Exploiting Global Input/Output Access Pattern Classification*

Tara M. Madhyastha Daniel A. Reed

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

1 Introduction

Despite continued innovations in disk design, input/output performance has not kept pace with concurrent increases in processor speeds. For input/output intensive applications, only parallel input/output can ameliorate the resulting input/output bottleneck. In a parallel file system, multiple application tasks concurrently access common files that are distributed across multiple disks. Each task has its own, local, access pattern, and the interleaving of these local patterns forms a global access pattern.

To minimize the execution time of an input/output dominated parallel application, a parallel file system must exploit not only the individual local access streams but also their aggregate, global properties. Although some current parallel file systems provide interfaces to express global access pattern information, application input/output characterization studies have repeatedly shown that application developers are often unaware of the detailed input/output behavior of their codes, particularly when input/output requests are mediated by input/output libraries (e.g., like HDF [17]).

To provide the requisite local and global access pattern information and to remove the intellectual burden of such specification from the application developer, we have developed a suite of automatic access pattern classification techniques that combine local input/output access patterns to create global ones. The file system classifies access patterns either dynamically throughout program execution, or at file open using information from previous program executions. It then uses the global classification to select file system policies that optimize the per file parallel input/output performance of the application as a whole.

The remainder of this paper is organized as follows. In §2 we motivate the need for global classification information to improve input/output performance. We describe local and global classification in detail in §3. Evaluation of global classification requires an experimental framework, described in §4. We present our experimental results in §5. Finally, §6-§7 place this work in context and summarize our results.

2 Access Pattern Classification Rationale

In previous work we showed the performance benefits of using local access pattern classification to tune file system policies [16] for sequential applications. In a parallel file system, the complexities of efficiently servicing concurrent, related, input/output request streams make exploitation of global behavior even more important to overall performance. Global access pattern information can be used to select globally optimal file system policies that cannot be determined from local access patterns.

*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).

For example, consider a global interleaved access pattern where local strided access patterns interleave during execution to create a sequential pattern. If the file is striped (the default declustering strategy used by most commercial parallel file systems) and request size is smaller than the stripe unit, the same stripe unit will be reread multiple times as the processors coordinate to access the file. Caching this stripe unit can potentially prevent many unnecessary disk accesses, and prefetching the file sequentially can improve throughput.

Application programming interfaces (APIs) for current commercial and experimental file systems provide limited support for application specification of global access patterns. In Intel's PFS [8] the programmer can select one of five modes that describe common global access patterns. MPI-IO [23], a proposed high-level API for parallel input/output, allows a developer to specify arbitrarily complex access patterns. IBM's PIOFS also provides an interface to describe file views. To provide this same functionality, while not requiring the application developer to specify the access pattern, we have developed real-time access pattern classification techniques that can dynamically identify local and global access patterns and choose appropriate file access modes and policies.

3 Access Pattern Classification

Abstractly, an access pattern is a qualitative statement describing future file accesses that can be used to select and tune file system policies. Applications often have qualitative, recognizable input/output access patterns (e.g., 'read-only sequential with large request sizes,' or 'write-only strided.') File systems are normally optimized for certain common access patterns, and performance is poor when other access patterns occur. For example, sequential prefetching is a poor policy when a file is accessed randomly.

Within a parallel application, there are two levels at which input/output access patterns can be observed. The first is at the local level (e.g., per parallel program thread), and the second is at the global level (e.g., per parallel program). For example, the threads of a parallel program

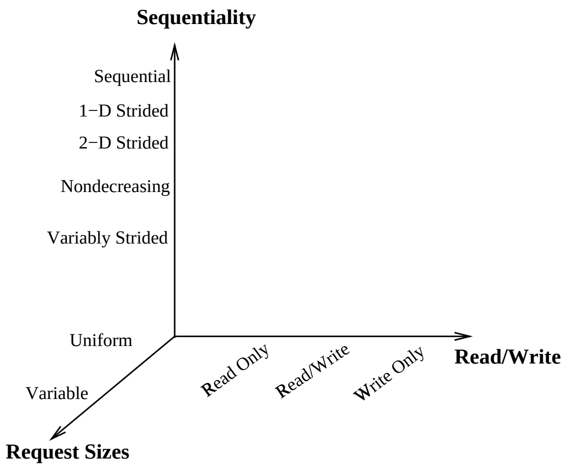

Figure 1: Access pattern space.

might access a parallel file in such a way that each thread appears to be accessing the file locally in strides, but the interleaved access stream is globally sequential.

