Intelligent, Adaptive File System Policy Selection*

Tara M. Madhyastha Daniel A. Reed

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Abstract

Traditionally, maximizing input/output performance has required tailoring application input/output patterns to the idiosyncrasies of specific input/output systems. In this paper, we show that one can achieve high application input/output performance via a low overhead input/output system that automatically recognizes file access patterns and adaptively modifies system policies to match application input/output needs. This approach reduces the application developer's input/output optimization effort by isolating input/output optimization decisions within a retargetable file system infrastructure.

To validate these claims, we have built a lightweight file system policy testbed that uses a trained learning mechanism to recognize access patterns. The file system then uses these access pattern classifications to select appropriate caching strategies, dynamically adapting file system policies to changing input/output demands throughout application execution. Our experimental data show dramatic speedups on both benchmarks and input/output intensive scientific applications.

1 Introduction

Input/output performance is the primary performance bottleneck of an important class of national challenge applications (e.g., global climate modeling). Many, if not all such applications exhibit complex, dynamic, often irregular input/output access patterns, rather than the regular, sequential patterns for which most file systems are optimized [3]. Often, to achieve acceptable performance, the developer is forced to tune the size, order and frequency of input/output requests to match the idiosyncrasies of a specific input/output system. Not only does this place a substantial cognitive burden on the developer, but such optimizations are system-specific and may be inappropriate for other systems or input/output configurations.

Several studies have demonstrated that file system policies can exploit knowledge of application access patterns to provide higher performance than is possible with policies that are oblivious to access patterns [6, 10, 14]. For example, on detection of random input/output access patterns, an adaptive file system might eschew the caching and sequential prefetching normally used for sequential accesses.

*Supported in part by the National Science Foundation under grant NSF ASC 92-12369, by the National Aeronautics and Space Administration under NASA Contracts NGT-51023, NAG-1-613, and USRA 5555-22 and by the Advanced Research Projects Agency under ARPA contracts DAVT63-91-C-0029 and DABT63-93-C-0040.

Unfortunately, *a priori* identification of effective file system policies is difficult because application access patterns may be data dependent or simply unknown. Furthermore, input/output requirements are a complex function of the interaction between system software and executing applications and may change unpredictably during program execution.

We believe one solution to this conundrum is an adaptive file system that automatically responds to changing application input/output requirements by choosing file system policies based on observed access patterns. This approach would reduce an application developer's input/output optimization effort by isolating input/output optimization decisions within a retargetable file system infrastructure. Moreover, input/output performance could be preserved across file systems and system architectures because the file system would observe application access patterns and dynamically select policies appropriate for the current circumstances. Simply put, access classification is application specific and platform independent, whereas file system policies are both access pattern and platform specific.

To validate the thesis that major performance improvements are possible by adaptively tuning file system policies using automatic access pattern classifications, we have built a lightweight file system policy testbed. This testbed contains automatic classification techniques based on trained neural networks, a suite of possible file policies, and a decision procedure to select policies based on the access pattern classification. Experiments with both benchmarks and large scientific applications show major performance increases over those achievable with vendor file systems.

The remainder of this paper is organized as follows. In §2, we describe the structure of PPFS, the user file system used for our experimental studies. We present a summary of the mechanics of intelligent access pattern classification in §3. As described in §4, PPFS uses this classification infrastructure to identify access patterns and tune caching policies to improve input/output performance. Following this, §5–§7 describe the validation of this approach using a set of synthetic benchmarks and input/output intensive applications. Finally, §8-§9 place this work in context, summarize our results, and outline directions for future research.

2 Portable Parallel File System (PPFS)

Quantitative assessment of adaptive file system policies necessarily presumes some underlying experimental infrastructure. To minimize implementation effort while focusing attention on the salient details, we have developed a user-level portable parallel file system (PPFS) [8]. PPFS is an input/output library, portable across parallel systems and workstation clusters, with a rich interface for application control of data placement and file system policies. Below, we describe the basic structure of PPFS, its component interactions, and the extensions necessary to support automatic access pattern classification and dynamic file policy selection.

2.1 Basic PPFS Infrastructure

In the PPFS input/output model, files are accessed as either fixed or variable length records, and the PPFS library has an extensible set of interfaces for specifying file distributions, expressing input/output parallelism, and tuning file system policies. For example, one can specify how file records are distributed across input/output nodes, how and where they are cached, and when and where prefetch operations should be initiated.

As Figure 1 shows, the user-level PPFS library satisfies input/output requests via the interaction of four basic components: I/O servers, metadata servers, global caching agents, and application clients. On a workstation cluster or parallel system, the PPFS servers and global caching agents are processes that mediate requests from application tasks that are linked with PPFS interface functions. On the I/O servers, all "physical" input/output is performed through an underlying UNIX file system.

In this model, an application client opens a file by first contacting a metadata server that stores or creates information about the file storage order on the remote I/O servers. With this information, the application can specify caching and prefetching polices for local client caches, global caching agents, and I/O servers.[1] During application execution, the client caches either satisfy requests or forward them to the caching agents or I/O servers.

Experiments using large research codes on both the Intel Paragon XP/S and IBM SP-2 have shown that tuning PPFS file system policies to application needs, rather than forcing the application to use inappropriate and inefficient file access modes, is the key to performance [3]. In short, simple access pattern hints and cache policy controls can yield large performance benefits [8]. However, to achieve these performance gains with PPFS, the application writer must understand both the application access pattern and the PPFS input/output cost model to specify appropriate policy controls and file data distributions.

2.2 PPFS and Dynamic Adaptation

As designed and implemented, the PPFS interface provides a rich set of manual file system policy controls and structured data access functions. However, to achieve high performance, the application writer must understand both the application access pattern and the PPFS input/output cost model to specify appropriate policy controls and file data distributions. As we argued earlier, the file system should *automatically* infer appropriate policies from low-level application access patterns, lessening the application programming burden and the likelihood of user misconfiguration. This goal motivated two major extensions to the original PPFS design: automatic access pattern classification and an adaptive file system policy selection mechanism.

The basic PPFS infrastructure supports a general record-oriented file access model, with both fixed and variable length records. To simplify the problem of automatic access pattern classification, we added PPFS support for the simpler UNIX model of unstructured byte streams.[2] In the byte stream model, each file access is completely specified by a byte offset, the operation type, and the size. Using the newly developed PPFS UNIX file interface, we augmented the manual PPFS file system controls with an adaptive access pattern classification scheme and a file system policy selection mechanism.

Figure 2 illustrates the interaction of these components with the original PPFS structure. An access pattern classifier and statistics module replace obviate the user specification of file policies and parameters. The statistics module monitors the byte-oriented access stream to generate the statistics needed by the access pattern classifier. The file policy selection mechanism then uses this classification to select and tune prefetching and caching policies.

