# Discovery of Application Workloads from Network File Traces

Neeraja J. Yadwadkar, Chiranjib Bhattacharyya, K. Gopinath
Department of Computer Science and Automation, Indian Institute of Science

Thirumale Niranjan, Sai Susarla
NetApp Advanced Technology Group

## Abstract

*An understanding of application I/O access patterns is useful in several situations. First, gaining insight into what applications are doing with their data at a semantic level helps in designing efficient storage systems. Second, it helps create benchmarks that mimic realistic application behavior closely. Third, it enables autonomic systems as the information obtained can be used to adapt the system in a closed loop.*

*All these use cases require the ability to extract the application-level semantics of I/O operations. Methods such as modifying application code to associate I/O operations with semantic tags are intrusive. It is well known that network file system traces are an important source of information that can be obtained non-intrusively and analyzed either online or offline. These traces are a sequence of primitive file system operations and their parameters. Simple counting, statistical analysis or deterministic search techniques are inadequate for discovering application-level semantics in the general case, because of the inherent variation and noise in realistic traces.*

*In this paper, we describe a trace analysis methodology based on* Profile Hidden Markov Models. *We show that the methodology has powerful discriminatory capabilities that enable it to recognize applications based on the patterns in the traces, and to mark out regions in a long trace that encapsulate sets of primitive operations that represent higher-level application actions. It is robust enough that it can work around discrepancies between training and target traces such as in length and interleaving with other operations. We demonstrate the feasibility of recognizing patterns based on a small sampling of the trace, enabling faster trace analysis. Preliminary experiments show that the method is capable of learning accurate profile models on live traces in an online setting. We present a detailed evaluation of this methodology in a UNIX environment using NFS traces of selected commonly used applications such as compilations as well as on industrial strength benchmarks such as TPC-C and Postmark, and discuss its capabilities and limitations in the context of the use cases mentioned above.*

## 1 Introduction

Enterprise systems require an understanding of the behavior of the applications that use their services. This application-level knowledge is necessary for self-tuning, planning or automated troubleshooting and management. Unfortunately, there is no accepted mechanism for this knowledge to flow from the application to the system. We can neither impose upon application developers to give hints, nor over-engineer network protocols to transport more semantics. Therefore, we need mechanisms for systems to *learn* what the application is doing *automatically*.

Being able to identify the application-level workload has significant benefits. If we can figure out that the client OLTP (online transaction processing) application is doing a *join*, we can tune the caching and prefetching suitably. If we can discover that the client is executing the *compile* phase of a *make*, we can immediately know that it will be followed by a *link* phase, that the output files generated will be accessed very soon, and that the output files can be placed on less-critical storage since they can be generated at will. If we can spot that the client is executing a *copy* operation, then we can derive data provenance information usable by compliance engines. If we can match the signature of a trace with that of known malware or viruses, that can be useful as well. We can employ offline workload identification for auditing, forensics and chargeback. We can help storage systems management by providing inputs to sizing and planning tools.

In this paper, we tackle a specific instance of the problem – given the headers of an NFS [4] trace, to identify the application-level workload that generated it. NFS clients send messages to the server that contain opcodes such as READ, WRITE, SETATTR, READDIR, etc., their associated parameters such as file handles and file offsets, and data. An NFS trace contains a timestamped sequence of these messages along with the responses sent by the server to the client. These traces can be easily captured [12, 1] for online or offline analysis, allowing us to develop a non-invasive tool using the methodology described here. Furthermore, the NFS trace contains all the interactions between the clients and the server. As all the necessary information is available, we can assert that any deficiency in tackling our use cases is solely due to the sophistication of the analysis methods.

However, given a trace captured at the server, it is non-trivial to identify the client applications that generated it. First, there could be noise in the form of background communication between the client and server. Second, messages could be interleaved with those from other applications on the same client machine. Third, the application's parameters may create variations in the trace. For instance, traces of a single file copy and that of a recursive file copy may look very different (see Tables 1 and 2), even though it is the same application. Fourth, the asynchrony in multi-threaded applications impact the ordering of messages in the traces. Therefore, we believe that deterministic pattern searching methods will not be able to unearth the fundamental patterns hidden in a trace. Methods originating in the Machine Learning domain have shown considerable promise in computational biology [16, 14] as well as in initial studies on trace analysis [19]. In this paper, we apply a well-known technique called *Profile Hidden Markov Model* (profile HMM) [16, 14] to this problem, and demonstrate its pattern-recognition capabilities with respect to our use cases.

The key contributions of this paper are as follows:

**Workload Identification** We show that profile HMMs, once trained, are capable of identifying the application that generated the trace. Using commonly used UNIX commands such as *make, cp, find, mv, tar, untar*, etc., as well as industry benchmarks such as TPC-C, we show that we are able to cleanly distinguish the traces that these commands generate.

**Trace Annotation** We show that our methodology is able to identify transitions between workloads, and mark workload-specific regions in a long trace sequence.

**Trace Sampling** We show that profile HMMs do not need the entire trace to work on. With merely a 20% segment of the trace, sampled randomly, we are able to discriminate between many workloads and identify them with high confidence. This will enable us to perform faster analysis. Further, we show how to use this ability to identify concurrently executing workloads.

**Automated Learning** We demonstrate a technique by which the profile HMMs can be trained automatically without manual labeling of workloads. We use the technique to train and then subsequently identify constituent workloads of a Linux kernel compilation task.

**Power of Opcode Sequences** We show that opcode sequences alone contain sufficient information to tackle many of the common use cases. Other information in the traces such as file handles and offsets are not sufficiently amenable to mathematical modeling, so this result is valuable.

Since the technique we use requires training on data sets followed by a recognition phase and also involves reasonable amounts of computation, it is best suited for those problems whose natural time constants are in the minutes or hours range (such as in system management, for example, detecting configuration errors). Algorithmic approaches, widely used, are still the best if the time constants are much smaller (such as in milliseconds or seconds).

The rest of the paper is organized as follows. Section 2 presents the current state of research in this area and places our work in context. Section 3 describes the mathematics behind our methodology, the workflow associated with it, and describes how it is used to identify workloads and mark out regions exhibiting known patterns in the trace. Section 4 offers experimental validation of our techniques. Finally, Section 6 summarizes our conclusions and proposes avenues for continuing this work.

## 2 Related Work

There is a rich body of work in which file system traces have been analyzed to get aggregate information about systems and to understand how storage is used over time [2, 17, 24, 11]. Our work differs from this body of work in that we focus on individual workloads running on the system and attempt to discover them. Since prior research efforts are oriented towards extracting gross behavior, counting-based tools suffice. The problem that we tackle in this paper requires more powerful methods.

Traces are a good source of information as they contain a complete picture of the inputs to a system and at the same time are easy to capture in a non-invasive manner. Ellard [10] makes a strong case that the information in NFS traces can be used to enable system optimizations. HMMs generated from block traces have been used for adaptive prefetching [27]. Traces have been used for file classification [19]. In that work, the authors build a decision tree based system that uses NFS traces to infer correlations between the create-time properties of files such as their names and the dynamic properties such as access patterns and size. In this paper, we do not attempt to classify files and data but focus more on the applications that access them.

The power of HMM as a tool to extract workload access patterns is known [18]. Our work is significantly larger in scope. While they restrict themselves to inferring the sequentiality of workloads using read and write headers in the block traces, we use all the opcodes available in NFS headers to discover the higher-level application that caused it. The sequentiality of a workload can perhaps also be discovered using our framework by including the file offsets as part of the alphabet through an appropriate scheme of quantization.

Magpie [3] diagnoses problems in distributed systems by monitoring the communications between black-box components, and applying an edit-distance based clustering method to group similar workloads together. Somewhat similar is Spectroscope [25], which uses clustering on request flow graphs constructed from traces to categorize and learn about differences in system behavior. Intrusion detection is another area where various such techniques are used. Warrender [29] surveys methods for intrusion detection using various data mining techniques including HMMs, on system call traces.

Our work is different from all of the above in that it is not only able to identify a higher-level workload, given a trace, but also to be able to accurately mark out workload regions in a composite trace.

## 3 Methodology

A key observation that motivates our approach to solving the problem is that NFS traces corresponding to a given workload class exhibit significant variability, yet have a characteristic signature. For instance, look at the four traces depicting a *cp* command, shown in Tables 1 and 2. The fuzziness in the repeating subsequences in the trace of *cp * dir/* and *cp -r dir1 dir* make us look at probabilistic methods.

