Input/Output Access Pattern Classification Using Hidden Markov Models*

Tara M. Madhyastha Daniel A. Reed

{tara,reed}@cs.uiuc.edu
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Abstract

Input/output performance on current parallel file systems is sensitive to a good match of application access pattern to file system capabilities. Automatic input/output access classification can determine application access patterns at execution time, guiding adaptive file system policies. In this paper we examine a new method for access pattern classification that uses hidden Markov models, trained on access patterns from previous executions, to create a probabilistic model of input/output accesses. We compare this approach to a neural network classification framework, presenting performance results from parallel and sequential benchmarks and applications.

1 Introduction

Input/output is a critical bottleneck for many important scientific applications. One reason is that performance of extant parallel file systems is particularly sensitive to file access patterns. Often the application programmer must match application input/output requirements to the capabilities of the file system. Because this match is so critical, many studies have demonstrated how a file system can exploit knowledge of application access patterns to provide higher performance than is possible with general file system policies [5, 12, 22].

To exploit application access pattern information, we investigate automatic input/output access pattern classification techniques that drive adaptive file system policies. Ideally, automatic classification can reduce an application developer's input/output optimization effort and increase performance portability across file systems and system architectures by isolating input/optimization decisions within a retargetable file system infrastructure. We have shown this approach to be successful, using an artificial neural network (ANN) based classifier [18].

This paper describes a complementary classification technique that uses hidden Markov models (HMMs) [24, 1] for modeling input/output access patterns, using training data from previous application executions. As we shall see, this method offers significant advantages over ANN access pattern classification, increasing resource utilization and providing more precise control over caching and prefetching, further improving performance.

The remainder of this paper is organized as follows. We describe the access pattern classification problem in §2. In §3, we describe hidden Markov models and how they can be used to classify input/output access patterns. Access patterns exist both locally and globally within the context of a parallel program; we describe this distinction and global classification in §4. We present extensions to our Portable Parallel File System (PPFS) to support adaptive file system policies in §5. In §6-§7, we present experimental data to evaluate HMM classification, comparing it to neural network classification, on sequential and parallel applications. We describe related work in §8. Finally, we summarize our plans for future work and conclude in §9.

2 Access Pattern Classification

A file access pattern is a description of past and future accesses (e.g., read-only, sequential, with a request size of 8 KB). Most current file systems are optimized for a small number of common access patterns and do not modify caching or prefetching policies when non-standard patterns occur. For example, most UNIX file systems are optimized for sequential access patterns and perform poorly for random accesses. By automatically recognizing access patterns and tuning policies to these access patterns, we can improve file system performance.

To be useful for controlling file system policies, a file access pattern description need not be a perfect predictor of future accesses. It simply needs to provide the file system with enough information to select suitable policies. To this end, in [18] we proposed an artificial neural network (ANN) classification framework that processes statistics calculated from a short sequence of input/output requests and generates qualitative, categorical classifications of access patterns (e.g., strided or random, read only or read/write). A neural network is a learning method

*Supported in part by the National Science Foundation under grant NSF ASC 92-12369 and by a joint Grand Challenge grant with Caltech, by the National Aeronautics and Space Administration under NASA Contracts NGT-S1399, NAG-1-613, and USRA 5555-22 and by the Defense Advanced Research Projects Agency under ARPA contracts DAVT63-91-C-0029, DABT63-93-C-0040 and DABT63-94-C-0049 (SIO Initiative).

that is well suited to this classification problem. These classifications in turn are used to select file system policies.

Our neural network classification framework was designed to provide efficient, real-time classification throughout program execution, and to create purely qualitative and platform-independent classifications from a low-level input/output access representation. Ultimately, qualitative classifications are supplemented by quantitative access statistics to guide policy control (e.g., to provide the quantitative stride length for a qualitative strided access pattern).

However, certain access pattern information that could be used to improve performance is unavailable from periodic observation of short sequences of input/output requests. Not all access patterns are amenable to brief, qualitative descriptions (e.g., sequential, random). For example, an irregular read-only access pattern might be repeatable or truly unpredictable. Metadata accesses to structured data files often have highly irregular patterns, but they are caused by library calls and are repeatable across program executions. A simple qualitative classification could specify only that an access pattern is highly irregular or random, and an intelligent prefetcher might use this knowledge to disable prefetching. However, when the pattern has a complex, repeatable and detectable structure, a prefetcher could exploit this knowledge.

A model that allows one to calculate and express the probability of accessing a given portion of a file in the future, having observed some access sequence, can supply more complex access pattern information for caching and prefetching policy decisions. While a neural network might be trained to do this, it is not a very efficient approach. We know precisely the function that we want to compute: the probability distribution function of future accesses, having already observed some access sequence. This knowledge allows us to choose a learning method more appropriate to the problem.

To create such a model automatically, we use file access data from previous executions to train hidden Markov models (HMMs) [24, 1]. HMMs are commonly used in speech recognition software, where the goal is to identify sub-words, words, or syntax based on probabilistic models. At a very high level, access pattern classification is similar to speech recognition, and similar techniques apply. By accurately modeling large scale access pattern behavior, this probabilistic approach supplements qualitative classifications made by observing a limited number of accesses.

3 Hidden Markov Models

A discrete-time Markov process [10] is a system that at any time is in one of N distinct states. At discrete times, the system changes states, or makes a transition, according to a set of probabilities associated with each state. Each state corresponds to a single observable event, or an observation symbol.

In a discrete Markov process, the observation symbol uniquely determines the next state. A hidden Markov process is a generalization of a discrete Markov process where a given state may have several exiting transitions, all corresponding to the same observation symbol. In other words, the "hidden" nature of a hidden Markov process represents the fact that the observation sequence does not uniquely determine the state sequence.

As a simple example, imagine a biased lottery where we select a winning number by choosing a numbered ball from one of four containers; the container is chosen according to some specific algorithm. After selection, the ball is replaced in the container from which it was taken. Each day the winning number is published, but the identity of its container remains a secret. Our observation sequence is the stream of winning lottery numbers, the current state represents the container, and from this we attempt to model the selection of the hidden container to better predict winning numbers. Analogously, in the domain of access pattern classification, the underlying program logic that generates the input/output pattern is not directly observable, but we can use data from previous executions to model this logic and predict future access patterns.

