

Workload Modeling of Stateful Protocols Using HMMs

Richard Honicky
University of California, Berkeley, CA
(honicky@cs.berkeley.edu)

Swami Ramany, Darren Sawyer
Network Appliance Inc., Sunnyvale, CA
(swamir@netapp.com, dsawyer@netapp.com)

There are several techniques for generating a workload consisting of a series of operations presented to a target device. Each has advantages and drawbacks when evaluated in terms of accuracy with respect to the real workload being modeled and computational or storage resources required to generate it. For workloads corresponding to “stateful” protocols, some techniques cannot be applied, as they fail to capture important ordering relationships between operations or fail to avoid generating an illegal sequence of operations as dictated by the protocol. This paper describes a new workload modeling and generation technique using Hidden Markov Models. The technique provides an accurate model of a workload while minimizing the resources required during simulated workload generation as well as conforming to stateful protocol sequencing restrictions. The application of the technique as applied to modeling CIFS network filesystem protocol workloads is presented, along with preliminary experimental results validating its effectiveness.

1 Introduction

In an attempt to create a workload generator that would produce a realistic stream of transactions over the Common Internet File System (CIFS) file system protocol, we were faced with a number of challenges. Simple random generation of the various types of transactions, using a distribution based on observations from traces collected from real systems, was insufficient. Aside from the fact that the sequence of operations may play a significant role in the performance observed on the target system, the protocol itself requires that operations maintain a certain set of sequencing relationships between each other. Any illegally ordered transactions would violate the protocol and result in errors from the targeted server. Alternatively, having the generator simply replay the collected traces had a different set of issues. While the proper sequencing of transactions was ensured with this method, the resource consumption on the workload generator, in terms of memory, storage, and compute resources, was unacceptably high. Furthermore, it was not obvious how to deliver a “composite” workload, which had the characteristics of a large number of traces taken from various

machines, a key design point for our generator.

Rather than accept the limitations of the methods we evaluated and compromise on our design point, we developed a new workload generation technique based on Hidden Markov Models (HMMs). After providing background information on workload generation techniques, the CIFS protocol, and HMMs, this paper will present a generally applicable technique for generating a workload consisting of transactions in a “stateful” protocol (such as CIFS). Experimental results based on an implementation of such a generator will be provided to show how the technique is effective in modeling a workload collected in a trace, and how sensitivity to the order in which transactions are presented to a server can affect its performance.

2 Background

This section will provide background on methods for generating synthetic workloads and provide motivation for a technique that is both accurate and efficient. It will also provide an introduction to Hidden Markov Models, the CIFS protocol, a stateful

	Trace Driven Generation	Random Op Generation	Hybrid Trace-Random Generation	HMM-Based Op Generation
Example Benchmark	Various	SPEC SFS	NetBench	None
Captures Protocol Correctness?	Yes	No	Yes	Yes
Captures Stateful Behavior	Yes	No	Yes	Yes
Storage Requirement	Very Large - trace file size in GBs	Negligible	Tradeoff with accuracy	Negligible
Pre-processing Computational Complexity	None	Very small - the right operation mix should be retrieved from the traces	None	$O(n^2)$ where is n is the number of states required in the HMM
Pre-processing Memory Requirement	None	None	None	$O(n^2)$ where is n is the number of states required in the HMM
Runtime Computational Complexity	Trace file needs to be actively read by clients - limits scalability	Very low - simple random number generation	Moderate - simple random generation with trace replay overhead	Low - multiple random number generation
Runtime Memory Requirement	High	None	Moderate	None

Table 1: Comparison of selected workload generation techniques

network filesystem protocol to which the modeling techniques presented later in the paper will be applied.

2.1 Workload Modeling and Synthetic Generation Techniques

The most desirable method to generate a workload against a target device is to capture one or more real-life traces from the environment of interest and support these within a trace replay tool. While this approach actually preserves the exact spatial and temporal ordering of various operations it has the handicap of requiring large amounts of storage and also places a high runtime CPU and memory requirement on the clients that replay the trace. To overcome this problem many benchmarks (e.g., SPEC SFS) actually derive various statistics from real life workload traces and then randomly generate various operations to conform to the derived statistics. However such an approach will not work for modeling stateful protocols which require certain order of operations for protocol correctness. One easy way to solve this problem is to have a hybrid approach that preserves the correct order of operations (based on observed traces) and at the same time randomly chooses between various clusters of operations to generate a workload that conforms to the right overall statistic. Even this approach has a clear accuracy versus required

storage tradeoff. Additionally this approach still requires significant runtime resource on the client side. Table 1 compares these techniques to a HMM-based workload generation method that is discussed in detail in the next few sub sections.

