were made during training, *ARM* might build the following prediction table (shown with many rows omitted):

Antecedent	Consequent
A	B,C,Z
...	...
BC	A,Z,Y
...	...
ABC	Z
...	...
BCAZ	Y

By definition, the *ARM* approach should include every rule generated by the *p-s-gram+*-based models. In addition, *ARM* will generate rules about files associated a few accesses apart, in any order.

Each rule also has a value for *support* (how frequently all files mentioned in the rule appear together, compared to all files that appear together) and *confidence* (what percentage of the time the consequent of the rule appears, if the antecedent of the rule appears).

There are many types of *Association Rule Mining*, including many specialized implementations such as sequence mining or incremental association rule mining. In this paper, we use a novel moving window association rule mining algorithm developed for mining evolving data [13].

When used in prediction, *ARM* uses recent file accesses to generate a list of possible predictions. This list is sorted by support, then by confidence, and finally by length of the antecedent of the rule. (Sorting by support first yielded slightly better results on average during the prefetching cache tests than sorting by either of the other two sort keys first). Finally, the list is trimmed to a configurable maximum number of predictions, and the remaining predictions are made.

3.6.5 Prefetching Cache

Using the prediction models discussed above, a *prefetching cache* can be built. After each file access is made, a cache system could use a prediction model to determine what files are expected to be accessed soon. If the prediction model returned any predictions and there was spare I/O bandwidth, the prefetching cache could begin to pull the data for the predicted files into cache, in the hopes that future real accesses for the predicted files would result in cache hits. Finally, the system could make updates to its prediction model to reflect patterns in recent accesses. A prefetching filesystem is especially relevant for networked file systems; with their increased latency, the impact of I/O savings resulting from successful prefetches would be all the more dramatic. A simulation of prefetching caches is presented in Section 4.

3.6.6 Automatic Compressed Filesystems

An effective filesystem that supports automatic (i.e. transparent to the user) compression and decompression of files should have two primary goals. First, like other filesystems, it should maximize I/O throughput for files accessed frequently. Second, it should be able to minimize space occupation for files accessed infrequently.

The file access histories can determine whether a given file is accessed frequently or not, and should be the primary

tool in determining which files to compress. Obviously, a large file that is accessed rarely is a file that ought to be compressed. But even smaller files that are accessed rarely are good files to compress. The file access histories used in this paper revealed many files that were only accessed once during the tracing period. In fact, a majority of the files accessed during this time were accessed two times or less. More than its size, a file's access frequency should be used to determine whether it is selected for compression.

For files that are accessed frequently, it makes sense to store them uncompressed, since that is a format where they are more readily accessible. However, for files that are accessed only occasionally, it makes sense to store them compressed, to save space. However, that increases the access overhead, since the data for these files needs to make a transition from a compressed to an uncompressed state, before the data can be read or written. Using prediction models, this overhead can theoretically be lessened.

Similar to the way the prefetching cache operates, an automatic compressed filesystem could make predictions about what compressed files are likely to be accessed soon, based on recent accesses. This way, the filesystem can immediately begin the work of uncompressing a compressed file; hopefully, by the time the real access for the file is made, the data will have fully made the transition to an uncompressed state. A simulation of a prefetching cache with automatic decompression can be found in Section 4.1.7.

3.6.7 Intrusion Detection

Finally, we consider using the file access histories and prediction models as part of an intrusion detection system (IDS). The primary goal of an IDS is to accurately distinguish legitimate users of a system from intruders, and signal alarms when hostile users are determined to be present.

IDS systems can be characterized as *signature-based* or *anomaly-based* [14]. *Signature-based* IDS systems identify intruders by watching for usage patterns known to be intrusive. *Anomaly-based* systems identify intruders by looking for aberrations from normal usage patterns. Theoretically, the file access histories and prediction models in this paper could be leveraged as part of an *anomaly-based* IDS.