[1] Clients, global caching agents, and I/O servers each have their own caches and configurable prefetch engines.
[2] We anticipate extending our classification scheme to the full set of PPFS record access methods.

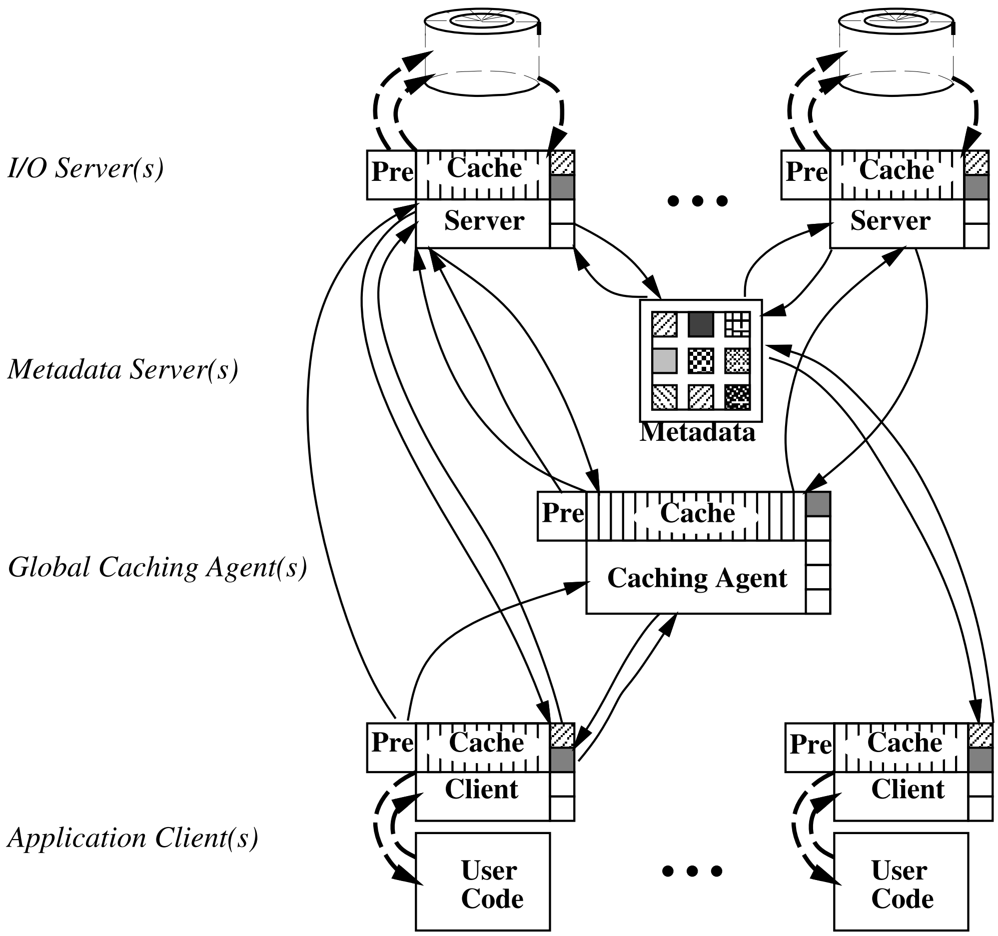


Figure 1: Basic PPFS Infrastructure

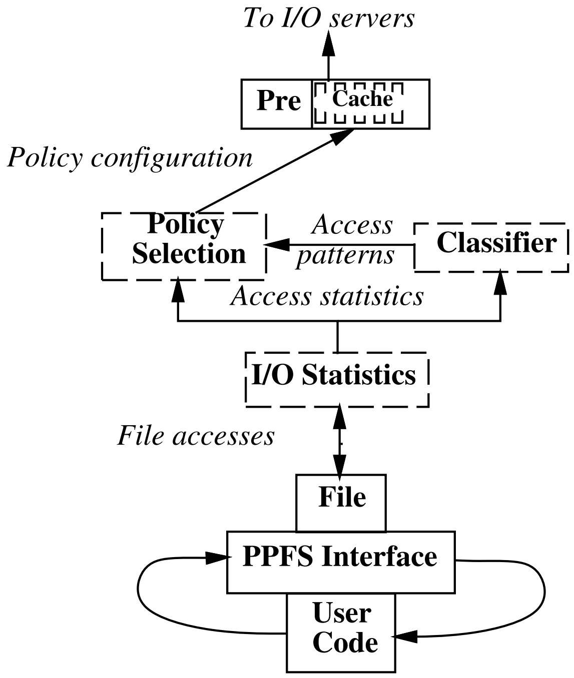


Figure 2: PPFS Classification and Policy Selection Extension

3 Automatic Access Pattern Classification

Abstractly, a file access pattern is a qualitative description of future accesses that could be used to tune file system policies. However, most current file systems are optimized for a small number of common access patterns (e.g., sequential, read only or write only) and implement policies appropriate for these regardless of the actual access patterns.

To be effective, a file access pattern description need not be a perfect predictor of future accesses; mere statistical correctness often suffices. For example, one might classify an input/output pattern as "sequential and read only" even if there are occasional file seeks and writes — this would suffice to correctly choose a sequential prefetching policy. Such a qualitative description is difficult to obtain based on heuristics alone. Instead, one needs a general classification methodology that can adapt to exigencies (e.g., the need to refine the partition of possible access patterns).

3.1 Access Pattern Partitions

Based on our ongoing characterization of scientific application input/output patterns as part of the Scalable I/O initiative [3, 15], we partitioned access patterns based on three broad features: read/write mix, sequentiality, and request size; see Table 1. This ternary partition is not unique, and other useful partitions can be added (e.g., inter-request latency). However, these features define a wide range of common scientific access patterns, and, as we shall see in §7, suffice to effectively select and tune several file policies.

The read/write patterns of Table 1 qualitatively describe the mix of input/output operations to a particular file. Of these, the read only and write only patterns are both self-explanatory

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

and frequently occurring. Superficially, the read-update-write pattern, where a file region is read and then rewritten, might seem excessively baroque. However, our characterization studies [3, 17] showed that this pattern occurs quite frequently in scientific codes.

The second category, sequentiality, describes access orders. In addition to sequential accesses, strided accesses to large, out of core arrays are a frequent component of scientific computations. A one-dimensionally strided pattern is non-decreasing access of non-contiguous file portions that are separated by a unique, nonzero distance. Two-dimensionally strided patterns are similar, but with two unique distances between strides.

