AUTOMATIC CLASSIFICATION OF INPUT/OUTPUT ACCESS PATTERNS

BY

TARA MAJA MADHYASTHA

B.A., Rutgers State University, 1990
M.S., University of Illinois at Urbana-Champaign, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

Abstract

Despite continued innovations in disk design, input/output performance has not kept pace with concurrent increases in processor speeds. Much research has focused on developing algorithms to avoid input/output or hide input/output latency in an attempt to redress this widening gap. Many studies have shown that with advance knowledge of access patterns, file systems can improve input/output performance by selecting policies appropriate for the resource demands. Unfortunately, access patterns may be complex or data dependent, and therefore unknown *a priori.*

Our thesis is that the file system can automatically detect qualitative file access patterns both locally (per parallel program thread) and globally (per parallel program) and use this information to dynamically choose appropriate file system policies. We propose two complementary methods for automatic classification, based on neural networks and hidden Markov models, respectively. Global classifications are created from a combination of local classifications and additional access pattern information. We map qualititative classifications and quantitative statistics to file system policies shown to improve performance for those access patterns.

We have implemented this classification framework as extensions to the Portable Parallel File System (PPFS) testbed. Experimental results on sequential and parallel scientific applications demonstrate the utility of this approach.

Acknowledgments

There are many people to acknowledge for their technical and personal support of this work.

First I need to thank my advisor, Daniel Reed. Over seven years, he taught me far too many things to enumerate, and no one has had a greater influence over my views on science and research. He made the Ph.D. process both exciting and rewarding. For this I cannot thank him enough.

My committee members, Andrew Chien and Mike Heath, also offered many suggestions that improved the quality of this work, and lots of support along the way.

Mike Lake discussed learning methods with me and suggested the use of hidden Markov models for classification.

Many people contributed tools, scientific applications, and characterization data to this effort. Roger Noe wrote the original Pablo trace library that I used extensively and built upon to gather trace data. Ruth Aydt wrote the current input/output extensions and tools to gather statistics from them, which made data collection much less painful. Terry Pratt and Gene Feldman of NASA Goddard Space Flight Center helped me to obtain the SeaWiFS application, and Frank Chen patiently explained how it worked. Peter Smith of NASA Goddard provided the Pathfinder application. Robert Ferraro of NASA Jet Propulsion Labororatory and his group explained to me the input/output resource demands of applications they were studying. Ruth Aydt, Phyllis Crandall and Evgenia Smirni collected characterization data from various applications from the Scalable I/O Initiative suite that I used extensively.

Anda Harney and Bonnie Howard made all sorts of administrative things happen for me automatically.

My last year in the Pablo group has been one of my happiest because of the Three Graces (Evgenia Smirni, Kris Grossi, and Ying Zhang). They epitomize for me everything that it is to be a woman and a scientist; I am fortunate to know them. Evgenia in particular provided me excellent career advice and was a powerful role model, refusing to let me wallow in thesis despair for more than a few days, suggesting I write a proposal or paper instead. Luiz De Rose also has been a great friend and a source of much practical graduate school advice (and fantastic chocolate).

Kristie Hirschenberger and Will Scullin accompanied me on numerous fun summer adventures, helping me to comprehend highlights of midwestern culture such as Rockhome Gardens, and to appreciate simple pleasures like cheap pizza and the X-Files (the one *before* the other).

Over the years, Ruth Aydt has helped me out in critical situations. I am incredibly grateful to her for inviting me to my first SIO meeting, graciously sharing her room with me while I edited a paper until two in the morning.

Chris Elford has been a good friend and "opponent" for many years. Thanks to Brian Totty and Thomas Kwan for being such wonderful officemates. Huseyin Simitci and Randy Ribler offered many useful comments on my papers, presentations, and dissertation. Countless other members of the Pablo group, past and present, contributed their support, feedback, and technical expertise to further my efforts over the years I've been in graduate school. In particular, I thank Taed Nelson, Jim Arendt, Bob Olson, Lawrence Rauschwerger, Bobby Nazief, Dave Jensen, Keith Shields, Dave Kohr, Phil Roth, Matt Gardner, Luis Tavera, Mario Medina, Celso Mendes, Eric Schaffer, Shannon Whitmore, Mike Mesnier and Jim Oly. They have also made life in Champaign-Urbana much more pleasant and bearable.

Outside the group, Ron Spinner has fielded frantic phone calls from me for years. Lincoln Smith supported my poetical endeavors. Josh Vaughn identified technical terms to describe my endearing personality quirks.

And in the animal and vegetable department, the Picasso group office plants basically gave their lives for my thesis. As I write, however, the group fish, Grommit and Andreas-fiche, are still alive (and probably hungry).

Finally, this work would have been impossible without the love and support of my parents.

This work was funded 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).

Table of Contents

Chapter

List of Tables

List of Figures

Chapter 1

Introduction

Despite continued innovations in disk design, input/output performance has not kept pace with concurrent increases in processor speeds. Moreover, the increase in processor speeds has enabled the study of larger problems with more complicated algorithms, revealing input/output to be a crucial performance bottleneck.

To ameliorate this input/output bottleneck, many researchers have tried to exploit the high bandwith afforded by input/output parallelism, building file systems that decluster data over many disks. To be successful, this approach requires clever orchestration of input/output requests. In a uniprocessor system, requests must be issued well in advance of the time data is needed to exploit the available bandwidth. In high performance parallel systems, 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. These related file requests must be coordinated to maximize system throughput.

File access patterns are thus an important factor in obtaining high throughput from both sequential and parallel file systems. Many studies have shown that file systems can utilize knowledge of input/output access patterns to improve input/output performance by selecting policies appropriate for the resource demands [26, 41, 68, 46]. Due to the complexity of input/output optimization in parallel file systems, access pattern information is even more critical to performance than in sequential file systems.

In this thesis we propose an approach to optimizing parallel input/output where automatic file access pattern classification is used to tune architecture-dependent file system policies locally and globally. Automatic classification and policy selection relieve the programmer of responsibility for input/output performance while increasing code and performance portability.

1.1 The Input/Output Problem

Because of their cost effectiveness, magnetic disks have dominated secondary memory design since 1965 [67]. A magnetic hard disk consists of a set of platters that rotate on a spindle. Each disk surface is divided into concentric circles called tracks. An arm with a magnetic read/write head moves just above the surface to access data from the tracks. There are four main components to each disk access [67]. The first is *seek time* to position the head over the correct track. Once positioned, the head must wait until the the requested data moves beneath the head; this is the *rotational latency.* The third component is the time to transfer the data, or *transfer time.* Finally, there is *controller time* imposed by the disk controller. Reliance on mechanical systems makes overall access times much slower than that of dynamic random access memory.

Improvements in disk access times have not kept pace with microprocessor performance, which has been improving by 50% or more per year. Although magnetic-media densities have increased, reducing disk transfer times by approximately 60-80% per year [76], overall improvement in disk access times, which rely upon advances in mechanical systems, has been less than 10% per year. Amdahl's law implies that the speedup obtained from faster processors is limited by the slower system components; therefore, if input/output accounts for some fraction of execution time, it is impossible to speed up the program by more than the reciprocal of that fraction without also improving input/output performance.

File systems address the huge and growing performance gap between processors and input/output through smart algorithms, caching, and prefetching to reduce or hide input/output access latencies. Developing such approaches requires an intimate understanding of the file system workload and disk characteristics; for example, sequential blocks of a file are usually organized sequentially or close together on disk to avoid costly seeks between accesses.

Studies of typical UNIX workstation environments show that 90% of file accesses are to sequential data, 80% of file accesses are to files smaller than 10 KB, and 67% of accesses are reads [66]. Large caches can satisfy many of the small read only requests and sequential prefetching can hide input/output latency for this workload. Because the workload is relatively uniform, a single set of policies provides good performance for the majority of accesses.

Supercomputer input/output is dominated by accesses to very large, sequential files [55, 64]. Input/output is due to compulsory reads of initialization data and writes to large checkpoint files. Therefore, sequential supercomputer input/output systems are typically optimized for high throughput.

Uniprocessor input/output, whether from scientific applications or engineering workstations, is quite regular. In contrast, multiprocessor input/output access patterns exhibit greater variation [18, 81], making it more difficult for file systems to optimize input/output. Furthermore, as processor speeds increase, it becomes computationally feasible to examine larger problems and new application areas, exacerbating the input/output bottleneck. This bottleneck seriously impacts an important class of national challenge applications (e.g., global climate modeling) because many such applications exhibit complex, dynamic, often irregular input/output access patterns.

Current parallel file systems are usually optimized for some expected workloads, and even slight deviations from the expected can degrade performance by orders of magnitude [20, 80]. Often, to achieve acceptable performance, the programmer 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 programmer, but such optimizations are system-specific and may be inappropriate for other systems or input/output configurations. For example, the optimal request size for a striped parallel file system may be much larger than the optimal request size for a UNIX sequential file system. Ideally the programmer should not have to tune the request size for each platform.

Studies have demonstrated that file system policies that exploit knowledge of application access patterns can provide better performance than a default set of strategies [26, 41, 68]. 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. Recognizing the importance of such information to performance, many parallel file system application programming interfaces (APIs) enable the application to specify file access patterns. 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.

1.2 Application Input/Output Requirements

Application level characterization studies [18, 81] demonstrate that there is wide variability in input/output access patterns, and moreover, performance is extremely sensitive to these variations. Many of these characterization efforts are part of the Scalable Input/Output (SIO) Initiative, a multidisciplinary collaboration of the Caltech Center for Advanced Computing Research (CACR) created to investigate issues in input/output performance on massively parallel computer systems. Application developers and computer scientists have identified a representative set of applications and we leverage this work to examine typical access pattern and performance characteristics.

As an example of performance sensitivity, consider input/output timelines from two applications targeted by the Scalable Input/Output Initiative. These applications were executed on partitions of an Intel Paragon XP/S, a supercomputer with a number of i860 processor nodes connected by a high-speed interconnect network.

Figure 1.1 shows the input/output timeline of ESCAT [93], which models low-energy electron-molecule collisions, a topic of interest in many contexts, including aerospace applications, atmospheric studies, and material processing. After an initial burst of compulsory reads to load the problem definition and initial data, all nodes participate in calculation of the quadrature data set, causing a series of compute/write cycles. Processors channel their output to node 0, which writes the output sequentially.

Figure 1.2 shows input/output activity from phase three of QCRD [94, 50], a quantum chemical reaction dynamics code. This code has logical input/output requirements similar to that of ESCAT, however, processors coordinate in the read and write cycles to read and write matrices using a global interleaved sequential pattern (each processor accesses the file in strides but the interleaving of accesses across all processors is sequential). Note that seeks dominate

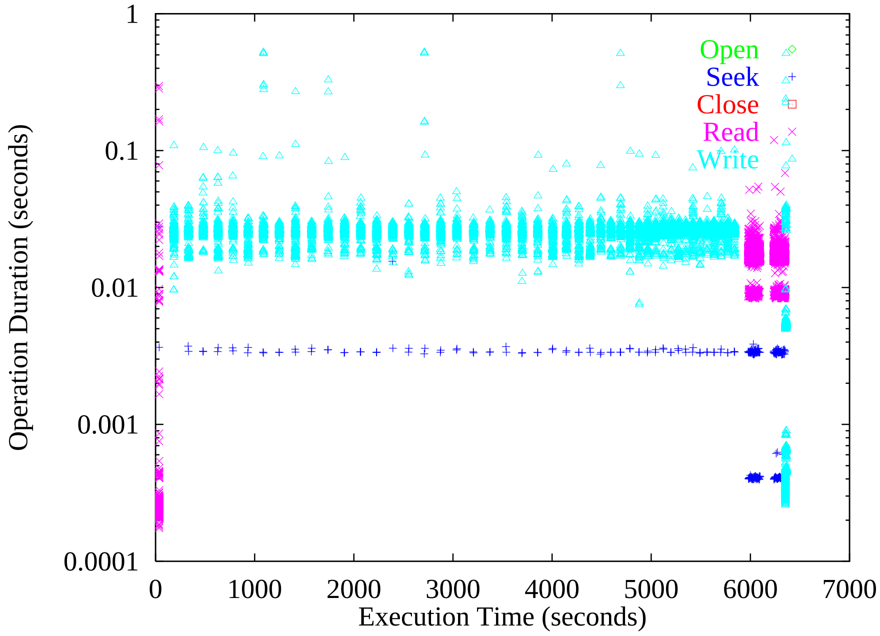


Figure 1.1: ESCAT on 128 processors.

input/output time, and there is an order of magnitude difference between write durations of the two codes in the compute/write cycles.

Input/output performance of QCRD is much worse than ESCAT because the file system software is not optimizing performance for the interleaved access pattern. Without knowing that the interleaved writes are disjoint, the file system serializes them to preserve UNIX atomicity semantics. This manifests itself by extremely expensive seeks; file operations are executed first come, first serve to avoid potentially leaving a file in an inconsistent state. This serialization destroys the potential access parallelism that could ordinarily improve throughput. Moreover, the information that the global interleaved pattern is sequential could be used to tune prefetch and writebehind policies, prefetching blocks that will be needed in the global sequence or batching write requests to preserve sequential access, thereby improving performance.

These two applications are typical of the scientific codes in the SIO application suite. Parallel input/output access patterns observed in current applications are dominated by combinations of sequential and one or two dimensionally strided accesses issued by all or a subset of processors, with a wide range of request sizes. Some file systems provide interfaces to specify one of a set of supported access patterns in an attempt to alleviate performance problems. However, present input/output patterns are limited by UNIX-compatible parallel file system interfaces

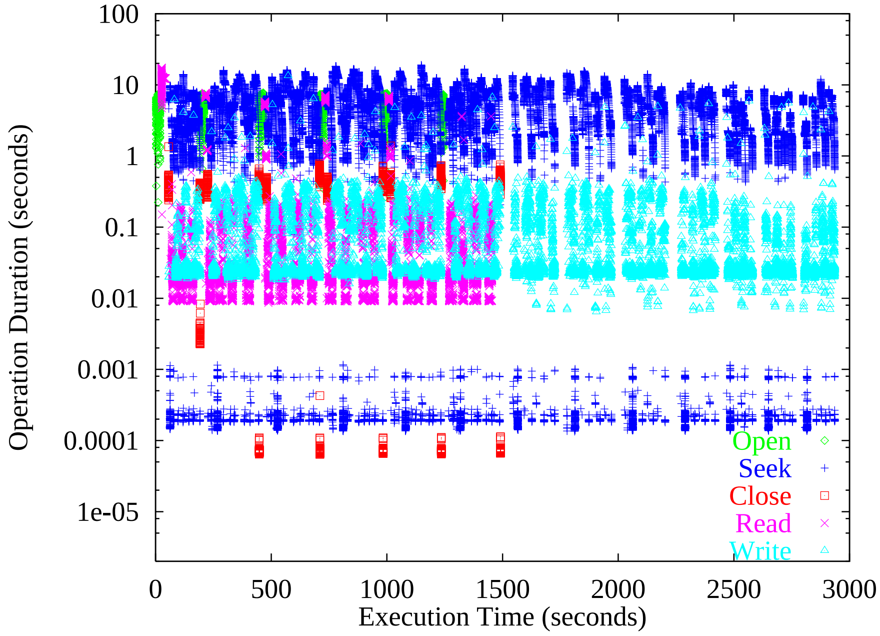


Figure 1.2: QCRD on 64 processors.

and do not necessarily represent what is desirable. As file systems improve to meet the resource demands of current applications, applications will stress them in new ways.

1.3 Adaptive File System Policies

Clearly, application input/output access patterns are important to file system performance. Unfortunately, it may be difficult to specify file access patterns with accuracy in advance. Characterization studies have demonstrated that programmers are frequently unaware of the access patterns in their code; they may be hidden by higher level structured file abstractions or they may be data-dependent. Furthermore, the idiosyncrasies of different file system architectures may influence what access pattern characteristics (e.g., sequentiality, request size) are most critical to performance.

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, the access classification is application specific and platform independent, whereas file system policies are both access pattern and platform specific.

This thesis demonstrates that we can automatically classify access pattern features that are important to file system performance. It is possible to structure a file system so that automatic classification information may be used to select and tune file system policies, and this approach can dynamically improve performance. Early and accurate access pattern classification is important to obtaining good speedups, especially on parallel file systems. However, historical execution information can be used to generate such early classifications. Even if previous execution data is not available, classification based policy control can still improve performance significantly beyond default that of file system policies.

1.4 Thesis Organization

The remaining chapters of this thesis are organized as follows. In Chapter 2 we review related work and place this thesis in context.

In Chapter 3 we describe and validate our methodology for local automatic classification. We introduce two complementary and novel classification methods based on artificial neural networks and hidden Markov models.

Chapter 4 describes how a file system uses these classifications to select appropriate policies. We present the results from experiments calibrating a policy selection algorithm for the uniprocessor experiments in this thesis.

Chapter 5 presents results of classification using our two methodologies on sequential benchmarks and applications. Our application domain is the area of low level satellite data processing applications. These applications are input/output bound and exhibit complex file access patterns due to their reliance on high level input/output libraries.

Having demonstrated that local classifications can be used to improve input/output performance, in Chapter 6 we provide a rationale for global classification. In a parallel file system, the complexities of efficiently servicing concurrent, related input/output request streams make exploitation of global behavior especially critical to overall input/output performance.

In Chapter 7 we present results showing how global classification can be used to improve input/output performance of parallel scientific applications and benchmarks.

One of the methods for providing access pattern information to a file system, included in the Scalable Input/Output Initiative's low level parallel input/output API, is to supply hints describing future access patterns. Our work can be used to generate hints automatically; we describe the relationship between automatic classification and hints in Chapter 8.

Finally, in Chapter 9 we describe directions for continuation of this work, and conclude by summarizing our results in Chapter 10.

Chapter 2

Background and Related Work

Current parallel file system research centers on understanding application input/output requirements and determining how to consistently deliver close to peak input/output performance. This challenge necessitates re-examining the traditional interface between the file system and application, examining ways of using application access pattern information to improve performance.

In §2.1 we present a survey of some major characterization efforts relevant to massively parallel systems. Existing characterization data and future projections of application demands mandate that file systems must be flexible; §2.2 describes approaches to application control of file system policies. Most of these approaches have relied heavily upon programmer input; we describe some more intelligent techniques in §2.3.

2.1 Access Pattern Characterization

An understanding of expected workload is essential to file system design and optimization. Uniprocessor workloads are well characterized. Studies of typical engineering workloads on workstations under UNIX 4.2 BSD [61] and the Sprite distributed file system [3] reveal that most file accesses are small, sequential, and read only.

Scientific workloads on sequential supercomputers have also been extensively studied. Miller and Katz [55] characterized the behavior of seven computational fluid dynamics and climate applications. File accesses were sequential and input/output was bursty due to the iterative nature of the underlying algorithms. They identified three main categories of input/output requests: compulsory, cyclic checkpointing, and data staging. Pasquale and Polyzos [64, 65] analyzed input/output intensive applications on the Cray Y-MP and C90 at the San Diego Supercomputer Center; they concluded that their behavior over time was regular and cyclic. Reddy and Banerjee [72] characterized implicit and explicit input/output activity of five parallelized applications from the Perfect benchmark suite [54] and also found it to be mostly sequential; input/output itself was not parallelized.