Hidden Markov models can "learn" the hidden behavior of the application that generates input/output requests from observation of the request stream. Efficient algorithms exist to train the model on observed sequences and calculate the probability of future sequences. In §3.1 we describe our approach to modeling input/output access patterns using HMMs. In §3.2 we outline the HMM training process. Finally, §3.3 describes how to use HMMs for policy control.

3.1 Modeling Input/Output Patterns

There are many possible ways to model access patterns using hidden Markov models; the choice of model description is critical to its predictive ability. One must determine what application behavior corresponds to a state, the number of states in the model, and the allowed observations. Our approach is to construct a hidden Markov model where each state corresponds to a segment of the file.

Segments are contiguous file regions of fixed or variable size. Ideally, segment size is chosen to correspond to an underlying file system caching unit or the application request size. Within a segment, the access pattern is assumed to be sequential. For the purposes of this paper, we assume that a segment is the same as a fixed size file block, and each state corresponds to a block. For example, a 10 megabyte file has 1280 8-Kbyte file blocks, and accesses to this file could be modeled by an HMM with 1280 states.

Observations are reads or writes that change the current state with some probability to a new state (a new current file block). For example, Figure 1 illustrates an HMM that models a sequential read pattern for a file with five blocks. Application requests can be smaller than the block size; we process a trace of input/output requests so that consecutive reads or consecutive writes that access the current block do not incur reflexive transitions.

We prevent state space explosion by consolidating sequential accesses into a single state and pruning unlikely transitions. In practice, we have observed HMM memory requirements for regular access patterns to grow at most linearly with file size. For example, the HMM for a 238 MB strided file used in the Pathfinder application (described in 6.3) is less than 61 KB without any state space compression.

In Figure 1, the file access pattern is deterministic. Figure 2 illustrates a pattern where there is a conditional branch in the underlying program. Suppose that based

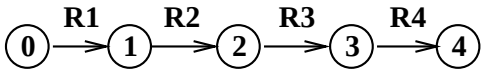

Figure 1: Markov model describing sequential reads.

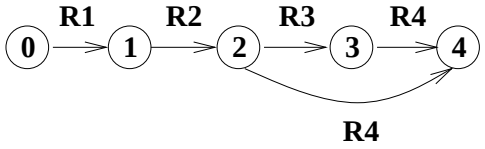

Figure 2: Degenerate HMM describing two possible control paths.

on data read from block 0, the program either reads the rest of the file sequentially or skips block 3. Trained on previous executions, the HMM computes the probabilities of each transition, information that can help the file system determine if it should prefetch block 3.

In the above examples, the Markov model is completely observable (i.e., given an observation sequence the system can be in only one state), and the only issue for complete specification is to determine, at each state, the probabilities of a read or a write to every other file block. Thus, the HMM is degenerate and can be expressed as a Markov chain.

Markov chains are sufficient to model many scientific input/output file access patterns; however, they fail when several predictable patterns are possible for a single file. For example, consider the case where the application may access a file with either a sequential or a mixed sequential and strided access pattern, depending on program input.

Figures 3a and 3b illustrate two possible models of this behavior. The HMM in Figure 3a is a poor predictor because it does not remember the path it took to reach each even-numbered block. Figure 3b shows a trained HMM that accurately models the two threads of control. Before training, all transitions are possible. We create two states for each file block, one each in groups A and B, noting that now many state sequences are possible given an observation sequence. At each state, an additional, unspecified parameter determines whether a read or write to a particular file block causes transition to a state in group A or group B.

Thus, after observing a read of blocks 1 and 2, one could be in either state 2A or 2B, with the next access being a read of block 3 or 4. The probabilities of the transitions determine which of these is more likely. Figure 3 can also be viewed as a composite of two individual HMMs; in essence the HMM clusters sequences, as described in [26].

3.2 Training Hidden Markov Models

Each application has one hidden Markov model per file; these models are trained online by running an application executable that has been linked with a training module. The HMM segment size is selected in advance based on the underlying cache block size, and the training module maintains counts of all block read or write transitions. Most files involve only a single access pattern*; in this

* Characterization studies of applications from the SIO initiative suite reveal no files that are accessed throughout an application lifetime with more than one unique access pattern.

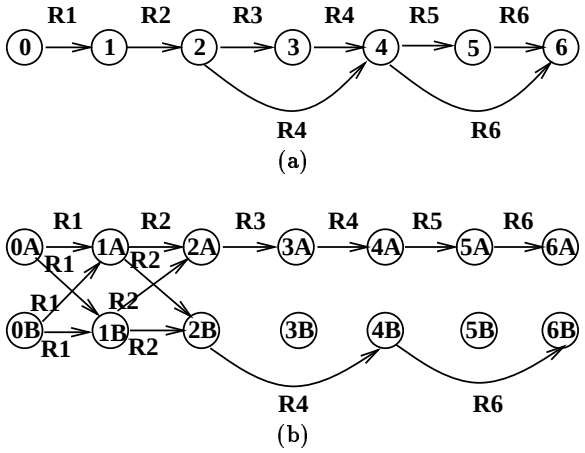

Figure 3: Two HMMs for modeling conditional access patterns.

case, a degenerate HMM with one state per file block suffices to model them and training is a trivial calculation. The probabilities for each transition are calculated by dividing the number of occurrences of the transition by the total number of transitions from each block, and only a single training execution is required. All examples in this paper utilize this training algorithm.

When a file has several significantly different access patterns (e.g., Figure 3), the training process is more complicated. Although we have not observed this to be a common in practice, HMMs elegantly model this uncertainty. To reflect the multiple threads of control, we construct a composite HMM from the HMMs trained on each unique access pattern. In essence, HMM construction is a clustering algorithm that assumes the number of clusters (i.e., unique access patterns) is known in advance, the user must provide this. For example, for two threads of control, the HMM structure would resemble Figure 3b. For details of the training process see [17].