Finally, request sizes qualitatively describe the relative size of accesses. Here, we opted to partition requests only based on their uniformity. A finer gradation might include small, medium, and large sizes, making possible selection of file policies based on expected cache sizes and memory copy overheads.

3.2 Neural Network Classification

Although many techniques have been proposed to classify and identify observed patterns, most sophisticated learning techniques attempt to generalize from examples and detect common features. To automatically classify file access patterns, we have developed a feedforward artificial neural network [7] capable of producing a qualitative access pattern description. The inherent imprecision of neural networks makes it possible to usefully classify patterns that are "very close" to a well-defined pattern (e.g., a pattern close to sequential can still be classified as sequential). This is an advantage, allowing a file system to learn, through training, how it should classify new access patterns.

To train the neural network to recognize the access patterns of Table 1, we compute input/output statistics on small fixed-size windows of the raw access patterns. These statistics include the request type, size, and byte offset. Although neural network training is expensive, once training is complete, the network's classification is very efficient, producing outputs that correspond directly to the features in Table 1.

The neural network output is a vector of ten bits, where each bit indicates the presence or absence a feature in Table 1. One and only one bit is set for each of the three categories (i.e., a set of accesses cannot be both read only and write only). If this condition is not met, no classification is made.

4 Intelligent Policy Specification

As Figure 2 shows, file access statistics and file access pattern classifications provide the qualitative and quantitative data needed to select and tune file system policies; each plays an important role. The qualitative classification of the access pattern is system independent but may be unique to a particular application execution (e.g., if the access pattern is data dependent); it provides the high-level data needed to choose file policies suited to the observed access pattern. The complementary quantitative data provides the measures of overhead necessary to configure the selected file policies for a particular hardware configuration.

4.1 Implementation Alternatives

Together, quantitative and qualitative data make possible dynamic selection and configuration of file policies based on access pattern characteristics. By choosing policies to match each application's needs, a file system should provide higher performance than by enforcing a single system-wide policy. Moreover, if the policy selection and configuration mechanism is configurable, one can optimize the file system on a new system or hardware configuration simply by replacing the file policy decision procedures.

Application specification of resource management policies has proven valuable in many domains, and many modern microkernels now support user control or extension of operating system services (e.g., SPIN [2] and *exokernel* [4]). We have adopted a similar approach in PPFS; a user-specified code fragment defines the set of possible PPFS file management policies and an algorithm for choosing among those policies based on the access pattern classification. Below we describe a simple policy selection algorithm to illustrate this general approach.

4.2 Algorithmic Selection

Abstractly, PPFS continuously monitors and classifies the input/output request stream. This classification is passed to the user policy suite for file policy selection and configuration. For example, when the access pattern is classified as sequential and read only, the user policy suite might prefetch aggressively. Likewise, when the access pattern is regularly (1-D or 2-D) strided, the policy suite might prefetch anticipated blocks based on the stride size and adjust the cache size to increase the hit ratio.

Figure 3 shows a simple, parameterized example of the kinds of policy decisions that are possible given qualitative and quantitative access pattern information. In this example, the default cache block replacement policy is LRU, and default cache and block sizes are chosen to give good performance on small sequential reads. Policy parameters are adjusted whenever they might potentially yield performance improvements given the access pattern classification. Quantitative values for the parameters of Figure 3 depend on the particular hardware configuration and must be determined experimentally.

The algorithm of Figure 3 is but one simple possibility for policy control. A much richer control structure could be built given access pattern information and an even more accurate model of input/output costs. However, in §5 and §7 we show that even this simple policy suite suffices to yield large performance increases over that possible with standard UNIX file policies.

```
if (sequential) {
      if (write only) {
         enable caching
         use MRU replacement policy
      } else if (read only && average request size > LARGE_REQUEST)
        disable caching
      } else {
          enable caching
          use LRU replacement policy
      }
}

if (variably strided || 1-D  strided || 2-D strided) {
     if (regular request sizes) {
         if (average request size > SMALL_REQUEST) {
              disable caching
         } else {
            enable caching
            increase cache size to MAX_CACHE_SIZE
            use LRU replacement policy
         }
     } else {
         enable caching
         use LRU replacement policy
     }

}
```

Figure 3: Dynamic File Policy Selection (Example)

5 Access Pattern Benchmarks

To quantify the overhead for automatic access pattern classification and dynamic policy selection, we conducted a series of performance studies using both simple benchmarks with a single fixed access pattern and more complex benchmarks with time-varying access patterns.[3] Each is briefly described below.

5.1 Regular Access Patterns

As a first assessment of PPFS performance and overheads, we constructed a set of simple sequential, strided, and random access pattern benchmarks. Each benchmark read or wrote a 20 million byte file in fixed size records that ranged from 20 to 200 thousand bytes. Figure 4 shows the PPFS speedups, relative to UNIX buffered input/output, for these benchmarks.

For sequential accesses, the performance of PPFS is roughly that of UNIX because both systems use similar policies, buffering small input/output requests and flushing larger requests. In contrast, PPFS performance is substantially higher for the strided and random access patterns than that achieved by UNIX.

The strided benchmark views the file as a two-dimensional array of records with varying x and y dimensions. Records are stored in row-major order and are accessed in column-major order, along the y-axis, creating a strided access pattern. By the algorithm of Figure 3, if the record size is large (e.g., greater than 8K bytes for this benchmark), caching is disabled; otherwise, the cache size is increased. In most cases, this allows PPFS to buffer blocks that will be re-read or to consolidate writes if requests are small.

Finally, for the random access pattern, PPFS either enlarges the file cache if the average record size is small, or disables caching if it is large. As with the regularly strided benchmark, this policy change allows PPFS to buffer small reads or writes and minimize memory copy costs for large accesses.

5.2 Time-Varying Access Patterns

The regular access pattern benchmarks of §5.1 showed that the overhead for real-time access pattern classification and policy selection was modest and that careful matching of policies and policy parameters to access patterns could yield performance greater than that achievable by a single UNIX file system policy. Moreover, by continuously ensuring that file system policies are suited to the access pattern, PPFS can respond to application needs dynamically throughout program execution.

To illustrate how dynamic adaptation can a performance advantage, we constructed a benchmark that reads the first half of a 40 million byte file sequentially and the second half randomly. For both access patterns, the request size was 2000 bytes. We executed this benchmark twice using the PPFS file system, once with adaptive policy selection, and once with a single, default system policy.

Figure 5 shows the read durations for this benchmark with and without PPFS policy adaptivity. The sequential reads complete in approximately 3.38 and 3.4 seconds, respectively, using a 1 MB cache with 128 KB cache blocks and an LRU cache block replacement policy. In Figure 5(a) the

[3] All experiments were conducted on a Sun SPARC 670 running SunOS 4.1.3.