Clearly, file access histories could play a role in the establishment of normal usage patterns. Once normal usage patterns were established and the prediction models were sufficiently trained, the prediction models could measure how frequently their predictions came true. For a given user, if very few predictions came true, then the signaling of an alarm could be considered. If many predictions came true, then the system could more confidently characterize the user as legitimate, and could reduce false alarms. However, a comprehensive IDS would also need to incorporate user command histories as well as file access histories. In short, we believe file access histories could strengthen an IDS, but an IDS could not rely solely on such information.

4. EXPERIMENTAL RESULTS

The effectiveness of the prediction models was tested via a filesystem cache simulation and an intrusion detection simulation.

4.1 Cache Simulation Results

Two sources of data were used for the cache simulation. One data set, referred to in this paper as *vis-fstrace*, was comprised of a single user's file access traces over a 2-week period, totaling around 20,000 file accesses; we have performed the same experiments using data sources from other users as well, with comparable results. All of the file accesses in *vis-fstrace* were captured during normal day-to-day use of a filesystem. The other data set, referred to in this paper as *dfstrace*, was a multi-user data set and was the same data set used by Kroeger and Long in their work [2].

In the experiments that follow, *cachesize* defines the number of files the cache can hold. In this paper, we only consider whole-file caching. *Training amount* defines what percentage of the file accesses were used as training data to build the prediction model. During training, no statistics on cache hits were kept.

There were a number of caching strategies that were simulated, including the *non-predicting* model, the *1-gram* model, the *2-gram+* model, the *1-5-gram+* model, and the *association rule mining* model. In addition, we built a *hybrid* model out of the top-performing *ARM* and *1-5-gram+* models. The *hybrid* model simply consults each of its sub-models for predictions, and makes any predictions that either submodel offers.

The *non-predicting* approach is a baseline cache that has no knowledge of the file access histories. This cache simply adds recently accessed files to the MRU end and removes items from the LRU end as necessary to make room for new entries in cache. When a file in cache is accessed, it is moved to the MRU end. The *non-predicting* approach can still yield a high hit rate, given sufficient re-use of recently accessed files.

The other four caches were built by augmenting the baseline cache with one of the forms of prediction discussed in this paper. The following table shows a direct comparison of the caching strategies:

Model	Cache Hit Rate
Non-predicting	63%
1-gram	84%
2-gram+	87%
1-5-gram+	90%
ARM	90%
hybrid (ARM/1-5-gram+)	90%

A closer look at the *ARM*, *1-5-gram+* and *hybrid* hit rates reveals how close to optimal the predictors can be. These caches missed 10% (100% - 90%) of the time. However, 75% of these misses were due to accesses during testing for files that did not exist during training. For these files, it would be impossible to generate any predictions anyway, since no patterns involving these files have been seen. So, the optimal hit rate with this data set would be about 92.5%. The 2.5% difference between both predictors' 90% hit rate and the optimal 92.5% can be attributed to the infrequent case of new patterns being exhibited among previously seen files. This result strongly supports the idea that past file access patterns can be used to predict new ones.

The above table used the *vis-fstrace* data, with *cachesize* set to 70 files and *training amount* set to 55%. For the *ARM* system, *max predictions* were set to 9, *support threshold* was set to 0.005 and *confidence threshold* was set to 0.5. Unless otherwise specified, the remaining experiments all use this configuration.