3.1 Local Access Patterns

An input/output trace of file accesses from a single processor may be represented as a stream of tuples of the form

$$< \textit{byte offset}, \textit{request size}, \textit{read/write} > .$$

Based on this portable input/output representation and our ongoing characterization of scientific application input/output patterns as part of the Scalable I/O Initiative [2, 22, 20], we partitioned access patterns based on three broad features: read/write mix, sequentiality, and request size. Figure 1 shows an access pattern space described by these axes; each point in the three-dimensional space represents a unique access pattern. Many functions can be computed on the trace tuple components to yield identifiable access patterns in the access pattern space. However, access patterns that are predictive and can be used to influence file system policy selection are the most interesting to identify. Figure 1 shows certain categories meeting these criteria along each axis that can be used to label all points in the access pattern space. Additional categories can be added as necessary to each axis to further refine the access pattern space.

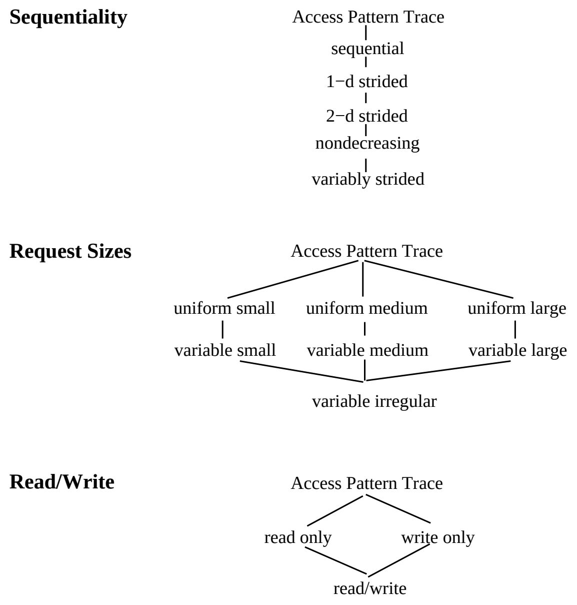

Figure 2: Hasse diagrams for local classifications.

The set of classifications on each axis $C = a, b, \ldots$ is a partially ordered set $< C, \subseteq >$, where $\subseteq$ is the relationship "is a less precise predictor." For example, a classification of "Read/Write" is a less precise predictor than either a "Read Only" or "Write Only" classification. Figure 2 shows the Hasse diagrams for local classifications. Each category of local classifications forms a lattice, because every pair of elements $a, b \in C$ has a greatest lower bound and a least upper bound. The greatest lower bound, or *meet* operator is denoted by $a \otimes b$ and the least upper bound, or *join*, by $a \oplus b$. For example, if there are two access patterns, read only and write only, from Figure 2 the greatest lower bound of the two is a read/write access pattern. The least upper bound of the two classifications is the access pattern trace, or the precise description of the accesses themselves, which would be a perfect predictor.

3.2 Local Classification Methodology

Many techniques can be used to classify and identify observed patterns using the lattice model. One simple approach is to apply heuristics to the sequences of input/output requests generated by an application and attempt to locate the pattern in the access space. Unfortunately such an approach is not easily extensible.

We have investigated more sophisticated classification approaches that "learn" to classify access patterns. One approach is to train a feed-forward artificial neural network [6] (ANN) to classify patterns. We provide the neural network with examples of access patterns and their corresponding classifications; once trained, the network can correctly classify new access patterns.

Another approach is to represent a file as a collection of blocks, and construct a probabilistic model of transitions between blocks from previous execution traces using an extended Markov model, or hidden Markov model (HMM) [21, 1]. With such a model, classification is performed by computing the probability of detecting certain features based on previous execution information. A detailed description of classification methodologies is beyond the scope of this paper; for more details, see [15].