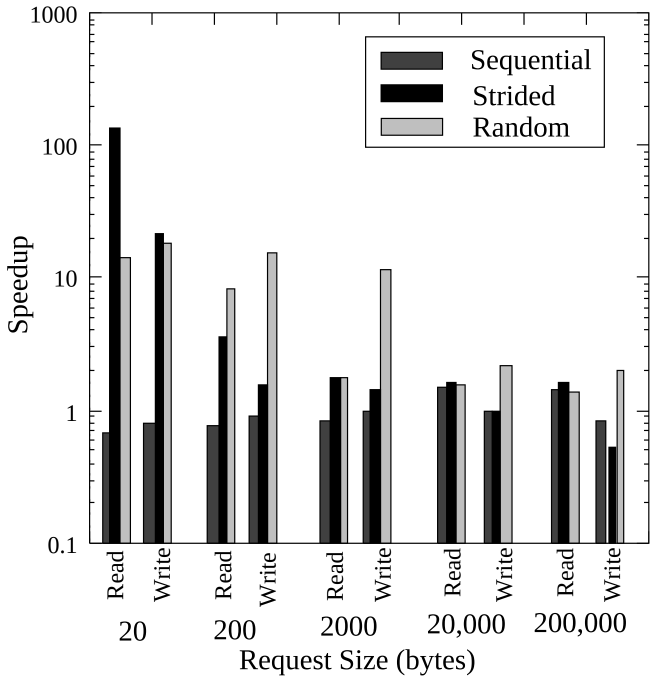
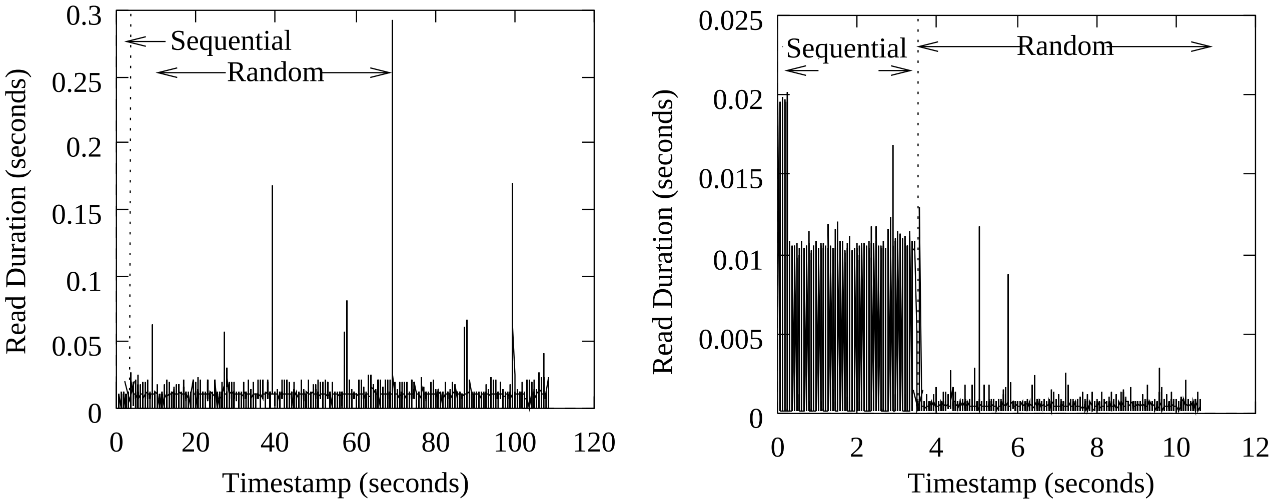


Figure 4: Synthetic Benchmark PPFS Speedups Relative to UNIX

same policies are used for both the random and the sequential reads. In contract, Figure 5(b) shows the dramatic improvements possible when the access pattern change is automatically detected and caching is disabled for the random access pattern.

Figure 6 shows detailed read durations at the point where the access pattern changes from sequential to random. In Figure 6(a) the transition between sequential and random access occurs at 3.4 seconds. In this execution, PPFS policies are never modified for the random access pattern, and almost every random read incurs a cache miss. In Figure 6(b) the transition occurs at 3.38 seconds. One can see that the read times in the sequential access pattern alternate between one large access time and several very small ones. The file is cached in blocks of 128 KB, but reads are only 2000 bytes; each spike represents a cache miss. In the period between 3.38 and 3.455 seconds the first ten random accesses occur; every access is a cache miss and reads are very inefficient. When this pattern is detected, caching is disabled and read performance improves; however, the average read time for random access is still larger than that for sequential access.

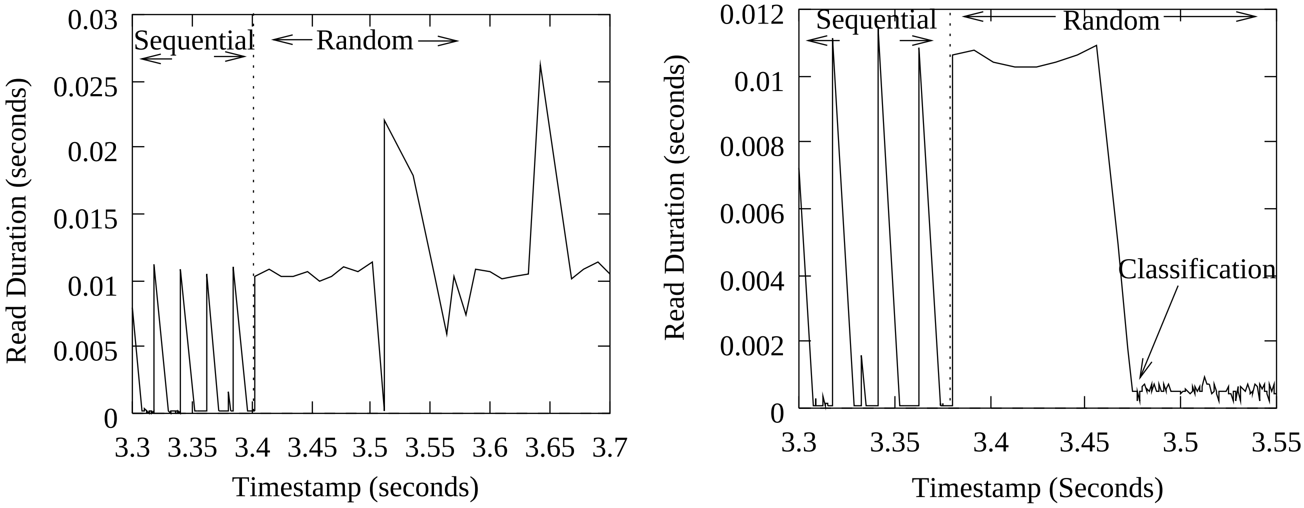
The results of these benchmarks demonstrate that automatic classification and adaptive file system policies provide more uniform performance over a wider range of access patterns, and in some cases yield dramatic performance improvements. In §6, we describe two large scientific codes that benefit from these adaptive techniques.