An HMM is appropriate for probabilistic modeling of sequences, and has been used in similar settings in the past [14]. However, in our case, the sequences of the same workload show additions, deletions and mutations between them that are not easily modeled by an HMM. A *cp foo bar* differs from *cp foo dir/* – the latter has an extra *lookup* operation, as seen in Table 2. Our method should have the power to ignore this extra operation since that operation must not be used for discrimination. A variant of the HMM called the profile HMM [8] offers exactly this ability, via *non-emitting* (or *delete*) states. Therefore, we conjecture that profile HMM will be a good method to use for classifying NFS traces. In the rest of this section, we first outline the theory behind the profile HMM and then describe the workflow of our workload identification methodology.

### 3.1 Profile HMMs for Modeling Opcode Traces

It is well known and empirically verified, e.g., Table 1, that opcode traces of the same command are often very similar but not exactly the same. It is also known that traces corresponding to different commands are dissimilar. These observations motivate the development of mathematical models that are capable of discovering a command/workload by merely looking at the trace it generates (e.g., opcode se-

| cp * dir/ | cp -r dir1 dir |
|---|---|
| GETATTR Call, FH:0x0eb18814 | ACCESS Call, FH:0xc5914d40 |
| READDIRPLUS Call, FH:0x0eb18814 | LOOKUP Call, DH:0xc5914d40/dir |
| READDIRPLUS Reply (Call In 9) ... | LOOKUP Reply Error:NFS3ERR_NOENT |
| LOOKUP Call, DH:0xe003db8b/tqslwiz.h | MKDIR Call, DH:0xc5914d40/dir |
| LOOKUP Reply Error:NFS3ERR_NOENT | GETATTR Call, FH:0xc5914d40 |
| GETATTR Call, FH:0x21b1a714 | GETACL Call |
| ACCESS Call, FH:0x21b1a714 | ACCESS Call, FH:0xc5914d40 |
| CREATE Call, DH:0xe003db8b/tqslwiz.h | LOOKUP Call, DH:0xc5914d40/dir |
| SETATTR Call, FH:0x6bd9e67c | LOOKUP Reply, FH:0x3fb1b914 |
| GETACL Call | GETATTR Call, FH:0x0eb18814 |
| GETATTR Call, FH:0x6bd9e67c | ACCESS Call, FH:0x0eb18814 |
| READ Call, FH:0x21b1a714 ... | READDIRPLUS Call, FH:0x0eb18814 |
| WRITE Call, FH:0x6bd9e67c ... | READDIRPLUS Reply .  .. |
| COMMIT Call, FH:0x6bd9e67c | ACCESS Call, FH:0x3fb1b914 |
| GETATTR Call, FH:0xe003db8b | MKDIR Call, DH:0x3fb1b914/hh |
| **LOOKUP Call, DH:0xe003db8b/TrustedQSL.spec** | GETATTR Call, FH:0x3fb1b914 |
| **LOOKUP Reply Error:NFS3ERR_NOENT** | GETACL Call |
| **GETATTR Call, FH:0x2fb1a914** | GETATTR Call, FH:0x3fb1b914 |
| **ACCESS Call, FH:0x2fb1a914** | GETATTR Call, FH:0x36b1b014 |
| **CREATE Call, DH:0xe003db8b/TrustedQSL.spec** | ACCESS Call, FH:0x36b1b014 |
| **SETATTR Call, FH:0x65d9e87c** | READDIRPLUS Call, FH:0x36b1b014 |
| **GETATTR Call, FH:0x65d9e87c** | READDIRPLUS Reply .  .. |
| **READ Call, FH:0x2fb1a914 ...** | **GETATTR Call, FH:0x39b1bf14** |
| **WRITE Call, FH:0x65d9e87c ...** | **ACCESS Call, FH:0x39b1bf14** |
| **COMMIT Call, FH:0x65d9e87c** | **ACCESS Call, FH:0x3db1bb14** |
| LOOKUP Call, | **CREATE Call, DH:0x3db1bb14/contacts.csv** |
| DH:0xe003db8b/TrustedQSL.spec.in | **SETATTR Call, FH:0x33b1b514** |
| LOOKUP Reply Error:NFS3ERR_NOENT | **GETACL Call** |
| GETATTR Call, FH:0x23b1a514 | **GETATTR Call, FH:0x33b1b514** |
| ACCESS Call, FH:0x23b1a514 | **READ Call, FH:0x39b1bf14 ...** |
| CREATE Call, | **WRITE Call, FH:0x33b1b514 ...** |
| DH:0xe003db8b/TrustedQSL.spec.in | **COMMIT Call, FH:0x33b1b514** |
| SETATTR Call, FH:0x67d9ea7c | GETATTR Call, FH:0x21b1a714 |
| GETATTR Call, FH:0x67d9ea7c | ACCESS Call, FH:0x21b1a714 |
| READ Call, FH:0x23b1a514 ... | CREATE Call, DH:0x3fb1b914/tqslwiz.h |
| WRITE Call, FH:0x67d9ea7c ... | SETATTR Call, FH:0x35b1b314 |
|  | GETATTR Call, FH:0x35b1b314 |
| COMMIT Call, FH:0x67d9ea7c | READ Call, FH:0x21b1a714 ... |
|  | WRITE Call, FH:0x35b1b314 ... |
|  |  |
|  | COMMIT Call, FH:0x35b1b314 |

**Table 1.** Two *cp* NFS trace headers. The first one copies 3 files into a directory, while the second one is a recursive copy. These traces illustrate that workloads repeat some elements of the trace, with one region being underlined. However, the repetition of symbols is not strict and cannot be captured by a finite state automata model. There is sufficient variability that warrants a fuzzy or probabilistic pattern recognition algorithm such as an HMM. Figure shows only the client→server requests, not the responses. The sole exception is that of responses to LOOKUP since they will help the reader understand the traces.

| cp contacts.csv con.csv | cp contacts.csv dir/con.csv |
|---|---|
| ACCESS Call, FH:0xe003db8b | **LOOKUP Call, DH:0xe003db8b/dir** |
| LOOKUP Call, DH:0xe003db8b/con.csv | **LOOKUP Reply, FH:0x0eb18814** |
| LOOKUP Reply Error:NFS3ERR_NOENT | ACCESS Call, FH:0x0eb18814 |
| LOOKUP Call, | LOOKUP Call, DH:0x0eb18814/con.csv |
| DH:0xe003db8b/contacts.csv | LOOKUP Reply Error:NFS3ERR_NOENT |
| LOOKUP Reply, FH:0x71d9fc7c | LOOKUP Call, |
| GETATTR Call, FH:0x71d9fc7c | DH:0xe003db8b/contacts.csv |
| ACCESS Call, FH:0x71d9fc7c | LOOKUP Reply, FH:0x71d9fc7c |
| CREATE Call, DH:0xe003db8b/con.csv | GETATTR Call, FH:0x71d9fc7c |
| SETATTR Call, FH:0x58d9d57c | ACCESS Call, FH:0x71d9fc7c |
| GETACL Call | CREATE Call, DH:0x0eb18814/con.csv |
| GETATTR Call, FH:0x58d9d57c | SETATTR Call, FH:0x14b19214 |
| READ Call, FH:0x71d9fc7c ... | GETACL Call |
| WRITE Call, FH:0x58d9d57c ... | GETATTR Call, FH:0x14b19214 |
|  | READ Call, FH:0x71d9fc7c ... |
| COMMIT Call, FH:0x58d9d57c | WRITE Call, FH:0x14b19214 ... |
|  |  |
|  | COMMIT Call, FH:0x14b19214 |

**Table 2.** Two *cp* NFS trace headers. The second one differs from the first in an extra LOOKUP operation (underlined), showing the need for a methodology that can suppress or ignore certain elements in traces. Profile HMM is one such candidate. Figure shows only the client→server requests, not the responses. The sole exception is that of responses to LOOKUP since they will help the reader understand the traces.

quence), and checking for its similarity with prior traces of the same command with various arguments. The problem of constructing such models is complicated as there is no unique trace for every command. Similar issues arise in many other areas, notable among them being computational biology. The study of designing efficient sequence matching algorithms has received a significant impetus from computational biology where one needs to align a family of many closely related sequences (typically genetic or protein sequences). These sequences diverge due to chance mutations at certain points in the sequence while, at the same time, conserving critical parts of the sequence.

The similarity of two symbol sequences can be measured via the number of mutations needed to make them identical, also called the *edit distance*. Hence, to measure the similarity of a sequence to a set of sequences, one could first align them to be of the same length by adding, deleting or replacing the minimal number of symbols, and then use the smallest edit distance.