4.1.1 Varying Training Data Amount

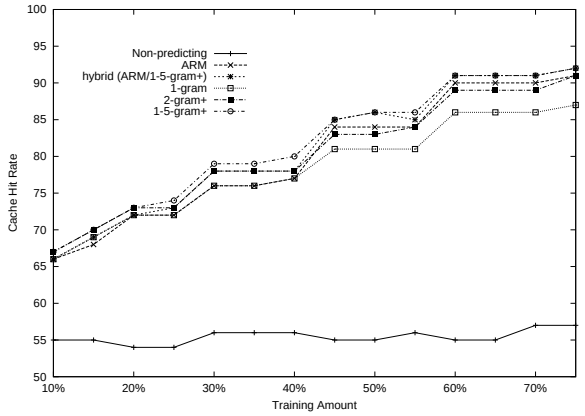

Figure 4: Varying Training Data Amount

Figure 4 shows the performance of the various approaches to caching as training size varies. Not surprisingly, all of the approaches that actually trained performed better with a larger training data amount. The results were obtained by averaging together 5 unique random samples, each using 50% of the available records for both training and testing purposes.

For all training amounts larger than 50%, all predictive approaches yielded hit rates about 1.5 times greater than the *non-predicting* approach.

4.1.2 Varying Cache Size

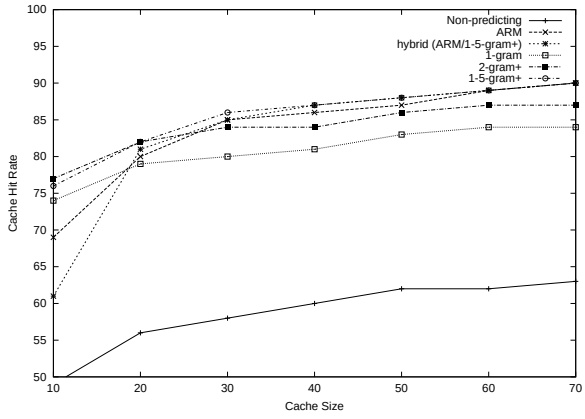

Figure 5: Varying Cache Size

Figure 5 shows the performance of all the cache systems as cache size varies. Not surprisingly, in all cases the hit rate increased gradually as cache size increased. For all cache sizes greater than 30, the the *ARM*, *1-5-gram+* and *hybrid* yielded hit rates about 1.45 times greater than the *non-predicting* approach.

4.1.3 Varying Maximum Predictions Made by ARM

Unlike the *1-gram* and *2-gram+*, the *association rule mining* approach can make several predictions at each file event. Figure 6 shows how the performance of the *ARM* approach varies as it is allowed to make an increasing max number of predictions. The experiment was repeated for three different cache sizes.

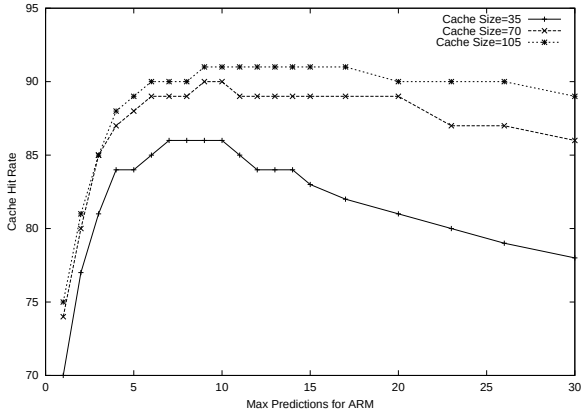

Figure 6: Varying Maximum Predictions Made by ARM

This figure shows that there is a "sweet spot" for how many predictions to make. If too few predictions are made, the hit rate is less than optimal since the system is less likely to pull in the right successor. However, if too many predictions are made, the cache turnover increases, reducing temporal locality, and therefore lowering the hit rate. Another reason to keep the max predictions as small as possible is that in a real file system the I/O generated by prefetching may block regular (non-prefetching) I/O, causing an I/O bottleneck.

It is surprising to discover that the peak hit rate occurred at the same maximum number of predictions (9) for all cache sizes simulated. This implies that the *max predictions* is not dependent on cache size, a fact that could simplify cache configuration on a real system. Also this potentially suggests that recent work on temporally constrained associations [15] might be beneficial for this problem.