(a) Non-adaptive

(b) Adaptive

Figure 5: Sequential and Random Access Comparison



(a) Non-adaptive

(b) Adaptive

Figure 6: Sequential and Random Access Comparison (Detail)

6 HDF and Satellite Data Processing Applications

The NCSA Hierarchical Data Format (HDF) [12] is a self-documenting physical file format that facilitates data transfer between machines and operating systems. It contains interfaces for storing and retrieving many types of scientific data, including images and n-dimensional data sets, together with information about the data.

The HDF application interface routines were originally designed to be simple and machine independent, with little expectation that an HDF file would be accessed many times during program execution. Based on this use model, the HDF library implementation is stateless. Each time an HDF file is accessed, the underlying UNIX file first must be opened, all the metadata is then read, and finally the desired data is read or written. Therefore, writing data to an HDF file involves a file open, a sequence of alternating reads and seeks, and finally, the actual write.

HDF input/output performance is particularly important because HDF is the standard data format for all NASA EOS (Earth Observing System) data products. The goal of the Earth Observing System Data and Information System (EOSDIS) [1] is to manage data from NASA's Earth science research satellites, providing researchers with a uniform information resource that can be used to assess global change. The collective data rate from satellites in this project is expected to rise as high as 500 terabytes per day by the year 2000. It is essential that manipulation of HDF files be as efficient as possible to meet this data processing challenge.

Among the problems facing EOSDIS is the need to reprocess large multi-year data sets derived from satellite instrument measurements. Satellites periodically send "raw data" back to ground stations. This raw data is archived and processed to produce a set of data products by combining subsets of raw data from different instruments in various ways, or by combining low level products to get higher level products (e.g., averages over time). As scientists refine algorithms for data processing after the satellite has been in orbit for some time, they reprocess the archived raw data to generate more accurate data products.

Below, we describe two typical applications used to process low-level satellite data: Pathfinder, from the NOAA/NASA Pathfinder AVHRR (Advanced Very High Resolution Radiometer) data processing project, and `l0to1a`, from the Sea-viewing Wide Field-of-view Sensor (SeaWiFS) project. Both applications exploit the HDF library to generate data products, albeit in decidedly different ways.

6.1 Pathfinder

The goal of the Pathfinder project is to process existing data to create global, long-term time series remote-sensed data sets that can be used to study global climate change. Although there are four types of Pathfinder AVHRR Land data sets (daily, composite, climate, and browse images) for simplicity, we consider only the creation of daily data sets. Each day, fourteen files of AVHRR orbital data, approximately 42 megabytes each, in Pathfinder format are processed to produce an output data set that is approximately 228 megabytes in HDF (SDS) format.

During Pathfinder execution, ancillary data files and the orbital data file are opened, and an orbit is processed 120 scans at a time. Although the orbit file is accessed sequentially, the access patterns for other ancillary data 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 to create

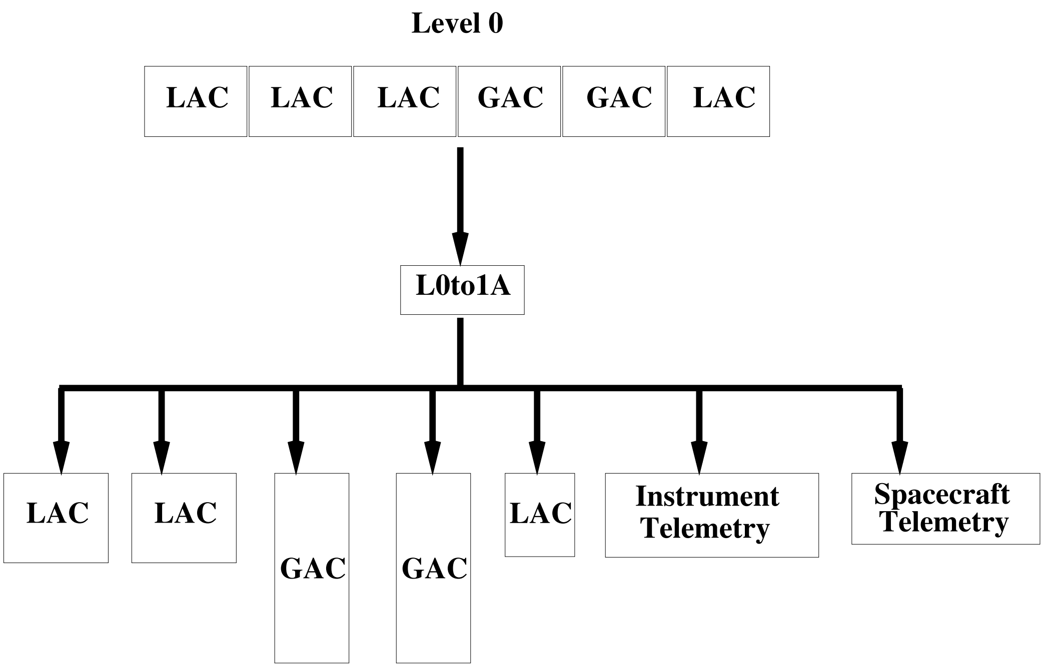


Figure 7: Overview of SeaWiFS Level-0 to Level-1A Processing

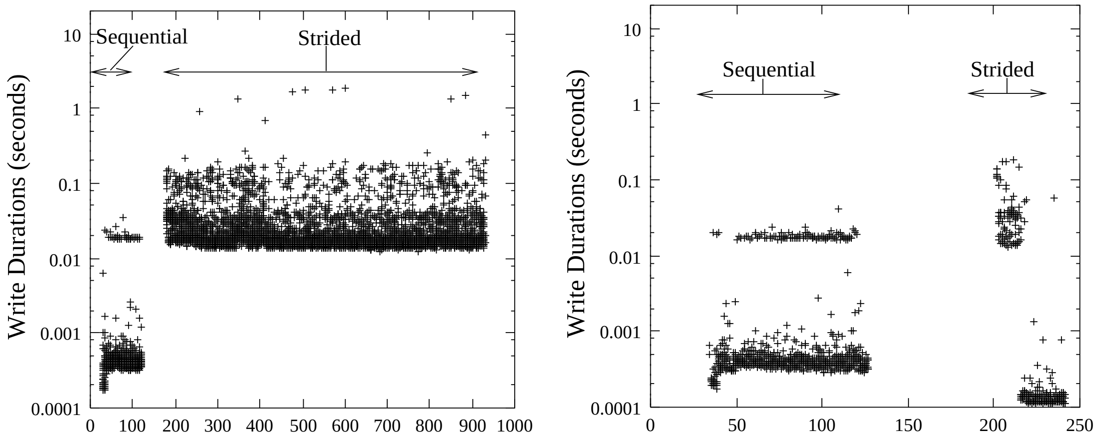
three 8-bit and nine 16-bit layers. Based on our input/output analysis, over seventy percent of the Pathfinder execution time is spent in UNIX input/output as part of the HDF library routines.