As of today there are quite a few techniques for sequence matching, ranging from deterministic [13] to probabilistic approaches [6]. Deterministic approaches are based on dynamic programming, which often leads to algorithms that have prohibitively high time complexity for large symbol sequences: $O(N^r)$ to match with r sequences, each of length N. Probabilistic approaches such as Profile HMMs [6] have emerged as faster alternatives to deterministic methods and have been proven to be very effective for computational biology problems. The key observation behind our work is that trace-based workload identification and annotation maps well to the sequence-matching problem in computational biology, and hence can benefit from similar techniques. Profile HMMs are special Hidden Markov models (HMMs) developed for modeling sequence similarity occurring in biological sequences. Next, we provide a high-level intuitive understanding of HMMs, profile HMMs and their use for sequence matching.

An HMM [23] is a statistical tool that captures certain properties of one or more sequences of observable symbols (such as NFS opcodes) by constructing a probabilistic finite state machine with artificial hidden states responsible for emitting those sequences. During training, the state machine's graph and its state transition probabilities are computed to best produce the training sequences. Later, the HMM can be used to evaluate whether a new unseen "test" sequence is "of the same kind" as the training data, with a score to quantify confidence in the match. The test sequence gets a higher score if the HMM has to traverse higher-probability edges in its state machine to produce that sequence. Thus, the HMM's state machine encodes the commonality among various opcode sequences of a given application workload by boosting the probabilities of the corresponding state transitions. It identifies a new workload by measuring how well its opcode sequence makes the HMM to make high-frequency transitions.

A profile HMM is a special type of HMM with states and a left-to-right state transition diagram specifically designed, as explained in Section 3.4.2, to efficiently remember symbol matches as well as tolerate chance mutations (i.e., inserts and deletes) in observed symbol sequences. Unlike a fully connected state graph of a traditional HMM, the profile HMM's left-to-right transition graph enables very fast $O(N)$ matching of a test sequence against known workload patterns.

In this paper, we consider two specific problems where existing sequence-matching techniques are applicable:

- *Workload identification*: we are told that samples are only from one workload but not told which one. Can we say which workload it is from?
- *Annotation*: we are told that distinct workloads ran sequentially one after another. Can we mark the boundaries when the workloads were switched?

In the following sections, we provide a more formal description of the HMM construct, including the concept of sequence alignment and how it is central to do approximate matching of large symbol sequences like opcode traces.

## 3.2 A Brief Review of HMMs

An HMM is defined by an alphabet $\Sigma$, a set of hidden states denoted by $Z$, a matrix of state transition probabilities $A$, a matrix of emission probabilities $E$, and an initial state distribution $\pi$. The matrix $A$ is $|Z| \times |Z|$ with individual entries $A_{uv}$, which denotes the probability of transiting to state $v$ from $u$. The matrix $E$ ($|Z| \times |\Sigma|$) contains entries $E_{ut}$, which denotes the probability of emitting a symbol $t \in \Sigma$ while in hidden state $u$. Let $\lambda$ be the model's parameters; these depend on $\Sigma, Z, A, E$ and $\pi$ and hence written as $\lambda = (\Sigma, Z, A, E, \pi)$. If we see a sequence $X$, an HMM can assign a probability to it as follows (assuming a model $\lambda$):

$$P(X|\lambda) = \sum_z \prod_k A_{z_k, z_{k+1}} E_{z_k, X_k}$$

The (inner) product terms arise from the probabilities of transition from one state ($z_k$) to another state ($z_{k+1}$) in the sequence of states under consideration whereas the (outer) sum of terms arises from having to sum all the possible ways of emitting the sequence $X$ through all possible sequence of states. There is an iterative procedure based on expectation maximization algorithms for determining the parameters $\lambda$ from a training set [23]. Popularity of HMMs stems from the fact that there are efficient procedures such as *(a) Viterbi algorithm* [23]) to compute the most probable state $Z$ given a sequence $X$, i.e. compute $Z$ to maximize $P(Z|X)$ *(b) forward and backward procedures* [23] to compute the likelihood, $P(X)$ and *(c) Expectation Maximization procedures* [23] to learn the parameters, $(A, E, \pi)$ given a dataset of independent and identically distributed sequences.

## 3.3 Problem Definition

At this point we can state the problem more formally as follows. Let $\{S_1, S_2, \dots, S_r\}$ be a set of traces obtained by executing $r$ times a particular workload, say $W$. The traces are different as they are obtained by executing the workload with different parameters; they may also be different due to some stochastic events in the system. The $j$th symbol $s_{ij}$ of the sequence $S_i$ is generated from the alphabet $\Sigma$ of all possible opcodes. Let the sequence $S_i$ be of length $n_i$, i.e the index $j$ varies from 1 to $n_i$. We consider the task of constructing a model on these $r$ sequences such that when presented with a previously unseen sequence, $X$, the model can infer whether $X$ was generated by executing workload $W$.

## 3.4 Profile HMMs for identifying workloads

We will begin by recalling a few definitions related to sequence alignment. We will then discuss profiles and Profile HMMs, finally ending with a scheme for classifying workloads using them.

### 3.4.1 On Aligning Multiple Sequences

Let $S_i = s_{i1}s_{i2}\dots s_{in_i}$ $(i = 1, 2)$ be two sequences of different lengths $n_1$ and $n_2$ generated from an alphabet $\Sigma$. An *alignment* of these two sequences is defined as a pair of new equal length sequences $S_i^* = s_{i1}^* \dots s_{in}^*$ $(i = 1, 2)$ obtained from $S_1(S_2)$ by inserting "$-$" states in $S_1(S_2)$ to record differences in the two sequences. Let $n$ be the length of $S_1^*$ (which is also that of $S_2^*$) with $(n_1 + n_2) \geq n \geq max(n_1, n_2)$. We will call $s_{1k}$ and $s_{2l}$ as **matched** if for some $j$ , $s_{1j}^* = s_{1k}, s_{2j}^* = s_{2l}$. On the other hand if $s_{1j}^* =$ "$-$",$s_{2j}^* = s_{2m}$ then we will say that there is a *delete* state in $S_1$ and *insert* state in $S_2$.

The **global alignment** problem is posed as that of computing two equal length sequences $S_1^*$ and $S_2^*$ such that the matches are maximized and insertions/deletions are minimized. This problem can be precisely formulated for suitably defined score functions and solved by dynamic programming based algorithms [20]. Global alignment is a good indicator of how similar two sequences are.

The problem of **local alignment** tries to locate two subsequences one from each string such that they are very similar. This problem can be formulated as that of finding two subsequences which are maximally aligned in the global sense for a suitably defined score function. It also admits a dynamic programming based algorithm [26] and can be solved exactly.

However both global and local alignment are defined for a pair of sequences. As mentioned before, our interest is in inferring similarities in more than two sequences. This will require the notion of multiple alignment, which generalizes the notion of alignment to more than two sequences. **Multiple alignment** is defined as the set $\mathcal{S} = \{S_1^*, S_2^*, \dots, S_r^*\}$ where, as before, $S_i^*$ is obtained from $S_i$ by inserting "$-$" states so that the length of all the resulting $r$ sequences are equal, say $n$. Multiple alignment can be visualized as a $r \times n$ matrix where each row consists of a specific string and each column corresponds to specific position in the alignment. Each matrix entry can take values in $\Sigma \cup$ "$-$". Multiple alignments are useful in detecting similar subsequences which remain conserved in sequences originating from the same family. Thus multiple alignment can decide the membership of a given new sequence with respect to a family represented by the multiple alignment. Figure 1 shows an alignment of ten traces of opcodes generated by an *edit* workload. Each symbol in the alignment represents a particular opcode. The alignment shows regions of high conservation where more than half of the symbols in the column are present. These conserved regions capture the similarity between the traces of this workload. When identifying a previously unseen trace generated by the same workload, it would be desirable to concentrate on checking that these more conserved columns are present.

One can extend the dynamic programming based solutions for the pairwise case to the problem at hand. Unfortunately they are prohibitively expensive, $O(n^r)$ in both time and space [13], and are not very practical for detecting large file operation sequences (100s to 1000s) typical in networked storage workloads.

### 3.4.2 Introduction to Profile HMMs

A *profile* is said to be a representation of a multiple alignment (such as that of multiple proteins that are closely related and belong to the same family). One can attribute the slight differences between family members to chance mutations, whose underlying probability distribution is not known. It has been empirically observed that HMMs are extremely useful in building profiles from biological sequences [6].