4.1.4 Varying Maximum State Occupation

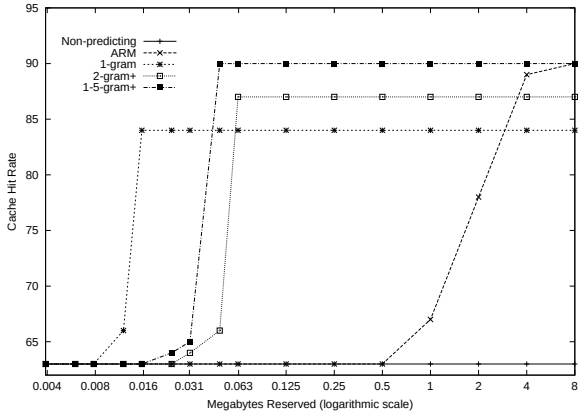

Figure 7: Varying Maximum State Occupation

Prefetching systems cannot realistically expect to be able to consume an unlimited amount of state for prefetching purposes. Figure 7 shows the performance of the various systems when an upper bound on state occupation is en-

forced. When running in this mode, each prefetching system was modified to expire older rules as necessary to make room for more recent rules.

The graph reveals that the *ARM* model requires more space to be effective, since it generates more rules. The *p-s-gram+*-based models reach their optimal numbers much sooner, confirming the fact that the *p-s-gram+* models require less state.

4.1.5 Comparing p-s-gram+ Possibilities

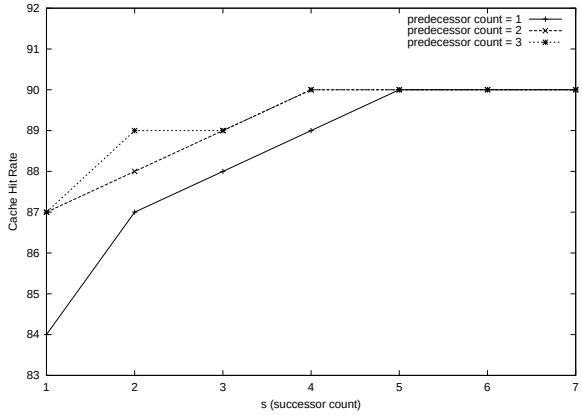

Figure 8: Comparing p-s-gram+ Possibilities

The *p-s-gram+* model offers a choice of values for both p and s. Figure 8 shows that with sufficiently large s, regardless of the choice of p, the peak hit rate is achieved.

But the choice of p is important. With $p = 1$, the memory occupation of the prediction table is bounded in the worst case by $O(N)$, where N is the number of unique files on the system. But with $p = 2$, the worst case memory occupation grows to $O(N^2)$. And in general, the worst case memory occupation will be $O(N^p)$. So the best approach appears to be to choose $p = 1$ and then choose s to be as large as necessary to reach the peak hit rate. Figure 8 shows that a *1-5-gram+* is a good choice, according to the approach just mentioned.

4.1.6 Prefetch Arrival Time

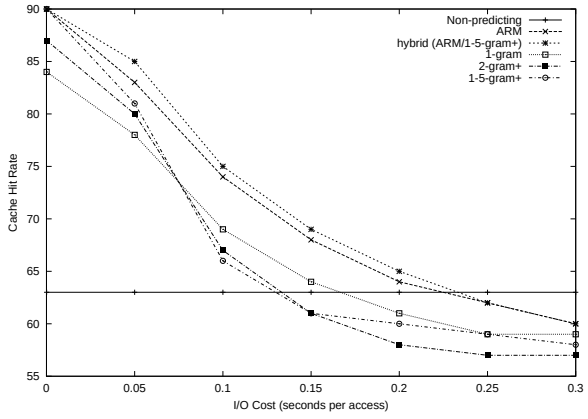

Figure 9: Varying the Cost of I/O

When a prefetch is made, time may be spent pulling the file into cache to ensure it is available in cache for a later fetch. It is crucial that most of the time, prefetched files arrive in cache before the later fetch is made. If the prefetch is still in progress when the predicted fetch occurs, then the prefetch was not only a waste of time, but also may have caused unrelated I/O to block needlessly.