These two classification approaches are complementary. Artificial neural network (ANN) classification detects access patterns automatically throughout execution without any execution history. HMM classification can recognize a wider range of access patterns and features that occur on a long time scale (e.g., file reuse), but to be useful, the HMM must be trained on previous application executions. Most importantly, HMMs can provide classifications at file open based on previous execution information, while neural network classification must observe some window of accesses before detecting local and global patterns.

3.3 Global Classification

Local classification is powerful tool for tuning file system policies in a sequential file system [16]. However, local classification is a small part of a global classification problem. As we noted earlier, local access patterns within the parallel program merge during program execution to create a global access pattern; it is essential to recognize qualitative patterns within the interleavings of the local input/output requests. In a parallel file system, the complexities of efficiently servicing concurrent, related input/output request streams make exploitation of global behavior critical to overall performance. Global access pattern information is necessary to select file system policies to optimize total system throughput that cannot be determined from local access patterns.

For example, consider a global sequential access pattern for a small initialization file (i.e., each processor reads the file individually). If a parallel file system stripes file blocks across disks and the request size is smaller than the stripe size, all input/output requests will be serviced by a single disk, causing a bottleneck. A better policy choice is to read the file once, providing each processor with its own local copy. However, this bottleneck cannot be determined without knowledge of the global access pattern, the data distribution, and the request size.

3.4 Global Classification Methodology

Our global classification infrastructure is based on an access pattern temporal algebra. We combine local classifications and other local data to make global classifications. However, merging this information is difficult because local information must be coordinated in time and space to identify the global file access pattern at a given point in time.

Local and global classifications are valid during a specific time interval corresponding to the duration between the first and last accesses within the observation period. 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 p local classifications. If $max(t_s) \geq min(t_e)$, the local classifications do not all overlap, so we cannot make a global classification with cardinality p.

When this happens, it implies that the local classification windows are too small and are staggered in time or that multiple access patterns are present. In the first case, it is difficult to predict *a priori* an appropriate duration of local classification windows to ensure a global classification window, because it is dependent upon the dynamic input/output rates of the individual processors. These rates change between and during program executions, varying with input data and system configuration.

If the local classifications overlap in time, a global access pattern classification is determined by a combination of local classifications. In addition, to identify global sequentiality, quantitative information about the input/output access stream is used to "correlate" the local classifications within the global file context. For example, if every local access pattern is sequential, the beginning and end of every sequential stream is used to determine whether the global pattern is *global sequential* (every process reads the entire file sequentially) or *partitioned sequential* (the entire file is read in disjoint, sequential segments).

Each category of the classification (i.e., each axis of Figure 1) except for sequentiality can be determined as the meet of the local classifications for that category. For example, the read/write global classifications are straightforward; if each local access pattern is read only, the global access pattern is read only. If the local access patterns collectively involve reading and writing, the strongest global classification is read/write.

Global classifications involving sequentiality require additional information about the bytes accessed by each client. As an example, we formulate a classification for a global interleaved sequential access pattern below, assuming a nonzero range $(max(t_s), min(t_e))$.

Interleaved sequential *Processors each access the entire file in strided patterns which interleave to form a global sequential pattern.*

Figure 3 shows byte ranges accessed by each of four processors during an interleaved sequential access pattern. We view the access pattern through a time window; the white regions are accesses used to make a global classification and the shaded accesses have not yet occurred.

We make this classification under the following conditions:

- $max(t_s) < min(t_e)$
- $\otimes L$ = 1-D strided
- $\cap$ bytes accessed = $\emptyset$
- If we merge the input/output requests from the different processors by file position, there should be at least p contiguous requests, where p is the number

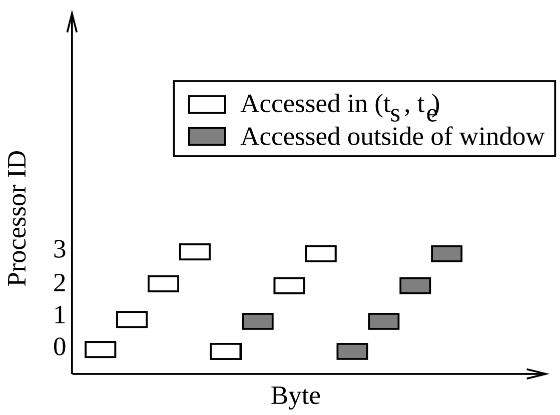

Figure 3: Global interleaved sequential classification.

of processors involved in the classification). In Figure 3, the first 5 requests are contiguous, satisfying this requirement.