**Profile HMMs:** For modeling alignments, a natural choice for hidden states correspond to Insertions, Deletions and Matchings. In a Profile HMM, each insert state $I_i$ and match state $M_i$ has a nonzero emission probability of emitting a symbol, whereas the delete state $D_i$ does not emit a symbol. The non-emitting states make Profile HMMs different from traditional HMMs. From an insert state, it is possible to move to the next delete state, continue in the same insert state or go to the next match state (Figure 2). Each diamond, circle, and square represents insert, delete and match states respectively. From each insert, delete or match state, the possible state transitions are as follows:

```
- G A G L L G G A A   G G G G - - A A C C   S S S S - - G G V V   G D D S W W S - - -   - - - - S W G - - G
- - - - - - - - - -   - - - - - - - - - -   - - - - - - - - - L   - - - - W W - - - -   - - - - - - G - - G
- L W W S S R R L L   - - G G - - L L C C   S S S S G G G G V V   S S S S W W S S L L   S S W W S S G - - G
- - W W S S R R L L   A A G G - - L L C C   S S S S G G G G V V   - - - S W W S S L L   S S W W S S G - - G
S S W W S S G G V V   A A G G L L L L C C   S S S S G G G G V V   - - - S W W S S L L   S S W W S S G - - G
- - W W S S R R L L   G G - - - - L L C C   S S S S G G G G V V   - - - S W W S S L L   S S W W S S G - - G
- S W W L L G G L L   G G G G - - - - C C   S S S S - - G G V V   G D D - - W G G - -   - - - - A A G - - G
- S W W L L G G L L   G G G G - - - - C C   S S S S - - G G V V   L D D - - W G G - -   - - - - S G G - - A
- S W W L L G G L L   G G G G - - - - C C   S S S S - - G G V V   G D D S W W S G - -   - - - - G G G - - A
- S W W - - G G - -   G G G G - - - - C C   S S S S - - G G V V   G D D S W W S G - -   - - - - G G G G A G
    + + + + + + + +   + + + +         + +   + + + +     + + + +         + + + + +             + + +     +
```

**Figure 1.** An example of multiple alignment of ten NFSv3 traces generated by an *edit* workload using the wireshark [5] tool. Here, G is getattr, S setattr, L lookup, R read, W write, A access, D readdirplus, C create, M commit, V remove, etc. Aligned columns are annotated at the bottom by a '+' if the opcodes in those columns are highly conserved. These columns will be modeled as match states in the profile HMM.

$$\begin{array}{lllll} I_i & \rightarrow & D_{i+1}, & I_i, & M_{i+1}, \\ D_i & \rightarrow & D_{i+1}, & I_i, & M_{i+1}, \\ M_i & \rightarrow & D_{i+1}, & I_i, & M_{i+1}. \end{array}$$

Profile HMMs are essentially *Left-Right* HMMs (Figure 2). Unlike fully connected state machines, Left-Right HMMs have a more sparse transition matrix and are often upper triangular. Inference on such machines is much quicker and hence often preferred in many applications such as speech processing [23].

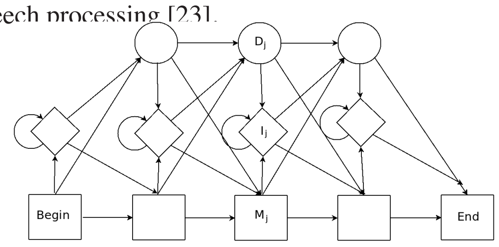

**Figure 2.** The transition structure of a profile HMM [8]. For example, from an insert state (diamond), we can go to the next delete state (circle), continue in the insert state (self loop) or go to the next match state (rectangle). Note that while multiple sequential deletions are possible by following the circle states, each with a different probability, multiple sequential insertions are only possible with the same probability.

It is straightforward to adapt the traditional HMM algorithms such as Viterbi algorithm, Forward-Backward procedure and Expectation Maximization based learning procedure [23] to profile HMMs [6, 8].

These models provide flexibility in modeling closely related sequences by the choice of more complex score functions. This has made profile HMMs extremely popular for comparing biological sequences.

**Learning a Profile HMM from data:** The parameters of profile HMMs are the emission probabilities and the state transition probabilities. This is easy to compute if one knows the multiple alignment. In such a case, the state transition probabilities are given by $a_{uv} = \frac{AN_{uv}}{\sum_v AN_{uv}}$ and the emission probabilities are given by $e_{ut} = \frac{EN_{ut}}{\sum_t EN_{ut}}$ where $AN_{uv}$ denotes the number of transitions from the state $u$ to $v$ and $EN_{ut}$ denotes the number of emissions of $t$ given a state $u$(see [6]).

### 3.4.3 Profile HMM for identifying workloads

Let us now revisit the problem as defined in subsection 3.3. Assume that we have pretrained many Profile HMMs, each for a workload. Now consider the problem of identifying the underlying workload when a new trace is presented. Using Profile HMMs one can consider solving such a problem by the decision rule

$$y(X) = argmax_k P(X|\lambda_k)$$

where $X$ is the unseen sequence, $\lambda_k$ denotes the model for the $k$th workload and $y(X)$ is prediction for the underlying workload which generated the sequence $X$. Using the forward-backward procedure we can compute this decision rule easily. This can be understood as globally aligning the profile with the unseen sequence. Though there is no confidence measure with respect to prediction, the input is rejected (no prediction is made) if a confidence threshold is not crossed.

Now consider the problem of annotating a huge trace of opcodes generated by sequentially running workloads. As before assume that we have pretrained models of individual workloads. This would be equivalent to computing a local alignment of each profile with the bigger trace.

It is thus clear that the Profile HMM architecture chosen should be versatile enough to solve such problems. The architecture shown in Figure 2 will require some tweaking or the inference mechanism needs to be modified for such problems.

**A Specific Implementation for Profile HMMs:** For our work here, we have used the open source HMMER [7] implementation of a profile HMM whose architecture (Figure

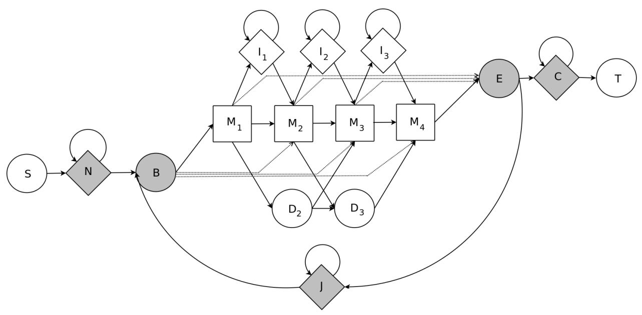

**Figure 3.** Architecture of HMMER [7]. Squares represent match states w.r.t. an alignment, diamonds are insert and ignored emitting states (N,J,C), circles are delete and special begin/end states (B,E,S,T). Note that there are no D to I or I to D transitions in HMMER.

3) allows flexibility in deciding between global and local alignments by adjusting the parameters of self-transitions involving nodes $N$ (at the beginning), $C$ (at the end), and $J$ (in between). These self-transitions model the unaligned (or "ignored") part of the sequences. The set of states with their abbreviations are as follows:

| | |
|---|---|
| $M_x$ | Match state $x$, emitter. |
| $D_x$ | Delete state $x$, non-emitter. |
| $I_x$ | Insert state $x$, emitter. |
| $S$ | Start state, non-emitter. |
| $T$ | Terminal State, non-emitter. |
| $N$ | N-terminal unaligned sequence state in the beginning of a sequence, emitter. |
| $B$ | Begin state (for entering main model), non-emitter. |
| $E$ | End state (for exiting main model), non-emitter. |
| $C$ | C-terminal unaligned sequence state at the end of a sequence, emitter. |
| $J$ | Joining segment unaligned sequence state, emitter |

If the loop probability modeling the transition between $N \rightarrow N$ is set to 0, all alignments are constrained to start at the beginning of the model. If the probability of transition from $C \rightarrow C$ is set to 0, all alignments are constrained to end at the last node of the model. Setting $E \rightarrow J$ to 0 forces a global alignment. If it is not set to 0, the model can start at any point in a larger sequence and end some distance away for effecting local alignments. This option can be used for the sequence annotation task mentioned before by aligning the model locally against a large sequence. Furthermore, the transition $J \rightarrow J$ can be used to control the gap between local alignments. One can do the reverse, i.e., globally aligning a smaller sequence to a part of the model, by controlling the transitions between $B \rightarrow M$ and $M \rightarrow E$. HMMER is an extremely versatile and powerful sequence alignment tool. It can thus be very useful in locating sequences of opcodes from traces.