Like uniprocessor workstation workloads, access patterns for scientific codes are primarily sequential; however, the input/output volume of scientific workloads is much larger, yielding larger request and file sizes, and the proportion of writes (due to checkpoints and output of results) is much greater. Furthermore, the underlying application structure of most scientific applications generates predictable input/output demands.

Unfortunately, scientific workload characterizations of supercomputers with sequential input/output do not automatically apply to multiprocessor systems, where the input/output requests themselves are parallelized over multiple disks. Furthermore, until relatively recently, little emphasis has been placed on characterizing parallel input/output access patterns. Crockett [19] and Kotz [43] have proposed taxonomies of access patterns (e.g., sequential, strided, and global equivalents), but experimental evidence to demonstrate the importance of these patterns is still scant. Few input/output bound multiprocessor applications exist due to inadequate file system performance; those that do exhibit architecture-centric access patterns. Several studies attempt to address this dearth of characterization data by gathering application input/output access pattern information at the system and application level.

The CHARISMA project [44, 60] has examined input/output accesses on the iPSC/860 Concurrent File System (CFS) and the CM5 Scalable Disk Array to obtain some generalizations of access patterns in parallel input/output workloads. They have observed predominantly write accesses, small request sizes, and generally sequential requests.

The CHARISMA characterization effort focused on system-level input/output; however, in-depth study of application-level input/output is also necessary to understand programmer input/output requirements. Many researchers have addressed this problem, studying the application behavior of a variety of scientific parallel applications.

Using the the Pablo [73, 74] performance analysis software, researchers have characterized the application level behavior of a wide variety of parallel applications on the Intel Paragon

XP/S. Application areas include modeling of electron-molecule collisions, a 3-D numerical simulation of the Navier-Stokes equations, an implementation of the Hartree-Fock self consistent field method to calculate the electron density around a molecule, and quantum chemical reaction dynamics [18, 81, 82]. These applications compose a snapshot of current input/output practices on scalable parallel machines and reflect the programmers' input/output design choices based on the capabilities of available input/output systems. They are but a small part of the Scalable Input/Output Initiative's (SIO) code suite [69].

Thakur *et al* [90, 89] used Pablo to characterize the behavior of a three-dimensional parallel astrophysics code running on PIOFS on the IBM SP/2 and PFS on the Paragon. This application uses several three-dimensional arrays that are allocated to processors using a (block,block,block) distribution across three dimensions.

Baylor and Wu [6, 5] characterized four applications running on the IBM Vesta file system. They determined that the number of small requests was very large, yet there was a wide request size variation. There was significant file sharing within each application.

These studies revealed that multiprocessor scientific applications exhibit a wide variety of input/output request patterns, with both very small and very large request sizes, reads and writes, sequential and non-sequential accesses, and a variety of temporal variations. While many applications still exhibit cyclic behavior, cycles are more irregular due to variations in input/output performance across nodes. Often the programmer's desired input/output access style is inefficient, necessitating input/output restructuring in future application revisions to improve performance.

2.2 Application Policy Control

Given the natural variation in parallel input/output patterns, 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 [41, 46, 47, 26]. 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.

There are several very closely related approaches to application policy control. Categorizing these loosely, we describe prototype systems that utilize explicit policy control and access pattern information supplied at various levels of the application programming interface.

2.2.1 Explicit Policy Control

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 [8] and *exokernel* [21]). SPIN achieves application resource control by allowing applications to define extension services that are dynamically linked into the kernel virtual address space. The *exokernel* operating system simply defines a low-level interface to resources that handles only protection issues, not management, similar to the application-specific approach to operating systems proposed by Anderson [2].

A file system is a particular example of an operating system abstraction that can be restrictive when performance is an issue. Database systems have historically bypassed native file systems, seizing control of the raw resources in order to provide high performance. Park *et al* suggest using custom virtual memory page replacement policies to optimize out-of-core input/output [63]. Cao *et al* recommend that applications use their own page replacement policy and describe an approach to buffer allocation to support application-controlled page replacement [9, 10].

Similarly, it is well-accepted that a parallel file system must be flexible to provide reasonable performance to a variety of applications. Several research file systems have been designed with explicit policy control in mind. HFS, the Hurricane File System [45, 47], supports a variety of file structures, file system policies, and interfaces. HFS allows the file structure to be optimized for a particular access pattern, with commensurate prefetching, locking, and file cache management policies. Galley [59] is a flexible file system with a well defined low-level input/output interface, upon which multiple high-level input/output interfaces can be written. The second generation Galley file system goes a step further, allowing user code to be executed on input/output nodes.

Although applications with predictable resource demands can effectively control prefetching or caching policies, there are many problems with this approach. Application policy control requires the application programmer to have a detailed understanding of system and application performance characteristics, and is usually non-portable. The best choice of policies might not be known *a priori*; 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.

2.2.2 Encapsulated Access Pattern Information

A more portable approach to providing application access pattern information to a file system is to encapsulate it within a file system interface. The file system can invoke (possibly system dependent) policies that optimize input/output accordingly. For example, the ELFS system [26] proposes an object-oriented interface where object classes represent particular file access patterns.

A particularly important kind of optimization that has been targeted by this approach is collective input/output. If the mapping of file data to disks does not match the processors' view of that data, small, redundant, and extremely inefficient requests may result. Collective input/output operations alleviate this problem by requiring operations to be performed by every processor, permitting requests to be aggregated across processors and optimizing input/output accesses at the expense of explicit synchronization. Performance improvements have been demonstrated using collective input/output strategies including two-phase input/output[20] and disk-directed input/output[40].

In two-phase input/output, the application uses a particular interface to alert the file system to a collective operation. In a two-phase read, for example, data is read from disks in the most efficient manner and redistributed to the processors involved in the collective. A variation of two-phase input/output called extended two-phase input/output[88] uses collective input/output in conjunction with dynamic partitioning of the input/output workload among processors to balance load and improve performance.

Disk-directed input/output [40] is another method for optimizing collective requests. The complete collective request is passed to the input/output processors, which sort the list of transfers and use remote memory "put" and "get" messages to pipeline the transfer between application memory and disks.

Many input/output libraries have been developed that rely upon collective input/output optimizations to improve performance of multiprocessor input/output systems. Jovian [7] is a runtime library for Single Process Multiple Data (SPMD) codes (processors execute the same code on different data). The programmer chooses the appropriate data distribution and input/output operations are implicitly collective. A variable number of coalescing processes aggregate the requests to perform them efficiently under the chosen distribution. PASSION [13, 87, 86] (Parallel And Scalable Software for Input-Output) is another runtime library targeted for SPMD applications with routines to efficiently perform out-of-core operations. The Panda [79, 12] library utilizes server-directed input/output and a collective interface to achieve high performance on array accesses.

APIs can encapsulate information that allows optimizations that are impossible otherwise. For example, another example of an API-specified access pattern is the *Read Any* mode of the Portable Parallel File System (PPFS) [31]. Using this mode, an application specifies that the processors want to collectively read disjoint records of the entire file, but in no particular order. With this knowledge, the file system can balance application requests and set global prefetching policies.

2.2.3 Access Pattern Hints

Access pattern information provided through application programming interfaces is often tightly coupled to the interface itself. Another way to provide the file system with access pattern information is to use application hints (possibly inaccurate access information) to guide a proactive file system. Hints need not be specified in order for the file system to work, but they can improve performance if provided.

Patterson *et al* demonstrate the success of providing hints to guide prefetching and caching of files that will be accessed in the future [68]. The application reveals future accesses through hints, allowing the file system to exploit input/output bandwidth. A cost-benefit analysis evaluates the reduction in input/output service time per buffer access to decide when to allocate buffers.

This is an elegant approach for dealing with multiple applications that compete for shared caching and prefetching buffers. Kimbrel *et al* [35] and Tomkins *et al* [92] compare several integrated caching and prefetching policies using hints for systems with multiple disks.

In parallel file systems, the way that data is partitioned among processors is closely linked to the access pattern. For example, in a global sequential access pattern, processors each access identical portions of a file sequentially. In a global sequential interleaved access pattern, processors access disjoint interleaved blocks in node order. Hints may be specific input/output modes to describe common partitioning strategies, or they may be implicitly specified through the use of particular data distributions.

For example, Intel PFS [32] has file access modes that describe a variety of common global access patterns. A popular attempt to define a standard user-level application programming interface (API) called MPI-IO [91, 70] uses MPI-derived datatypes [24] to express the mapping of file data to processors. IBM's Vesta [16] (marketed as PIOFS) file system gives applications control over declustering that is geared towards two-dimensional distributions. With this information, the file system can optimize performance for the specified patterns.

2.3 Intelligent Policy Control

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 exhibiting global sequentiality; this information is used to guide non-sequential prefetching within a file [42]. Fido is an example of a predictive cache that prefetches by using an associative memory to recognize access patterns over time [62]. Knowledge based caching has been proposed to enhance cache performance of remote file servers [37]. Mowry *et al* [56] propose a method for using input/output access patterns determined by the compiler to hide input/output latency by prefetching, assuming the application relies upon paged virtual memory for all input/output.

These studies focus on application input/output, but the method can also be applied to implicit input/output. An automatic policy for page prefetching due to implicit input/output was proposed by Song and Cho [84]. This approach uses a history of previous page faults to predict subsequent page faults when an application repeatedly access pages in the same order.

Some intelligent approaches to policy control have focused on construction of models of file access that guide prefetchers. Some approaches are purely statistical; in automatic prefetching [25] the operating system creates a probability graph based on frequency counts to allow prefetching of files that are likely to be accessed soon after other files. Kroeger and Long [48] use a method based on data compression to create first, second, and higher order context models of file relationships. Lei and Duchamp [51] create file access trees for program executions and use recurrent access patterns heuristically to guide prefetching. The SEER system [49] correlates files by "semantic distance" to decide what files to cache in a portable computing environment.

Although our focus is on file prefetching, a statistical model based on Markov predictors has been proposed for prefetching between on-chip and off-chip cache [33]. Memory references are modeled as a Markov process that is continually rebuilt as the program executes. Multiple memory references are prefetched at once. To simplify the model, the fan-out from each state is limited, and the transition probabilities are approximated using an LRU mechanism. This kind of predictor is very effective in reducing overall execution stalls.

2.4 Summary

Input/output characterization of parallel scientific codes reveals a large variation in access patterns, to which file system performance is extremely sensitive. The demonstrated benefits of application supplied access pattern information and flexible file system policies fuel research on the best ways to maximize performance while maintaining a simple programming interface. Unfortunately, there is still no consensus on what information applications must specify to the file system to improve performance.

Many existing parallel file systems and proposed APIs provide some control over data layout, in the hope that the application can achieve high input/output performance by matching the data layout to the access pattern. Still, achieving peak performance in this manner is hampered by inability to control static, system-wide file system policies, and by the difficulty of understanding performance tradeoffs.

Intelligent approaches seek to automatically construct models of file access to guide file system policies, relieving the programmer of responsibility for specifying hints or using interfaces to improve performance. We have described research in this area most relevant to our proposed methodology.

Chapter 3

Automatic Access Pattern Classification

Accurate access pattern information is an important factor in optimizing input/output performance; unfortunately, programmers may have difficulty specifying access patterns in advance. Access patterns may be data dependent or masked by input/output libraries. Automatic access pattern classification and policy selection can improve input/output performance without increasing programming difficulty.

Consequently, our research rests on the following two premises:

- Access patterns can be detected automatically at local (e.g., per parallel task) and global (e.g., per parallel program) levels.
- System-independent access patterns can be used by system-dependent decision procedures to select and tune file system policies.

In §3.1 we describe our rationale for automatic classification. We explain our model for access pattern classification in §3.2. In §3.3 we describe the classification problem, motivating complementary classification approaches based on neural networks and hidden Markov models that are described in depth in §3.4–§3.5.

3.1 Access Pattern Classification Rationale

Applications often have qualitative, recognizable input/output access patterns (e.g., 'read-only sequential with large request sizes,' or 'write-only strided.') File system policies are normally optimized for certain common access patterns. When other access patterns occur, performance can be much poorer than that achievable using advance knowledge of the unusual requests. For example, a cache optimized for a read-only sequential access pattern might employ the space-efficient MRU (Most Recently Used) cache block replacement policy. Such a policy is inappropriate for random access requests that exhibit locality; an LRU (Least Recently Used) pattern would be more suitable. These kinds of decisions are impossible without access pattern information and the ability to use it to control file system behavior.

An access pattern is a qualitative statement that can be used to select and tune file system policies. It need not be a perfect predictor of future accesses to improve file system performance. For example, one might classify an input/output pattern as "sequential" even if a few bytes are skipped, and elect to use an MRU cache replacement policy. However, if the number of bytes skipped is large and regular, the access pattern might be strided. If we expect the skipped bytes to be reread on subsequent strides, an LRU replacement policy may be more suitable.

Such a qualitative description is difficult to obtain on the basis of heuristics alone (e.g., the access pattern is sequential if over 90 percent of the bytes are accessed in increasing order). Many alternate heuristic descriptions using a variety of statistics could be proposed and there would inevitably be exceptions to each one. For example, is an access pattern that reads every file block twice in increasing order also sequential? Although it does not satisfy the above heuristic, for purposes of policy selection it is useful to treat it as a sequential access pattern. We require a classification methodology that can adapt to such exceptions when they occur. Our research investigates automatic learning mechanisms that periodically produce file access pattern classifications. The file system uses these classifications to dynamically tune caching and prefetching policies throughout program execution.

Within a parallel application, there are two levels at which input/output access patterns can be observed. The first is at the local (e.g., per thread) level, and the second is at the global (e.g., per parallel program) level. For example, a parallel file might be striped across the threads of a parallel program in such a way that each thread appears to be accessing the file locally in strides, but the interleaved access stream is globally sequential. Global classification is especially important because optimal policies for the local access patterns can be different from those that would benefit the file access stream as a whole.

3.2 Access Pattern Classification Model

Access patterns are closely tied to both the application programming interface (API) used to generate input/output requests, and the underlying file structure. For example, a record-based file that can only be read or written in fixed size records will never have unpredictable request sizes. Small requests generated by calls to the UNIX standard buffered input/output library are translated into fixed size input/output calls to the underlying file system.

In our approach, we attempt to divorce the access pattern description from these higher and lower-level software layers. To accommodate a variety of high-level interfaces and underlying file structures, we describe access patterns assuming a byte stream file representation. File accesses are made using UNIX style *read*, *write*, and *seek* operations, and access patterns are determined at the application level. Using this representation, an input/output trace of file accesses may be represented as a stream of tuples of the form

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

This approach is portable, because both local and global classifications are independent of API, file structure or layout. Input/output requests made using any API can be translated into UNIX style operations, although information might be lost. By classifying accesses at the application level, classifications are completely independent of file system policies. For example, the file system might choose the same policies to optimize a strided access pattern as a sequential access pattern if the stride granularity is small compared to the underlying file system block size.

Access patterns exist within a three-dimensional access pattern space, pictured in Figure 3.1. Each dimension corresponds to patterns observed in the time-varying values of the trace tuple components, and each point in the three-dimensional space represents a unique access pattern. This ternary partition is not unique; we have selected these axes because they are the architecture-independent categories most commonly used to describe access patterns at a small scale (i.e. from observing a small window of accesses). Other useful categories can be added

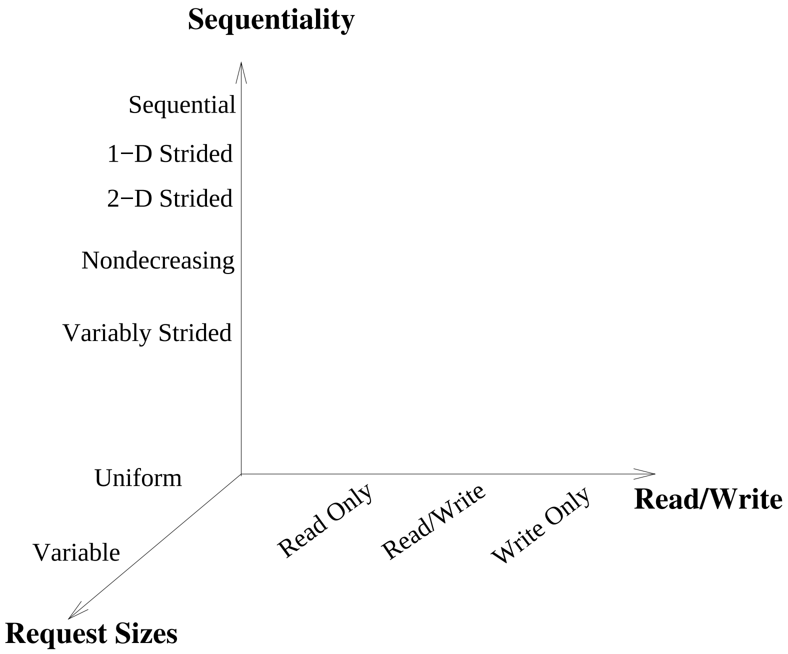


Figure 3.1: Access pattern space.

(e.g., file reuse). Relaxing our desire for architecture independence opens additional possibilities, such as interaccess latency. However, these axes embrace a wide range of common scientific access patterns, and, as we shall see in Chapter 5, suffice to effectively select and tune several file policies.

Many functions can be computed on the trace tuple components to yield identifiable features with which to label the three-dimensional access pattern space. However, access patterns that are predictive, can be determined from a small window of accesses, and can be used to influence file system policy selection are the most interesting to identify. The three axes of Figure 3.1 are labeled with certain features meeting these criteria that can be used to classify all points in the access pattern space. Additional features can be added as necessary to each axis to further refine the access pattern space.

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 3.2 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

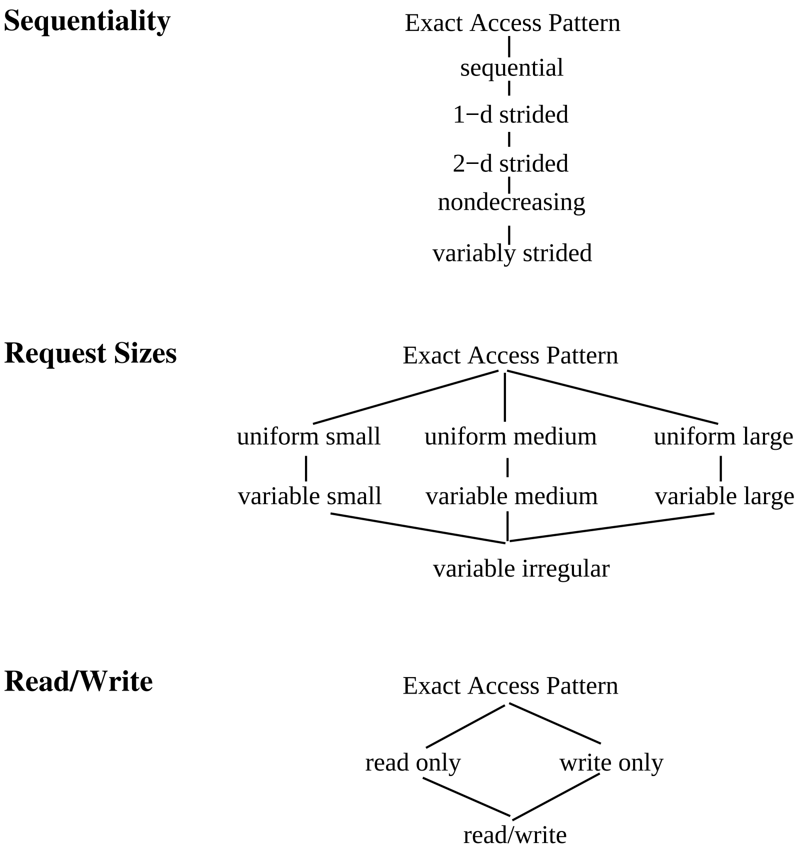


Figure 3.2: Hasse diagrams for local classifications.