4 Portable Parallel File System

The Portable Parallel File System (PPFS)[7] 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 4 shows the PPFS components and their interactions. Application clients invoke PPFS interface functions to initiate input/output. 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. Processors access files using a UNIX style read/write interface with individual file pointers. Each client has a local classification module that generates qualitative and quantitative information used to control local caching and prefetching policies from input/output access streams.

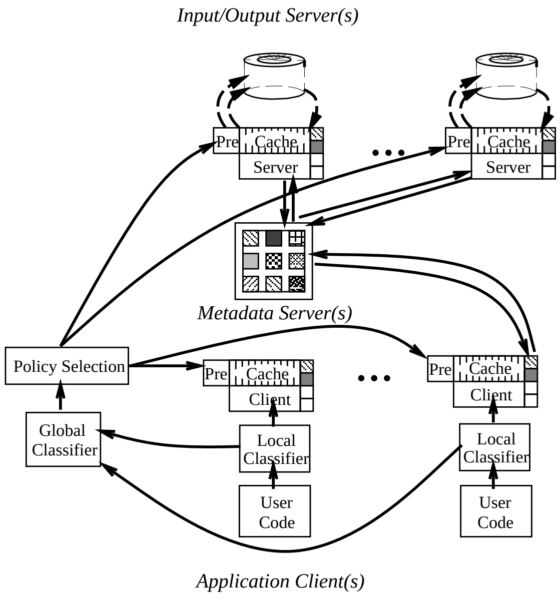

Figure 4: PPFS with global classification.

To support global classification, we added a global classification server to consolidate local classifications and necessary access statistics, as described in §3.3. When the global server classifies an access pattern, it updates policies accordingly on the input/output servers and the clients. Depending on the selected local classification method, clients can also consult the global classifier at file open for an initial global policy based on previous execution data.

5 Global Classification Experiments

To demonstrate the performance improvements from global classification, we show results from benchmarks and parallel applications from the Scalable I/O Initiative application suite on the Intel Paragon XP/S. We used PPFS as an intermediary between the application requests and the underlying Intel PFS file system [8].

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 and have limited control over input/output node buffering. In our hybrid system, our access pattern classification toolkit identifies access patterns and automatically chooses appropriate PFS file modes.

The PFS modes we manipulate below are M_UNIX (the default, atomic UNIX style input/output), M_RECORD (efficient access for interleaved sequential accesses), M_ASYNC (which does not preserve input/output atomicity) and M_GLOBAL (for global access patterns, where all processors read the same file bytes). Despite recommendations for matching PFS mode to pattern, the choice of "optimal" Intel mode depends upon other factors, such as request size, input/output configuration, and the number of processors doing input/output.

PFS also supports buffering on the input/output nodes. The default buffering strategy, intended for requests larger than 64 KB, is to disable buffering. However, buffering can be enabled on a per file basis to improve performance for smaller input/output requests.

Using global classification, we automatically select optimal PFS input/output modes and buffering strategies and compare performance to the default mode. Our motivation for this approach is not to optimize PFS performance, but rather to demonstrate the feasibility and importance of timely access pattern detection in conjunction with the well-defined set of policies. Emerging APIs like high-level MPI-IO interface and the Scalable I/O low-level interface provide a much richer set of controls, enabling a larger set of future optimizations using access pattern classifications.