3.3 Classification and Policy Control

In earlier work we examined the utility of purely qualitative classifications using a neural network based classifier. Based on our ongoing characterization of scientific application input/output patterns as part of the Scalable I/O Initiative [3, 25, 23], we partitioned access patterns based on three broad features: read/write mix, sequentiality, and request size; see Table 1.

At periodic intervals corresponding to some number of accesses or number of bytes accessed, the neural network based classifier produced qualitative classifications of these access pattern features. These classifications, supplemented by quantitative information about input/output requests (e.g., average request size), defined an access pattern space. Regions within this space were mapped to changes to the default caching and prefetching policies. This classification approach proved successful in improving performance of input/output intensive applications [18].

Qualitative classifications are limited to describing a simple set of structured access patterns. Many patterns are complex but repeatable, such those generated by appli-

Category	Category Features			
Read/Write	Read Only	Write Only	Read-Update-Write	Read/Write Mix
Sequentiality	Sequential	1-D Strided	2-D Strided	Variably Strided
Request Sizes	Uniform		Variable	

Table 1: File access pattern features.

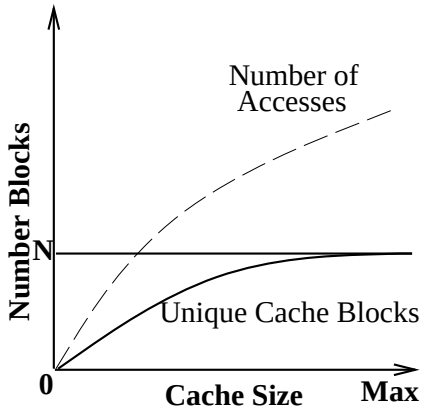

Figure 4: Allocating cache space.

cation calls to the NCSA hierarchical data format (HDF) library [20]. Such input/output patterns defy simple qualitative classification, but they have structure that is invariant across executions. Qualitative classification does not easily lend itself to description of these more complicated access patterns.

In contrast, hidden Markov models use data from previous executions to model access patterns probabilistically. A trained hidden Markov model with one state per file block learns the probability distribution function of accessing the next block given the current block. We have written a library of functions that uses an HMM, together with quantitative information computed from the input/output accesses, to estimate the probability of observing each of the features shown in Table 1 until the next classification.

For example, request size variability is determined heuristically from examining the number of unique request sizes in a recent window of accesses. On the other hand, the probability of sequentiality is estimated from the HMM by multiplying the probability of transitions between consecutive blocks for some number of accesses from the starting position. If the probability exceeds some threshold (e.g., 0.9) we adopt the corresponding policy.

HMM functionality is not limited to choosing among qualitative categories. For example, an HMM can be used to estimate a working set size by determining the minimum number of cache blocks that can capture cyclic access (e.g. multiple file reads, nested strided accesses). Figure 4 illustrates the heuristic. Intuitively, as the cache size increases, the application will be able to access more blocks without replacement. This is shown by the curve labeled "Number of Accesses." If there is locality, a small number of blocks suffices to capture the working set. The curve labeled "Unique Cache Blocks" denotes the number of unique blocks accessed as cache size increases. The maximum number of unique cache blocks accessed is a good estimate for minimal cache space allocation (N cache blocks). An HMM can be used to compute estimates of these curves by taking the most probable transitions through the states from any starting position within the file and recording the total number of block accesses and the number of unique blocks accessed for a range of hypothetical cache sizes.

An HMM can also be used to control classification frequency. Ideally classification need occur only when the access pattern changes. Neural network classification periodically reclassifies an access pattern; reclassification frequency is a user-specified parameter that must be carefully chosen to balance classification overhead and responsiveness to pattern changes. In contrast, using an HMM, a pattern change is possible only when the probability of accessing the next anticipated block using the current pattern falls below some specified threshold. Thus, the HMM can be used to monitor the input/output request stream, automatically recognizing access pattern changes.

Finally, the availability of data from previous executions can be used to make classifications for better policy selection at file open. In parallel applications involving many processors and large, disjoint, input/output operations, early access pattern classification and policy selection can make a significant difference in overall input/output performance.

HMM classification offers many advantages over ANN classification; however, the cost is the overhead of executing the application once or more to train the HMMs. This cost must be amortized over the performance improvements of subsequent program executions. For this approach to be profitable, we need to execute an application multiple times on the same data files (or the data files need to have similar structure and access patterns). In practice, this is a common mode of usage. Frequently, scientists rerun applications on the same data set, changing the algorithm slightly without changing the input/output access pattern. Often the access pattern is identical across different data sets.

4 Global Classification Issues

Until now, we have described modeling of a single input/output access stream; this is local classification, or classification per parallel program thread. Though sufficient for controlling uniprocessor caching and prefetching policies, local classification is but a small part of a larger classification problem. The local access patterns within each thread of a parallel program merge during execution, creating a global access pattern. Global knowledge is especially important for tuning file system policies, because coordinated access patterns cannot be detected with only local information. For example, consider a globally interleaved access pattern; each processor accesses a file by strides but the global pattern is sequential. A sequential prefetching policy at the input/output nodes can improve

performance.

Global classification is difficult because local information must overlap in time and space; i.e., bytes accessed must overlap or be disjoint and the local accesses must occur within the same timeframe. Although we can combine local information to make global classifications without temporal coordination, the classification will not be very useful in guiding file system policies. Simply put, each processor's local access pattern classifications must overlap in time and space to identify the global file access pattern at a given point in time.

We achieve temporal coordination by ensuring that local classifications overlap in time. We represent the valid time interval of a local classification as (t_s, t_e), where t_s is the start time and t_e is the end time. A global classification is valid for $(max(t_s), min(t_e))$ over all local classifications.

We ensure spatial coordination and determine a global access pattern by combining local classifications according to an access pattern algebra [19, 17]. For example, for a global access pattern to be read only, each local access pattern must be read only. Global classifications involving sequentiality require additional information about the bytes accessed by each client. For example, to make a classification of global sequential, an access pattern where all processors access a file sequentially, we must verify that the processors access overlapping bytes.

During execution, each processor periodically computes a local classification; these classifications, their time windows, and the byte ranges accessed are consolidated for global classification. Our framework for implementing and testing local and global classification and dynamic policy selection is described below.