6.2 SeaWiFS `l0to1a` Translation

The second application is a preliminary version of the raw satellite data to HDF format translation program (**`l0to1a`**) from the Sea-viewing Wide Field-of-view Sensor (SeaWiFS) mission on a simulated data set. The goal of the SeaWiFS project is to provide a five year data set of global ocean color data to improve our understanding of the role of ocean primary production in the global carbon cycle [5].

The **`l0to1a`** application from the SeaWiFS processing software converts raw (level 0) satellite data to level 1A data products. These data products contain unprocessed 4 km by 4 km resolution Global Area Coverage (GAC) and 1 km by 1 km resolution Local Area Coverage (LAC) data; all spacecraft and instrument telemetry data is retained in raw form. In addition, instrument telemetry, selected spacecraft telemetry, and geolocation data are reformatted and appended; Figure 7 shows a diagram of this process.

During program execution, the level 0 file is first read in segments of 1 KB (1 frame) and all HDF output files are created with metadata and some file level data. The instrument telemetry files are created at this time. After creating the initial level 1A HDF files, the level 0 file is rewound and rescanned. In this pass, one or more frames may be processed at a time to create the actual level 1A data.



(a) PPFS (Non-adaptive) (b) PPFS (Adaptive)

Figure 8: Pathfinder Write Durations (Sun SPARC 670)

7 Application Experiments

To validate the promising benchmark results of §5, we retargeted the Pathfinder and SeaWiFS applications to use the adaptive version of PPFS. During execution, PPFS automatically classified the access patterns and selected file policies using the algorithm of Figure 3.

7.1 Pathfinder Input/Output

To convert Pathfinder to use PPFS, its buffered UNIX input/output calls were replaced with the PPFS library equivalents. Only the conversion of the temporary output file to HDF format was left unchanged (using buffered UNIX input/output), because that phase accounted for such a small fraction of the overall execution time (approximately 3.35 percent).

Table 2 shows the relative execution times for Pathfinder using UNIX buffered input/output and PPFS on a Sun SPARC 670. The dynamic adaptation of PPFS yields a speedup of approximately 1.87 with the policies of Figure 3.[4]

During Pathfinder execution, the PPFS automatic classifier detected that the output file access pattern was initially write only and sequential, with large accesses, and that the pattern later changed to write only, strided, with very small accesses. In consequence, PPFS chose an MRU cache block replacement policy for the first phase. In the second phase it enlarged the cache, retaining the working set of blocks. In contrast, UNIX buffered input/output forced a write of 8 KB for every one or two byte access.

Figure 8 illustrates the dramatic benefits of dynamic policy adaption for Pathfinder's execution. The first cluster of accesses is the write only sequential phase. Performance for the first phase is roughly equivalent using either MRU or the default, non-adaptive LRU replacement policy. However, enlarging the cache in the second phase substantially decreases the average write duration.

[4] We did, however, disable caching for small, variably strided reads.

Experimental Environment	System Time	User Time	Total	Total Instrumented
UNIX	1578.2	1781.1	4299.3	6201.6
PPFS	400.4	1270.4	2300.8	4054.4

Table 2: Pathfinder Execution Times (seconds)

Experimental	Read		Write				
Environment	**Count**	**Bytes**	**Count**	**Bytes**	**Lseek**	**Open**	**Close**
UNIX	3,030,382	24,824,700,000	4,077,265	625,698,239	10,961,293	41	41
PPFS	3,957,852	629,901,726	3505	766,433,994	3,897,144	42	42

Table 3: Pathfinder Input/Output Operations

Table 3 shows the number of system-level input/output operations and bytes read and written for the PPFS and UNIX Pathfinder executions. PPFS decreases the total number of bytes read by two orders of magnitude.[5] Similarly, the number of writes declines by three orders of magnitude via PPFS's buffering of small writes. This write caching is extremely successful with a hit ratio exceeding 0.99.

Finally, Figure 9 shows a histogram of the Pathfinder write times, obtained by instrumenting input/output calls with the Pablo instrumentation library [16]. Each bar in the UNIX histogram represents the time spent performing buffered UNIX input/output. Likewise, each bar in the PPFS histogram reflects the time spent in the PPFS input/output. The success of PPFS caching redistributes physical writes, changing the shape of the write time histogram. In the PPFS version, less time is spent in other input/output operations (reads and opens) and is used for writes instead.

Because of the high overhead for obtaining timestamps, the software instrumentation overhead stretches the execution time. Thus, the times in Figure 9 correspond to the instrumented times of Table 2.

7.2 SeaWiFS Input/Output

As with the Pathfinder code, we retargeted the SeaWiFS `l0tol1a` processing program to use PPFS. All HDF library calls (for writing and access to output files) use PPFS to perform input/output. For simplicity, all other input/output (the reading of the input file) was left unchanged, using standard UNIX buffered input/output.

Table 4 shows the SeaWiFS `l0tol1a` execution times for both UNIX and PPFS, with operation counts and data volume shown in Table 7.2. As with the Pathfinder code, PPFS access pattern classification and dynamic policy selection reduces total execution time and dramatically reduces the number of input/output calls.

[5] The number of PPFS reads increases slightly because tiny variably strided reads are not buffered in PPFS. Each time a buffered UNIX read call is issued by the PPFS library and the bytes are not in the current buffer, a new 8 KB block is read.