2.2 Hidden Markov Models

One common tool for analyzing streams of discrete values that depend on an underlying stateful process is a Hidden Markov Model (HMM) [AIMA, Speech89]. HMMs are a well understood way of modeling a stateful process which is partly or fully “hidden”.

Like other probabilistic graphical models, HMMs map the conditional independence structure of a joint probability distribution to a graph. With that mapping, we can then utilize graph theory to implement efficient probabilistic inference algorithms which exploit the conditional dependencies in the problem. As the name suggests, one assumption implicit and central to HMMs is the Markov property: the past is independent of the future, given the present. Less formally, this means that if we know the current state of the model, knowing the past states will give us no more information about what our future state will be.

This characterization can be slightly confusing,

however, because we can model the current state to encode information about the past. Consider Figure 1, where we see a simple HMM in which state transitions can lead us down two paths. If we find ourselves in one of the top states, then we know that in the past we made the choice to take the top path. This does not violate the Markov property. In fact, it is implicit in the Markov property, because information about the past is contained in the state itself. The Markov property simply states that knowing the past cannot give us *more* information about the future than is already encoded in the current state.

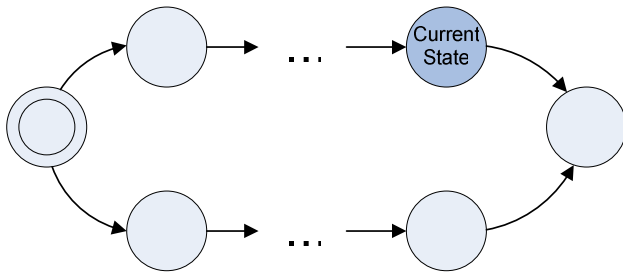


Figure 1: A HMM with two possible paths

One other important property of HMMs, mentioned briefly above, is that the true state of the model is generally hidden. In other words, even if we know the right model, the best we can do is make probabilistic inferences on what the current state is, based on what sequence of symbols we have seen. Thus, HMMs can model a process which is obscured by some (arbitrarily complex) randomization function, and which eventually outputs a sequence of symbols from the range of the function. Of course, often times (as is the case in this paper), the true model is also unknown, and we are forced to approximate it.

There are many good introductory texts on HMMs. Stuart Russell and Peter Norvig provide a good overview in their artificial intelligence text [AIMA]. The paper which was used as the basis for our HMM implementation covers one of the most prevalent applications of HMMs: speech recognition [Speech89]. Several good web pages also exist [HMM].

2.3 CIFS and other “Stateful” Protocols

Over the years, protocols like CIFS have also become increasingly important to storage vendors. CIFS is standard remote file system access protocol that enables groups of users to work

together and share documents across the internet or intranets [CIFSRe02]. CIFS is an enhanced version of Microsoft’s Server Message Block (SMB) protocol, the native file sharing protocol in Microsoft windows operations systems. A CIFS file system is different from a traditional Network File System (NFS) file system in many ways. The NFS Version 3 (NFSv3) protocol [NFSv3p95] is stateless and in most cases clients are oblivious to short server reboots. Conversely, CIFS protocol is stateful and it emphasizes locking. Strict locking is required for sustained connection and it is important that active sessions are not interrupted. NFS Version 4 (NFSv4) protocol [NFSv4p03] is in many ways similar to the CIFS protocol in terms of its stateful behavior.

CIFS and NFSv4, however, lack a fair, informative and representative benchmark [Ramany01]. This absence causes problems not only for performance testing and internal evaluation, but also for resource and capacity planning, and performance troubleshooting. Hence it is important to support stateful protocols like CIFS and NFS v4 within industry standard benchmarks.