5 Portable Parallel File System

The Portable Parallel File System (PPFS)[8] is a portable input/output library designed to be an extensible testbed for file system policies. It has a rich interface for control of data placement and file system policies that can be manipulated by the application or by an automatic classifier.

Figure 5 shows the PPFS components and their interactions. Application clients initiate input/output via invocation of PPFS interface functions. To open a file, the application first contacts the metadata server, which stores or creates information about the file layout on remote disk servers (input/output nodes). With this information, the application can issue input/output requests and specify caching and prefetching polices for all levels of the system. Clients either satisfy the requests locally or forward them to servers (abstractions of remote input/output devices). Clients and servers each have their own caches and prefetch engines. All "physical" input/output is performed through an underlying UNIX file system.

To provide automatic policy control based on local and global classification, we extended the basic PPFS design. Each client has a local classification module that generates qualitative and quantitative information used to control local caching and prefetching policies.

To support global classification, we added a global classification server to consolidate local classifications and necessary access statistics, as described in §4. When the global server classifies an access pattern, it updates policies accordingly on the input/output servers and the clients.

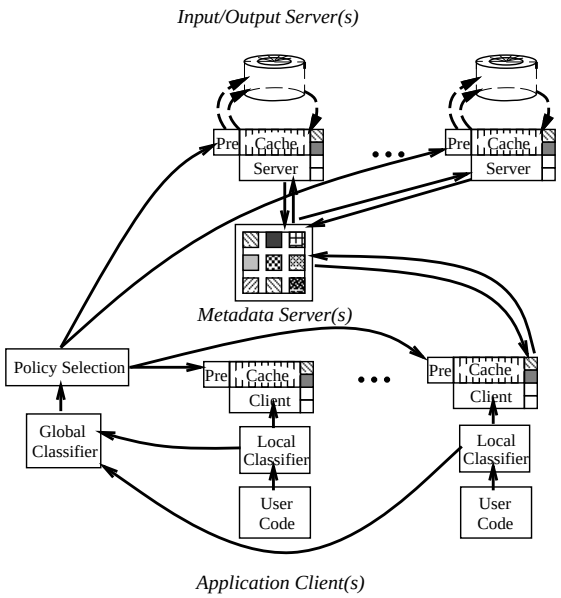

Figure 5: PPFS with global classification.

Under HMM classification, clients can also consult the global classifier at file open for an initial global policy based on previous execution data.

6 Local Classification Experiments

To compare ANN and HMM local classification, we conducted a series of uniprocessor performance studies using both access pattern benchmarks and an input/output intensive satellite data processing application. The experimental platform is PPFS on a Sun SparcServer 670 running SunOS 4.1.3, with 64 MB of physical memory and a local SCSI disk. Local classification guides policy control, as described in §4. Table 2 shows how access patterns are matched to policies. The average request size thresholds, default cache block size (32 KB), and large and small cache sizes were determined through experimentation and are platform-dependent.

6.1 Dynamic Access Patterns

One of the main advantages of dynamic classification is the ability to refine policy selections as the access pattern changes. In this section, we describe the performance of a benchmark that reads the first half of a 40 million byte file sequentially and the second half pseudorandomly. For both access patterns, the request size is 2000 bytes. We executed this benchmark three times using PPFS, twice with adaptive file system policies guided by neural network based classification and hidden Markov model based classification, and once with a single, default system policy.

Figure 6 shows the throughput (computed every 10 accesses) for the benchmark. The periodic dips in throughput before the access pattern change are caused by cache misses. When the access pattern changes from sequential to random, the policy selection of Table 2 is to disable caching, reducing the byte volume of physical input/output calls and improving throughput.

Because the frequency of ANN classification is a user-

Sequentiality	Read/Write	Request Size	Action
Sequential	ReadOnly	$\geq$ 8 KB	disable cache
Strided/Random	ReadOnly	$\geq$ 1.5 KB	disable cache
Strided/Random	ReadOnly	$<$ 1.5 KB	LRU, enlarge cache
Sequential	WriteOnly	any	MRU
Strided/Random	WriteOnly	$\geq$ 8 KB	disable cache
Strided/Random	WriteOnly	$<$ 8 KB	LRU, enlarge cache

Table 2: Classifications and policy control.

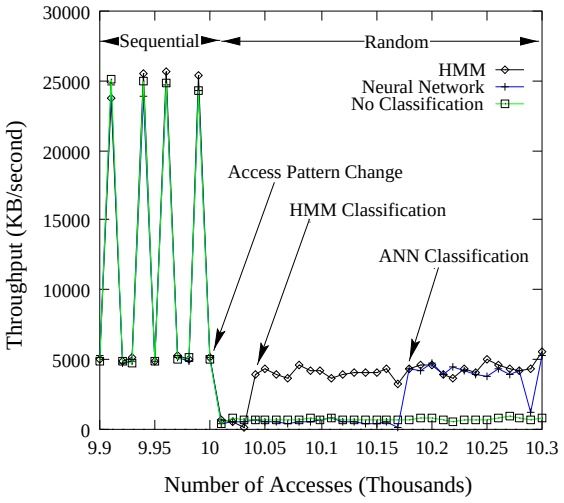

Figure 6: Adapting to a changing access pattern.

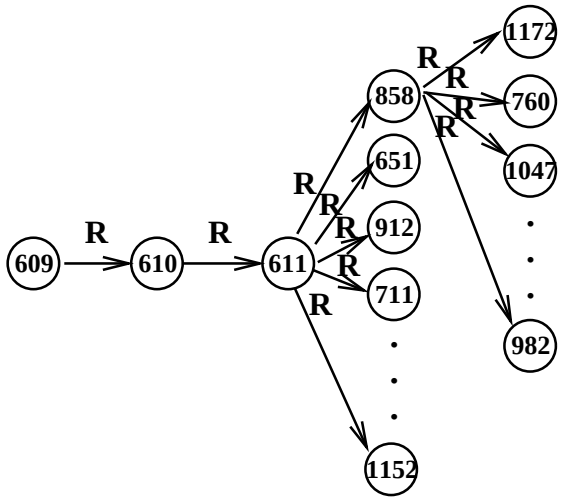

Figure 7: HMM at access pattern change.

specified parameter, responsiveness of the ANN classifier is limited by the selected classification frequency. In this PPFS configuration, the HMM classifier is able to detect this access pattern shift sooner than the ANN method. The ANN classifier automatically reclassifies the pattern every 640 KB, so it begins to reclassify the random access pattern after reading 20316160 bytes (i.e., 31 640 KB blocks). Figure 7 illustrates the portion of the HMM where the access pattern changes; the symbol **R** denotes a read of the block indicated by the arrow. After detecting a sequential access pattern, the HMM classifier computes the probability that the sequential access pattern will continue. At block 611 the probability of accessing block 612 is 0.0, so the access pattern can no longer be sequential. Because blocks are 32 KB, reclassification occurs after reading 20021248 bytes, the last byte in the block that marks the end of the sequential run.