5.1 Access Pattern Benchmarks

To explore the responsiveness and overhead of our automatic classification framework, we conducted a series of parallel experiments to automatically classify global sequential and global interleaved access patterns. PPFS selects PFS mode M_GLOBAL when the global access pattern is read only global sequential, and mode M_RECORD when the global access pattern is read only interleaved sequential. We vary the request size and number of processors. The PFS

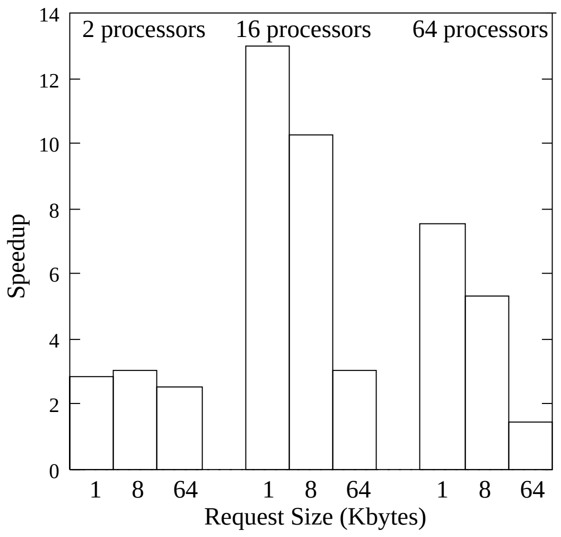

Figure 5: Global interleaved sequential speedups (PPFS classification to select PFS M_RECORD vs. M_UNIX).

configuration has 16 input/output nodes, each controlling a 4 GB Seagate disk.

Figures 5 and 6 show the speedups obtained by using the PPFS library and classification framework to automatically select an appropriate PFS mode during execution, as compared to the default mode M_UNIX. The input file sizes are 120 MB for the interleaved access pattern and 20 MB for the global sequential access pattern, large enough to ensure time to classify the global access pattern.

In general, when a file system is given information that an access pattern is global interleaved sequential, it can parallelize accesses, prefetch file data sequentially, and utilize cache space more effectively. In PFS, mode M_RECORD improves performance of interleaved input/output access patterns by guaranteeing that processors will read fixed length records stored in node order, allowing the file system to parallelize accesses and optimize for interleaved sequential file access.

In file systems that decluster file blocks, global sequential access patterns often perform poorly because processors contend for the same disk. Using a global sequential classification, we can select PFS M_GLOBAL, which alleviates this bottleneck by synchronizing requests across all

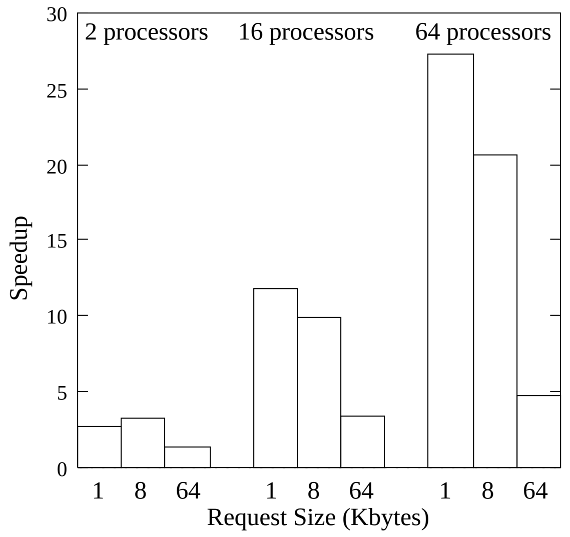

Figure 6: Global sequential speedups (PPFS classification to select M_GLOBAL vs. M_UNIX).

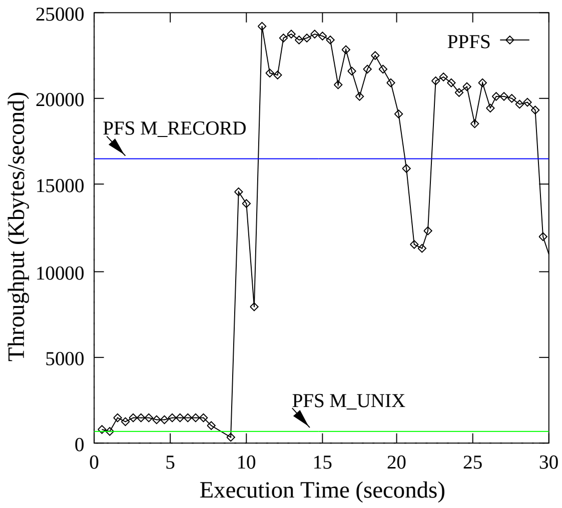

Figure 7: Global interleaved access pattern, 16 processors, 8 KB request size.

processors and having one processor perform input/output on behalf of all.