Supporting these protocols within a benchmark requires accurate capture of the order of operations observed in various traces for both protocol correctness and to emulate the significant impact that statefulness can have on performance of a storage device. For example, an application or kernel module that periodically issues flushes after some number of writes will have a significantly different performance profile than one that does not. Also an application that typically does a series of reads to a small dataset would see a large portion of its reads being cached in processor and system caches due to inherent locality in accesses. A stateful modeling technique is really essential to learn and simulate these kinds of state-dependent behaviors.

The rest of this paper illustrates how HMMs can effectively be used to capture the stateful behavior of CIFS traffic exhibited in a series of traces, and then generate load that both obeys protocol correctness, and also generates realistic sequences of operations with low runtime CPU and storage requirement on the clients.

3 Using HMMs to Model Stateful Workloads

The statefulness of CIFS and NFSv4 protocols means that these protocols require that operations occur in a specific order: opens must precede reads

and writes and should be followed by closes, find_first must precede find_next, etc. This differs from NFSv3, since in NFSv3 operations can come in any order.

In order to mimic the stateful behavior exhibited by a CIFS or NFSv4 client, Hidden Markov Models (HMMs) can be used to learn and simulate the operation sequences found in several traces. This section gives some background on HMMs and how they apply to our workload generator.

An HMM is similar to a Probabilistic Finite Automata [Hopcraft79], except that the emissions of the model are chosen via a distribution of symbols which is associated with a particular state, rather than via a single symbol associated with each edge. HMMs are typically represented using one matrix to describe the HMMs “transition probabilities” (the probability of transitioning to some state, given that we are in some other state), and a second matrix to describe the “emission probabilities” (the probability that a given state will emit a given symbol). There exists a significant body of theory on how to train or “re-estimate” models based on the likelihood (in the statistical sense) that the given model produced the string of symbols used to train it.

The training process is called re-estimation because the most common algorithm for training is a special case of the Expectation Maximization (EM) algorithm, which will only locally maximize the likelihood of the model in the parameter space. This means that a good initial guess at the parameters is important to obtaining a good final model.

Specifying the model too rigidly (i.e. making the initial guess at the parameters too specific), on the other hand, may result in a model that misses important features of the stream of data.

3.1 Trace Analysis

Before discussing the model training process, we will briefly describe the process for collecting and analyzing traces. Analysis software was developed that can utilize traces in any format that ethereal can read. This includes libpcap (i.e. tcpdump), snoop, netmon, and lanalyzer formats, to name a few. Alternatively, text formatted traces can be used, assuming that the trace includes:

- source and destination addresses (appropriately anonymized)
- filename or file ID or search ID (appropriately anonymized)
- user ID

- process ID
- multiplex ID
- packet type
- sub-command

Each trace is run through a script that groups operations on the same file handle (and hence also on the same client) together. For example, if some client reads a file, it might issue an open, a few reads and then a close. If the server is busy, these operations might be separated in the trace by operations from another client. In order to capture the stateful behavior of a client, the streams from each client must be separated, and operations reordered so that operations that occur by the same client between an open and close from that client are also adjacent in the trace. A similar process is applied to search operations.

This approach obscures situations where a client is operating on two different files in parallel. This should not be a problem since the same effect can be achieved (with respect to the workload) by generating a workload in which different processes access different files at the same time.

Once the operations have been separated by file (or search key), they are turned into a stream of symbols, which are then used to train the HMM.

3.2 Model Training

The model training process is designed to capture the hard-and-fast requirements of the CIFS protocol while remaining flexible enough to learn to generate sequences of operations which are similar to those in the traces.

Once a file system connection has been negotiated, there are three types of CIFS operation:

1. Filename operations: these operations can occur at any time.
2. File ID operations: these operations must occur after an open-type operation, and must supply a valid File ID
3. Search ID operations: these operations must occur after a find_first operation, and must supply a valid Search ID.

Because of preprocessing, an additional restriction that only a single file or search is open can be imposed at a given time for a given process. This allows a drastic simplification in the model.