6.2 Sequential File Reuse

In many cases, an application reads a file repeatedly. A common optimization for this access pattern is to retain as many of the file blocks in cache as possible to avoid rereading them. Both ANN and HMM classifiers can recognize sequentiality, but because it classifies access patterns on a much larger scale, the HMM classifier can also predict reuse of sequentially accessed blocks and determine an appropriate cache size, using the method described in §3.3.

To illustrate the potential performance improvement, consider a benchmark that reads a 20 million byte file five times sequentially. The request size is 2000 bytes, and classifications occur every ten thousand accesses. Both ANN and HMM classifiers detect that the access pattern is read-only and sequential. The HMM classifier recognizes that the file will be re-read (because the access pattern is sequential and the transition from the last block is a read of the first block) and that a cache size of 20021248 bytes (611 cache blocks of size 32 KB) can retain the file in memory, so the cache is resized.

Figure 8 shows the throughput (computed over every 100 accesses) for the benchmark executing with both classification methods. During the initial file read, throughput using both classification methods is roughly equivalent (minor performance variations are due to system level cache effects). After the file is read once, throughput using both classification methods increases because the file is in the UNIX file system cache. However, under HMM classification the file is also retained in the PPFS user-level cache, further improving throughput.

6.3 Pathfinder

In this section we explore the performance of Pathfinder, a typical sequential UNIX application used to process low-level satellite data. Pathfinder is from the NOAA/NASA Pathfinder AVHRR (Advanced Very High Resolution Radiometer) data processing project. Our analysis focuses on the generation of daily data sets, created from fourteen files of AVHRR orbital data (approximately 42 megabytes each) processed to produce an output data set that is approximately 228 megabytes in HDF format.

During program execution, ancillary data files and the

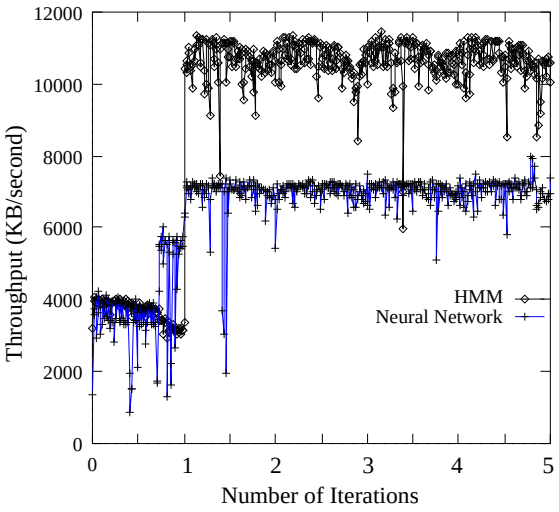

Figure 8: Detection of file reuse.

orbital data files are opened, and each orbit is processed sequentially. The access patterns for ancillary files range from sequential to irregularly strided. The result of this processing is written to a temporary output file using a combination of sequential and two-dimensionally strided accesses. Finally, the temporary file is re-written in HDF format. Because they are processed in a main program loop, we focus on the processing of only one of the fourteen orbital data files.

One of the problems with automatic policy selection using the algorithm of Table 2 is that limited physical memory prohibits indiscriminately enlarging file caches. Two Pathfinder files are accessed with small, strided requests; one must decide how to allocate the available cache space between these files. The ANN classifier cannot predict working set sizes, so one can either allocate space to one file, or split it evenly. The HMM classifier can generate a working-set size prediction that one can use to allocate cache space.

Table 3 shows the execution times for Pathfinder using UNIX buffered input/output and PPFS using ANN and HMM classification and different cache allocation strategies. When the ANN classifier detects that the output file is accessed with small strided accesses, the output file cache is enlarged. Without enough memory to enlarge the cache for small strided reads, caching is disabled for the input file accessed with that pattern. This policy selection achieves a speedup of approximately 1.75.

Another solution to the problem of limited cache space is to partition space evenly between the two files. In this example, this approach is superior, producing a two-fold improvement in performance over the original execution time. Using HMM classification and the cache space allocation heuristic described in §3.3, we can estimate the cache space needed for the input file and allocate the remaining memory to the output file. This strategy yields the best performance, a speedup of 2.27.

Table 4 shows the UNIX system level input/output operation counts and bytes for Pathfinder using both classification methods. PPFS with ANN classification enlarges the file cache to 25 MB for small, strided writes, and disables caching for small variably strided reads. This contributes to the reduction in number of bytes read while causing an increase in the read count. The write count is reduced by two orders of magnitude, although the total write volume increases because of the larger cache block size (32 KB vs. 8 KB for buffered UNIX input/output). When the cache is evenly split, there are fewer physical reads, but the additional block replacements necessitated by reducing the cache space available to the output file increase the number of writes and the read and write volume.

Experimental Environment	Total Time
UNIX	4299.3
PPFS (ANN)	2452.0
PPFS (ANN, split cache)	2174.4
PPFS (HMM)	1891.6

Table 3: Pathfinder execution times (seconds).

With the usage prediction provided by the HMM classification, 4 MB is allocated to the input file and the remaining 21 MB to the output file. The number and volume of writes is identical to that obtained by allocating the entire 25 MB cache to the output file, so write performance has not been sacrificed by increasing the input file cache. However, the number of physical reads decreases, and due to block replacements, the total read volume is only slightly greater than when input file caching is disabled.