On the test system, the cost of each file access was determined to be between 0.001 and 0.002 seconds. However, the cost of performing I/O is system-dependent, and varies with each instance, depending on available I/O bandwidth at access time.

Figure 9 shows the cache hit rate as the cost of I/O varies. As the cost of I/O increases, performance degrades for all of the prefetching approaches. With sufficiently slow I/O, all of the prefetching systems degrade to worse than the non-prefetching approach.

For all I/O costs tested, the *ARM*-based approaches (*ARM*, *hybrid*) performed, on average, 1.4 times better than the *p-s-gram+*-based approaches.

4.1.7 Incorporating Automatic Decompression

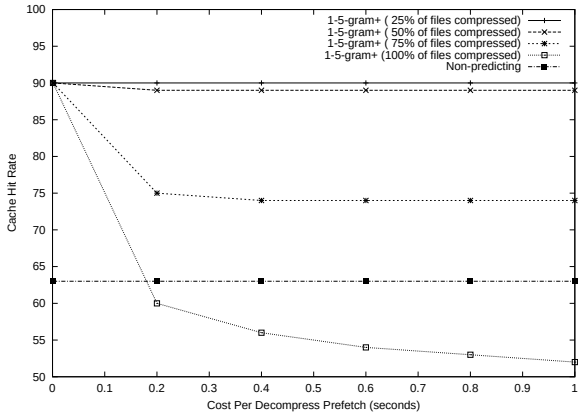

Figure 10: Incorporating Automatic Decompression

A filesystem which automatically compresses files has several issues to contend with. First, it needs to choose which files to compress. To model this, we sorted the frequency of accesses for each file, and chose the files with the least frequency. As mentioned in Section 3.6.6, we discovered during data collection that a majority of the files were accessed only two times or less.

Second, it needs to compress those files. We assume this can be done when there is idle CPU and spare I/O bandwidth, so we don't model its cost.

Finally, it needs to pay the cost of decompressing compressed files during prefetch time. For this, we determined the average time it takes to decompress an average-sized file and used this result in our experiment. For our test system, it was determined to be around 0.035 seconds, or about 20-30 times slower than only performing a normal read.

Figure 10 shows the results of simulating the addition of an automatic decompression feature to the filesystem cache. It fixes the I/O cost for normal prefetches to the test system's typical cost (about 0.0015 seconds), and then varies the I/O cost for prefetches which also perform decompression. It shows results for the *1-5-gram+* while the number of files compressed varies among 25%, 50%, 75% and 100%. (When only part of the files are chosen to be compressed, the

least frequently accessed ones are chosen). It also compares against the baseline non-predicting model.

This figure reveals only slight deterioration in cache hits with 50% of the files compressed and the cost of a decompression prefetch at the maximum. Only when 75% and 100% of the files are compressed is there a significant reduction in cache hits. In other words, the amount of files chosen for compression is a more important determining factor of the effect on cache hits than the decompression cost.

Another factor which could affect the viability of automatic decompression is file size. For very small files, the benefits of compression are small relative to the compression/decompression overhead. The process of selecting which files to compress should eliminate files that are too small to be useful.