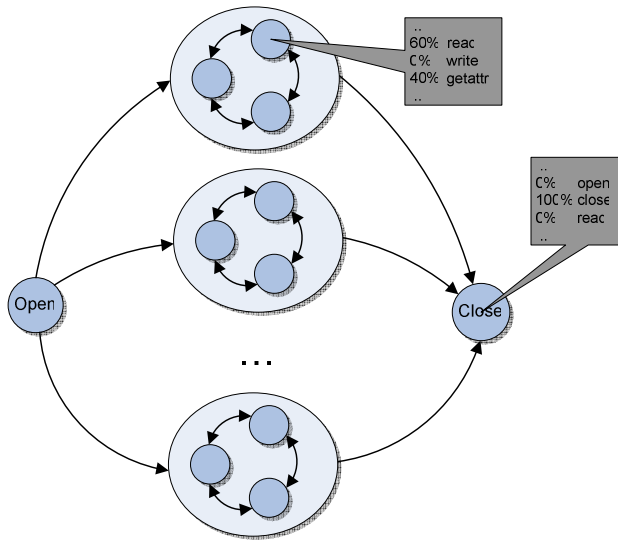


Figure 2: File operations are represented by an open state, several “clusters of operations” states, and a close state

Figure 2 illustrates a part of a HMM which can be used to learn and generate CIFS operations which operate on files. One can see an open state, several clusters of states that are fully connected (cliques) inside the cluster, but are unconnected to states outside the cluster, and finally a transition to a close state. The states inside a cluster are initialized with randomly assigned emission probabilities and transition probabilities. This provides an unbiased framework within which the re-estimation can take place.

Once re-estimation has taken place, each cluster of states, in conjunction with the open and close state, corresponds roughly to a “cluster of operations”.

Now let us consider the rest of the states, including those outside of the open and close. In Figure 3 notice the open and close sections of the HMM at the top. Below it is the find_first and find_next states, and below that is a cloud representing one or more states corresponding to all of the path-type operations. This section of the diagram is represented as a cloud because it may be possible to represent this portion of the diagram as one state, or it may not. This depends on how state dependent these operations are. There is no restriction imposed by CIFS on the ordering of these operations, so the model has been implemented using a single state to emit all of the path-type operations. This technique seems to give reasonably good results, but warrants further exploration.

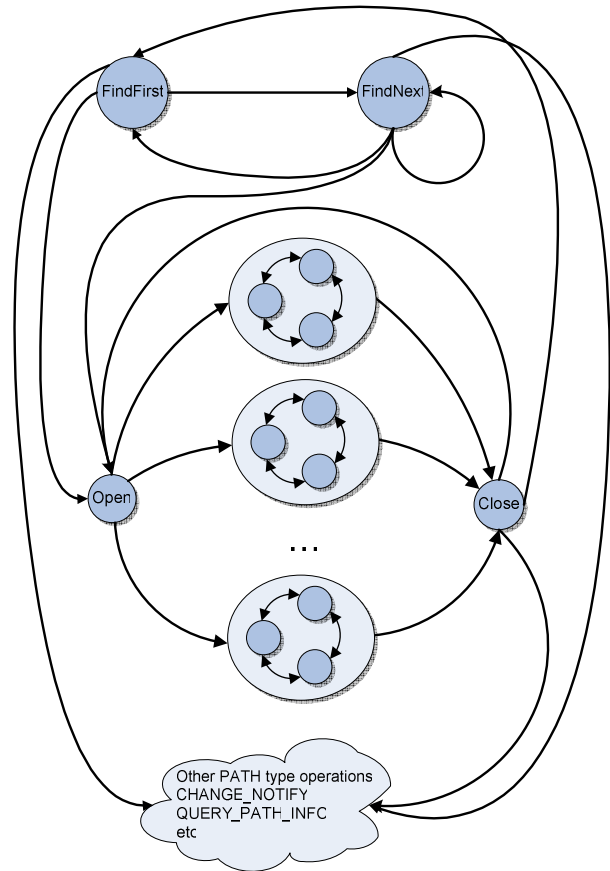


Figure 3: The entire HMM model

We should note that adding states to the HMM can significantly increase the time and memory required to train the HMM. The time and memory requirements of the standard Baum-Welch parameter re-estimation algorithm is $O(n^2)$ in the number of states, and linear in the number of observations. Furthermore, Baum-Welch is an EM algorithm, and hence it is iterative. More complex models (with stronger conditional independence structures) typically require more iterations to converge. Thus, we must trade off between modeling the workload accurately and the computational resources at our disposal.