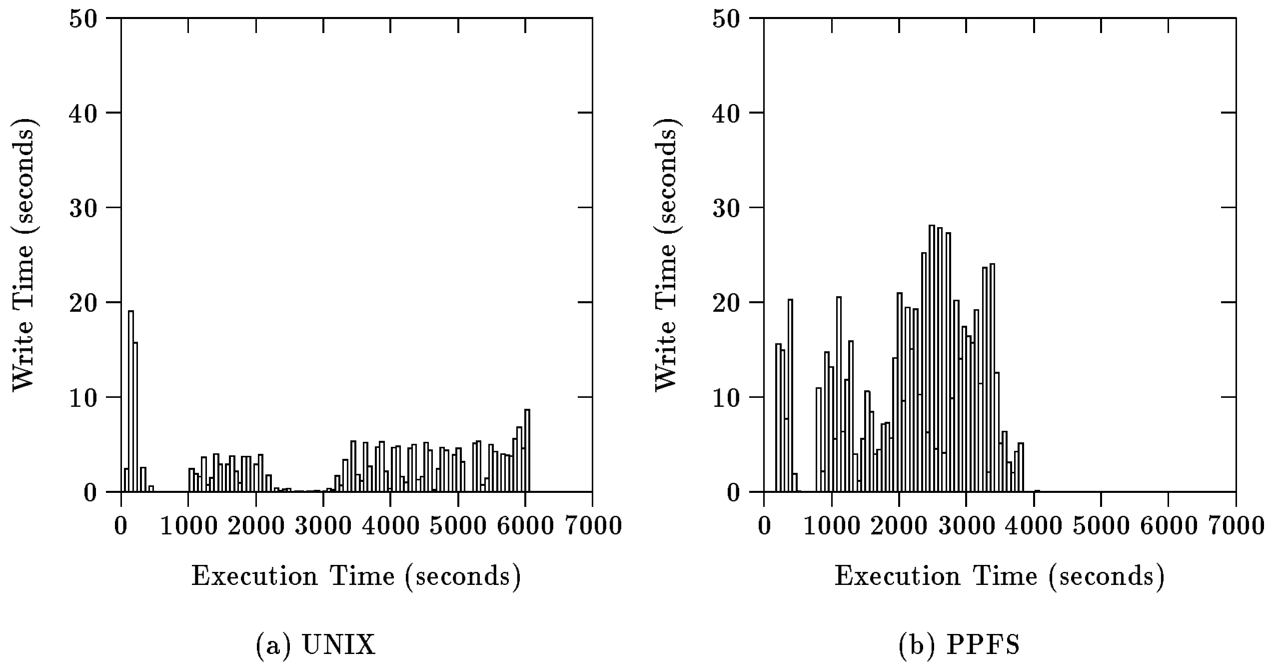


Figure 9: Pathfinder Write Duration Histograms

Experimental Environment	System Time	User Time	Total	Total Instrumented
UNIX	74.4	195.1	327.8	443.6
PPFS	26.5	140.9	180.8	290.2

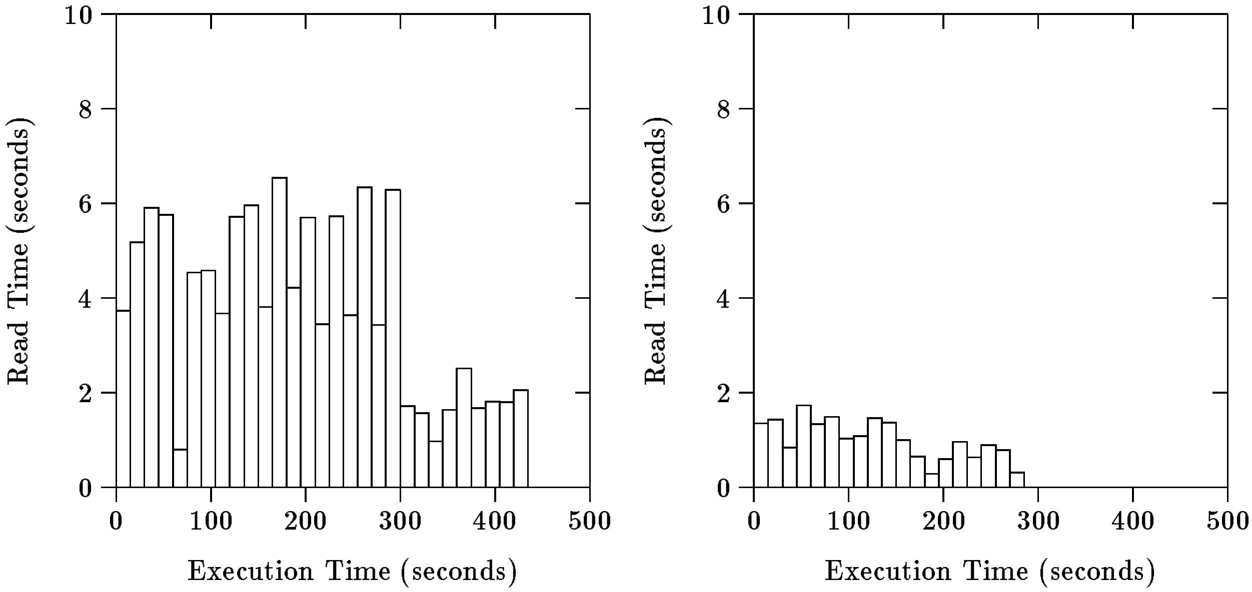
Table 4: SeaWiFS `l0to1a` Execution Times (seconds)

Two PPFS optimizations are responsible these reductions. First, PPFS correctly classifies the access pattern for each file as either read/write no-update or write only with irregular request sizes and stride sizes. Based on the algorithm in Figure 3, PPFS then uses its default cache size and an LRU cache replacement policy for all files. Due to the large PPFS cache, a fraction of the HDF-generated file seeks and writes are captured, reducing the input/output costs. Thus, the hit ratio for cached files in `l0to1a` is 0.116 (i.e., 11.6 percent of all requests are satisfied from the cache). This improvement over the UNIX buffered input/output policies is responsible for a portion of the performance improvement.

The second factor contributing to speedup is the virtual file interface of PPFS, which allows file data to remain in user memory between file opens and closes. A large percentage of time in the `l0to1a` code is spent opening and closing the files for every access. In practice, this is unnecessary. PPFS implements virtual file opens and closes, capturing a true file open/modification/close sequence and maintaining a file cache across logical file access boundaries. This cache is particularly important because the offsets to data blocks are stored at the head of an HDF file.

Experimental	Read		Write				
Environment	**Count**	**Bytes**	**Count**	**Bytes**	**Lseek**	**Open**	**Close**
UNIX	183,350	1,305,612,282	71133	107,941,938	392,928	5745	5766
PPFS	23,601	210,176,176	38770	114,290,111	1547	63	83

Table 5: SeaWiFS `l0to1a` Input/Output Operations



(a) UNIX

(b) PPFS

Figure 10: SeaWiFS l0to1a Read Duration Histograms

Finally, Figures 10–11 show histograms of the SeaWiFS `l0to1a` read and write times. Although the general shapes of the histograms are the same, the PPFS histograms are "compressed" in time and the peaks are not as high, confirming the overall reduction in input/output operation counts and volume.