4.1.8 *Varying Training Data with dfstrace*

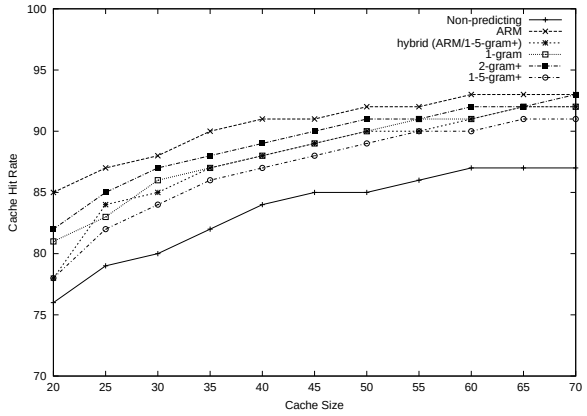

Figure 11: Varying Training Data with dfstrace

Figure 11 shows the results of running the various predictors on the *dfstrace* data set while varying the amount of training data. The *dfstrace* data set captured file access traces from many simultaneous users. This means that different users' file accesses may be interleaved with one another on the logs, disrupting the predictive accuracy of the models. Despite the possibility of disruption, the results in figure 11 clearly show an advantage of all of the predictive models over the non-predictive model. In particular, the *ARM* model yields, on average, 1.08 times more cache hits than the *non-predicting model*. This advantage could be heightened were the predictive models built on a per-user basis rather than globally.

4.2 Intrusion Detection Results

The effectiveness of the prediction models was also tested via an intrusion detection simulation. However, instead of predicting over file access patterns, predictions were made over user command histories.

For this experiment, our data set was user command data captured from eight UNIX users at Purdue University [16]. The data is in the form of sanitized shell history files, captured over a period of up to two years, and appears in order by the date the command was issued. Although each command is not timestamped, the entry and exit points to shells sessions were marked. We filtered out common commands such as `ls, cd, cp` and `mv`, etc. We believe that these common commands are used in so many different contexts that they are not useful in making predictions about what commands should appear after them.

Two prediction models were used, the *p-s-gram+* model and the *ARM* model. A series of tests were performed; each test used 50% training data and 50% testing data. For each configuration evaluated, we conducted two distinct kinds of tests. The first test evaluated how accurately the model could predict future UNIX commands for a user, after training on that user's UNIX command histories. The second test evaluated how accurately the model could predict future UNIX commands for a user, after training on a different user's UNIX command histories. The further these two test results are from each other, the greater the model's ability to distinguish true users from intruders.

Dividing the model's ability to predict correctly for true users by its ability to predict correctly for intruders yields a quotient we are terming the *predictive gap*. The larger the *predictive gap*, the more users and intruders are distinguished by the predictive model. Results from these experiments are listed in the following table, presented in decreasing order of performance:

Model/Configuration	Predictive Gap
1-1-gram	10.94
2-1-gram+	9.01
3-1-gram+	8.66
2-3-gram+	7.10
1-2-gram+	6.49
1-5-gram+	6.44
ARM (1 prediction)	5.86
ARM (2 predictions)	5.64
ARM (3 predictions)	5.63
ARM (5 predictions)	4.61

The *p-s-gram+* model was tested over several configurations for predecessor and successor count. In this case, a low predecessor count, and particularly a low successor count improves the predictive accuracy. In fact, the optimal tested configuration was for *p=1, s=1* (i.e. the last-successor model). For the *p-s-gram+*, predictive accuracy topped out at 10.94.

The *ARM* model was tested over several configurations for maximum predictions made. As with the *p-s-gram+*, the numbers indicate that a smaller number of predictions are superior to a larger number of predictions. For the *ARM* model, predictive accuracy topped out at 5.86.

5. CONCLUSIONS AND FUTURE WORK

We have shown a sample of applications of file usage patterns that can benefit the user directly. Users can see how they have used their files in the past, which can help them restructure their files to such that they are easier to access. Users can also learn how their programs are using their files, which can provide insight into how those programs function, and can aid in configuration and debugging of those programs. We have also shown a sample of applications of file usage patterns that can benefit the system. This includes a prefetching filesystem cache with optional automatic decompression that outperforms a non-prefetching filesystem cache across the board in simulation. We also explore the idea of using file usage patterns as part of an intrusion detection system. These systems benefits will indirectly benefit the user as well, in the form of better performing filesystems and improved security.