Finally, once the model has been trained, it can be exported as a series of matrices and loaded by the workload generator. Generating load using the model is fast, simple and easy to understand. The random initialization of the many of the transition probabilities and mixture weights ensures that this process remains flexible enough to adapt to different, unanticipated workloads. The general structure of the HMM guarantees that CIFS operations are generated in a legal order.

3.3 Measuring HMM Accuracy

Ideally, in order to measure the effectiveness of Hidden Markov Models at generating stateful workloads, we would use a comparison function between the distribution of operations generated by the HMM versus the distribution of operations observed in the traces used to train the model. This value could then be tested for correlation with performance. An obvious candidate for comparing two distributions is relative entropy [Cover01]. This approach has proved difficult, however, because even relatively large traces don't exhibit improbable but not impossible combinations of operations frequently enough to generate a statistically significant distribution, hence skewing the relative entropy measure.

Instead, for this paper we choose a less intuitive, but easier to calculate measure of the accuracy of the model: the likelihood of the model given the trace. Likelihood is distinct from probability in that likelihood refers to events in the past. Bayesian statisticians routinely utilize likelihood to measure the probability that a given model generated some data, especially in inference problems [Gelman03]. In fact, the Baum-Welch re-estimation algorithm we use to train the HMM utilizes the likelihood of the model to estimate the parameters of the HMM. We are in essence taking a Bayesian perspective by using the likelihood as a measure of the accuracy of the model¹.

Conveniently (and by definition), the likelihood of the model given the data and the probability of the data given the model turn out to be the same quantity. So given a model, and some data, we can easily calculate the likelihood of the model.

For example, suppose we have a sequence of symbols A,B,D,A,C,D, and the HMM shown in Figure 4. If we consider the likelihood of each path through the HMM model of this sequence, and sum those likelihoods together, we get the likelihood that the model generated the data we have observed.

Sequence	Product	Likelihood
"A","B","D","A","B","D"	$1 \times 0.5 \times 1 \times 1 \times 0.5 \times 1$	0.25
"A","B","D","A","C","D"	$1 \times 0.5 \times 1 \times 1 \times 1 \times 1$	0.5
"A","C","D","A","B","D"	$1 \times 0 \times 1 \times 1 \times 0.5 \times 1$	0
"A","C","D","A","C","D"	$1 \times 0 \times 1 \times 1 \times 0 \times 1$	0
Total:		0.75

Table 2: The likelihood calculation for the HMM in Figure 4

¹ The Baum-Welch algorithm, however, has both a Frequentist and a Bayesian interpretation.

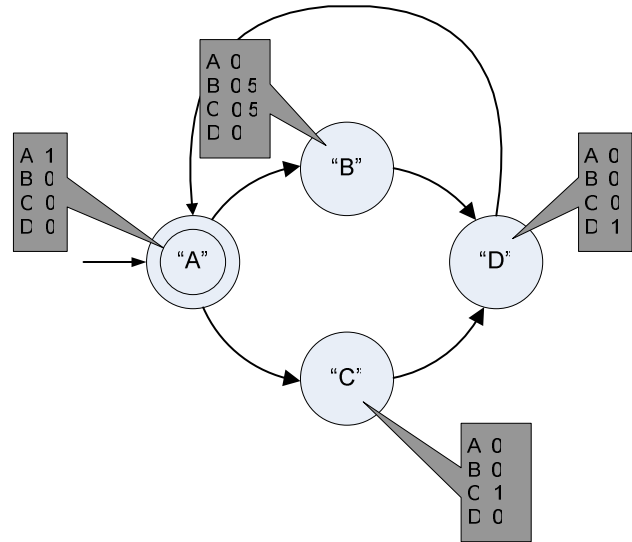


Figure 4: A simple HMM with start state "A"

Error! Reference source not found. shows this calculation, yielding a likelihood of 0.75. Now consider what happens if we adjust the probabilities, so that B and C have probabilities 0.8 and 0.2 respectively in node "B", B and C have probabilities 0.3 and 0.7 respectively in node "C". In that case, shown in Table 3, our model yields a likelihood of 0.99.