To learn the parameters of the model, it may be useful to use a small set of multiply aligned sequences. We have used an open source implementation of multiple alignment provided in [9] for this purpose.

### 3.5 Workload Identification Workflow: An Overview

In this section, we give an overview of our methodology using profile HMMs. Figure 4 gives the workflow for building a profile HMM model of a given workload. We need to supply one or more opcode sequences corresponding to traces of different runs of an application workload. These opcode sequences need to be encoded into a limited-sized alphabet that the HMM model works with. This is done by the alphabetizer module. The encoded sequences pass through a multiple alignment module (explained in Section 3.4.1), which creates a canonical aligned sequence for training. We use an open-source tool called Muscle [9] for this purpose. We then use HMMER [7] to generate a profile HMM model of the workload based on the aligned sequences.

To annotate the occurrences of a set of trained workloads in an arbitrary NFS trace, we extract the NFS opcode sequence from the trace, alphabetize it and pass it to the HMMER's pattern search tool called *hmmpfam* along with the profile HMM models of the workloads that we want to identify within the trace. The tool outputs the indices of the subsequences that it matched with various workloads along with a fractional score (in the range 0 to 1) indicating its confidence in the match relative to other workloads. We have written a script to post-process this output to produce the final annotation of the test sequence. The post-processing phase involves the following steps:

1. Merge two contiguous matches of the same workload.
2. Remove the matching subsequence with very low score (less than 0.1 percent of the average score for the matching subsequences of the same workload).
3. Again, merge any two new contiguous matching subsequences of the same workload.
4. If more than two workloads are reported for the same region, report the workload with a higher score.

## 4 Evaluation

In this section, we illustrate the capabilities of our profile HMM based methodology including its ability to identify and mark out the positions of high-level operations in an unknown network file system trace as well as its ability to isolate multiple workloads running concurrently. We also evaluate the training and pattern recognition performance of the methodology via micro-benchmarks.

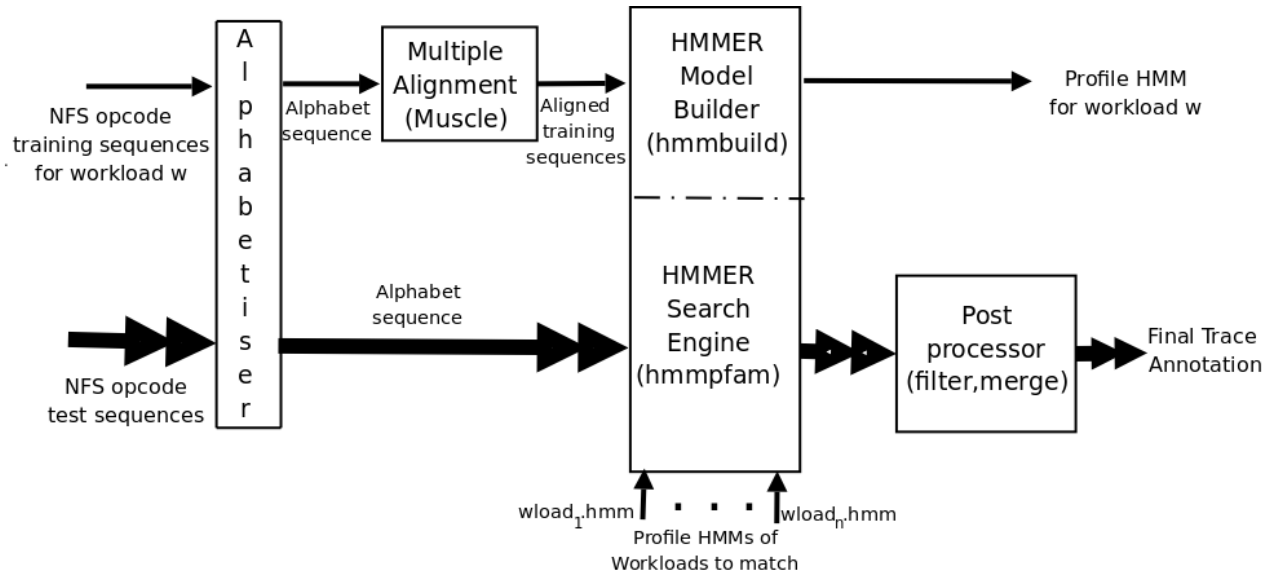
NFS opcode training sequences for workload w
Alphabetiser
Alphabet sequence
Multiple Alignment (Muscle)
Aligned training sequences
HMMER Model Builder (hmmbuild)
Profile HMM for workload w
NFS opcode test sequences
Alphabet sequence
HMMER Search Engine (hmmpfam)
Post processor (filter,merge)
Final Trace Annotation
wload$_1$.hmm
wload$_n$.hmm
Profile HMMs of Workloads to match

**Figure 4.** Profile HMM Training and usage workflow. Given a set of opcode traces of a given workload *w* with various parameters, this workflow produces a profile HMM model in the file *w.hmm*. Muscle and HMMER are existing open source tools, whereas the alphabetizer and post processor are modules that we developed. The bottom flow represents trace identification, where we input the workload models developed by the training workflow above into the HMMER search engine.

## 4.1 Experimental Setup and Training Method

For our evaluation, we choose several popular UNIX commands and user operations on files and directories as our application workloads: *tar, untar, make, edit, copy, move, grep, find, compile*. The UNIX commands access subsets of 14361 files and 1529 directories up to 7 levels deep stored on a Linux NFSv3 server from one or more Linux NFSv3 clients. For a more realistic evaluation, we also incorporated TPC-C [22] workloads. TPC-C is an OLTP benchmark portraying the activities of a wholesale supplier, where a population of terminal operators executes transactions against a warehouse database. Our TPC-C configuration used 1 to 5 warehouses with 1 to 5 database clients per warehouse. The database had 100,000 items.

The NFS clients are located on the same 1 Gbps LAN with NFS client-side caching enabled. The caching effects across multiple experiments were eliminated by mounting and unmounting the file system between each experiment. We capture the NFS packet trace at the NFS server machine's network interface using the Wireshark tool [5], and filter out the data portion of the NFS operations. For all experiments in this paper, we only use the opcode information in the NFS trace. Hence, we use the term *trace* in the rest of this section to refer only to the opcode sequences.

We build profile HMMs for each of the UNIX commands as follows. First, we run the UNIX command many times with different parameters and capture their traces. The number of captured traces for each command along with their average length in opcodes, is shown in Table 3. Next, we build the profile HMM for the command with increasing numbers of randomly selected traces as outlined in Figure 4, each time cross-validating its recognition quality by testing with the remaining traces. We stop when the improvement in the model quality metric diminishes below a threshold. We found that ten traces of each command were sufficient. We call those sequences as our *training sequences*, and the rest as *test sequences*.

## 4.2 Workload Identification

Our first experiment evaluates how well profile HMM can identify pure application-level workloads based on past training. We feed the test sequences to the trained profile HMM for identification. Table 3 shows the results in the form of a "confusion" matrix. Each row of the matrix indicates a test command and each column under the "models" umbrella indicates a command for which profile HMM got trained. Each cell indicates how well the profile HMM labeled the sequence as the given command, the ideal being 100%. Commands were recognized correctly much of the time with a few exceptions.

For instance, about 9% of the *copy* workloads are mislabeled as *edit* workloads. These were primarily single file copies and they share similarities with *edit* workloads that we trained with; they both exhibit an even mix of reads and writes. Copies of multiple files or recursive copies were not confused with *edit* workloads. The results also show that 11.3% of *grep* workloads are getting mis-labeled as *tar* workloads. Upon close inspection, we discovered that many

| Trace | Models | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Command | make | find | grep | tar | untar | copy | move | edit | tpcc |
| **make** | **91.7** | 1.2 | | 1.2 | 2.4 | 3.6 | | | |
| **find** | | **91.8** | 2.1 | | | 3.1 | 1 | | 2.1 |
| **grep** | 1 | | **72** | 22 | 5 | | | | |
| **tar** | | | | **100** | | | | | |
| **untar** | | | | 1.2 | **98.8** | | | | |
| **copy** | | 1 | 1 | | 6 | **82** | 1 | 9 | |
| **move** | | 5.6 | 0.8 | 0.8 | | 2.4 | **89.6** | 0.8 | |
| **edit** | | | | | | | | **100** | |
| **tpcc** | | | | | | | | | **100** |

**Table 3.** Recognizing a single workload using the profile HMM on a test opcode sequence. Confusion matrix gives entries indicating the percentage of instances recognized correctly; the rows add up to 100%. The profile HMM recognized most commands correctly.

of the single-file *grep* commands ("grep foo bar.c") were being identified as *tar*'s. The combined multiple alignment model shows that the initial subsequence of *tar*, where a single file is being read from beginning to end, is very much like that of a single-file *grep*. That could have led to the profile HMM making an error. The diversity of the training set is critical. For instance, when we manually picked the *grep* training traces to have diverse command traces, we could improve the accuracy from 72% to 85%.