7 Global Classification Experiments

The experiments in the previous section exercised local classification and policy control. Local classification is only part of a larger classification problem; as described in §4, an important factor in parallel input/output performance is exploiting characteristics of the combined, or global input/output access stream, by making a global classification.

To ensure good performance, parallel file systems sometimes require that the application specify global access pattern information. Global classification can automatically recognize global access patterns. To demonstrate the potential performance improvement from global classification, below we show results from two typical parallel applications from the Scalable I/O Initiative application suite and a benchmark representing complex array access. Our experimental platform is the Intel Paragon, using PPFS to perform physical input/output using PFS[9].

PFS is a parallel file system that stripes data over disks on input/output nodes using a default 64 KB stripe size. In normal usage, applications provide access pattern information by specifying PFS modes. Using global classification, we automatically select an optimal PFS input/output mode for a file access pattern and compare performance to the default mode.

The PFS modes we manipulate below are M_UNIX (the default, atomic UNIX style input/output), M_ASYNC (which does not preserve input/output atomicity) and M_GLOBAL (for global access patterns, where all processors read the same file bytes).

Experimental Environment	Read		Write		Lseek
	Count	Bytes	Count	Bytes	Count
UNIX	3,030,382	2.48247e+10	4,077,265	625,698,239	10,961,293
PPFS (ANN)	3,669,087	1,098,883,277	27,102	888,014,784	3,640,123
PPFS (ANN, split cache)	58,582	1,595,272,028	41,562	1,361,840,064	71,307
PPFS (HMM)	44,134	1,121,577,820	27,102	888,014,784	43,279

Table 4: Pathfinder operatation counts and bytes accessed.

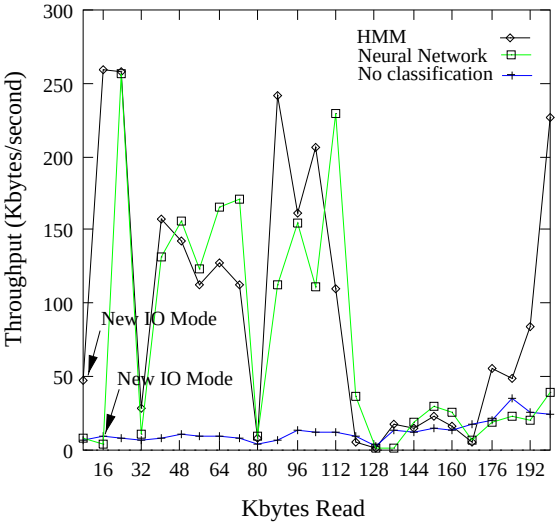

Figure 9: PRISM initialization file throughput.

7.1 PRISM

PRISM, a computational fluid dynamics code, is a parallel implementation of a 3-D numerical simulation of the Navier-Stokes equations [6, 7]. The parallelization is by apportioning the periodic domain to the processors, with a combination of spectral elements and Fourier modes used to investigate the dynamics and transport properties of turbulent flow. We focus on the first phase, in which every processor reads three initialization files. In an underconfigured input/output system, this kind of workload (global sequential access) causes a bottleneck at the input/output servers that are overrun with identical requests from each processor. To create such a bottleneck, we ran PRISM on 16 processors of an Intel Paragon running OSF/1 1.4 with one input/output node.

Figure 9 shows total throughput, calculated per 8 KB block, for access to one initialization file using the default M_UNIX PFS input/output mode (no classification) and using HMM and ANN classification to select the optimal mode M_GLOBAL. Throughput using the default mode M_UNIX is very poor, because in the PFS implementation of this mode seeks are serialized.

The ANN classifier improves performance by recognizing, after the first ten accesses, that the access pattern on each processor is sequential and read-only. It sends this information (along with the byte ranges accessed) to the global classifier. The global classifier determines from the sequential classifications and overlapping byte streams that the pattern is global sequential, and recommends each processor select mode M_GLOBAL. Because the PFS call to reset the input/output mode is synchronizing, and it must be issued when all processors are at the same file position, there is some synchronization overhead for a policy change. This accounts for the low throughput at the first measurement under the new policy. The policy change occurs approximately two seconds into input/output execution.

In contrast, the HMM classifier can query the global classifier for a recommended mode at file open. The global classifier computes an initial policy recommendation based on the individual processor HMMs and sends this recommendation (use of mode M_GLOBAL) to processors upon file open. While the initial ANN classifications could be saved between executions to similarly provide policy recommendations, they would also be based on previous execution data, and organization of potential multiple file access patterns would have to be supported externally.

After selecting mode M_GLOBAL, the overall file throughput under both classifiers is the same. Periodic dips in throughput occur because this mode requires input/output requests to be synchronized among the sixteen processors. Therefore, should one processor creep ahead of the others, it must stall until they synchronize, causing abnormally large read request times.

7.2 QCRD

QCRD [28, 14] is a quantum chemical reaction dynamics code used to study elementary chemical reactions. Parallelization is accomplished by data decomposition; all processors execute the same code on different portions of the global matrices. The matrices are large enough to necessitate an out-of-core solution; the chosen data decomposition results in strided and cyclic input/output access patterns.

We limit our analysis to the first phase of QCRD. In this phase, 64 processors coordinate to write 13 global output matrices, each with a global interleaved access pattern (the interleaving of the individual processors' strided accesses is sequential). We executed this application on a 64 node partition of a 512 node Intel Paragon XP/S. Output was to a parallel file system with 16 input/output nodes.

When using the default mode M_UNIX, input/output operation time accounts for 10.6 percent of the total execution time. Selecting mode M_ASYNC, which does not preserve UNIX file sharing semantics, reduces input/output time to less than one percent of overall execution time.