Sequence	Product	Likelihood
"A","B","D","A","B","D"	$1 \times 0.8 \times 1 \times 1 \times 0.2 \times 1$	0.16
"A","B","D","A","C","D"	$1 \times 0.8 \times 1 \times 1 \times 0.7 \times 1$	0.56
"A","C","D","A","B","D"	$1 \times 0.8 \times 1 \times 1 \times 0.2 \times 1$	0.06
"A","C","D","A","C","D"	$1 \times 0.8 \times 1 \times 1 \times 0.2 \times 1$	0.21
Total:		0.99

Table 3: The revised likelihood calculations

Based on these calculations, we can compare the two models and decide that it is more likely that the second model generated the sequence in question than the first.

Note, however, that the likelihood will decrease as a sequence gets longer, since each new symbol means that we multiply by another probability. Thus likelihood serves as a good metric to compare different models for the same sequence, but cannot be used to compare different sequences over different models. The optimal value for the likelihood depends not only on the model, but also on the sequence.

Additionally, it is customary to take the *log* of the likelihood (often referred to as the log-likelihood) to avoid numerical problems, so likelihoods are typically expressed as negative numbers. Log is a monotonically increasing function, however, so we still always want to maximize log-likelihood.

While log-likelihood may seem unsatisfactory with

comparison to some distance metric between the operations sequence generated by the HMM and the trace file used to train the HMM, it serves as a suitable means to compare different models of the same trace. Empirically, we also observe a strong correlation between the log-likelihood of our model and server performance, as we shall see in Section 4.

4 Experimental Verification

In this section, we discuss three sets of experiments we have conducted to verify the computational cost and effectiveness of HMM in modeling an observed trace. In the first experiment, we show how the computational time required for model training as we increase the complexity of the model. In the second experiment, we apply the likelihood metric presented in Section 3.3 to a series of models of various complexity against the trace from which the models were derived. Finally, we show the results of performance tests run using workloads generated from several HMM-based models of various complexity, and compare these to a completely random (but legal) stream of operations to demonstrate the effect of context sensitivity on performance.

4.1 Model Training Time

By using a HMM as opposed to a trace replay to generate load, we replace runtime CPU and memory requirements (e.g. while the load is being generated) with preprocessing requirements. Since these requirements are polynomial in the number of states as well as linear in the length of the sequence, the complexity of the model can have an impact on the ability of our system.

To measure the time to train our models on a short trace with 6242 operations, we trained models with between 1 and 5 clusters of operations, and between 1 and 5 states per cluster (see Figure 2).

In Figure 5, we can see that the training time per iteration indeed increases polynomially with the number of states in the HMM, but that exponent of the polynomial is approximately 1.6555. These efficiencies are due to optimizations in the specific HMM implementation we use [LogiLab,Speech89].

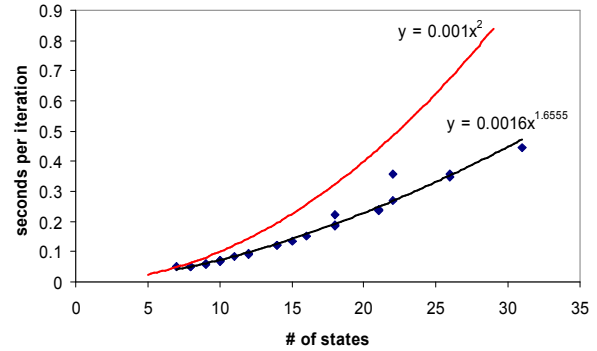


Figure 5: Training time for various models of different complexity

The complexity of the model also affects the number of iterations required for the training algorithm to converge. In Figure 6, we see that as the number of states and hence the complexity of the model increases, that there is a general trend towards more training iterations. Since the number of training iterations required is highly dependent on the (randomized) initial guess for the parameters of the HMM, the training time for the model can vary drastically, hence the outliers in Figure 6. The trend line from Figure 6 excludes these outliers.

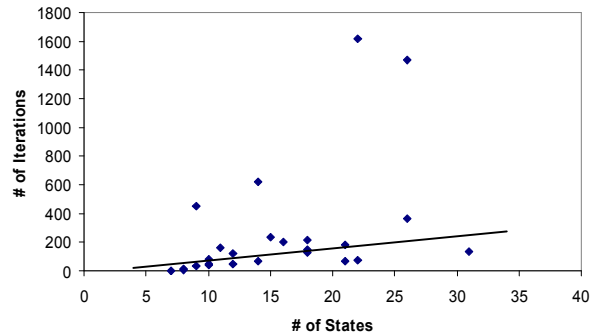


Figure 6: Training iterations for various models

Increasing the complexity of the model can thus increase both the cost of an iteration and the number of iterations required. This increasing cost must be balanced against the increased accuracy of the model. As we shall see in the next section, increasing complexity has diminishing returns.