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 3.2 the greatest lower bound of the two is a read/write access pattern. The least upper bound of the two classifications is the exact access pattern, or the precise description of the accesses themselves. As we will see in Chapter 6, this ordering helps us to combine multiple classifications to form global classifications.

3.3 Classification Methodologies

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. Exceptions will occur for any selected heuristics, and these will have to be identified and added by hand.

Instead, we have investigated more sophisticated classification approaches that "learn" to classify access patterns. One approach is to train a feedforward artificial neural network [30] (ANN)[1] 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.

Our ANN classifier operates within certain domain-specific assumptions and constraints. First, we demand that classification occur efficiently in real-time throughout program execution, although training can be performed off-line. We also assume that the training data, although it might be synthetically generated, must consist of statistics computable from the components of our trace representation (i.e., byte offset, request size, read/write). The output classifications are qualitative and are supplemented by trace statistics to control file system policies. Within these constraints, the ANN classifier produces a qualitative classification that locates an access pattern within the access pattern space of Figure 3.1.

Classifications made within these constraints have some limitations. A classification made from observation of a small window of accesses cannot capture many important access pattern characteristics that occur on a larger scale. For example, if a file is being read once sequen-

[1] Usually the descriptor "artificial" is dropped when it is clear from context that the neural network is not biological in origin. However, we retain the acronym ANN for its irresistible assonance with HMM, our complementary classification method.

tially, blocks that have already been read can be replaced, but if it is being reread repeatedly and is small enough, it should remain cached. File reuse and other large scale access pattern characteristics are unrecognizable from a limited subset of accesses. For example, an access pattern might appear highly irregular but in fact is consistent across application executions. Classification with no historical information will classify the access pattern as random. In reality, detailed knowledge of the repeatable pattern can be used to guide caching and prefetching policies.

Because of these limitations, we sought an alternative method for classification 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. To this end, we made an additional domain-specific assumption about scientific input/output access patterns. Namely, we expect that scientific applications will frequently exhibit the same file access patterns as in previous executions. All of the scientific applications in the Scalable I/O Initiative application suite have this property.

A probabilistic model of file accesses can supply more detailed access pattern information for caching and prefetching policy decisions. To create such a model automatically, we use file access data from previous executions to train extended Markov models, or hidden Markov models (HMMs) [71, 11]. 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.

Using HMMs, we view a file as a collection of blocks, and construct a probabilistic model of transitions between blocks from previous execution traces. With such a model, classification is performed by computing the probability of detecting certain features based on previous execution information. This gives us a powerful language for expressing access patterns that are complex yet repeatable.

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. More system-dependent information is encoded in an HMM; the access pattern is modeled at the level of file segments rather than application-level accesses. Most importantly, HMMs can provide classifications at file open based on previous execution

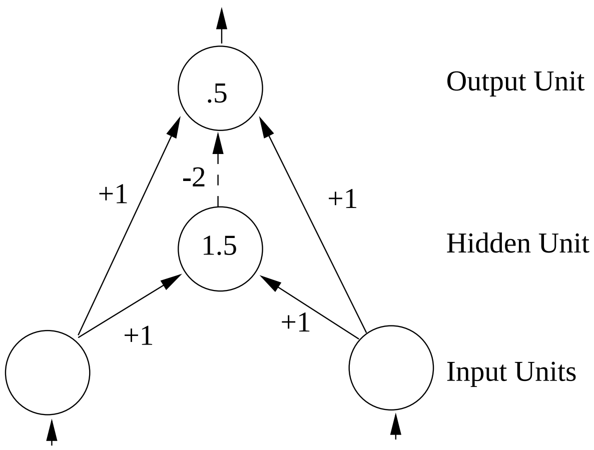


Figure 3.3: An XOR neural network.

information, while neural network classification must observe some window of accesses before detecting local and global patterns. Although the results of ANN classification could be saved and reused to provide initial classifications, the method does not statistically differentiate between multiple different access patterns. In other words, if the file has different classifications during separate executions, the probability of each classification must be maintained externally.

3.4 Neural Network Classification

Our first classification approach was to train a feedforward artificial neural network [30] to classify patterns. Although neural networks are expensive to train initially, once training is complete, classification is very efficient. Neural networks are particularly suited to classification problems where training data is readily available and some degree of imprecision must be tolerated.

3.4.1 Artificial Neural Networks

An artificial neural network consists of many simple processors (units) that are linked to each other through communication channels (links) that carry numeric data. Each link has a weight, or strength, associated with it. The units produce outputs that are computed from the weighted input data from their input links.

Figure 3.3 shows a simple network that computes the XOR function on two inputs [77]. The circles represent the units, the arrows represent the links, and the number by each arrow is the associated link weight. The numbers inside the hidden and output units are thresholds used in computing the unit output. The value presented to each input unit is multiplied by the outgoing weights, creating weighted input data for other units. For example, if the inputs are both 1, the inputs to the hidden unit are also both 1. To compute their outputs, the hidden unit and output unit each sum their weighted inputs, and output a 1 if the sum is greater than the unit threshold, or 0 if it is less.

Most neural network training methods are gradient descent methods. Random initial estimates between are made for the link weights, and the net trains on each training pattern, iteratively refining the weights until the error is acceptably low. Training on a pattern proceeds as follows. In the forward pass, the input pattern is presented at the input units, and data flows forward in the network through all units as they evaluate their outputs. In the backward pass, the output of the output layer is compared with the target output pattern to compute the error. This error is used together with the output of the source unit to compute the changes to the link. Errors are propagated backward through the network according to the formulas below, which are those for standard backpropagation as introduced in [77, 78].

Let the error measurement be the square difference between the target and the output, or mathematically,

$$E_p = \frac{1}{2}\sum_j (t_{pj} - o_{pj})^2$$

for input pattern p, where t_{pj} is the target output for the jth component of the output pattern, and o_{pj} is the actual output for the jth component of the output pattern.

Consequently, the total error over all patterns is

$$E = \sum_p E_p$$

Our goal is to train the link weights in such a way as to minimize E. The formula used to update the weights, called the generalized delta rule, implements gradient descent; although there is a danger of reaching a local minimum while training, this does not often happen in practice.

To implement gradient descent in E we adjust our weights according to

$$\Delta_p w_{ji} = -\eta g_{pj}$$

where $\Delta_p w_{ij}$ is the change to be made to the weight from the ith to the jth unit following presentation of pattern p, η is a constant learning rate and g_{pj} is the partial derivative of the error measure with respect to w_{ji}.

3.4.2 Neural Network Classification Methodology

Our neural network configuration is a small feedforward network with thirteen input units, ten output units, and one hidden layer consisting of twelve units. While the number of inputs and outputs determines the number of input and output units, the number of hidden units can vary without significantly affecting the neural network output. Hidden units recode the input data so that arbitrarily complex mappings between inputs and outputs can be learned; if there are too few, the network cannot be trained, and if there are too many, some are redundant. To determine the number of hidden units in our network, we employed a standard heuristic: begin with many hidden units and gradually reduce the number until the network is the smallest size that can still be accurately trained.

The training algorithm is standard backpropagation as described in §3.4.1, a very common learning algorithm. To successfully train the neural net to recognize access patterns within the space shown in Figure 3.1, we must process the trace of input/output accesses so that training can be performed efficiently and accurately. Our approach is to compute input/output statistics on a small fixed number of accesses and normalize them so that they fall within the range -1 to 1. Figure 3.4 depicts read and write accesses to a file, labeled with variables of interest. Statistics computed on these variables are listed, with descriptions, in Table 3.1. Note that n is the total number of bytes accessed, N is the number of input/output requests in the window, a capitalized variable represents the set consisting of the individual members (e.g., D is the set of the individual distances d between requests), and $k(X)$ denotes the cardinality, or number of elements in, set X.

These statistics represent the access pattern in a compact, normalized form suitable for input to a neural net. In our experiments, we have chosen to compute statistics on relatively small windows of accesses, so that periodic reclassification can detect changes in the access pattern.

The output of our trained neural net is a ten bit vector; each bit indicates the presence or absence of one of the features of Table 3.2. A bit is one if the value of the corresponding unit

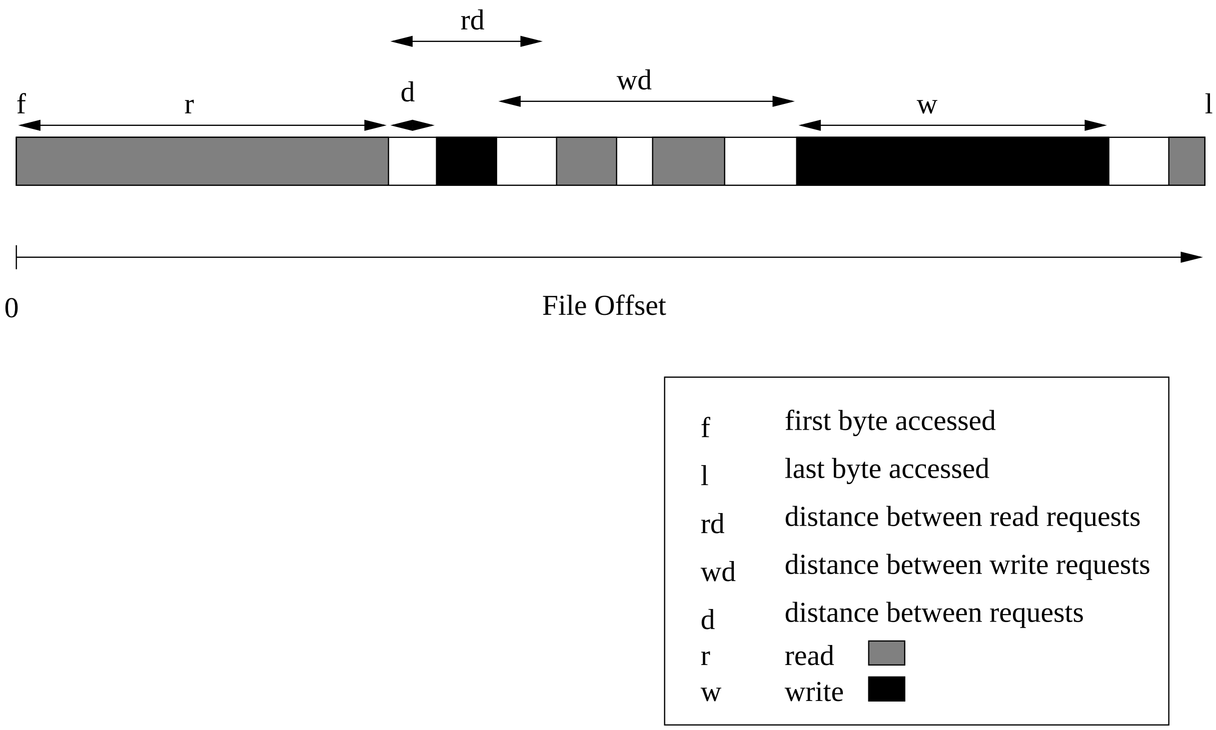


Figure 3.4: Input/output requests.

output is $\geq$ 0.9 and zero if the value is $\leq$ 0.1. If the unit output is in between, neither the presence nor the absence of the corresponding feature can be identified.

The read/write patterns of Table 3.2 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 and frequently occurring. The read-update-write pattern, where a file region is read and then rewritten, also occurs frequently in scientific codes. Read/write nonupdate includes any other pattern with both reads and writes.

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,

Statistic	Description
$(\sum d)/n$	distance in bytes between requests
$(\sum r)/(l-f)$	percentage bytes read
$(\sum w)/(l-f)$	percentage bytes written
$l > l_{prev}$	has the file grown in size?
$k(D)/N$	percentage of unique distances between requests
$k(RD)/N$	percentage of unique distances between read requests
$k(WD)/N$	percentage of unique distances between write requests
$k(R)/N$	percentage of unique read request sizes
$k(W)/N$	percentage of unique write request sizes
prob(rr)	probability of read followed by a read
prob(rw)	probability of read followed by a write
prob(ww)	probability of write followed by a write
prob(wr)	probability of write followed by a read

Table 3.1: Input/output statistics used to train neural network.

medium, and large sizes, making possible selection of file policies based on expected cache sizes and memory copy overheads.

Most of the features of Table 3.2 correspond to axis categories of Figure 3.1, with the exception of read-update-write and read/write nonupdate. Although these two categories span sequentiality and read/write composition in terms of the algebra we have described, they are meaningful terms when describing input/output access patterns. For convenience we consider them read/write patterns only and do not include them in our algebra.

To make a successful classification, one and only one bit should be set for each category, and the other bits should be zero. For example, a set of accesses cannot be both read only and write only, or if it is classified as read only, it must definitely not be write only. This condition is enforced to "double check" the validity of the classification with respect to the neural network training.

The inherent imprecision of neural nets allows us to train a net to produce a useful classification for patterns that are "very close" to some well-defined pattern (e.g., an almost sequential pattern can be classified as sequential). This is an advantage; it allows the file system to learn, through training, how it should classify new access patterns. Unfortunately, this flexibility makes training very difficult. It is impossible to classify and train on every possible trace

<table>
<tr><th>Category</th><th colspan="4">Category Features</th></tr>
<tr><td>Read/Write</td><td>Read Only</td><td>Write Only</td><td>Read-Update-Write</td><td>Read/Write Nonupdate</td></tr>
<tr><td>Sequentiality</td><td>Sequential</td><td>1-D Strided</td><td>2-D Strided</td><td>Variably Strided</td></tr>
<tr><td>Request Sizes</td><td colspan="2">Uniform</td><td colspan="2">Variable</td></tr>
</table>

Table 3.2: Input/output trace features.

Bit	Interpretation
0	read only
1	write only
2	read-update-write
3	read/write non-update
4	uniform request size
5	variable request size
6	sequential
7	1d strided
8	2d strided
9	variably strided

Table 3.3: Neural network classification key.

subsequence; therefore, classification is not always accurate. However, in practice, incorrect classification yields performance no worse than the system defaults.

3.4.3 Training the Neural Network

We generated an input training set that consisted of a suite of local access patterns covering a range of key areas in the three-dimensional access pattern space. Tables 3.4–3.7 list these patterns and their corresponding classifications. Table 3.3 shows how to interpret the classification identifiers.

For each test pattern, the file size is picked from a uniform random distribution to be a size between 1 KB and 10 MB. The access pattern modifier *partial* indicates that only a randomly selected subset of the file is accessed. An *overlapping* access pattern is sequential except that every access re-reads or re-writes a subset of the bytes accessed in the previous request. Our selection of parameters and patterns is fairly arbitrary. If the neural network is effective, our specific choices are inconsequential; the network will identify the salient features and generalize from them to interpolate gaps.

Pattern	Read/Write	Request Size	Classification
sequential	read only	random (1–2048)	0-5-6
sequential	write only	random (1–2048)	1-5-6
partial sequential	read only	random (1–2048)	0-5-6
partial sequential	write only	random (1–2048)	1-5-6
partial sequential	read only	fixed (1 KB)	0-4-6
partial sequential	read only	fixed (2 KB)	0-4-6
partial sequential	read only	fixed (8 KB)	0-4-6
partial sequential	write only	fixed (1 KB)	1-4-6
partial sequential	write only	fixed (2 KB)	1-4-6
partial sequential	write only	fixed (8 KB)	1-4-6
overlapping sequential	read only	random (1–2048)	0-5-9
overlapping sequential	write only	random (1–2048)	1-5-9
sequential	read only	fixed (1 KB)	0-4-6
sequential	read only	fixed (2 KB)	0-4-6
sequential	read only	fixed (8 KB)	0-4-6
sequential	write only	fixed (1 KB)	1-4-6
sequential	write only	fixed (2 KB)	1-4-6
sequential	write only	fixed (8 KB)	1-4-6

Table 3.4: Sequential neural network training patterns.

For each of these input test patterns, we computed input/output statistics on windows of ten accesses. Kotz [38] suggests that a smaller number of accesses suffices to predict sequential runs; however, we wish to recognize strided and random patterns, implying that a larger number of accesses is necessary. At least 3–4 accesses are necessary to detect a strided access pattern. One or two more accesses must be observed to accurately classify a two-dimensionally strided access pattern. With this intuition, we selected a moderately generous window size of ten. Ten accesses is small enough to respond quickly while being large enough to make accurate classifications of important access patterns.

We ignored the first and last five accesses, in turn, from each pattern trace and selected two windows from the beginning and end. If there are fewer than fifteen accesses in the trace (due to the randomly chosen file size) we generated new traces until there were enough accesses to classify the pattern according to these criteria.

Connection weights were randomly initialized to values in the range -1.0 to 1.0, and patterns were presented to the network in the order shown in Tables 3.4–3.7. The step width of the

Pattern	Read/Write	Request Size	Stride Size	Classification
strided	read only	1 KB	1 KB	0-4-7
strided	read only	2 KB	1 KB	0-4-7
strided	read only	8 KB	1 KB	0-4-7
strided	read only	1 KB	2 KB	0-4-7
strided	read only	2 KB	2 KB	0-4-7
strided	read only	8 KB	2 KB	0-4-7
strided	read only	1 KB	8 KB	0-4-7
strided	read only	2 KB	8 KB	0-4-7
strided	read only	8 KB	8 KB	0-4-7
partial strided	read only	1 KB	1 KB	0-4-7
partial strided	read only	2 KB	1 KB	0-4-7
partial strided	read only	8 KB	1 KB	0-4-7
partial strided	read only	1 KB	2 KB	0-4-7
partial strided	read only	2 KB	2 KB	0-4-7
partial strided	read only	8 KB	2 KB	0-4-7
partial strided	read only	1 KB	8 KB	0-4-7
partial strided	read only	2 KB	8 KB	0-4-7
partial strided	read only	8 KB	8 KB	0-4-7
strided	write only	1 KB	1 KB	1-4-7
strided	write only	2 KB	1 KB	1-4-7
strided	write only	8 KB	1 KB	1-4-7
strided	write only	1 KB	2 KB	1-4-7
strided	write only	2 KB	2 KB	1-4-7
strided	write only	8 KB	2 KB	1-4-7
strided	write only	1 KB	8 KB	1-4-7
strided	write only	2 KB	8 KB	1-4-7
strided	write only	8 KB	8 KB	1-4-7
partial strided	write only	1 KB	1 KB	1-4-7
partial strided	write only	2 KB	1 KB	1-4-7
partial strided	write only	8 KB	1 KB	1-4-7
partial strided	write only	1 KB	2 KB	1-4-7
partial strided	write only	2 KB	2 KB	1-4-7
partial strided	write only	8 KB	2 KB	1-4-7
partial strided	write only	1 KB	8 KB	1-4-7
partial strided	write only	2 KB	8 KB	1-4-7
partial strided	write only	8 KB	8 KB	1-4-7

Table 3.5: One-dimensionally strided neural network training patterns.