Consider another example: *find* and *tar* need to traverse a directory hierarchy in its entirety, except that in our case, *tar* additionally reads the file contents and writes the tar file. This distinction was enough for profile HMM to successfully distinguish *find* from *tar* in 100% of the cases. Overall, our methodology is able to distinguish workloads well based on small differences in their trace patterns.

An interesting result here is that the *tpcc* workload was identified correctly 100% of the time. The intuition behind this result is that, a complex workload contains unique patterns in its traces that can be accurately recognized. A simple workload may not have a strong signature in its traces, leading the profile HMM to mis-identify it occasionally.

**Discrimination between TPC-C and Postmark:** We also wanted to see how two large applications can be accurately distinguished using the NFS traces; we selected TPC-C and Postmark for this experiment. Postmark [15] is a synthetic benchmark that has been designed to create a large pool of continually changing files and measure the transaction rates for a workload approximating a large Internet electronic mail server.

Postmark traces were generated by running the benchmark 60 times with varying parameters. The file sizes were varied between 10000 bytes and 300000 bytes, the fraction of creations vs. deletions was varied between 10% and 100%, and the fraction of reads vs. appends was varied between 10% and 100%. Out of this set of traces, 10 were randomly picked for training, and 50 traces for testing. Similarly, 20 traces of previously unknown TPC-C workload were attempted after training with 4 traces. The TPC-C traces were from the previous experiment. The results of the workload identification are given in Table 4.

| | TPC-C | Postmark |
|---|---|---|
| TPC-C | 100% | 0% |
| Postmark | 0% | 100% |

**Table 4.** Workload identification accuracy with TPC-C and Postmark loads.

In both cases, there were no misclassifications. This experiment shows the capability of profile HMMs in discriminating between two complex and large workloads.

## 4.3 Trace Annotation

Our next experiment evaluates how profile HMM can mark out the NFS operations constituting various commands in a long but not earlier seen NFS packet trace. It tells us how accurately it can detect the start and end of commands just by observing the NFS operations. We run sequences of commands to simulate a variety of common user-level activities, collect their NFS opcode traces and query the profile HMM to identify the commands and their positions in each trace, as outlined in Figure 4. We then compare them with the known correct positions. Profile HMM is able to detect the boundaries of a command's opcode sequence to within a few opcodes in many cases.

Figure 5 shows the trace annotation diagram with both the detected and actual command boundaries for a command sequence <*untar;make;edit;make;tar*> that attempts to simulate the process of downloading the HMMER source package, compiling it, modifying it, compiling it again, and then *tar*'ing up the resulting package. The bottom-most bar in the figure shows the actual command boundaries, while the other bars show the annotation made by the profile HMM. We see that the quality of annotation is high. The NFS operations corresponding to the *untar*, the two *make*'s

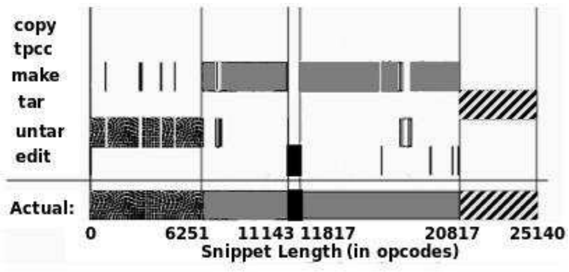

**Figure 5.** Visualization of the annotated trace for a sequence of user commands: <*untar*; *make*; *edit*;*make*; *tar*>. The bottom-most bar in the figure shows the actual sequence in the trace, while the other bars above show the annotation by the profile HMM. The vertical lines indicate workload transition boundaries. The visualizations in this figure show that the annotation is reasonably accurate. *make* is a harder command to classify because it invokes other commands.

and *tar* commands are accurately marked.

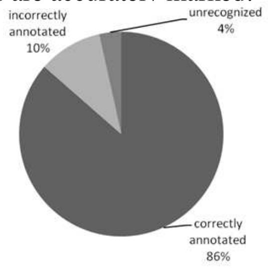

**Figure 6.** Overall Trace Annotation Accuracy for a random sequence of UNIX commands.

We then ran a comprehensive experiment, so that our results can be more statistically significant. We generated 100 traces, where each trace contained a run from a sequence of 100 commands, each picked randomly from our available pool of commands. We analyzed the traces using profile HMM, and annotated each opcode with its identified command. The results are presented in Figure 6. The annotation accuracy is a measure of how much of the trace is marked correctly with respect to start and end of the traces (and unrelated to confusion matrix entries computed for workload identification). 86% of the opcodes were annotated correctly; 10% of them were marked as belonging to a wrong command; and, 4% were identified as not belonging to any of our commands. Figure 7 shows the results broken down on a per-workload basis. Here we notice that opcodes belonging to *grep* and *move* were often incorrectly annotated. Both these workloads perform poorly in the sampling experiments above as well, implying that their characteristic patterns are not very unique.

In summary, profile HMMs are able to make use of subtle differences in workload traces to accurately identify transitions among workloads and annotate opcodes with the higher-level operations that they represent. The minor discrepancies observed were likely caused by not having

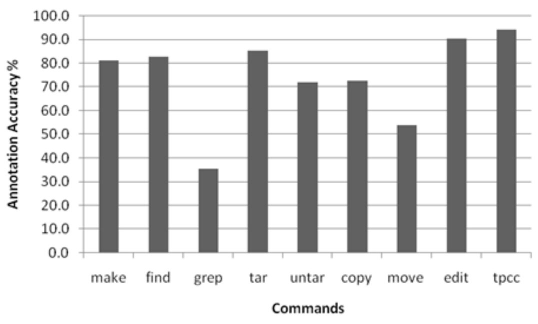

**Figure 7.** Trace Annotation Accuracy on a per-command basis. Note that it is lower than that for identification as the starting and ending of the traces have also to be marked correctly.

enough diversity in the selected training traces. Note that for single workload identification described in 4.2, manually picking the *grep* training traces to have diverse command traces resulted in accuracy improvement from 72% to 85%. Further work is needed to figure out how to select traces for improved discrimination.

## 4.4 Trace Processing Rate

Next, we measure the rate at which the profile HMMs can process (identify or annotate) a trace by applying it on a trace of length 50000 opcodes. Such a trace is constructed randomly using traces in our test sequence set. For identification, each model in turn reports how many instances of its family are present in the whole trace as well as a score that indicates how well it matches with its training set. For annotation, each model marks out its portion in the trace and a post-processing procedure decides which workload is assigned to a segment of the trace (based on a score).

Profile HMMs are not particularly fast – they processed the trace at a rate of 356 opcodes per second on a Intel Quad-Core CPU at 2.66 GHz and 3 GB of memory running Ubuntu Linux, kernel version 2.6.28. We then isolated each model and measured their performance individually on the same trace. The results are shown in the "processing rate" column of Table 5. We find that the models differ markedly in their speed (*make* and *tpcc* being the slowest). We see a strong inverse correlation between the speed of the model and the maximum sequence length of the training traces. This is understandable: shorter training sequences will likely build a profile HMM with fewer states and transitions. One could speed up the models by choosing shorter traces for training, provided they do not jeopardize the identification accuracy. This is a tradeoff worth exploring in the future.

| Trace Command | # Test Traces | Trace Length min. | Trace Length mean | Trace Length max | Processing rate (opcodes/sec) |
|---|---|---|---|---|---|
| **make** | 84 | 23 | 2653 | 32175 | 2971 |
| **find** | 98 | 33 | 10683 | 66093 | 135893 |
| **grep** | 100 | 19 | 4784 | 24024 | 121701 |
| **tar** | 98 | 67 | 1255 | 19578 | 49430 |
| **untar** | 81 | 85 | 2082 | 28013 | 24680 |
| **copy** | 100 | 35 | 8665 | 97789 | 21408 |
| **move** | 125 | 9 | 26 | 39 | 667714 |
| **edit** | 127 | 657 | 670 | 687 | 22177 |
| **tpcc** | 24 | 1289 | 12665 | 61430 | 565 |

**Table 5.** Trace processing rates. Since each model has different number of states in its profile HMM, the processing rates differ.