It is easy to see the two execution phases from the histograms. During the first, the read duration peaks are high and the write duration peaks are low; this is the first phase when the output HDF file is being created. Read time accumulates when the HDF files are opened and the header data is read by HDF library calls. Write time is low by comparison; only a few bytes of metadata are actually written. During the second phase, the data is actually written to the output HDF files, causing the write time peaks.

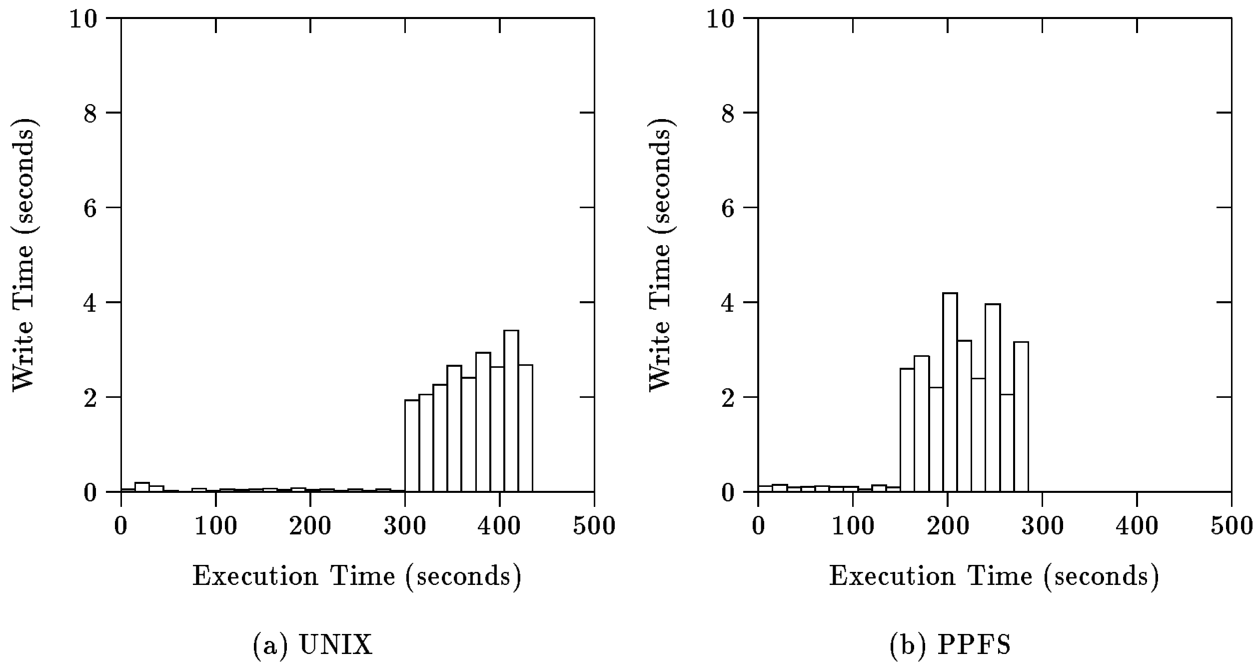


Figure 11: SeaWiFS l0to1a Write Duration Histograms

7.3 Summary

We began with the thesis that file systems must exploit application input/output access pattern information to achieve acceptable performance. Our experimental results using PPFS and two large satellite data processing applications demonstrate that it is possible to automatically classify access patterns and to use qualitative and quantitative access information to dynamically select and tune file system policies. Classification costs are far outweighed by the benefits of correct policy selection. Even simple access pattern information and rudimentary policy controls are useful; more complex classification and policy selection is likely to generate even better performance.

8 Related Work

Recent studies of high-performance parallel applications have shown that they exhibit a wide variety of input/output access characteristics [3]. Given such natural variation, 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.

Our approach is to provide a structure for allowing the file system to select appropriate policies by observing application behavior. There are many related approaches. One of the most direct ways to control policies is to require that the application recommend a management policy that has been previously determined to improve application performance. However, this is highly system-dependent and limits portability.

Applications do not need to directly specify policies; they can 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 [14]. This is portable, but requires the application programmer (or an intelligent compiler) to specify knowledge of the application input/output behavior.

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 [11]. Fido is an example of a predictive cache that prefetches by using an associative memory to recognize access patterns over time [13]. Knowledge based caching has been proposed to enhance cache performance of remote file servers [9]. Our work uses a trained neural net to recognize higher-level file access patterns from simple reads and writes; like other approaches based on learning algorithms it is extensible and can be trained to include more patterns as the need arises.

9 Conclusions

Through automatic input/output pattern identification and adaptive control of file system policies, we have achieved dramatic performance improvements over UNIX buffered input/output on a wide range of real and synthetic applications with non-sequential access patterns. Correct identification of simple access pattern categories enables a file system to choose caching policies best suited to that access pattern. Moreover, automatic classification and policy selection lessens the programming burden and increases code and performance portability.

To date, we have demonstrated the tenability of this approach on sequential applications. We currently are studying classification of parallel input/output access patterns, which have complicated performance characteristics that make this approach especially attractive.

Acknowledgements

We are extremely grateful to Peter Smith of NASA Goddard Space Flight Center for providing access to the Pathfinder code and to Gene Feldman for the `l0to1a` translation program and SeaWiFS test data. Figure 7 is based on Feldman's explanation of the `l0to1a` translation algorithm.

Thanks also to Frank Chen for patiently explaining SeaWiFS code operation.

References

[1] EOSDIS: EOS Data and Information System. National Aeronautics and Space Administration, 1992.

[2] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Eggers, S., and Chambers, C. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (December 1995).

[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] Engler, D. R., Kaashoek, M. F., and Jr., J. O. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (December 1995).

[5] Feldman, G. C. Seawifs Project Homepage. NASA Goddard Space Flight Center, Available at `http://seawifs.gsfc.nasa.gov/scripts/SEAWIFS.html`, 1996.

[6] 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* (1991), p. 177.

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

[8] Huber, J. V., 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* (July 1995), pp. 385–394.

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

[11] 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.

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

[13] 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.

[14] 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). To appear.

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

[16] Reed, D. A. Experimental Performance Analysis of Parallel Systems: Techniques and Open Problems. In *Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (May 1994), pp. 25–51.

[17] Smirni, E., Aydt, R. A., Chien, A. A., and Reed, D. A. I/O Requirements of Scientific Aapplications: An Evolutionary View. To appear (1996).