4.2 Accuracy vs. Complexity

As we increase the complexity of our model, the accuracy of our model increases. In Figure 7, we see a strong positive correlation between increasing complexity and increasing accuracy. This curve, however, is sub-logarithmic (on a log-scale), as shown by the trend line. Thus increasing complexity

has both diminishing returns and increasing cost.

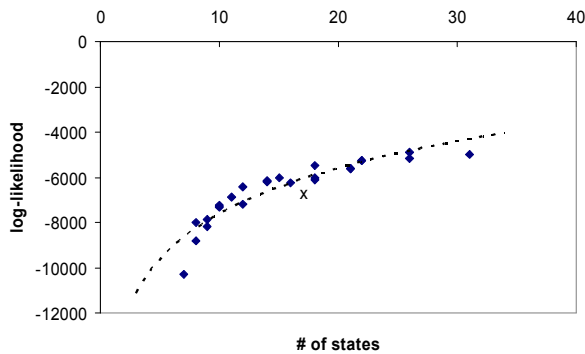


Figure 7: Likelihood of models with different complexity

4.3 Effectiveness of HMMs

In order to determine whether the stateful workloads generated by the HMMs actually have an impact on performance, we did an experiment which examines the performance of a machine while servicing operations generated by an HMM as described in the previous sections on one hand, and an HMM which generates operations with the correct mixtures of operations, but which only keeps the minimal amount of state necessary to adhere to the CIFS protocol on the other.

For a specific trace collected from a storage server, we train a series of four HMMs. First, we train an HMM with only a single state between the OPEN and CLOSE states. This is the simplest possible HMM which obeys the CIFS protocol, and essentially corresponds to randomly generating operations. Next, we train a single cluster of three states between the OPEN and CLOSE states, then three clusters, then and finally five clusters.

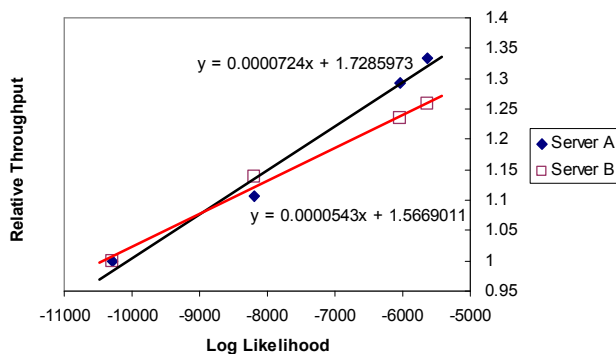


Figure 8: Relative throughput of two different servers vs. the complexity of the HMM generating the workload expressed in terms of the Log Likelihood metric

If the complexity of the HMM generating operations makes a significant impact on the performance of a server, then we would expect to see a correlation (either positive or negative) between the complexity of the HMM and the maximum throughput of the server. Furthermore, in order to demonstrate that the impact on performance is significant and relevant to benchmarking, we would like see that the impact of the complexity of the HMM on the performance of different servers is different. If this were not the case (e.g. if the percentage difference in performance on all servers was the same), then greater accuracy in the generated workload would only serve to assist in capacity planning, but would not have any purpose for comparison between servers. Note that even if a complex HMM serves no important purpose, a simple HMM (such as the single, one state clique HMM described above) is still useful since it obeys the requirements of the protocol, something that randomly generated operations such as those found in NFSv3 cannot do.

To measure the relationship between HMM complexity and maximum throughput, we ran the same benchmark with the same client configurations for two different servers, varying only the HMM configurations. In these tests Server A is a higher performance server than Server B.