Figure 7 shows dynamic throughput, computed over windows of length 0.5 seconds, for the global interleaved sequential benchmark running under PPFS with automatic classification, compared to the sustained throughput obtained from PFS with modes M_RECORD and M_UNIX. After the global classifier obtains local classifications from the individual processors, it combines them to form a global classification of interleaved sequential, and updates the local processes with the new policy selection (M_RECORD). Because of the underlying PFS implementation, the new mode cannot be selected immediately; all processors must synchronize at the same position in the file to change the mode. Our PPFS implementation synchronizes at the next classification point (every ten accesses) to change the input/output mode. Once M_RECORD is selected, PPFS throughput increases to that obtained by the native PFS less the overhead of PPFS library calls.

5.2 QCRD

QCRD [24, 14] is a quantum chemical reaction dynamics code used to study elementary chemical reactions. The code uses the method of symmetrical hyperspherical coordinates and local hyperspherical surface functions to solve the Schrödinger equation for the cross sections of the scattering of an atom by a diatomic molecule. 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.

QCRD has five qualitatively similar phases; we limit our analysis to the second phase of QCRD, using a moderately sized data set. In phase two, 64 processors collectively read 13 matrices (written in phase one). All processors read the first 2412 bytes of each matrix and coordinate to read the remainder with a global interleaved sequential access pattern; most matrices are read twice without reopening. The output of this phase is 12 smaller matrices, written using a global interleaved access pattern.

Because of the initial global reads, mode M_ASYNC, which has performance comparable to mode M_RECORD, is the best mode for all file access patterns. Furthermore, because the request sizes are small (2400 KB), enabling I/O server buffering further improves performance.

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

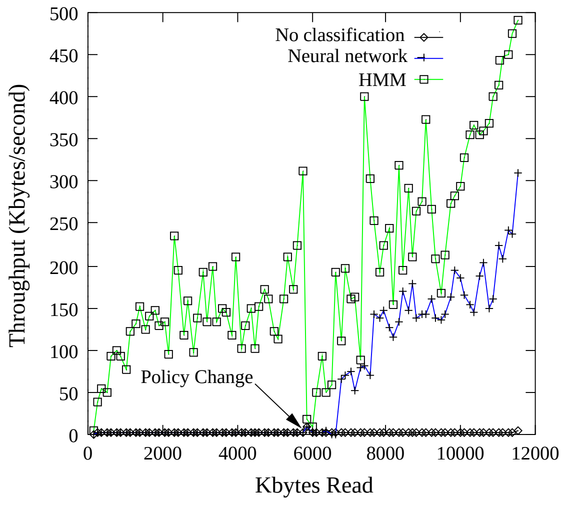

Figure 8: QCRD phase two throughput.

nodes, each controlling a 4 GB Seagate disk.

Figure 8 shows the total throughput, calculated every 128 KB, for one of the input matrices. Using the two classification methods described in §3.2, the artificial neural network (ANN) and hidden Markov model (HMM) classifications, we obtain classifications for each file and change policies as soon as the classification is available. Using ANN classification, each processor must locally classify a window of ten accesses before the global classifier can determine that the pattern is globally interleaved. Using HMM classification, the global classification determined from previous execution data is available at file open.

Throughput using mode M_UNIX (no classification) is poor because file operations are serialized. The ANN classifier switches to mode M_ASYNC approximately halfway through execution, dramatically improving throughput. The lag between the policy change and the performance improvement is due to the synchronization overhead of 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. However, throughput even with improved initial policies dips briefly midway through file lifetime, corresponding to when the file is rewound and reread.

Table 1 shows the input/output and execution times for this phase of QCRD. The most substantial performance improvement comes from recognizing that the global file access pattern is interleaved and selecting mode M_ASYNC. A secondary performance improvement is possible using the observation that the global average request size is small. A global interleaved sequential pattern with requests smaller than the stripe unit size will benefit from caching at the input/output nodes, because several processors will request disjoint segments from the same block. However, for large interleaved requests, the input/output server cache may begin thrashing, degrading input/output performance, making input/output server buffering a poor policy choice.