Figure 10 shows the throughput, calculated every 64 KB, for one of the output matrices. Throughput using mode M_UNIX (no classification) is poor because in the PFS implementation of this mode, seeks are synchronized and are very expensive. The ANN classifier switches to mode M_ASYNC approximately halfway through execu-

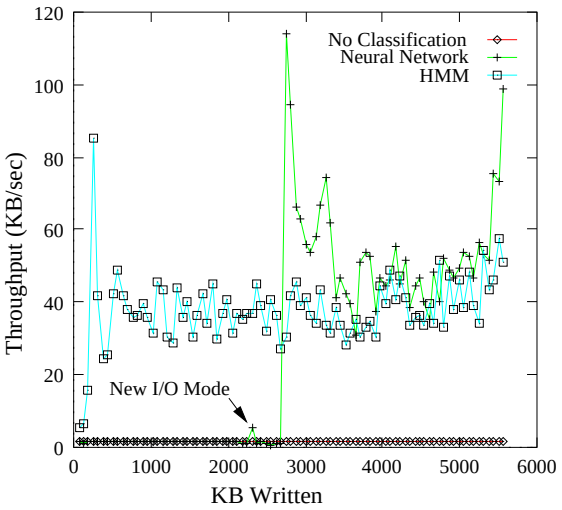

Figure 10: QCRD phase one throughput.

tion, dramatically improving throughput. The lag between the policy change and the performance improvement is due to synchronization overhead incurred by resetting the input/output mode. The HMM classifier can obtain a better initial default policy from the global classifier at file open, improving performance throughout the file lifetime.

In this application, the ability to make an informed initial policy selection based on previous execution information and a trained HMM significantly improves overall input/output performance, as seen in Table 5. The neural network classifier needs to observe some number of accesses (ten in this implementation) to recognize a global interleaved access pattern. Because of this delay, a classification and the ensuing global policy change cannot be made until half of the file is already written. Thus, total input/output time is slightly more than half of the time without classification. By using data from previous executions to determine the optimal PFS mode at file open, the HMM classifier reduces input/output time by an order of magnitude.

7.3 Complex Global Access Patterns

We have emphasized that HMM classification is able to classify a wider range of access patterns than is possible by ANN classification. This makes it particularly useful for guiding a prefetcher; at every access it can generate the probability of the most likely block to be accessed next.

In this experiment we consider a benchmark based on file access patterns exhibited by applications from Caltech for global climate modeling and modeling of the earth's interior [16]. These applications read and write data from a two or three dimensional grid, partitioning data in blocks of contiguous particles among the processors.

Figure 11 shows an example of such a partitioning on a two dimensional grid; each numbered block contains cs^2 elements allocated to the processor with that number. Processors simultaneously read their allocated chunks in segments of the particle size. Therefore, each processor exhibits a one-dimensionally strided access pattern. Globally, they coordinate to read the entire file, but the global access pattern is not interleaved sequential nor is it partitioned sequential; it is a mix of these that is unrecognizable as any predefined global pattern. The programmer cannot simply optimize this input/output pattern by changing the file layout so the matrix is accessed globally sequentially, because the allocations of particles to processors might change and can be quite complex.[†]

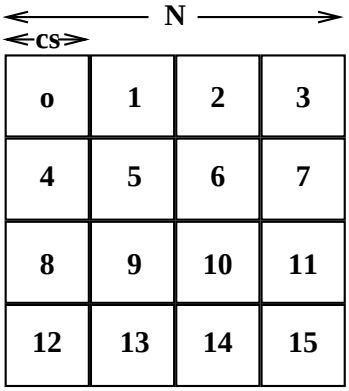

Figure 11: Block-block matrix partitioning.

Prefetching can improve read performance by overlapping input/output latency with computation, but prefetching must correspond to the access pattern. Because HMMs can be trained to recognize any arbitrary repeatable access patterns, the locally trained HMMs are useful for guiding prefetching.

To demonstrate this capability, our benchmark distributes a 64×64 matrix of 64 KB blocks across 64 processors according to the the block-block distribution shown in Figure 11. Each processor reads a block in row major order, and "computes" for 100 ms before continuing to the next block. The input file is striped on 12 input/output nodes, each controlling a Seagate drive. PPFS implements prefetching by using asynchronous PFS reads to initiate fetches for some number of blocks in advance of the access stream (the prefetch depth). Table 6 shows the input/output times for PFS operations for this benchmark. Prefetching one block ahead is not enough to keep up with the request stream, and has no significant effect on overall execution time. However, prefetching four blocks ahead significantly reduces read time and improves performance.

8 Related Work

Because performance of parallel input/output systems is extremely sensitive to access pattern characteristics, tailoring file system policies to application requirements is a natural approach to improving performance. One system-independent way of specifying application requirements is to provide hints (possibly inaccurate access information) to guide a proactive file system. Patterson *et al* demonstrate the success of providing hints to guide prefetching of files that will be accessed in the future [22, 27]. This approach is portable, but requires the application programmer to describe the application input/output behavior. We view access pattern classification as a potential way to automatically provide these hints.

[†]Ultimately an application of this kind will prove an excellent candidate for testing the efficacy of HMMs trained on multiple access patterns.

Experimental Environment	Write Time	Seek Time	Total
PPFS (no classification)	3468.92	0.64	3469.56
PPFS (ANN)	1896.06	0.35	1896.41
PPFS (HMM)	172.98	0.11	173.09

Table 5: QCRD input/output times (seconds) for a single output file.

Experimental Environment	Read	Seek	I/O Wait	Execution Time
PPFS (prefetch disabled)	266.47	0.21	N/A	23.14
PPFS (prefetch depth=1)	11.74	0.21	142.05	24.67
PPFS (prefetch depth=4)	14.58	22.35	6.85	13.43

Table 6: Input/output times for prefetching according to a block-block distribution (seconds).

Many groups have explored intelligent techniques to construct higher level models of file access automatically. Kotz has examined automatic detection of global sequentiality to guide non-sequential prefetching within a file [12].

Exploitation of relationships between files has also been a significant research topic. Fido is an example of a predictive cache that prefetches by using an associative memory to recognize access patterns over time [21]. Knowledge based caching has been proposed to enhance cache performance of remote file servers [11]. Some approaches use probabilistic methods to create models of user behavior to guide prefetching [15, 4, 13]. This work is similar in spirit to our HMM classification methodology. The difficulty with modeling user behavior probabilistically is that recently accessed files are more likely to be re-accessed than frequently accessed files. Access patterns within files are usually highly regular.