We have several specific conclusions to make related to a prefetching filesystem caches. The *1-5-gram+* model and the *ARM* model are both excellent, near-optimal predictive models for file accesses. A hybrid combination appears to work well, as predictions missed by one model are caught by the other. The *ARM* model showed superior results as the cost of I/O increased, indicating that it was making more accurate predictions. However, the lower space requirements and simpler efficient pattern analysis make the *1-5-gram+* model attractive as well. The *ARM* model performs best when it makes a limited number of predictions; unrestricted, *ARM* would have so many predictions to make that it would end up polluting the cache.

An important result from our automatic decompression simulation is that a majority of a user's files are accessed infrequently. This means that a majority of a user's files could be selected for compression. Our results show that even with 50% of a user's files being compressed and the cost of each decompression being high, the cache hit rate can remain near-optimal. This can provide significant space savings for each user.

While making multiple predictions increased cache hits for the filesystem cache simulation, single predictions were more beneficial for intrusion detection. Our intrusion detection results show that the *1-1-gram+* or *last-successor* model best distinguishes true users from intruders. The *1-1-gram+* is able to predict the next file more than 10 times more accurately for a true user compared to an intruder.

We believe that the applications featured here are just the beginning. When filesystems become sensitive to the unique patterns that identify each user, there are many new opportunities that arise. We hope our simulations in this paper and our sample implementations have convinced the reader of the advantages of making filesystems more personalized.

Future Work: Using the results generated through simulation, future work includes implementing a prefetching cache with automatic decompression on top of an existing file system. Our implementation work will involve replacing the existing file system's cache infrastructure with our own, and measuring changes in performance.

Currently, this paper only considers whole file caching. Although this is somewhat justified since small files are the majority of files [17], we still need to consider the fact that our whole file caching is impractical for large files. For our prefetching cache implementation, we will want to consider partial file caching. We plan on monitoring which byte ranges are accessed within a given file, so we can decide which file blocks to prefetch.

Finally, we would like to either build or modify an existing Intrusion Detection System. We believe that file access histories and the prediction models from this paper can assist in the establishment of normal usage patterns and in the recognition of intruders.

6. REFERENCES

[1] Ben Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on Graphics*, 11(1):92–99, Jan 1992.

[2] T. M. Kroeger and D. D. E. Long. The case for efficient file access pattern modeling. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[3] Jim Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *USENIX Summer*, pages 197–207, 1994.

[4] K. F. Lee and S. Mahajan. *Automatic Speech Recognition: The Development of the SPHINX System.* Kluwer Publishers, 1989.

[5] Ye Lu Zhong Su, Qiang Yang and Hong-Jiang Zhang. Whatnext: A prediction system for web requests using n-gram sequence models. In *Proceedings of the First International Conference on Web Information System and Engineering Conference*, 2000.

[6] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of 21st ACM Symposium on Principles of Database Systems (PODS 2002)*, 2002.

[7] K. Sequeira and M. Zaki. Admit: Anomaly-based data mining for intrusions. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002.

[8] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[9] Bing Liu, Wynne Hsu, and Yiming Ma. Integrating classification and association rule mining. In *Knowledge Discovery and Data Mining*, pages 80–86, 1998.

[10] Witold Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on VLDB*, 1980.

[11] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufmann, 2001.

[12] Christian Borgelt and Rudolf Kruse. Induction of association rules: A priori implementation.

[13] M. Carvalho B. Possas S. Parthasarathy A. Veloso, W. Meira and M.Zaki. Efficiently mining approximate models of associations in evolving databases. In *ECML/PKDD*, 2002.

[14] Sandeep Kumar and Eugene H. Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, 1994.

[15] S. Parthasarathy H. Yang and S. Reddy. On the use of temporally constrained associations for web log mining. In *WEBKDD*, 2002.

[16] Terran D. Lane. *Machine Learning Techniques for the computer security domain of anomaly detection.* PhD thesis, Department of Electrical and Computer Engineering, Purdue University, August 2000.

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