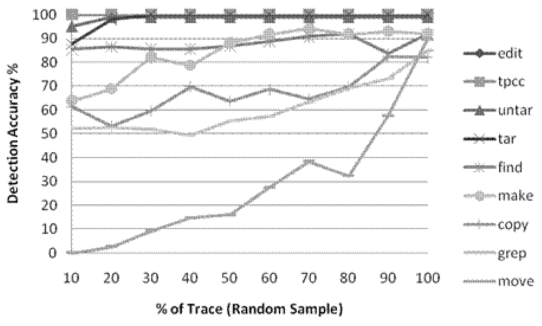

**Figure 8.** Sensitivity of profile HMM to the length of the trace sample analyzed for various commands when sample picked randomly from the whole trace. Y-axis indicates the percent of runs (out of hundred runs) where the command was correctly recognized.

## 4.5 Identification of Randomly Sampled Partial Traces

In a real system, we will not have the entire trace of a single command or a neatly ordered sequential set of commands to analyze. They will typically be interleaved because of concurrent execution. Therefore, we must be able to detect an application operation just by observing a snippet of a command's trace. Further, for online behavior detection and adaptation, we should be able to quickly detect an application operation, which implies that we should need to analyze small amounts of traces to identify workloads.

Our next experiment evaluates how much of a randomly sampled NFS trace the profile HMM methodology needs to be able to correctly recognize a high-level operation. For this experiment, we feed the profile HMM with contiguous substrings of the pure test sequences — of various lengths and at random locations in the full sequence — and measure how often it detects the command correctly. Figure 8 contains plots of profile HMM's sensitivity to trace snippet size for various high-level commands. As the graphs indicate, profile HMM is able to recognize most workloads with

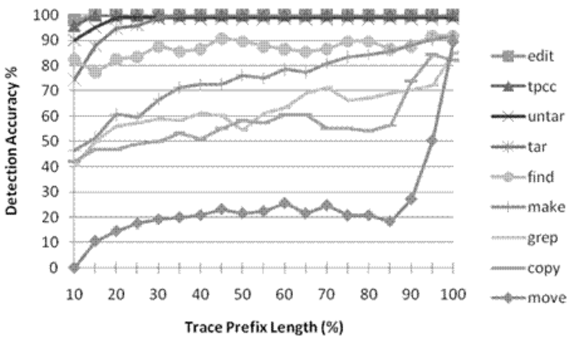

**Figure 9.** Sensitivity of profile HMM's accuracy to the length of the trace prefix analyzed for various commands. The Y-axis indicates the percent of runs (out of hundred runs) where the command was correctly recognized.

80% accuracy by examining a small fraction of the trace. The *move* command generates a small trace to begin with. Therefore, the profile HMM requires a large fraction of its trace to be examined to correctly identify it.

The characteristic patterns of a workload may be concentrated at some locations for certain commands, while they may be distributed better for other commands. Having characteristic patterns at various locations in the trace is useful for online behavior detection, since there is a larger likelihood of identifying a workload from a random sample. To understand the distribution of characteristic patterns in our workloads, we tested the profile HMM with varying length prefixes of traces. Figure 9 shows the results. We see that the predictive value of small prefixes of traces is quite high. For some commands like *copy* and *move*, the end of a trace seems to have strong characteristics.

This evaluation suggests that in real scenarios, some workloads may be identified by examining just a small snippet, while other workloads may need a large fraction of their traces to be analyzed before identification.

## 4.6 Automated Learning on Real Traces

Validating our approach using real traces from real deployments is important. Our approach is based on a classification-based methodology that requires that the training data be labeled. Unfortunately, real traces are typically not labeled with workload information. Therefore, we will neither be able to train with the real trace nor be able to validate our results.

To tackle this problem, we use the LD_PRELOAD environment variable on the client to interpose our own library that intercepts all process invocations ("exec" family of calls in UNIX) and forces a sentinel marker in the trace by doing an operation that can be spotted. Whenever we see an "exec", we "stat" a non-existent file – the file name encodes the identity of the exec'ed program. The NFS re-

|  | gcc | cat | mv | ld |
|---|---|---|---|---|
| gcc | 80.5 | 1.9 | 0.9 | 16.8 |
| cat | 3.1 | 77.9 | 0.8 | 18.2 |
| mv | 0.6 | 0.5 | 62.5 | 36.4 |
| ld | 13.3 | 1.2 | 1.7 | 83.8 |

**Table 6.** Workload identification accuracy on live traces.

sponse that the file does not exist (ENOENT) with the coded filename is enough for us to mark the boundaries of the trace segment generated by each of the command invocations. Here we need to ensure that the invocation is "atomic", i.e., it does not result in exec'ing of other programs that are of interest independently for identification (otherwise, we will mark a only a subtrace as belonging to the invocation and mark some part of the following trace as belonging to the subprocess). We used an open-source tool called *Snoopy* [21] and modified it to suit our purposes.

As an example, we used the compilation of Linux 2.6.30 source as the generator of a real trace. We instrumented the client with the above interposition library, collected the traces for a certain amount of time and constructed our training trace data automatically. Our sentinel markers in the trace also give us an easy way to validate our results.

The following commands were detected in the Linux source compilation on the Ubuntu 9 system[1]: "gcc", "rm", "cat", "mv", "expr", "make", "getent", "cut", "mkdir", "bash", "run-parts", "sed", "date", "whoami", "hostname", "dnsdomainname", "tail", "grep", "cmp", "sudo", "objdump", "ld", "nm", "objcopy", "awk", "update-motd", "renice", "ionice", "basename", "landscape-sysinfo", "who", "stat", "apt-config", "ls". Since commands like "make" initiate, for example, many gcc compiles, it is not possible to demarcate the beginning and end of the trace that "make" contributes as we are interested in "gcc" as a workload in itself. We eliminated such composite commands and those that do not contribute to NFS traces (eg. "date"), and ended finally by selecting 4 commands in the live trace.

For workload identification, we considered the 105 minute live trace of the Linux source compilation discussed earlier with training on approximately 3 minutes of the trace. The results are given in Table 6.

To understand how learning is improved with larger number of training traces used, we chose 30 sec, 40 sec, 50 sec, 1 min, 2 min, 3 min, 4 min and 5 min durations of the trace and used the specific workload found in these durations for training that workload. From Figure 10, we notice that the accuracy of the workload identification improves with increase in the number of training sequences used, thus demonstrating learning in the system. Commands that generate a small amount of traces, such as *cat* and *mv* pose difficulties for our methodology. In this experiment, the output of the *cat* commands were for */dev/null* and for a single specific file; because of client-side caching, the traces did not have a strong signature. We need traces with good signatures (like *gcc*) to get good results. This is acceptable from a practical standpoint as bigger application workloads, in general, are of more interest in the systems community.

[1]"landscape-sysinfo" provides a quick summary about the machine's status regarding disk space, memory, processes, etc. "run-parts" runs a number of scripts or programs found in a single directory.

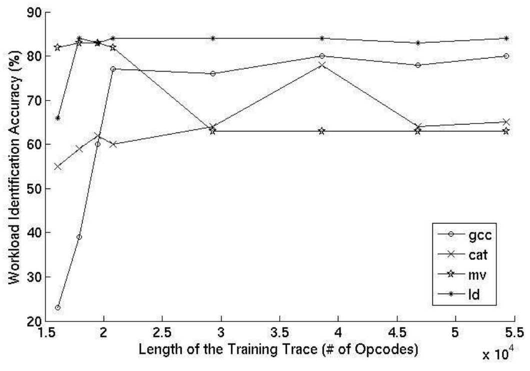

**Figure 10.** Online learning on live traces.

The value of the profile HMM as a practical tool will be significantly enhanced if we can automatically generate a labeled trace, with each of its constituent workloads demarcated, for training. The LD_PRELOAD mechanism is a way to do this. On new clients or clients running new applications, the interposition library could be introduced to generate new training sets. The library could subsequently be removed after sufficient training data has been generated.

## 4.7 Concurrent Workloads

Shared storage systems almost always serve multiple concurrent workloads. Therefore, the server-side trace contains the trace sequences of multiple application-level operations interleaved with each other in time. However, while a shared storage system may serve files to thousands of clients in an enterprise deployment, the NFS trace contains client IDs that can be used to tease the interleaving apart. Therefore, we need automated tools only to separate out the traces due to requests from a single client. Typically, the number of concurrent applications at a single client invoking NFS operations to the same backend server are small.