Figure 8 shows the impact of the complexity of the HMM (expressed using the Log Likelihood metric as explained in Section 3.3) on the performance of two different servers. To allow comparison between different servers, values are relative to the performance of the server when loaded by the simple HMM with one clique of one state. From the above result it is clear that higher performance servers are better able to exploit the correlation between operations generated by more complex HMMs. The high end server is able to gain an additional 8% performance gain relative to the low end server. We suspect this is due to a larger cache size, but also may be related to increased parallelism in the processor, higher disk bandwidth, etc.

Interestingly, although the number of states between the rightmost two data points increases significantly (by six states) between the HMMs with three and five clusters, the throughput does not change significantly. This corresponds to the diminishing returns exhibited in Figure 7. This suggests that likelihood is a good metric for the accuracy of the model, since it corresponds closely to a real impact on performance. Of course, other training traces

may exhibit different “sweet spots”, depending on the amount of hidden state in the trace.

Even though the log likelihood metric has been useful to compare relative performance of HMMs with varying degrees of complexity, the above experiments do not clearly show how close we get to the performance of a server running against the actual trace from which the HMMs were generated. This requires that we run the trace and workloads generated using the HMMs against the same setup with exactly the same file layout under similar conditions. This is an area where significant future work would be focused on.

5 Summary

In this paper, we have developed a unique technique for analyzing and generating workloads for context sensitive protocols such as CIFS using HMMs that can be supported easily in any workload generator or benchmark.

Early results indicate that the HMM technique can accurately and realistically mimic a series of operations coming into a file server. Our experiments also show that accurately modeled workloads have a significantly different performance profile than inaccurately modeled workloads. We have also shown that context sensitivity can have different a different effect on different storage servers, and is therefore an important element of performance benchmarks. This suggests that even benchmarks of stateless protocols such as NFSv3 could benefit from a context sensitive benchmark.

This paper presents our principle but preliminary results. We are continuing to develop this research in the following ways. First, we intend to study the effects of statefulness over variety of different traces, taken in diverse environments. Secondly, we intend to resolve the problems we are currently having with our absolute metric of similarity between the training trace and the generated sequence of operations. Third, we plan to demonstrate more conclusively that our metrics of similarity to the original traces are sound using the methodology suggested in the previous section. And finally, we intend to further explore the extent to which statefulness affects different storage servers differently.

In conclusion, we believe that context sensitivity is essential to generating not only legal but also realistic storage server workloads. We are hoping to have context sensitive operation generation using

HMMs supported in industry standard benchmarks in the near future.

6 References

- [AIMA] Stuart Russell, Peter Norvig, “Artificial Intelligence, a Modern Approach,” Second Edition, Prentice Hall, 2003
- [CIFSRe02] “Common Internet File System Technical Reference, Version 1.0”, SNIA CIFS Documentation Work Group, March 2002, http://www.snia.org/tech_activities/CIFS/CIFS-TR-1p00_FINAL.pdf
- [HMM] Hidden Markov Models, http://www.comp.leeds.ac.uk/roger/HiddenMarkovModels/html_dev/main.html
- [NFSv4p03] Network File System Version 4 Protocol, RFC 3530, <http://www.ietf.org/rfc/rfc3530.txt>, April 2003
- [NFSv3p95] Network File System Version 3 Protocol, RFC 3530, <http://www.ietf.org/rfc/rfc1813.txt>, June 1995
- [Ramany01] Swami Ramany, “A Case for a New CIFS Benchmark”, Proceedings of the CMG 2001, Anaheim, CA, December 2001.
- [Speech89] LR Rabiner, B Juang, “A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition,” Proceedings of the IEEE, 1989
- [LogiLab] HMM, a fast python based HMM implementation, <http://www.logilab.org/projects/hmm>
- [Netb01] The Netbench Benchmark, <http://www.veritest.com/benchmark/netbench/default.asp>
- [SFS97] SPEC SFS97_R1 V3.0, <http://www.spec.org/sfs97r1/>
- [Gelman03] Andrew Gelman, John B Carlin, Hal S Stern, Donald B Rubin, “Bayesian Data Analysis”, CRC Press, 2003

[Cover01] Thomas M. Cover, Joy A. Thomas,
"Elements of Information Theory",
John Wiley & Sons, Inc. 1991

[Hopcraft79] Hopcroft, J. E. and Ullman, J. D.,
"Introduction to automata theory,
languages, and computation",
Addison-Wesley, Reading, MA,
1979