Pattern	Read/Write	Request Size	Stride 1	Stride 2	Classification
2-D strided	read only	1 KB	2 KB	1 KB	0-4-8
2-D strided	read only	2 KB	2 KB	1 KB	0-4-8
2-D strided	read only	8 KB	2 KB	1 KB	0-4-8
2-D strided	read only	1 KB	1 KB	2 KB	0-4-8
2-D strided	read only	2 KB	1 KB	2 KB	0-4-8
2-D strided	read only	8 KB	1 KB	2 KB	0-4-8
2-D strided	write only	1 KB	2 KB	1 KB	1-4-8
2-D strided	write only	2 KB	2 KB	1 KB	1-4-8
2-D strided	write only	8 KB	2 KB	1 KB	1-4-8
2-D strided	write only	1 KB	1 KB	2 KB	1-4-8
2-D strided	write only	2 KB	1 KB	2 KB	1-4-8
2-D strided	write only	8 KB	1 KB	2 KB	1-4-8
partial 2-D strided	read only	1 KB	2 KB	1 KB	0-4-8
partial 2-D strided	read only	2 KB	2 KB	1 KB	0-4-8
partial 2-D strided	read only	8 KB	2 KB	1 KB	0-4-8
partial 2-D strided	read only	1 KB	1 KB	2 KB	0-4-8
partial 2-D strided	read only	2 KB	1 KB	2 KB	0-4-8
partial 2-D strided	read only	8 KB	1 KB	2 KB	0-4-8
partial 2-D strided	write only	1 KB	2 KB	1 KB	1-4-8
partial 2-D strided	write only	2 KB	2 KB	1 KB	1-4-8
partial 2-D strided	write only	8 KB	2 KB	1 KB	1-4-8
partial 2-D strided	write only	1 KB	1 KB	2 KB	1-4-8
partial 2-D strided	write only	2 KB	1 KB	2 KB	1-4-8
partial 2-D strided	write only	8 KB	1 KB	2 KB	1-4-8

Table 3.6: Two-dimensionally strided neural network training patterns.

Pattern	Read/Write	Request Size	Classification
random	read only	random (1-16384)	0-5-9
random	write only	random (1-16384)	1-5-9
random	write only	fixed (1 KB)	1-4-9
random	write only	fixed (2 KB)	1-4-9
random	write only	fixed (8 KB)	1-4-9
random	read only	fixed (1 KB)	0-4-9
random	read only	fixed (2 KB)	0-4-9
random	read only	fixed (8 KB)	0-4-9
random	read/write	random (1-16384)	3-5-9
random	read/write	fixed (1 KB)	3-4-9
random	read/write	fixed (2 KB)	3-4-9
random read-update-write	read/write	random (1-16384)	2-5-9
random read-update-write	read/write	fixed (1 KB)	2-4-9
random read-update-write	read/write	fixed (2 KB)	2-4-9
random read-update-write	read/write	fixed (8 KB)	2-4-9
sequential read-update-write	read/write	random (1-16384)	2-5-6
sequential read-update-write	read/write	fixed (1 KB)	2-4-6
sequential read-update-write	read/write	fixed (2 KB)	2-4-6
sequential read-update-write	read/write	fixed (8 KB)	2-4-6

Table 3.7: Miscellaneous neural network training patterns.

Application	Sequentiality	Read/Write	Request Size
RENDER	Sequential	Read Only	Unpredictable
	Sequential	Read Only	Uniform
HTF	Sequential	Read/Write Nonupdate	Unpredictable
	Sequential	Read Only	Unpredictable
ESCAT	Sequential	Read Only	Unpredictable
	Sequential	Write Only	Unpredictable
	Sequential	Write Only	Uniform

Table 3.8: Application access patterns used in neural network verification.

gradient descent (η) was 0.2 and the momentum term α was 0. Training proceeded quickly and accurately (according to our validation of the trained network) using these default parameters.

3.4.4 Validating Neural Network Classification

To determine that the trained network could correctly classify real patterns it had not previously seen, we tested it on per-processor file traces obtained from characterization studies of parallel applications.[2] The applications we used were RENDER, HTF, and ESCAT.

All of these applications run on the Intel Paragon. RENDER [53] is a parallel application that generates three-dimensional views of planetary surfaces from satellite data and terrain elevation data. HTF [29], a version of the Hartree Fock algorithm, calculates interactions between subatomic particles. ESCAT [93] implements the Schwinger multichannel method for calculating low-energy electron-molecule collisions.

We manually inspected each file trace to determine the "correct" classification for each window, partitioned the input/output statistics computed from every ten accesses into test files for the correct classification, and verified that the neural net output was identical. Table 3.8 shows the correct classifications for input/output access patterns exhibited by these applications.

Although this experiment demonstrates that the neural network can correctly recognize real access patterns when trained on synthetic traces, it does not tell us anything about how the network will react when presented with traces that fall outside its training scope. To explore

[2] We omitted traces with fewer than ten accesses.

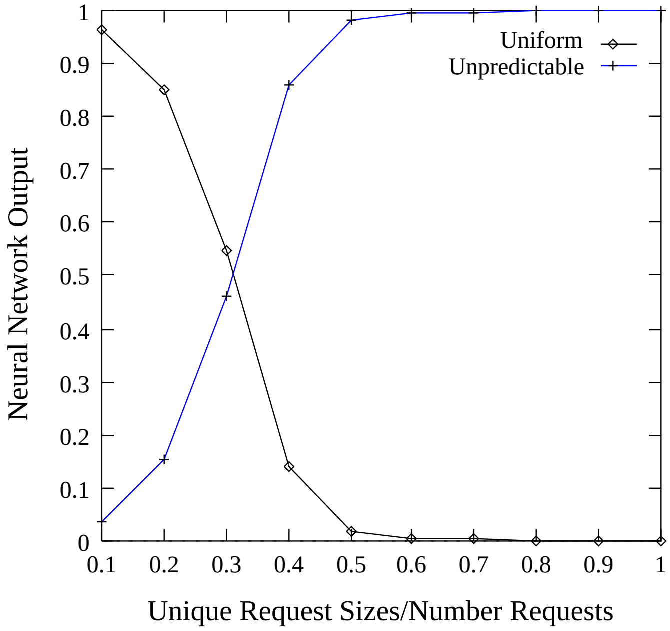


Figure 3.5: Varying the uniformity of request sizes.

this grey area, we varied the read/write mix, sequentiality, and request size uniformity of a random access trace and examined the values of the corresponding neural network outputs.

Figure 3.5 shows the effect of varying the number of unique request sizes on the neural network output corresponding to the uniform and unpredictable classifications. There is a large range for which no classification can be made, but when the number of unique request sizes is half of the total number of requests the pattern is classified as unpredictable.

Figure 3.6 shows the effect of varying the number of reads and writes on the read only and write only classifications. We start with a write-only random access pattern and gradually insert uniformly distributed reads into the mix. As the number of reads increases, the number of writes decreases so that they always sum to 10. Note that the distinction between read only, write only, and read/write no update is very precise; if both reads and writes occur in the trace, the classification is read/write no update. A valid classification exists for the entire range.

Finally, Figure 3.7 shows the result of varying the number of unique distances between random accesses on the neural net outputs related to sequentiality. If there are one or two unique distances between requests, this access pattern is classified as sequential. The crossover to unpredictably strided occurs at three accesses. The values for 1-D strided and 2-D strided remain (correctly) at low throughout.

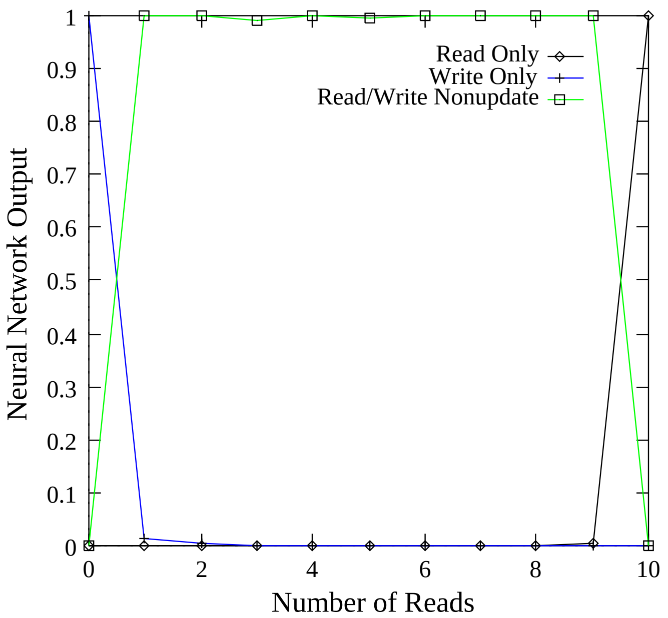


Figure 3.6: Varying the number of reads and writes.

3.4.5 Summary

Neural nets have many advantages. They are extremely efficient, once trained, and they can be re-trained to recognize new, important access patterns. For these reasons they provide an elegant framework for local classification. Unfortunately, this framework is potentially restrictive: learning and retraining can be imprecise, and neural networks cannot exploit information that is available about the functions we want to compute.

For example, there is no way to tell the neural net to compute the probability distribution function of the starting byte of the next access at every position in the file, information that would be useful in prefetching. For this reason, we examine an alternative method for classification based on hidden Markov models, described below.

3.5 Hidden Markov Model Classification

Qualitative classifications are limited to describing a simple set of structured access patterns. Many patterns are complex but repeatable, such as those generated by application calls to libraries such as the NCSA hierarchical data format (HDF) [57]. For example, often a process reads metadata from a structured file (small, localized accesses with irregular strides) and then

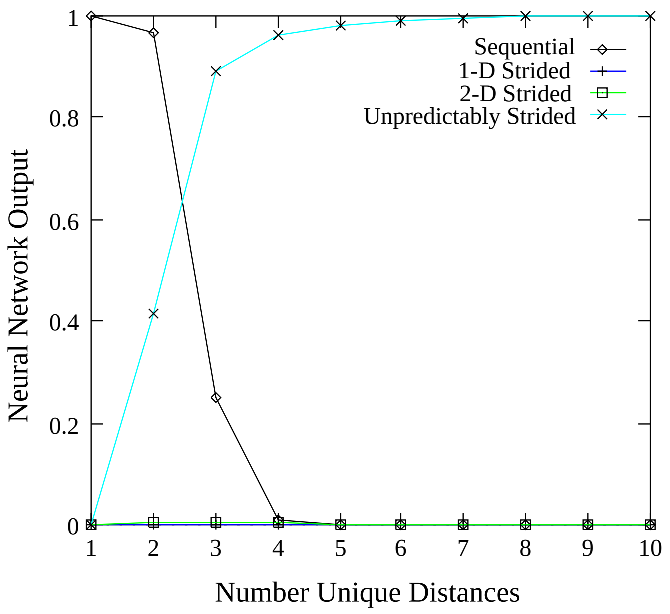


Figure 3.7: Varying the number of unique distances between requests.

accesses a data array within the file (with large sequential or strided accesses). This cycle may repeat several times as the HDF file is reopened and multiple arrays are read. Such input/output patterns defy simple qualitative classification, but they have definite structure that does not change between executions. Qualitative classification does not lend itself to description of more complicated access patterns.

Hidden Markov models [71] use data from previous executions to model access patterns probabilistically. HMMs are commonly used in speech recognition software, where the goal is to identify sub-words, words, or syntax based on probabilistic models. In its most general sense, the access pattern classification problem is similar to speech recognition; instead of calculating the probability that a spoken word or phrase has been correctly identified, we calculate the probability that a sequence of accesses matches a given classification.

A trained hidden Markov model with one state per file block has learned the probability distribution function of the next block given the current block. We can use an HMM in conjunction with quantitative trace information to compute probabilities of the features shown in Table 3.2 until the next classification. If the probability exceeds some threshold (e.g., probability of sequential access greater than 0.9) we change the corresponding policy.

We explain the theory behind HMMs in §3.5.1. In §3.5.2-§3.5.4 we describe our approach to modeling input/output access patterns using HMMs. In §3.5.5 we outline the HMM training process. Finally, in §3.5.6 we describe how to use HMMs for classification.

3.5.1 Hidden Markov Models

A discrete-time Markov process [36] 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.

3.5.2 Modeling Input/Output Patterns Using Markov Models

There are many possible ways to model access patterns using hidden Markov models; the choice of model description determines its predictive ability. One must decide what application

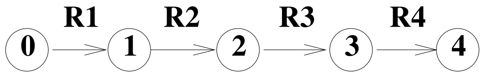


Figure 3.8: Markov model describing sequential reads.

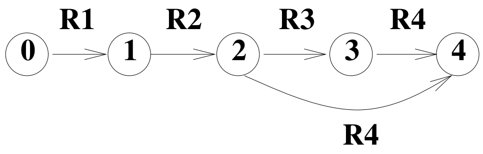


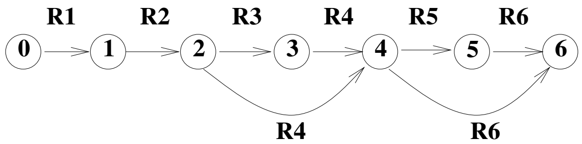
Figure 3.9: Degenerate HMM describing two possible control paths.

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.

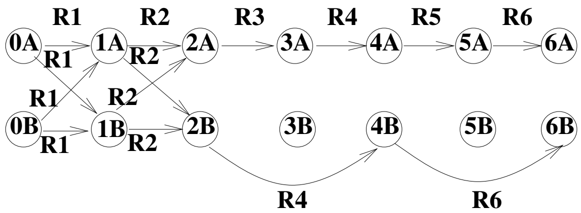
Segments are contiguous file regions of fixed or variable size. Ideally, segment size is chosen to correspond to an underlying file system caching unit or the application request size; within each segment, the access pattern is treated as sequential. Portions of the file that are regularly accessed can be consolidated into a single variable-sized state. For now, 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. The importance of permitting variable-sized segments to prevent state space explosion is discussed further in §3.5.4.

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 3.8 illustrates an HMM that models a sequential read pattern for a file with five blocks. The symbol **R1** denotes a read of block 1, **R2** a read of block 2, and so on. 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.

In Figure 3.8, the file access pattern is deterministic. Figure 3.9 illustrates a pattern where there is a conditional branch in the underlying program. Suppose that based 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.



(a)



(b)

Figure 3.10: Two HMMs for modeling conditional access patterns.

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.

3.5.3 Modeling Using Hidden Markov Models

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 3.10a and 3.10b illustrate two possible models of this behavior. The HMM in Figure 3.10a is a poor predictor because it does not remember the path it took to reach each even-numbered block. Figure 3.10b 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 3.10 can also be viewed as a composite of two individual HMMs; in essence the HMM clusters sequences, as described in [83].

3.5.4 Memory Requirements of Hidden Markov Models

Our chosen access pattern representation raises the issue of state space explosion. Maintaining the transition probabilities for all possible transitions between n small blocks of a large file can require up to $O(n^2)$ storage. To reduce this upper bound on HMM storage requirements, we exploit common access pattern characteristics.

First, our HMM implementation is designed to be memory-efficient for expected file access patterns. In regular access patterns where the request size corresponds roughly to the HMM segment size, there will be a relatively small number of transitions to and from each state. In other words, we expect the state transition matrix to be sparse. To save memory, transitions from each state are represented as a linked list rather than preallocating memory for a complete transition matrix.

Second, we postprocess a trained HMM to consolidate sequential accesses, which occur frequently in scientific applications, into segments that may be much larger than the underlying file system blocks. Associated with each state (representing a file segment) are its starting and ending bytes. When the sole transition out of a state is to a state representing consecutive bytes, we can simply merge the states. Transitions to either of the original states become transitions to the newly-created merged state. This allows an extremely compact representation for many access patterns with sequential components by reducing the number of segments for which transitions are stored.

Third, we can eliminate unnecessary transitions, which can consume a lot of memory and are not useful for prediction. For example, consider a hypothetical irregular access pattern, as represented by the HMM fragment pictured in Figure 3.11. In the general case, there are potentially n transitions from every state, where n is the number of segments in the file; in this example each state has only five transitions. However, the predictive power of this model is very limited. For example, the most likely state transition from the starting block (56, 24, 43) occurs with a probability of only 0.35. Other transition sequences are even less probable. Therefore,

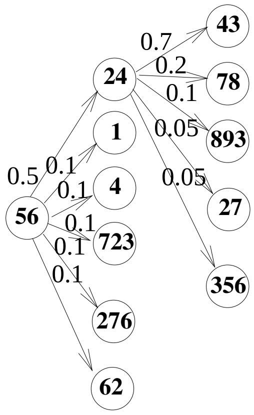


Figure 3.11: HMM fragment representing an irregular access pattern.

we can prune the state transition tree, eliminating unlikely transitions, without greatly reducing the utility of the model. The threshold for defining "unlikely" depends upon how the model will be used; i.e., the required accuracy of predictions.

Using these two optimizations, for access patterns common in scientific applications, HMM storage requirements grow at most linearly with the file size. Further reduction is possible by using more compact representations for regular access patterns (e.g., strided).

3.5.5 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. An initial segment size must be selected in advance; it should ideally correspond to the default underlying file system caching unit. The training module maintains counts of all block read or write transitions. Because most files involve only a single access pattern, a degenerate HMM with one state per file block (equivalent to a Markov model) suffices. In this case, 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. Only one application execution is necessary for training.

When a file has several significantly different access patterns (e.g., Figure 3.10), the training process is more complicated. We apply a simplified version of a clustering algorithm proposed in [83].

First we train an HMM on each individual access pattern as described above, yielding some set of K unique HMMs, each with the n states. A single composite HMM with $K * n$ states can be formed from the K components by creating a block-diagonal transition matrix A from the component HMM transition matrices $(A_1 \ldots A_K)$ as follows

$$A = \begin{pmatrix} A_1 & 0 & \ldots & 0 \\ 0 & A_2 & \ldots & 0 \\ 0 & 0 & \ldots & A_K \end{pmatrix}$$

For example, consider the two access patterns represented by Figure 3.10a. The sequential pattern has transition matrix

$$A_1 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

and the sequential/strided pattern has the transition matrix

$$A_2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

The initial state probabilities of the composite HMM are chosen to reflect the relative probabilities of each individual access pattern (by dividing the number of occurrences of the pattern by the total number of executions). These probabilities can evolve over time while executing the application in production mode.