Profile HMM's ability to detect high-level commands from small snippets of file system operations helps identify the various workloads running concurrently. Our next experiment evaluates this ability. We run sequences of commands from 2 to 6 NFS clients accessing the same NFS server, capture the NFS opcode trace at the server's network interface, remove the client ID (to simulate the effect of multiple applications from the same client), and feed it into the profile HMM for marking the commands' opera-

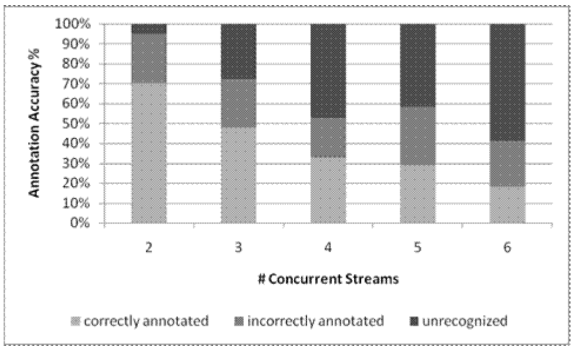

**Figure 11.** Concurrent sequences of commands were run from 2 to 6 clients. The graph shows the quality of the annotation.

tion sequences. We compare the result with the sequences identified manually based on the source IP address. Figure 11 shows the quality of the annotation. The amount of concurrency determines whether there will be long enough snippets for profile HMM to accurately annotate the trace. As expected, for a concurrency level of 2 or 3, the results are acceptable, but gets worse beyond that. The interesting point to note here is that the incorrect annotations do not increase with concurrency; only the proportion of unrecognized sequences do. The profile HMM's ability to explicitly tag unrecognized sequences as such helps the user rely on its output.

More than the exact marking of regions, the identification of constituent workloads in a mixed-workload scenario is itself of good value. This is because, for the typical administrator, a more compelling use case than unraveling the opcode sequences of interleaving workloads is to identify which workloads are running in a given interval of time. Note that TPC-C, a very concurrent workload, can be identified quite successfully as reported earlier (Sections 4.2, 4.3).

## 5 Limitations

During the course of our evaluation, we discovered a few limitations with this methodology. First, training the tool requires a diverse and representative sample of workloads. This is a fundamental characteristic of machine learning methodologies. Second, the open-source tools that we used to build our solution are from computational biology. The current off-the-shelf solutions have a limited alphabet space which may not be completely appropriate for systems applications. However, we believe that there are no fundamental mathematical limitations in the number of symbols, except that we may have to perform significantly more training if we use more symbols. Third, the level of concurrency at a client adversely affected the accuracy of the tool. The fine-grained interleaving resulting from a large number of concurrent streams can be tackled only if we are able to identify workloads using very small trace snippets. Finally, the profile HMM seems to be slow compared with the typical rates of NFS operations at a server, hampering online analysis. Many of these limitations may not be fundamental in nature, but pointers to future work.

## 6 Conclusions and Future Work

In this paper, we have presented a profile HMM-based methodology for analysis of NFS traces. Our method is successful at discovering the application-level behavioral characteristics from NFS traces. We have also shown that given a long sequence of NFS trace headers, it is able to annotate regions of the sequence as belonging to the applications that it has been trained with. It can identify and annotate both sequential and concurrent execution of different workloads. Finally, we demonstrate that small snippets of traces are sufficient for identifying many workloads. This result has important consequences. Because traces are going to get generated faster than one can analyze them, being able to infer meaningful information from periodic random sampling is very important for effective analysis.

Although profile HMM methodology looks promising for trace analysis, our experience indicates that we have not leveraged all its capabilities. For instance, we have not used all the information that is available in the NFS trace. There is a rich amount of data available in the form of file names and handles, file offsets, read/write lengths and error responses that throw more light on the application workloads. We have to investigate how to incorporate this information into a form amenable for multiple alignment and profile HMM. This will be the first step in extending our work.

NFSv4 introduces client delegations, offering clients the ability to access and modify a file in its own cache without talking to the server. This implies that an NFSv4 trace may not have all the information about application workloads. Investigating how profile HMMs work on NFSv4 traces is a clear extension of this work.

We also believe that our methodology is general enough that we can apply it to other source data such as network messages, system call traces, disk traces and function call graphs. This methodology can be a foundation to tackle use cases in areas such as anomaly detection and provenance mining, which are building blocks for next-generation systems management tools. Finally, we will look into other machine learning methods that overcome some of the limitations of profile HMMs.

**Acknowledgments:** We thank Bhupender Singh, Alex Nelson, and Darrell Long for reviewing the paper, Pavan Kumar for performing the PostMark experiments, and Alma Riska for shepherding the paper with thoughtful comments and guidance. We also gratefully acknowledge support from a NetApp[2] research grant.

## References

[1] Eric Anderson. Capture, conversion, and analysis of an intense NFS workload. In *Proccedings of the 7th conference on File and storage technologies,* pages 139-152, Feb. 2009

[2] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th Symposium on Operating Systems Principles*, Oct. 1991.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proc of the Seventh Symposium on Operating System Design and Implementation*, pages 259–272, Dec. 2004.

[4] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. Internet Request For Comments RFC 1813, Internet Network Working Group, June 1995.

[5] G. Combs. Wireshark network protocol analyzer. `http://www.wireshark.org`, 1998.

[6] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic models of proteins and nucleic acids*. Cambridge University Press, 1998.

[7] S. R. Eddy. HMMER: Sequence analysis using profile hidden Markov models. Available at `http://hmmer.wustl.edu/`.

[8] S. R. Eddy. Profile hidden Markov models. *Bioinformatics*, 14(9):755–763, 1998.

[9] R. C. Edgar. MUSCLE:multiple sequence alignment with high accuracy and high throughput. *Nucleic Acid Research*, 32(5):1792–1797, 2004.

[10] D. Ellard. *Trace-based analyses and optimizations for network storage servers*. PhD thesis, Cambridge, MA, USA, 2004. Adviser-Margo I. Seltzer.

[11] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST03)*, pages 203–216, 2003.

[12] D. Ellard and M. Seltzer. New NFS tracing tools and techniques for system analysis. In *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA)*, Oct. 2003.

[13] D. Gusfield. *Algorithms on Strings, Trees and Sequence*. Cambridge University Press, 1997.

[14] D. Haussler, A. Krogh, I. S. Mian, and K. Sjölander. Protein modeling using hidden markov models: analysis of globins. In *Proceedings of the 26th Annual Hawaii International Conference on Systems Sciences*, volume 1, pages 792–802. IEEE Computer Society, 1993.

[15] J. Katcher. Postmark: A new file system benchmark. Technical Report 3022, NetApp, 1997.

[16] A. Krogh, M. Brown, I. S. Mian, Sjölander, and D. Haussler. Hidden Markov models in computational biology: Applications to protein modeling. 235:1501–1531, 1994.

[17] A. Leung, S. Pasupathy, G. Goodson, and E. Miller. Measurement and analysis of large-scale file system workloads. In *Proceedings of the USENIX 2008 Annual Technical Conference*, June 2008.

[18] T. Madhyastha and D. Reed. Input/output access pattern classification using hidden markov models. In *Workshop on Input/Output in Parallel and Distributed Systems*, Nov. 1997.

[19] M. Mesnier, E. Thereska, G. Ganger, D. Ellard, and M. Seltzer. File classification in self-* storage systems. In *Proceedings of the First International Conference on Autonomic Computing*, May 2004.

[20] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[21] D. Packages. Snoopy. `http://http://packages.debian.org/lenny/snoopy`.

[22] F. Raab, W. Kohler, and A. Shah. Overview of the TPC benchmark C: The order-entry benchmark. `http://www.tpc.org/tpcc/detail.asp`.

[23] L. R. Rabiner. Tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of IEEE*, 77(2):257–288, 1989.

[24] D. Roselli, J. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of the USENIX 2000 Annual Technical Conference*, 2000.

[25] R. R. Sambasivan, A. X. Zheng, E. Thereska, and G. Ganger. Categorizing and differencing system behaviours. In *Hot Topics in Autonomic Computing*, June 2007.

[26] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:197–198, 1981.

[27] N. Tran and D. Reed. Automatic ARIMA time-series modeling for adaptive i/o prefetching. *IEEE Transactions on Parallel and Distributed Systems*, 15(4):362–377, Apr. 2004.

[28] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of ACM*, 21(1):168–173, 1974.

[29] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May 1999.

[2]NetApp, the NetApp logo, and Go further, faster, are trademarks or registered trademarks of NetApp, Inc. in the United States and/or other countries.