Early classification, exploiting the fact that input/output access patterns in this application are the same across program executions, affords the best performance. There are too few accesses to each output file to classify their patterns using our implementation of ANN classification, and the ability to select the optimal policies for the input files when they are opened, rather than during execution, reduces total read time by an order of magnitude.

5.3 PRISM

PRISM, a computational fluid dynamics code, is a parallel implementation of a 3-D numerical simulation of the Navier-Stokes equations [4, 5]. 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.

This kind of workload (global sequential access) causes a bottleneck at the input/output servers, which are overrun with identical requests from each processor. Because PFS files are striped across the available input/output nodes, increasing the number of disks does not parallelize input/output and does not improve performance. Because this bottleneck is prominent even in very small configurations, we present results from executing PRISM with a small dataset on 16 processors of an Intel Paragon XP/S with one input/output node controlling a single RAID.

Classification	PFS Mode	Buffering	Read Time	Write Time	Execution Time
None	M_UNIX	Off	51810.79	4908.84	1144.42
ANN	M_ASYNC	Off	42717.46	5664.19	996.148
ANN	M_ASYNC	On	34503.84	5150.02	858.35
HMM	M_ASYNC	Off	2478.51	433.31	284.20
HMM	M_ASYNC	On	1558.31	286.05	263.92

Table 1: QCRD phase 2 input/output times (seconds).

We focus our analysis on one initialization file that is read twice sequentially during startup by each processor. ANN and HMM classification classify the access pattern as global sequential. There are two optimizations PPFS supports given this classification.

The first possible policy selection is to select PFS mode M_GLOBAL, which synchronizing processors making identical input/output requests, internally issuing only one request for all processors and broadcasting the result to the others. Because the initialization file is read using UNIX buffered input/output, the input/output request size is 8 KB and synchronization occurs at every global read.

Another common global optimization to improve read performance of small initialization files is to have one processor read the entire file when it is opened and broadcast it to the others. We implemented this policy in conjunction with a local default policy selection of caching with 2 64 KB blocks, sufficient to retain the 127394 byte file. The block size is selected for input/output efficiency (it is the PFS stripe unit size). User-level buffering eliminates the synchronization overhead of M_GLOBAL and provides the best performance, however, this optimization can be selected only if the classification is known at file open.

Figure 9 shows average per processor throughput, calculated per 8 KB block, for access to one initialization file using the default M_UNIX PFS input/output mode (no classification), using HMM and ANN classification to select mode M_GLOBAL, and using HMM classification to enable caching and have processor zero read the file and broadcast it to the other processors.

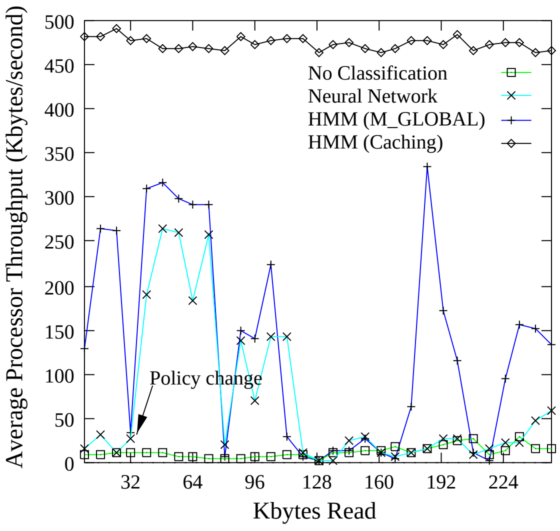

Figure 9: PRISM initialization file throughput.

Throughput using the default mode M_UNIX is very poor; in the PFS implementation of this mode nodes may not read the file in parallel. 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 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

Experimental Environment	Open Time	Read Time	Total
PPFS (no classification)	129.185	430.242	559.427
PPFS (ANN)	25.019	339.892	364.911
PPFS (HMM-M_GLOBAL)	28.315	263.363	291.678
PPFS (HMM-Caching)	238.56	8.416	246.976

Table 2: PRISM input/output times (seconds) for one initialization file.

the first measurement under the new policy. The policy change occurs approximately two seconds into input/output execution.