Suppose the access pattern described by A_1 occurs in practice with probability 0.75, and the pattern of A_2 occurs with probability 0.25. The composite matrix A formed from A_1 and

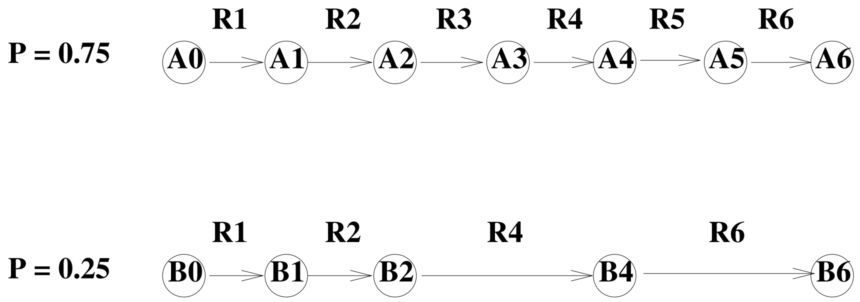


Figure 3.12: Composite HMM.

A_2 describes the HMM in Figure 3.12, where the initial probabilities determine the starting state (A0 or B0). Note, however, that there is no crossover between the patterns.

To obtain a better descriptive HMM, it is possible to retrain the composite HMM (with transition matrix A) with all the access patterns, using the classic Baum-Welch algorithm for training HMMs [71]. This composite HMM is no longer equivalent to a Markov model and cannot be trained by simply calculating transition probabilities; the Baum-Welch algorithm is an iterative procedure that adjusts the model parameters to maximize the likelihood of generating each of the sequences given the model. The goal of this additional training step would be to capture the underlying program logic in the state sequences of the trained HMM by observing the different sequences of input/output accesses that are generated. However, we believe that the number of unique file patterns observable in scientific applications is likely to be limited and clearly distinct, making this final step unnecessary. A more sophisticated approach to clustering access pattern sequences that automatically determines the number of sequences required to fit the data is proposed in [83].

Techniques described in §3.5.4 to condense the memory used by the HMM by consolidating sequential accesses or pruning low-probability transitions are applied to the model after training.

3.5.6 Qualitative Classifications and Hidden Markov Models

A trained HMM represents an access pattern differently than a neural network classification; it accurately models transitions between file blocks at the expense of detailed request information. For example, information about small accesses within a segment is lost. In §4.4 we explain these tradeoffs with respect to the utility of the model for policy control.

To enable comparison of HMM and ANN classification, we have defined a library of routines that, together with the input/output statistics calculated for neural network training, estimate probabilities of observing the features shown in Table 3.2 until the next classification. We say "estimate" because although one can compute exact probabilities based on the trained HMM, for specific features, reclassification intervals and HMM structures, the cost of doing so might be prohibitive.

Our approach is to exploit previously calculated quantitative statistics wherever possible and to avoid full searches of every path through the HMM, using the knowledge that precise probabilities are not important for tuning policies. In other words, when the file system makes a query about a certain access pattern feature, we attempt to round the answer quickly to a "yes" or "no" as defined by probability thresholds (defined arbitrarily to be $< .1$ is no and $> .9$ is yes).

All HMM classification routines accept either some number of accesses or a range of segments (blocks). A computed classification is valid for either that number of segment accesses or within the specified segments. In contrast to ANN classification, which uses the previous window of ten accesses to classify the pattern, HMM classification starts from the current file position and classifies based on the expected number of accesses until reclassification.

The algorithms described below assume there is only one access pattern per file. They would need to be modified slightly to use a complex HMM that represented several possible access patterns.

3.5.6.1 Read/Write Mix

To determine if an access pattern is read only within a specified range, we inspect all possible transitions from blocks in the given range to blocks within the range. If any of the valid transitions is a write, the pattern is not classified as read only. Write only classification is analogous to read only classification. An access pattern is read-write if within the specified range, there exists the possibility of both reads and writes.

Note that this is an unforgiving classification criteria; the possibility, however slight, of a single write in an otherwise read only file makes it read/write. We assume, based on characterization data from SIO applications, that if reads and writes both occur they will each happen at some point with high probability. Coincidentally, validation of ANN classification revealed

(Figure 3.6) that our trained neural network independently learned the same heuristic for the access pattern tested.

3.5.6.2 Request Size Uniformity

The trained HMM retains no information about application request sizes. We fix the HMM segment size at training time to correspond to the underlying file system cache block unit; this is an example of the fuzzy boundary between classification and policy control. The sizes of the application-level requests are important only in the extent to which they stress the file system.

Therefore, we rely upon the quantitative trace statistics to supply the average request size and the number of unique request sizes. Heuristics can be used with these metrics to decide if the request size is uniform or variable (e.g., the request size is variable if the number of unique request sizes is greater than three) or we can use ANN classification instead.

3.5.6.3 Sequentiality

We determine if an access pattern is sequential by computing the probability of accessing n blocks sequentially from the starting block. Although we cease computation if the sequential probability falls below 0.9, this probability can be computed exactly in $O(n)$ time.

One and two dimensionally-strided access patterns can similarly be identified by the HMM. For example, for a one-dimensionally strided pattern, if the application request size matches the HMM segment size, the gaps (as measured by number of blocks) between consecutive accesses will be identical. If the application request size does not equal the HMM segment size, the gaps are only approximately the same. Inspection of the HMM, in conjunction with the number of unique request sizes and number of unique stride sizes, identifies the pattern as regularly or variably strided.

3.6 Summary

Our motivation for access pattern classification is to generate a qualitative access pattern description that can be used together with quantitative access pattern statistics to select and tune file system policies. In this chapter, we have proposed an access pattern space and described two complementary methodologies for generating local classifications. The first method uses a neural network to produce a qualitative classification from input/output statistics. The second method uses previous execution information to train a hidden Markov model, in effect constructing a probabilistic representation of file accesses. As we will show, this latter approach can express a larger language of access patterns at the expense of training from previous execution data.

Chapter 4

Intelligent Policy Selection

A file access pattern classification is platform-independent and unique to a particular application execution. However, an optimal choice of file system policies for a particular access pattern is system-dependent. A portable file system uses the classification to tune file system policies for each input/output platform. By making policy decisions to suit the application requirements and the system architecture, not only is input/output performance portable over a variety of platforms, but the file system can provide better performance over a range of applications than it could by enforcing a single system-wide policy. This adaptivity occurs transparently, without application hints or optimizations.

To provide a framework for experiments involving automatic policy control, we have extended the Portable Parallel File System (PPFS), a flexible file system testbed, as described in §4.1. In §4.2 we describe a simple algorithm for policy control. We determine the thresholds for policy parameters experimentally; §4.3 describes calibration of this algorithm for the experimental platform used for all uniprocessor experiments. Policy selection using HMM classifications is slightly different from selection using neural network classifications, as described in §4.4. Finally, we summarize in §4.5.

4.1 Portable Parallel File System (PPFS)

Evaluation of local and global classification and adaptive policies necessitates a flexible file system testbed. For our experiments, we extended the Portable Parallel File System (PPFS) framework. PPFS is a portable input/output library designed to be an extensible testbed

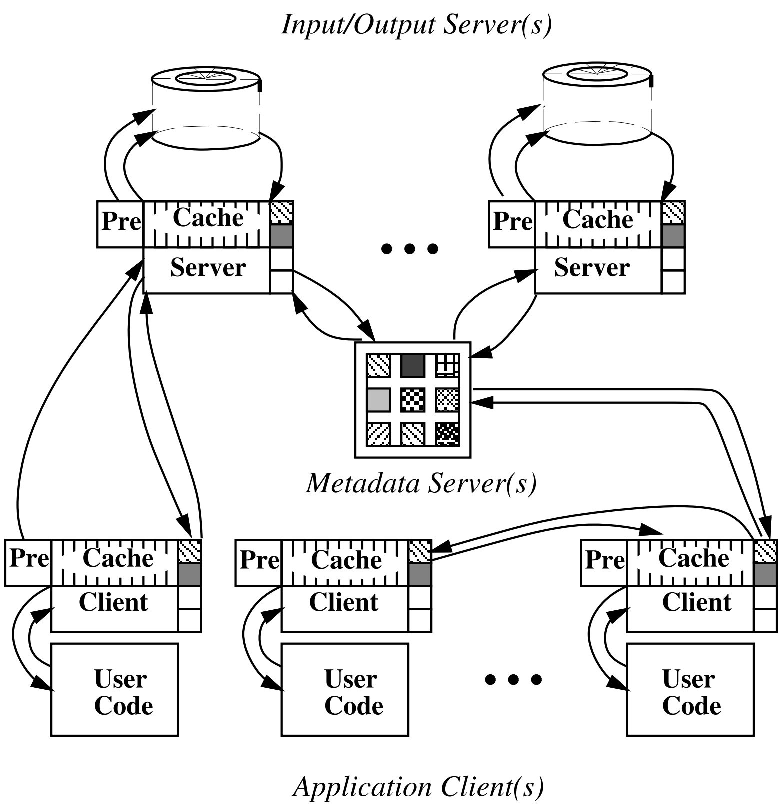


Figure 4.1: PPFS design.

for file system policies. A rich interface for application control of data placement and file system policies makes it exceptionally well-suited for the experiments conducted in this research. Below we describe the PPFS design and extensions completed to facilitate adaptive file policy experiments.

4.1.1 PPFS Design

Figure 4.1 shows the basic 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 is able to issue input/output requests and specify caching and prefetching polices for all levels of the system. Clients either satisfy the requests or forward them to caching agents (processes that manage intermediary caches) or servers (abstractions of remote input/output devices). Clients,

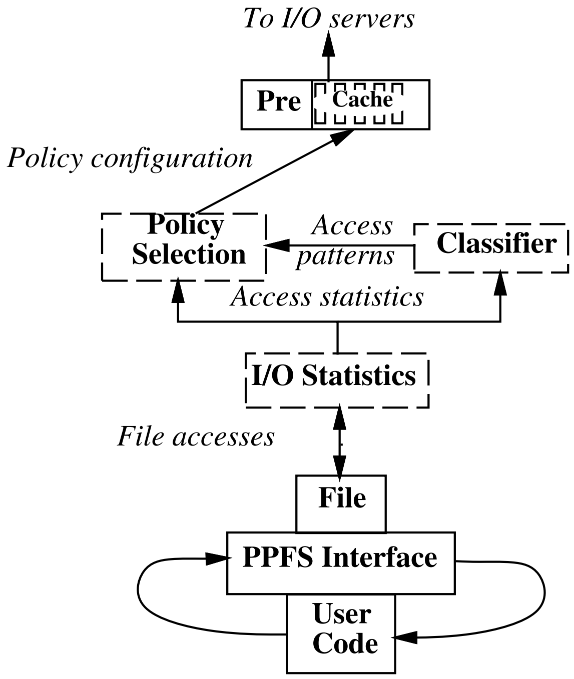


Figure 4.2: PPFS local classification extension.

caching agents, and servers each have their own caches and prefetch engines. All "physical" input/output is performed through an underlying UNIX file system.

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, the user 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.

Experiments using large research codes on the Intel Paragon XP/S 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 [18]. Simple access pattern hints and cache policy controls yield large performance increases [31].

4.1.2 PPFS Extensions

The original PPFS interface provides the application with a rich set of manual file system policy controls and structured data access functions, but the rules guiding their use are *ad hoc*. Ideally, the file system should *automatically* infer appropriate policies from low-level applica-

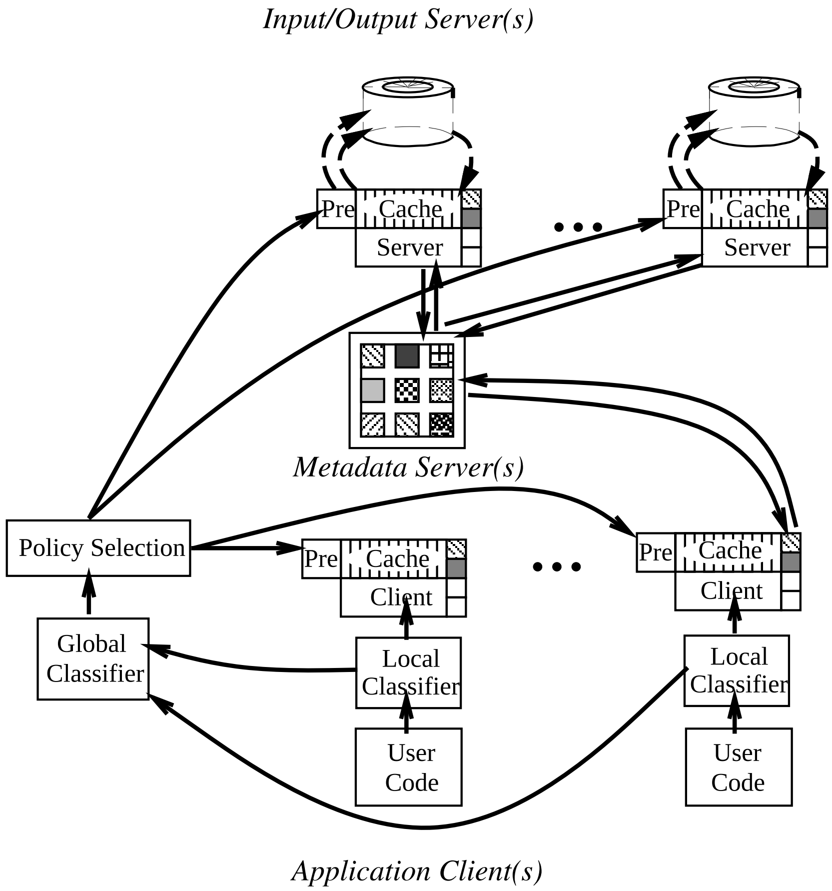


Figure 4.3: PPFS with global classification extension.

tion access patterns, lessening the application programming burden and the likelihood of user misconfiguration. This goal has motivated three major extensions to the existing PPFS design: a UNIX-style input/output interface, support for local classification and adaptive file system policy selection, and support for global classification and policy control.

PPFS files are accessed by fixed or variable-length records that determine the cache block size. Unfortunately, this application programming interface (API) already encodes a substantial amount of information about the access pattern (e.g., request size), as well as file system policy information (e.g., cache block size). To separate classification from the API and from policy decisions, we have initially focused on classification using a UNIX style I/O interface. UNIX imposes no file structure; files are simply streams of individually addressable bytes. Each file access can be represented as an offset, a read or write, and a request size. Because this interface is so simple, our classification system can use only this limited information to classify access patterns without loss of generality. We anticipate that in the future, more detailed file access information can be used to generate more complex classifications.

We have replaced PPFS file system controls in our extension by an adaptive access pattern classification and file system policy selection mechanism. During program execution, an input/output statistics module monitors the file access stream and computes the statistics needed by the classifier module. PPFS uses the classification to select and tune prefetching and caching policies. Figure 4.2 illustrates the interaction of the local classification extension with the original PPFS components.

Global classifications, described fully in Chapter 6, are formed from a combination of local classifications and additional access pattern information. To support global classification, we added a dedicated global classification server. Figure 4.3 shows PPFS with global extension modifications.

Each time a local classifier identifies an access pattern, it sends the file identifier, the local classification, the timestamps marking the start and end of the classification, and the byte ranges accessed during the classification period to the global classifier. The global classifier saves this information from each processor until it can classify the global file access pattern.

When the global server classifies an access pattern, it sends the input/output servers and clients a description of the appropriate policies. 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.

With the restricted UNIX file input/output interface, local and global classification and subsequent policy selection replace the manual specification of file system policies. This is a first step toward unification of manual and automatic file system policy selection; ultimately the user would be able to specify additional access pattern hints that the file system could incorporate into its decisions.

In summary, the PPFS extension is a simpler application interface to the original PPFS infrastructure. Instead of directly controlling file system policies and using structured data access routines, the application uses unstructured UNIX-style input/output, from which the file system infers file access patterns, and tunes policies accordingly.

4.2 A Simple Algorithm for Policy Control

PPFS provides the necessary framework for using classifications to tune file system policies. Abstractly, the file system 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 classification is sequential, the file system can assume that file access will continue to be sequential. If the classification is read only the file system can prefetch aggressively; if it is write only a write-behind policy might be efficient. When the classification is regularly (1-D or 2-D) strided, the file system can take advantage of this information to prefetch anticipated blocks according to the stride size and adjust the cache size.

Figure 4.4 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 4.4 depend on the particular hardware configuration and are determined experimentally.

4.3 Policy Selection Parameters

The system used for all of our uniprocessor experiments is a Sun SparcServer 670 running SunOS 4.1.3, with 64 MB of physical memory and a local SCSI disk. In this section, we describe how we identify the parameters for the policy selection algorithm of Figure 4.4.