9 Conclusions and Future Work

We have shown that hidden Markov models can be used for modeling input/output access patterns, using training data from previous application executions. Experiments with benchmarks and sequential and parallel applications demonstrate that this approach offers more precise control over caching and prefetching policies than neural network access pattern classification, an approach that classifies patterns based on periodic inspection of a small window of accesses.

HMM-based classification is extensible and has many potential applications. For example, to ensure that input/output classifications are platform independent, we do not include time as a parameter in input/output traces. However, the interaccess delay is important information when deciding how far ahead to prefetch, or which files value cache blocks more highly. We are currently investigating modeling interaccess delay to augment HMM-based spatial classification. Second, there is a natural correlation between ordered hints as described in the SIO low-level API [2] and HMM classifications; we are investigating using HMMs to supply these hints automatically in the next-generation PPFS II.

Acknowledgments

We wish to thank J. Michael Lake for his helpful discussions and for suggesting the use of hidden Markov models. Evgenia Smirni also provided many useful comments and suggestions.

Some data presented here were obtained from code executions on the Intel Paragon XP/S at the Caltech Center for Advanced Computing Research.

References

[1] Charniak, E. *Statistical Language Learning.* The MIT Press, 1993.

[2] Corbett, P., Prost, J.-P., Demetriou, C., Gibson, G., Riedel, E., Zeleka, J., Chen, Y., Felten, E., Li, K., Hartman, J., Peterson, L., Bershad, B., Wolman, A., and Aydt, R. Proposal for a Common Parallel File System Programming Interface Version 1.0, 1996. Available at `http://www.cs.arizona.edu/sio`.

[3] Crandall, P. E., Aydt, R. A., Chien, A. A., and Reed, D. A. Characterization of a Suite of Input/Output Intensive Applications. In *Proceedings of Supercomputing '95* (Dec. 1995).

[4] Griffioen, J., and Appleton, R. Reducing File System Latency Using a Predictive Approach. In *Proceedings of USENIX Summer Technical Conference* (June 1994), pp. 197–207.

[5] Grimshaw, A. S., and Loyot, Jr., E. C. ELFS: Object-oriented Extensible File Systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems* (December 1991), p. 177.

[6] Henderson, R. D. *Unstructured Spectral Element Methods: Parallel Algorithms and Simulations.* PhD thesis, Princeton University, June 1994.

[7] Henderson, R. D., and Karniadakis, G. E. Unstructured Spectral Element Methods for Simulation of Turbulent Flows. *Journal of Computational Physics 122*, 2 (1995), 191–217.

[8] Huber, J., Elford, C. L., Reed, D. A., Chien, A. A., and Blumenthal, D. S. PPFS: A High Performance Portable Parallel File System. In *Proceedings of the 9th ACM International Conference on Supercomputing* (Barcelona, July 1995), pp. 385–394.

[9] Paragon XP/S Product Overview. Intel Corporation, 1991.

[10] Kleinrock, L. *Queueing Systems, Vol. 1, Theory.* John Wiley, 1975.

[11] Korner, K. Intelligent Caching for Remote File Service. In *Proceedings of the 10th International Conference on Distributed Computing Systems* (May 1990), pp. 220–226.

[12] Kotz, D., and Ellis, C. S. Practical Prefetching Techniques for Multiprocessor File Systems. *Journal of Distributed and Parallel Databases 1*, 1 (January 1993), 33–51.

[13] Kroeger, T. M., and Long, D. D. E. Predicting File-System Actions From Prior Events. In *Proceedings of the USENIX 1996 Annual Technical Conference* (Jan. 1996), pp. 319–328.

[14] Kuppermann, A., and Wu, Y.-S. M. The Quantitative Prediction and Lifetime of a Pronounced Reactive Scattering Resonance. *Chemical Physics Letters 241* (1995), 229–240.

[15] Lei, H., and Duchamp, D. An Analytical Approach to File Prefetching. In *Proceedings of the USENIX 1997 Annual Technical Conference* (Jan. 1997), pp. 275–288.

[16] Li, P. ESS Grand Challenge Projects: The Earth's Interior Modeling and the Global Climate Modeling. In *HPCC ANNUAL REPORT* (1995). available at `http://olympic.jpl.nasa.gov/PERSONNEL/wangp/ping96one.html`.

[17] Madhyastha, T. M. Automatic Classification of Input/Output Access Patterns. Tech. Rep., University of Illinois at Urbana-Champaign, Department of Computer Science, Aug. 1997.

[18] Madhyastha, T. M., and Reed, D. A. Intelligent, Adaptive File System Policy Selection. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation* (1996), pp. 172–179.

[19] Madhyastha, T. M., and Reed, D. A. Exploiting Global Access Pattern Classification. In *Proceedings of SC'97* (November 1997).

[20] NCSA. *NCSA HDF, Version 2.0.* University of Illinois at Urbana-Champaign, National Center for Supercomputing Applications, Feb. 1989.

[21] Palmer, M., and Zdonik, S. B. Fido: A Cache That Learns to Fetch. In *Proceedings of the 17th International Conference on Very Large Data Bases* (Barcelona, September 1991), pp. 255–262.

[22] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. Informed Prefetching and Caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (December 1995), pp. 79–95.

[23] Pool, J. T. Scalable I/O Initiative. California Institute of Technology, Available at `http://www.ccsf.caltech.edu/SIO/`, 1996.

[24] Rabiner, L. R. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE 77*, 2 (1989).

[25] Smirni, E., Aydt, R. A., Chien, A. A., and Reed, D. A. I/O Requirements of Scientific Applications: An Evolutionary View. In *Fifth International Symposium on High Performance Distributed Computing* (1996), pp. 49–59.

[26] Smyth, P. Clustering Sequences with Hidden Markov Models. In *Advances in Neural Information Processing 9.* MIT Press, to appear.

[27] Tomkins, A., Patterson, R. H., and Gibson, G. Informed Multi-Process Prefetching and Caching. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems* (June 1997).

[28] Wu, Y.-S. M., Cuccaro, S. A., Hipes, P. G., and Kuppermann, A. Quantum Chemical Reaction Dynamics on a Highly Parallel Supercomputer. *Theoretica Chimica Acta 79* (1991), 225–239.