In contrast, the HMM classifier queries the global classifier for a recommended mode at file open. The global classifier computes an initial policy recommendation based on the classifications of previous executions. We use this classification to select mode M_GLOBAL or have one processor read the file and broadcast it to the others. Throughput is greatest using the second approach.

Table 2 shows the total input/output times for the four experiments. There is a large variance in the total open time because the open call is synchronizing. However, when HMM classification is used to select a policy of reading and broadcasting the initialization file, the open time reflects this significant overhead.

6 Related Work

Given the natural variation in parallel input/output patterns revealed by input/output characterization studies [22], tailoring file system policies to application requirements can provide better performance than a uniformly imposed set of strategies. Many studies have shown this under different workloads and environments [11, 13, 3]. For example, small input/output requests are best managed by aggregation, prefetching, caching, and write-behind, though large requests are better served by streaming data directly to or from storage devices and application buffers.

Although applications with predictable resource demands can explicitly control prefetching or caching policies, this can result in poor overall resource management. The performance of an application policy selection depends upon the total system environment, information unavailable to the application. For example, aggressive prefetching when another application requires a large percentage of the cache space might cause thrashing, degrading performance for both applications. Furthermore, file system optimizations are system and architecture dependent. A programmer must have a detailed understanding of both the application input/output characteristics and the architecture to tune the file system; an optimal selection of policies may be data-dependent or simply unknown.

A portable approach, supported by flexible file systems, is to utilize an application programming interface that encapsulates access pattern information. Collective input/output [10] is one such interface, as are input/output modes in Intel PFS [8]. A similar approach is to use application hints to guide a proactive file system. Patterson *et al* demonstrate the success of providing hints to guide prefetching and caching of files that will be accessed in the future [19].

Instead of requiring the application to specify hints, access patterns, or efficient file system policies, many have looked at the possibility of using intelligent techniques to construct higher level models of file access automatically. Kotz has studied detecting more complicated access patterns that are used to guide non-sequential prefetching within a file [12]. Fido is an example of a predictive cache that prefetches by using an associative memory to recognize access patterns over time [18]. Knowledge based caching has been proposed to enhance cache performance of remote file servers [9]. Our work uses trained neural networks and statistical models to classify global access patterns, and can be trained to recognize new patterns as the need arises.

7 Conclusions

Global access pattern information is critical to optimizing input/output performance of parallel applications. Recognizing this, many file systems allow the application to specify global access patterns. This information allows the file system to select corresponding file system policies that improve input/output performance. Because manual access pattern specification places a burden on the application programmer, we have proposed a method for automatic classification of global access patterns. Experiments with parallel benchmarks and applications demonstrate the ability of global classification to improve performance by automatically selecting appropriate file system policies.

Acknowledgments

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] 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).

[3] 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.

[4] HENDERSON, R. D. *Unstructured Spectral Element Methods: Parallel Algorithms and Simulations.* PhD thesis, June 1994.

[5] 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.

[6] HINTON, G. E. Connectionist Learning Procedures. *Artificial Intelligence 40* (1989), 185 – 234.

[7] 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.

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

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

[10] KOTZ, D. Disk-directed I/O for MIMD Multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation* (November 1994), pp. 61–74. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[11] KOTZ, D., AND ELLIS, C. S. Prefetching in File Systems for MIMD Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 1*, 2 (April 1990), 218–230.

[12] KOTZ, D., AND ELLIS, C. S. Practical Prefetching Techniques for Parallel File Systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems* (December 1991), pp. 182–189.

[13] KRIEGER, O., AND STUMM, M. HFS: A Flexible File System for Large-scale Multiprocessors. In *Proceedings of the 1993 DAGS/PC Symposium* (Hanover, NH, June 1993), Dartmouth Institute for Advanced Graduate Studies, pp. 6–14.

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

[16] 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.

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

[18] 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.

[19] 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.

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

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

[22] 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.

[23] The MPI-IO Committee. MPI-IO: A Parallel File I/O Interface for MPI, April 1996. Version 0.5.

[24] 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.