4.3.1 System Defaults

Because the most common UNIX access patterns consist of small, sequential reads, we selected as our default cache size and cache block size values that are optimal for read only sequential accesses in our PPFS implementation. Figure 4.5 plots a range of request sizes against throughput for a sequential read pattern. PPFS caching is disabled, so the throughput reflects the underlying UNIX file system. We can see that throughput begins to level off after the request size exceeds 32 KB, indicating that 32 KB is a good choice for the underlying cache block size.

```
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 4.4: Dynamic file system policy selection (example).

The amount of memory allocated to the cache does not directly affect performance of a sequential access pattern, because once a block has been accessed it is not reused. Therefore, we arbitrarily chose 1 MB to be the default cache size.

File cache sizes can vary across operating systems from a small fixed percentage of main memory (e.g. 10% for Ultrix) to a dynamic allocation that can be as high as 74% for SunOS 4.1 [67]. Our control over memory allocated to a particular file's cache is limited to two sizes, small (the default size) and large. We chose an upper bound on our experimental PPFS cache of 25 MB because this sufficed to eliminate PPFS cache thrashing for all our experiments without adversely perturbing the underlying, dynamicly resized file system cache. In essence, with this choice we attempted to simulate, at the user level, the dynamic file cache expansion that could occur in SunOS 4.1 with sufficient physical memory.

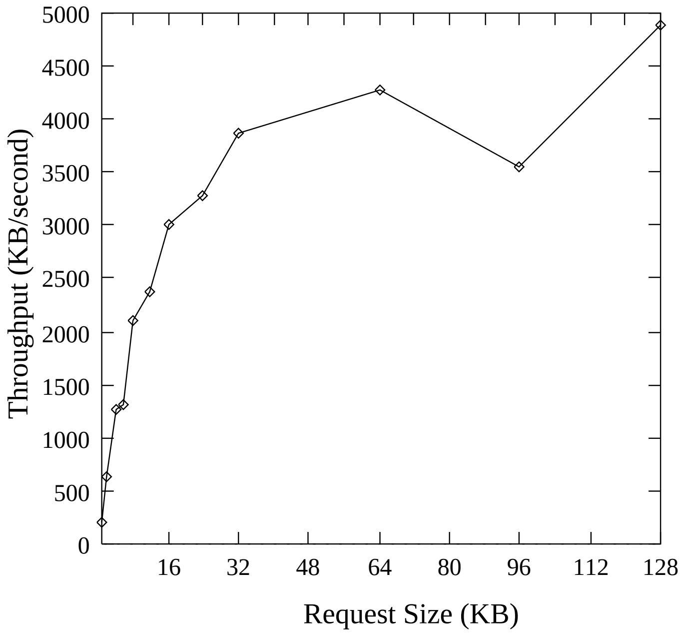


Figure 4.5: Read sequential access pattern.

Because PPFS cache management overhead has been highly optimized, small (< 8 KB) sequential reads perform well under these defaults. The default cache replacement policy is Least Recently Used (LRU). Adjustments are made whenever they might potentially yield performance improvements given the access pattern classification.

4.3.2 Policy Modifications for Sequential Access Patterns

When an access pattern is sequential, depending on the other classification features, there are several modifications to the default policies that may improve performance. For example, when the access pattern is sequential and write-only, we assume that the file is being written sequentially. A Most Recently Used (MRU) cache block replacement policy is more sensible than LRU given this implementation. Writes are buffered until the cache is full, and then only the most recently used block is written back as it is filled. This way, the first few blocks of the file remain cached should they be referenced (for header information) in the future.

The same logic holds for sequential read only patterns, yet we do not select MRU replacement for them. This is because caching is less useful for read only patterns; the PPFS buffering scheme cannot improve performance over UNIX unless requests are reasonably large, rendering UNIX standard input/output buffering unnecessary. Figures 4.6 and 4.7 show performance

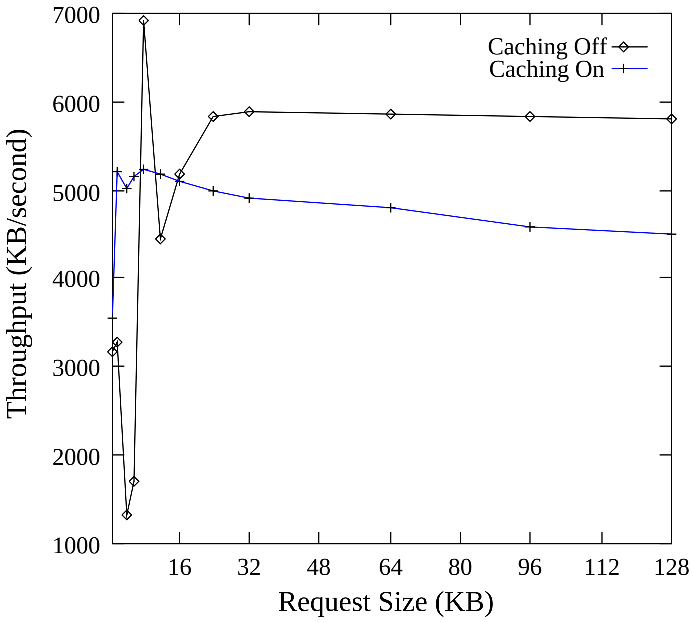


Figure 4.6: Read sequential access pattern.

for read only and write only sequential access patterns with and without caching. Based on Figure 4.6, we disable caching on sequential reads when the average request size is greater than or equal to 8 KB. Figure 4.7 shows that caching is of benefit in sequential write-only access patterns regardless of request size.

4.3.3 Policy Modifications for Nonsequential Access Patterns

File system policies must be modified for regularly and variably strided patterns to avoid cache thrashing. As with sequential access patterns, caching of large accesses is inefficient, but the thresholds are different than for sequential access patterns.

When the access pattern is regularly (1-D or 2-D) or variably strided, it is inefficient to cache large requests. The cost of performing the UNIX read or write is smaller than the overhead of managing the cache with such small chance of cache hits. However, if they are small, we increase the cache size to something reasonably large, in this case 25 MB, to avoid causing thrashing within PPFS that would not otherwise occur in a normal file system cache. Because blocks are allocated only as needed, the cache can never be larger than the file size.

Figure 4.8 shows this crossover point for a write only random access pattern with a large cache; it occurs between one and two kilobytes. Figure 4.9 shows the same data for a read only

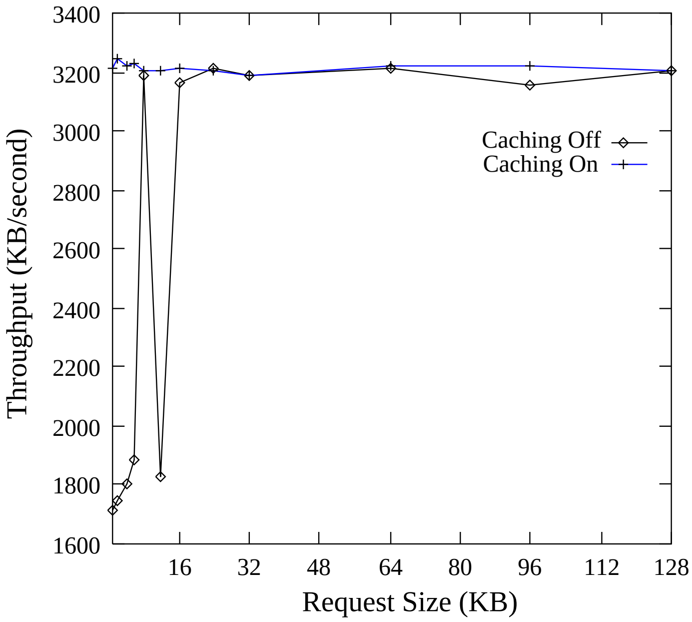


Figure 4.7: Write sequential access pattern.

random access pattern; random reads perform better without caching than with, but for small request sizes the performance differential is very small. Thus, we have set our threshold to be 1.5 KB for this architecture; this is the point at which we have determined that the overhead of cache management is greater than the cost of incurring a physical input/output if the hit ratio is likely to be low.

The algorithm of Figure 4.4 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 we show that even this simple policy suite suffices to yield large performance increases over that possible with standard UNIX file policies.

4.4 Policy Control Using Hidden Markov Models

The power to improve performance by tuning file system policies is limited by the descriptive and predictive power of a classification. Although we attempt to create a crisp boundary between classification and policy selection, in practice, it is less clear.

Qualitative classifications describe a simple set of structured access patterns. However, from a performance standpoint, certain access pattern features are more important than others. It may be irrelevant whether application request sizes are uniform or variable if a buffer cache

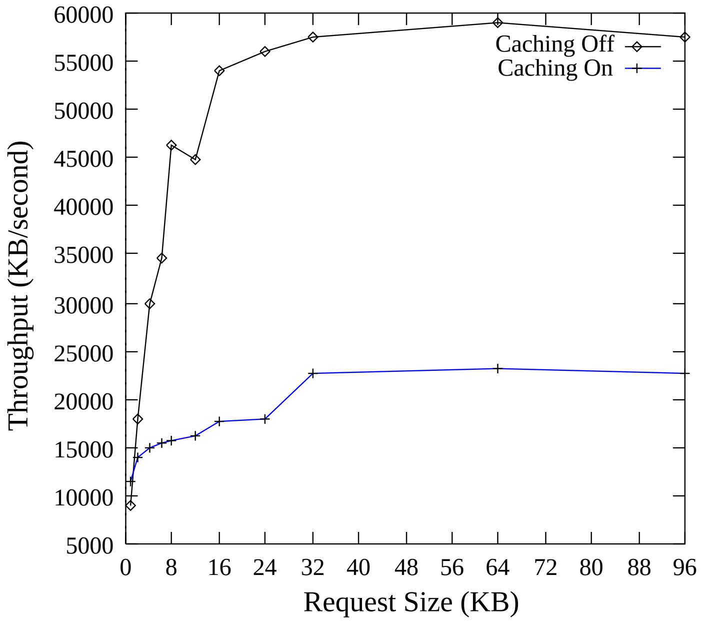


Figure 4.8: Write random access pattern, varying request size.

translates them to fixed size input/output requests. At the other extreme, the fundamental principle behind effective caching and prefetching is estimation of future accesses. Classifications of sequential and strided access patterns, together with quantitative request and stride sizes, provide this information.

Many patterns are complex but repeatable; such input/output patterns defy simple qualitative classification, but they have structure that is invariant across executions. One of the major reasons for complex structured access patterns is the use of higher level input/output libraries that hide file structure from the application. For example, NCSA's hierarchical data format (HDF) [57] is a platform independent data format for exchange of scientific data that has been adopted as a standard format for storing satellite data from NASA's Mission to Planet Earth project. The National Space Science Data Center's Common Data Format (CDF) [75] is another self-describing data abstraction for storage and manipulation of multidimensional data. CDF has been adopted by the space physics community as a standard for data sharing and analysis. Such interfaces already constrain performance of input/output intensive codes in sequential UNIX file systems; these libraries will exhibit these problems with even greater severity in parallel file systems.

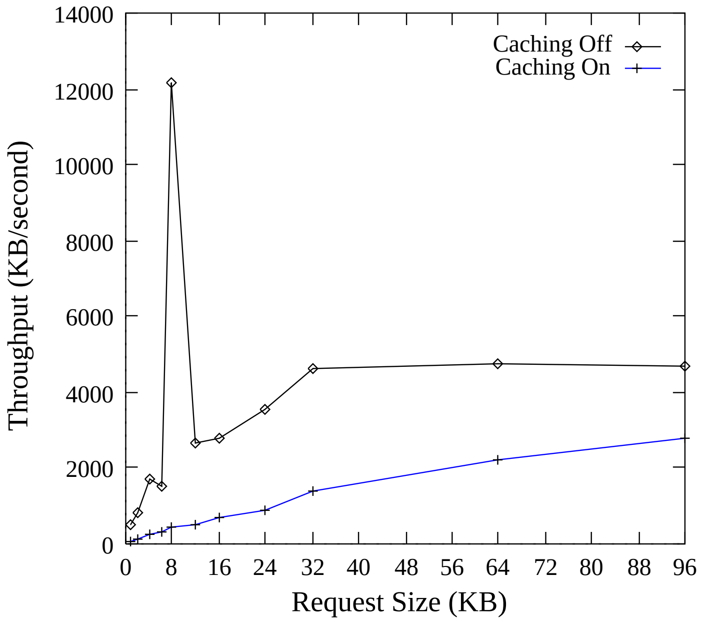


Figure 4.9: Read random access pattern, varying request size.

It is important to identify such structure for control of caching and prefetching policies. 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 HMM with one state per file block learns the probability distribution function of accessing the next block given the current block, learning with great accuracy a variety of likely request sequences.

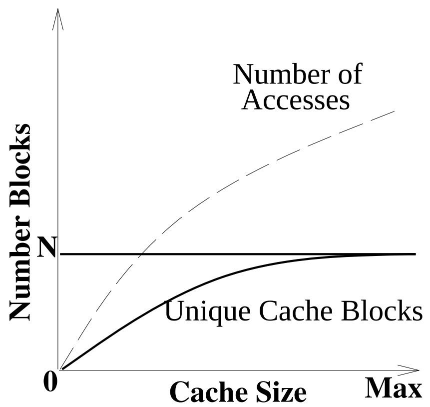


Figure 4.10: Allocating cache space.

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 4.10 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, 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. While ANN classifications can be saved between executions and used to provide initial policy decisions, this method still utiliizes previous execution data. Furthermore, since the same file might have multiple access patterns, access pattern probabilities would need to be maintained externally. A trained HMM models all observed access patterns.

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.

4.5 Summary

This chapter explains how to use automatically generated access pattern classifications to tune file system policies. To support our automatic policy control experiments, we modified the Portable Parallel File System (PPFS) testbed. We describe and calibrate an algorithm for policy control for a UNIX workstation. Finally, we investigate issues specific to using trained HMMs for policy control.

Chapter 5

Local Access Pattern Classification Experiments

Having defined and described the methodology for local classification and policy control, in this chapter we explore how this technique can be used in practice to dynamically improve performance. Neural networks and HMMs create access pattern classifications for uniprocessor benchmarks and applications. PPFS uses these classifications together with quantitative input/output statistics to select appropriate file system policies.

In §5.1 we examine local classification performance on simple benchmarks with fixed and varying access patterns. Hidden Markov model classification can recognize a wider range of access patterns than ANN classification; in §5.2 we describe circumstances where this is a significant advantage. Although UNIX file access patterns are most often sequential, portable input/output libraries such as NCSA's Hierarchical Data Format (HDF) hide physical input/output, yielding complex access patterns. HDF is a crucial building block for satellite data processing applications; we explore HDF performance in §5.3. Finally, in §5.4 we examine the performance improvements possible using local classification and intelligent policy control on input/output intensive satellite data processing applications.

The experimental platform for all experiments 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.

5.1 Access Pattern Classification Experiments

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.

5.1.1 Simple Benchmarks

To compare the performance of ANN and HMM classification and policy selection to UNIX buffered input/output, we constructed simple benchmarks of sequential, strided, and random access patterns. Each benchmark reads or writes a 20 million byte file in records of varying sizes. The UNIX times are obtained using buffered standard input/output. PPFS times include the overhead of PPFS library input/output calls that eventually make UNIX system calls, using HMM or ANN classification to classify the access pattern and select appropriate caching policies as described in §4. Classification and policy selection occurs after the first ten accesses, and every two million bytes thereafter. These timings are computed to a 95% confidence interval using a warm UNIX file cache.

5.1.1.1 Sequential Benchmarks

Table 5.1 shows performance for read and write sequential benchmarks with record sizes ranging from 20 to 2 million bytes. The PPFS policy selection strategy for this access pattern is to disable caching for requests larger than 8 KB. Cache management overhead causes write timings to be slightly worse under PPFS than UNIX. This is because UNIX buffers small input/output requests and flushes larger requests to disk; PPFS does the same thing, except that the buffer size is larger (1 MB vs. 8 KB). The overhead of user-level cache management in PPFS overshadows the savings obtained by generating fewer disk writes. For the read sequential benchmarks, we see that PPFS is slightly worse than UNIX for the 20–2000 byte request sizes; UNIX buffered input/output is optimized for such patterns and PPFS is not as efficient. However, performance is improved for request sizes greater than 2000 bytes. One notices that the performance improvement curve for sequential requests drops off at the largest request size; this is because the optimal behavior in the PPFS implementation is to disable

Benchmark	Record Size (bytes)	UNIX	ANN		HMM	
		Time	Time	Speedup	Time	Speedup
Read	20	4.28	6.32	0.67	6.69	0.64
Read	200	2.20	2.82	0.78	2.80	0.78
Read	2000	2.00	2.37	0.84	2.52	0.79
Read	20000	2.12	1.40	1.51	1.33	1.59
Read	200000	2.16	1.52	1.42	1.46	1.48
Read	400000	2.35	2.04	1.15	1.64	1.43
Read	800000	2.38	2.30	1.03	1.94	1.23
Read	1000000	2.57	2.70	0.95	2.05	1.25
Read	2000000	3.04	4.52	0.67	2.91	1.04
Write	20	7.26	8.27	0.88	8.04	0.90
Write	200	6.66	7.60	0.88	7.37	0.90
Write	2000	6.73	7.27	0.93	6.97	0.97
Write	20000	6.63	7.19	0.92	7.12	0.93
Write	200000	6.57	10.13	0.65	10.38	0.63
Write	400000	6.61	9.83	0.67	10.66	0.62
Write	800000	6.71	10.69	0.63	10.34	0.65
Write	1000000	6.78	10.20	0.66	10.36	0.65
Write	2000000	6.84	7.62	0.90	7.16	0.96

Table 5.1: Sequential access pattern (synthetic benchmark).

Benchmark	Record Size (bytes)	Dimensions X	Dimensions Y	UNIX Time	ANN Time	ANN Speedup	HMM Time	HMM Speedup
Read	20	1000	1000	895.13	34.50	25.95	34.32	26.08
Read	200	1000	100	35.74	8.89	4.02	9.02	3.96
Read	2000	1000	10	3.89	2.24	1.74	2.36	1.65
Read	20000	100	10	2.23	1.38	1.62	1.39	1.60
Read	200000	10	10	2.16	1.52	1.42	1.49	1.45
Write	20	1000	1000	422.37	38.56	10.95	38.78	10.89
Write	200	1000	100	24.05	13.83	1.74	14.20	1.69
Write	2000	1000	10	13.65	9.63	1.42	9.92	1.38
Write	20000	100	10	10.79	10.73	1.01	10.07	1.07
Write	200000	10	10	7.06	7.07	1.00	7.72	0.91

Table 5.2: Strided access patterns (synthetic benchmark).

caching. However, in this experiment there are only ten requests so this optimization is never performed.

5.1.1.2 Strided Benchmarks

Table 5.2 shows the results from the read and write strided benchmark. This benchmark reads a two-dimensional array of bytes with x and y dimensions as shown along the y-axis, in chunks of the specified record size. When the classifier detects that the access pattern is strided and the record size is larger than 1.5 KB, caching is disabled. If it is smaller, we increase cache size to prevent thrashing. In most cases this strategy provides better performance than UNIX buffered input/output because PPFS is able to buffer blocks that will be re-read in the future, or consolidate writes, if requests are small, despite the lack of locality. The classifiers differ in their abilities to recommend an enlarged cache size; the HMM classifier can calculate exactly how much space is necessary to retain a working set of blocks within some probability when it detects a strided access pattern. The ANN classifier must rely upon some default value that represents the maximum user cache size. However, in these benchmarks, because only one file is active, there is sufficient cache space that HMM classification poses no particular advantage.

5.1.1.3 Random Benchmarks

Table 5.3 shows performance for the random read and write benchmarks. The random benchmarks read or write a total number of bytes equal to the file size (20 million bytes) in

Benchmark	Record Size (bytes)	UNIX Time	ANN Time	ANN Speedup	HMM Time	HMM Speedup
Read	20	790.06	67.64	11.68	69.45	11.38
Read	200	79.68	9.84	8.10	11.33	7.03
Read	2000	8.90	4.29	2.07	4.42	2.01
Read	20000	2.43	1.52	1.60	1.54	1.58
Read	200000	2.06	1.43	1.44	1.41	1.46
Read	2000000	2.49	2.49	1.00	2.41	1.03
Write	20	606.64	64.45	9.41	67.58	8.98
Write	200	120.06	14.95	8.03	15.59	7.70
Write	2000	45.09	9.95	4.53	9.60	4.70
Write	20000	16.27	16.43	0.99	15.72	1.03
Write	200000	7.37	7.41	0.99	7.30	1.01
Write	2000000	5.99	5.97	1.00	6.53	0.92

Table 5.3: Random access pattern (synthetic benchmark).

segments of the specified record size. As for strided access patterns, PPFS selects either to enlarge the file cache if the average record size is < 1.5 KB, or to disable caching if it is greater than 1.5 KB. In most cases, PPFS performance either equals or surpasses that of UNIX standard input/output. Again, the observation that the pattern is random coupled with information about the average request size enables PPFS to choose more efficient caching policies. As with the strided access pattern, the HMM classifier can make a more specific prediction about the smallest cache size that will capture the locality of the program.

5.1.1.4 Overview

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

Classification costs vary with frequency of reclassification and with method. ANN classification cost is a constant factor each time it is invoked. In the read sequential benchmark with record size 20 bytes, ANN classification time accounts for approximately .03% of execution time. HMM classification cost varies not only with frequency of classification, but with the access pattern. However, for this benchmark HMM classification accounted for only .04% of execution time.

In addition, there is the added expense of loading the HMM description. We store the HMM description in an ASCII format that is readable, but not compact; each node is listed followed by its links. The file open time using HMM classification, which includes the time to load the HMM description, accounts for 1.3% of the execution time vs. 0.03% for ANN classification. For randomly accessed files, the size of the HMM description is larger (to represent the additional links) so loading it will take more time. Although there is room for much optimization in methods for HMM storage and classification, costs of both classification methods are still very small with respect to application execution time. Furthermore, as our experiments show, the speedups obtained through classification-based policy selection far outweigh the actual overhead of classification.

Classification costs are the only factor in any timing differences between the two classifiers. Although the HMM classifier is capable of more intelligent cache space allocation and is more responsive to access pattern changes, the performance of these simple benchmarks does not benefit from these advantages. Classifications and corresponding policy selections are identical. In 5.2 we explore the performance of HMM and ANN classification on a wider range of more complex access patterns.

5.1.2 Dynamic Adaptation

The regular access pattern benchmarks of §5.1.1 showed that the overhead for real-time access pattern classification and policy selection was modest and that an appropriate match of policies and policy parameters to access patterns could improve performance beyond 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 examine how dynamic adaptation can be a performance advantage, we conducted a series of performance studies using benchmarks with time-varying access patterns. We describe the performance of 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 ANN classification and adaptive policy selection, and once with a single, default system policy. Reclassification occurred every two million bytes (a user-specified parameter).

Figure 5.1 shows 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 5.1a the same policies are used for both the random and the sequential reads. In contrast, Figure 5.1b shows the dramatic improvements possible when the access pattern change is automatically detected and caching is disabled for random reads.

Figure 5.1c and 5.1d show detailed read durations at the point where the access pattern changes. In Figure 5.1c, PPFS policies are never modified for the random access pattern, and almost every random read incurs a cache miss. In Figure 5.1d, 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 suggest 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 §5.3, we describe two large scientific codes that benefit from these adaptive techniques.

5.2 HMM and ANN Classification Comparison

The ability to respond quickly to changing access patterns with suitable file system policies is critical to performance of an adaptive file system. As described in §4.4, our two classification methodologies differ significantly in their ability to express complex access patterns. If access patterns are similar across executions, HMM classification can adapt to changing patterns more quickly and can determine access pattern features on a larger scale (e.g., file reuse). In this section we compare HMM and ANN classification with an emphasis on these access pattern features.

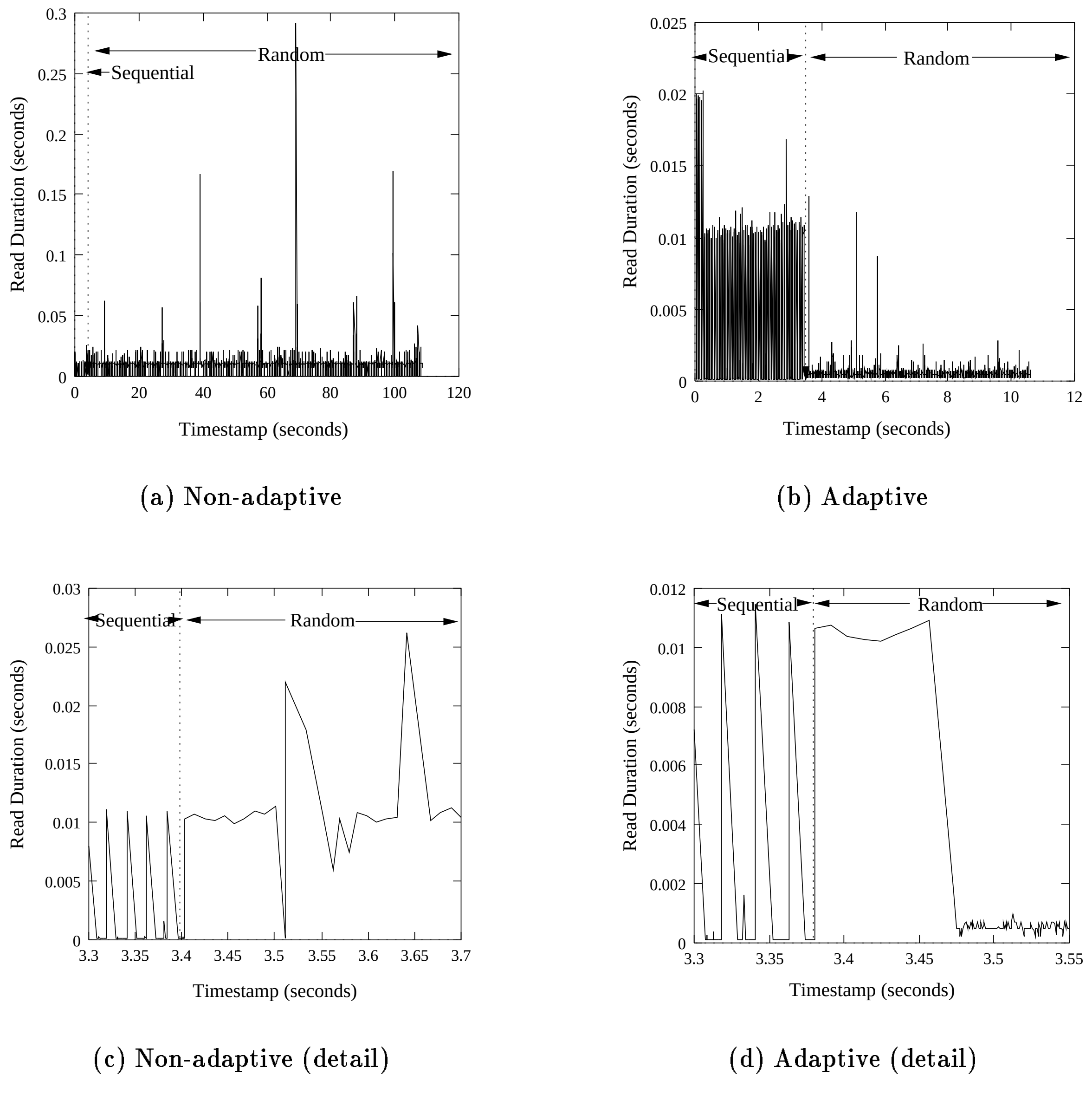


Figure 5.1: Dynamically adapting to changing access patterns.

5.2.1 Responding to Changing Access Patterns

We reexamine the dynamic classification benchmark of the previous section to emphasize the importance of classification responsiveness. Frequency of neural network classification is a user-specified parameter, and ideally reclassification should occur immediately after the access pattern changes. However, the tradeoff between the cost of too-frequent classification and the performance penalty of lags between reclassification and access pattern change are difficult to evaluate. Assuming previous executions accurately predict behavior, the responsiveness of an HMM classifier is bounded by the size of the file segments. This is because the classifier can detect a potential access pattern change from a change in the probabilities of transitions out of a file segment. Reclassification points occur naturally when the model predictions do not match actual accesses.

As in the experiment of §5.1.2, this benchmark reads the first half of a 40 million byte file sequentially and the second half randomly. For both access patterns, the request size is 2000 bytes. Unlike our experiments in §5.1.2, the ANN classifier automatically reclassifies the pattern every 640 KB instead of every two million bytes; a naïve reader might think this more frequent classification should be more responsive. We executed this benchmark three times using PPFS, twice with adaptive file system policies guided by neural network classification and hidden Markov model classification, and once with a single, default system policy.

Figure 5.2 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 algorithm of Figure 4.4 disables caching, reducing the byte volume of physical input/output calls and improving throughput.

Because the frequency of ANN classification is a user-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 5.3 illustrates the portion of the HMM where the access pattern changes. After detecting a sequential access pattern, the HMM classifier computes the probability that the sequential

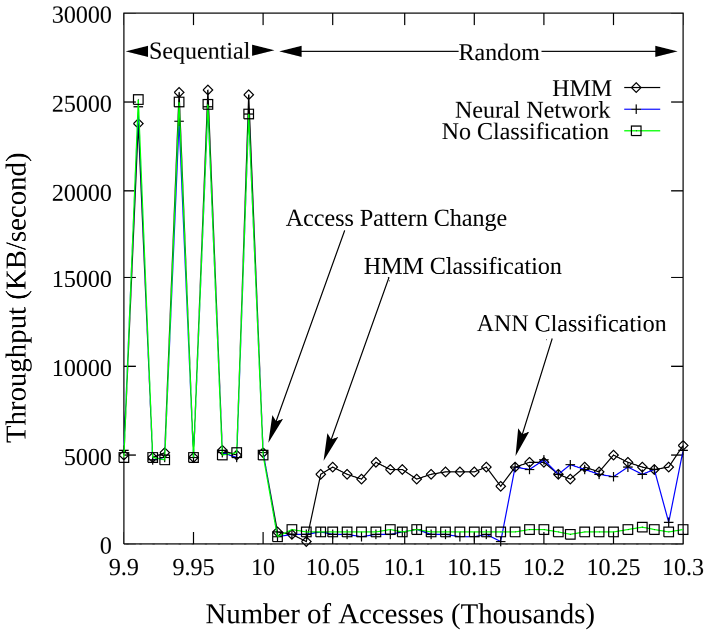


Figure 5.2: Adapting to a changing access pattern.

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.

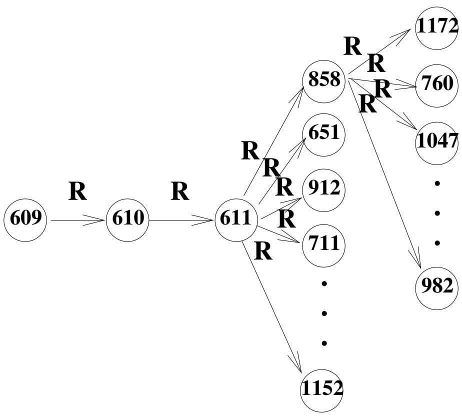


Figure 5.3: HMM at access pattern change.

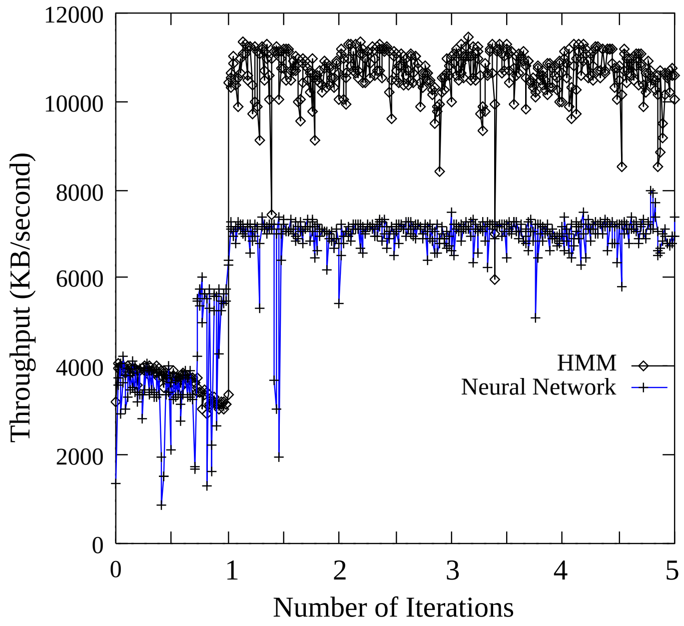


Figure 5.4: Detection of file reuse.

5.2.2 Sequential File Reuse

An important access pattern in scientific applications is repeated rereading of an input file. For example, two parallel implementations of the Hartree-Fock algorithm [29] exhibit this pattern. Hartree-Fock computes the electron density around a molecule by considering each electron in the molecule in the collective field of the others. The code iterates until all the electron fields are consistent. During the iterative third phase, calculation of the self-consistent fields, the algorithm sequentially reads the integral data, which is too large to fit into memory, multiple times.

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 the HMM classifier can also predict reuse of sequentially accessed blocks and determine an appropriate cache size, using the method described in §4.4. This can improve performance, particularly when reads are expensive or the file is re-read many times.

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. If the file is too large to fit into the available memory, the cache can make a different policy selection (e.g., to cache only a subset of the file).

Figure 5.4 shows the throughput (computed over every 100 accesses) for the benchmark executing with both classification methods. The benchmark reads the file once per iteration. During the first iteration, 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 buffer cache. However, under HMM classification the file is also retained in the PPFS user-level cache, further improving throughput. In other words, the difference between the two curves is the difference between the performance of user-level buffering and system-level buffering.

The ability to recognize and adapt to file reuse is an important factor for guiding caching policies. In sequential systems, a high cache hit ratio is an indicator of good cache performance. Cache replacement policies keep the cache size bounded while attempting to maximize the cache hit ratio. The optimal replacement algorithm (OPT) is to replace the file block that will not be used for the longest period of time; however, often this information is unavailable. The Least Recently Used (LRU) replacement policy, which replaces the block that has not been used for the longest period of time, is usually a good approximation.

However, in the case of sequential file reuse, when only a portion of the file can fit in cache, LRU is a very poor approximator to the optimal policy. It will systematically evict the least recently used block before it can be reused. In contrast, with advance knowledge of the access pattern, OPT would instead retain the first portion of the file in cache, to be reread at each iteration.

Our strategy of increasing the available cache space for the file is analogous; instead of using access pattern to guide a replacement policy we use it to estimate a working set size. Accurate predictions of future file block usage are key to chosing optimal caching policies.

5.3 HDF Access Pattern Classification Experiments

Most UNIX applications access files sequentially, and UNIX file systems are optimized for sequential access. However, strided and irregular access patterns are common when input/output is encapsulated by an input/output library. In this section we examine access patterns produced using the NCSA Hierarchical Data Format (HDF) [57]. HDF performance is important in its own right and is a key factor in the satellite data processing applications that we examine in §5.4.

5.3.1 HDF and Satellite Data Processing

HDF is a self-documenting physical file format that facilitates data transfer between machines and operating systems. HDF supports several different data models, each of which is a specific data type (e.g. multidimensional arrays, raster images, tables) and provides interfaces for accessing data and metadata related to the corresponding type. These interfaces are suitable for storing and retrieving many types of scientific data.

The HDF application interface routines are designed to be used in a certain way for maximum efficiency. In general, the application programmer should first open the HDF file, initialize the appropriate interface, access the data, terminate access to the interface, and close the file. However, nothing in the HDF implementation constrains programmers to this model of use.

In general, problems with HDF input/output performance arise because the portable format hides the underlying file layout, and therefore costs of HDF file operations, from the application programmer. For example, opening an HDF file and initializing interface-specific data structures are expensive operations. Each time an application opens an HDF file, the HDF library must open the underlying UNIX file, and read all the metadata, containing information about the objects in the file and where they are stored. Therefore, writing data to an HDF file involves a file open, a sequence of alternating reads and seeks, and finally, the actual write. Metadata accesses are expensive enough so that it is inefficient to open and close an HDF file for every small access. Yet, without an accurate cost model, this is a natural application organization for writing higher-level routines to perform logically distinct operations on objects within an HDF file.

As another example, the Scientific Data Set (SDS) interface provides access to multidimensional arrays. However, by hiding the physical data layout from the programmer, it is easy to use the SDS interface innefficiently, generating unnecessary seeks between data elements.

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. The codes we examine in §5.4 are representative low-level data processing applications.

5.3.2 HDF Performance

We conducted experiments that compare the performance of PPFS using automatic classification to buffered UNIX input/output on HDF benchmarks. Our test suite is a set of benchmarks that stress the HDF SDS interface in a variety of ways. The benchmarks access a 4-dimensional data set composed of a sequence of 240 byte 3-dimensional records according to the descriptions in Table 5.4. We have used two file sizes: 10000 and 100000 records. This benchmark is based on a code fragment that was brought to the attention of HDF support staff by a programmer who did not understand why its performance was abysmal [95].

SDS routines are used exclusively to access the data set. We read or write all records along dimension 0 or 4. Access along dimension 4 produces a strided access pattern; this is the most inefficient way to read the SDS data. The random access pattern is a randomly chosen sequence

Read/Write	Access Pattern	Dimension
Read	Sequential	0
Read	Strided	4
Read	Random	0
Write	Sequential	0
Write	Strided	4
Write	Random	0

Table 5.4: HDF SDS benchmarks.

Benchmark	UNIX Times		PPFS Times		Speedup
	Real	System	Real	System	
Read Sequential	1.21	0.40	1.48	0.47	0.82
Read Strided	182.94	31.09	14.79	0.51	12.34
Read Random	7.39	1.03	2.31	0.38	3.20
Write Sequential	3.36	1.37	2.58	0.58	1.30
Write Strided	48.40	20.70	16.75	0.63	2.89
Write Random	5.74	1.56	2.73	0.51	2.10

Table 5.5: 10,000 record HDF SDS benchmark.

of records (each record may be accessed more than once); the same sequence is used every time the benchmark is executed.

The HDF library hides the details of file access from the user; only minor modifications are necessary to convert HDF to invoke PPFS input/output calls instead of UNIX buffered input/output (the default for HDF 4.0 on Sun workstations). HDF defines macros to perform the system-dependent input/output operations. We simply changed the macro definitions to call the equivalent PPFS input/output operations, recompiled the HDF library, and re-linked the application programs. Non-HDF and non-file application input/output (e.g., terminal input) is unchanged.

Tables 5.5 and 5.6 show the benchmark results for 10,000 and 100,000 records, respectively. Each benchmark was run once with a cold UNIX file system cache (the file system was unmounted and remounted to flush any file data from memory) to obtain this data. Classification occurs after the HDF library has performed input/output generated by the `SDScreate` call,

Benchmark	UNIX Times Real	UNIX Times System	PPFS Times Real	PPFS Times System	Speedup
Read Sequential	12.6	3.0	10.5	4.3	1.2
Read Strided	1910.5	289.0	239.8	4.8	8.0
Read Random	122.1	50.3	21.8	4.8	5.6
Write Sequential	34.0	15.5	16.4	3.3	2.1
Write Strided	497.3	242.6	258.2	7.1	1.9
Write Random	132.4	38.2	30.1	7.0	4.4

Table 5.6: 100,000 record HDF SDS benchmark.

and is in a "steady state." This classification is then used to tune policies using the algorithm of Figure 4.4. Reclassification occurs every 1000 file accesses.

The execution times for all read and write benchmarks improve under PPFS due to caching. In the sequential and random access patterns, the record size is small (240 bytes) and the HDF implementation issues a seek to position itself before each record write. Under UNIX buffered input/output, this causes a flush of every record. The PPFS implementation does not issue `lseek` system calls unless they are necessary to read in a portion of the file. The adaptive caching strategy improves the performance of the random access and strided access patterns by buffering the entire file when it detects the random pattern and small accesses. Specifically, the write strided access pattern takes significantly longer than the other write benchmarks because it writes each record along dimension 4, in the least efficient way possible. This means that each individual access is only 8 bytes. PPFS detects this strided pattern and enlarges the cache to hold all file blocks, on the assumption that the cached blocks will be re-referenced. This strategy results in large speedups when accessing 10,000 records, and somewhat smaller speedups when accessing 100,000 records. This is because although the larger HDF file fits into the PPFS cache, UNIX virtual memory management causes paging to occur that results in a smaller speedup.

Unfortunately, the 10,000 record file size is too small to mask PPFS overhead on the read sequential patterns; the cache management overhead involved in sequential reads is not as efficient as buffered UNIX input/output.

5.4 Satellite Data Processing Applications

Having demonstrated the ability of local access classification to tune performance of access pattern benchmarks, we now examine real applications. In this section we classification results from two typical applications used to process low-level satellite data: `l0to1a`, from the Sea-viewing Wide Field-of-view Sensor (SeaWiFS) project, and Pathfinder, from the NOAA/NASA Pathfinder AVHRR (Advanced Very High Resolution Radiometer) data processing project. Both applications exploit the HDF library to generate data products, albeit in decidedly different ways. The file system uses local classification information to tune file system policies to each access pattern, dramatically improving performance.

5.4.1 SeaWiFS `l0to1a` Translation

The first application we examined 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. 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 [23]. Our input for this program is a simulated data set.

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 5.5 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.

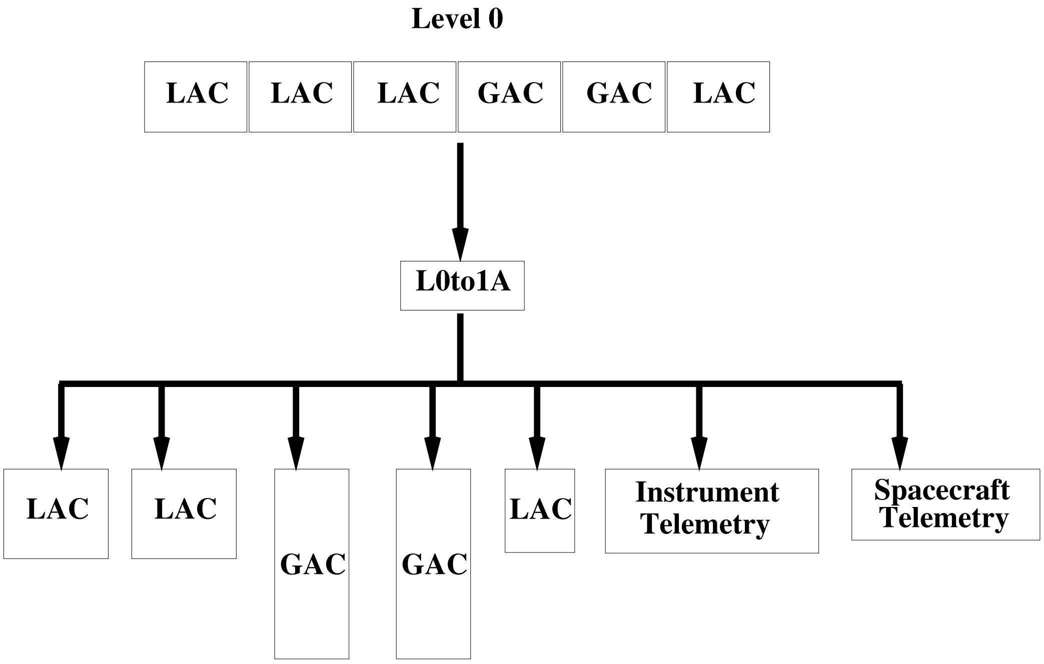


Figure 5.5: Overview of SeaWiFS level-0 to level-1A processing.

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 5.7: SeaWiFS `l0to1a` execution times (seconds).

5.4.2 SeaWiFS Input/Output

To validate the promising benchmark results of §5.1 and §5.3, we retargeted the SeaWiFS application to use the adaptive version of 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. During execution, PPFS automatically classified the access patterns and selected file policies using the algorithm of Figure 4.4.

Table 5.7 shows the SeaWiFS `l0to1a` execution times for both UNIX and PPFS, with operation counts and data volume shown in Table 5.8. As with the Pathfinder code, PPFS

Experimental Environment	Read		Write		Lseek	Open	Close
	Count	Bytes	Count	Bytes			
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 5.8: SeaWiFS `l0to1a` input/output operations.

access pattern classification and dynamic policy selection reduces total execution time and dramatically reduces the number of input/output calls.

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 4.4, 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.

Finally, Figures 5.6–5.8 show histograms of the SeaWiFS `l0to1a` read, write and seek 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

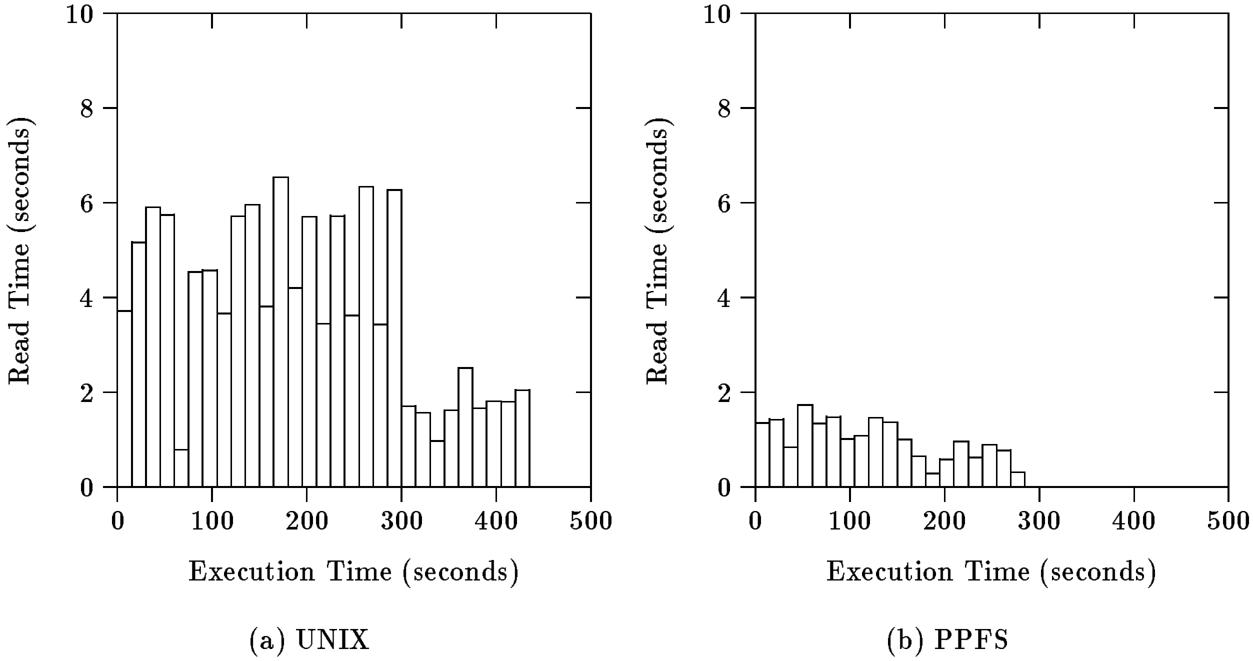


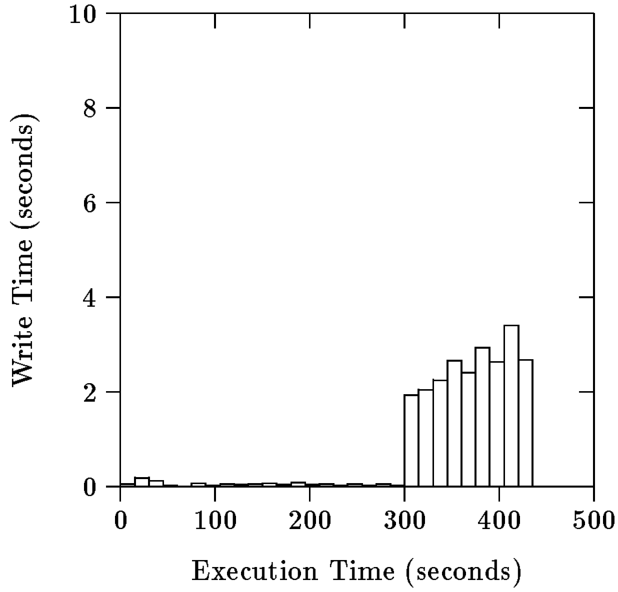
Figure 5.6: SeaWiFS l0to1a read duration histograms.

of metadata are actually written. During the second phase, the data is actually written to the output HDF files, causing the write time peaks.

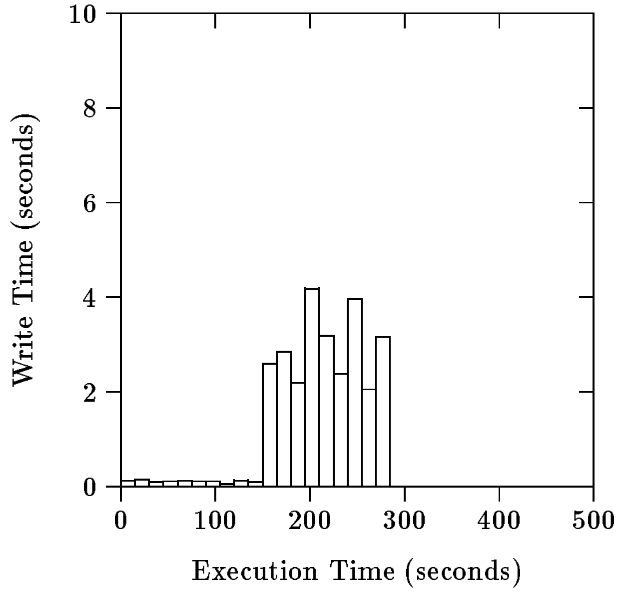
5.4.3 Pathfinder

Our second application is a low level satellite data processing code from the Pathfinder project. The goal of Pathfinder 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 program execution, ancillary data files and the 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



(a) UNIX



(b) PPFS

Figure 5.7: SeaWiFS l0to1a write duration histograms.

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.

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).

5.4.4 Pathfinder Input/Output

During Pathfinder execution, the PPFS ANN 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. Consequently, 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.

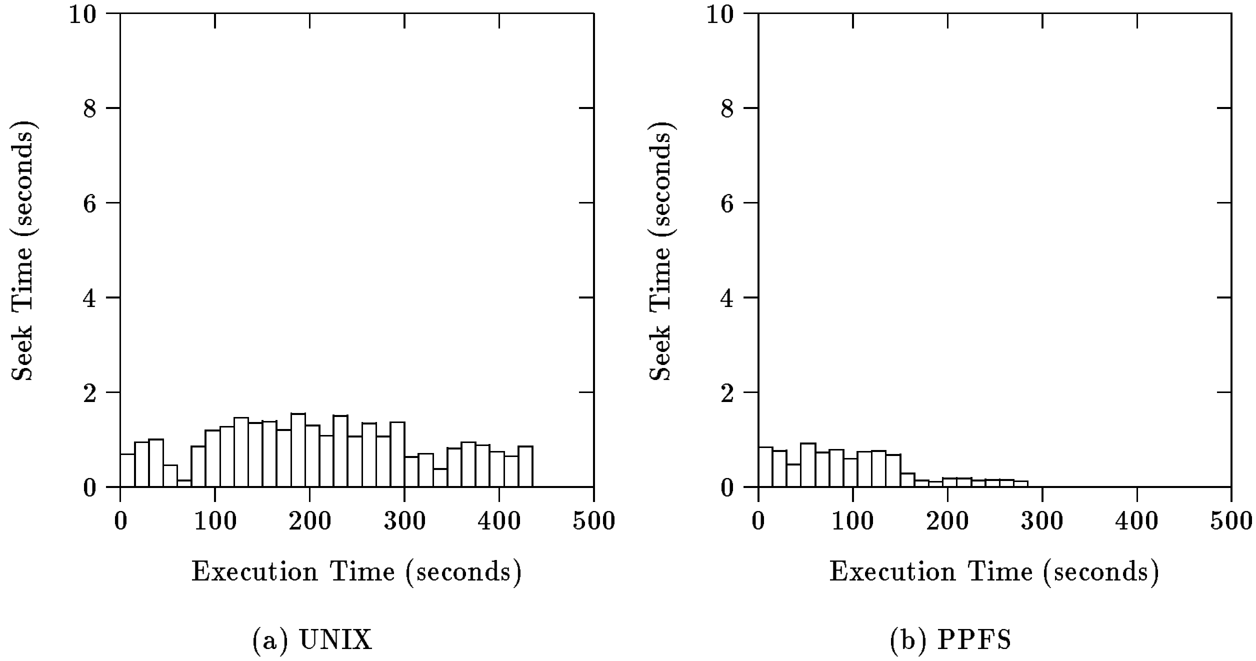
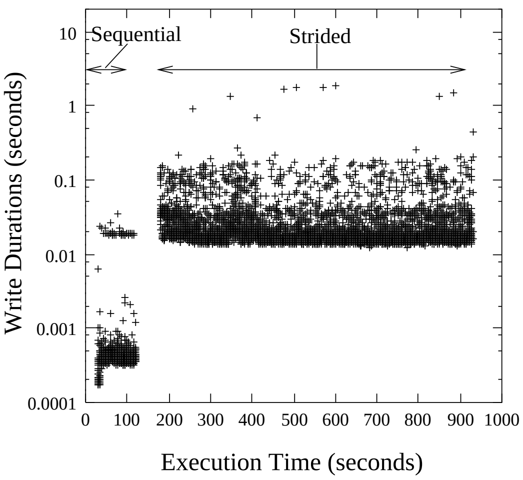


Figure 5.8: SeaWiFS l0to1a seek duration histograms.

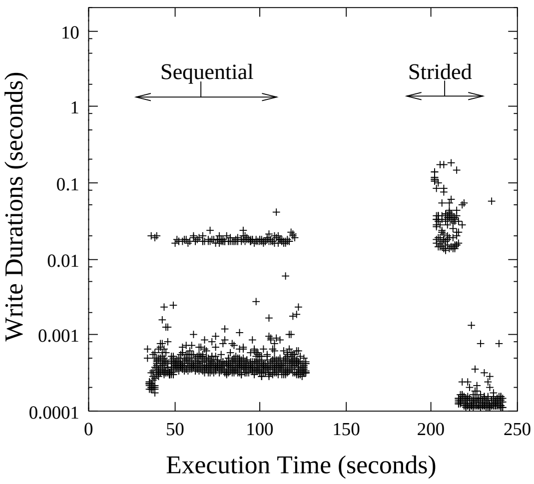
Figure 5.9 illustrates the dramatic benefits of dynamic policy adaption for Pathfinder's execution. Both Figures 5.9a and 5.9b show the same number of accesses to the output data file with and without classification and dynamic policy selection. The first cluster of accesses is the write only sequential 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.

One of the problems with automatic policy selection using the algorithm of Figure 4.4 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 5.9 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,



(a) PPFS (Non-adaptive)



(b) PPFS (Adaptive)

Figure 5.9: Pathfinder write durations.

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 §4.4, 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 5.10 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	System Time	User Time	Total
UNIX	1578.2	1781.1	4299.3
PPFS (ANN)	373.8	1265.1	2452.0
PPFS (ANN, split cache)	170.3	1243.2	2174.4
PPFS (HMM)	134.8	1161.8	1891.6

Table 5.9: Pathfinder execution times (seconds).

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 5.10: Pathfinder input/output operations.

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.

5.5 Summary

Our experimental results using a variety of benchmarks 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. The benefits of correct policy selection far outweigh classification costs. Even simple access pattern information and rudimentary policy controls are useful. More complex classification and policy selection, as evidenced by HMM classification of Pathfinder input/output, is capable of generating even better performance.

Chapter 6

Global Classification and Parallel File Systems

We have discussed local access pattern classification and demonstrated the performance benefits of using a local classification to tune file system policies. However, local classification is a small part of a global classification problem (i.e., recognizing qualitative patterns observed in 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 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. In this thesis we consider optimizing input/output performance for a single application and parallel file. This is only a subset of a much harder problem; that of balancing input/output resources among competing applications with multiple files and maximizing overall system throughput.

Current commercial and experimental parallel file systems use global access pattern information to improve input/output performance. In §6.1, we discuss the importance of global access pattern classification to performance in the abstract, before more closely investigating in §6.2 how specific file system implementations use global access pattern information. We describe how the PPFS uses global access pattern classifications in §6.3–§6.4.

6.1 File System Performance and Global Classification

Regardless of the specific file system implementation, there are several generic performance considerations specific to parallel input/output architectures. We consider an abstract input/output architecture that consists of multiple input/output devices attached to separate processors on an interconnection network. Files may be declustered over many disks.

We also limit our focus to compulsory file input/output. The Miller and Katz [55] study identified three reasons for input/output in scientific applications: compulsory (input/output to load data into memory for processing and write results), checkpointing data, and data staging. When staging data, the CPU may not need to access the data, because it is simply transferring it between devices. In this situation, direct connections to stream data between input/output devices can greatly improve performance. In our performance characterization of the SIO application suite, we have determined that compulsory input/output dominates the input/output mix. Checkpoints are conceptually similar.

One of the most important performance considerations for a parallel file system is exploiting parallelism to improve throughput, both within each request stream (per parallel program thread) and across request streams (per parallel program). Another is hiding input/output latency by prefetching and caching. The selected file declustering scheme is critical to these optimizations, because it determines the mapping of logical file requests to physical input/output requests.

In a uniprocessor file system with a single disk and single processor, file data is usually organized sequentially on disk and is accessed sequentially; sequential prefetching improves throughput of reads and writebehind improves write performance. This file system behavior, which matches the expected access pattern, is relatively efficient. However, when a program must access multiple files that might be at different disk locations (as when running "make" to compile a large system), input/output requests can be re-ordered for more efficient access, reducing overall execution time. This advance knowledge of input/output requirements must be provided to the file system (e.g., as hints). This is the motivation behind Transparent Informed Prefetching (TIP) [68] and dynamic sets [85]; the application reveals future accesses allowing the file system to use parallelism within the request stream to exploit input/output bandwidth and improve performance.

The same principles apply for exploiting parallelism across multiple input/output streams. Depending on the temporal and spatial distribution of the merged input/output requests, they can be reordered to obtain greater system throughput. This motivates optimizations such as two-phase [20] and collective input/output [40]; given global knowledge that some data must be read or written before all the processors can proceed, the input/output operations can occur as an efficient collective. Even when the collective operation is not performed explicitly, batching requests is a general principle that can be used to optimize input/output. Input/output requests for duplicate file blocks can be consolidated (via caching) and requests can be re-ordered to exploit input/output bandwidth.

To optimize system throughput by increasing parallelism of access, the file system must have available information about future requests from all processors, and when and if the processors need the data. Part or all of this information may be provided in the form of hints, prefetch requests, or a global access pattern classification. This information, in conjunction with file declustering information, allows the file system to reorder, consolidate, and batch requests.

Another approach to improving input/output performance of compulsory accesses is hiding or reducing input/output latency by prefetching and caching blocks that will be accessed in the future. Optimal uniprocessor caching and prefetching strategies rely upon knowledge of future block accesses. Similarly, in a parallel file system, it is impossible to make effective caching decisions without considering the global access pattern and file layout. For example, consider global interleaved access to a striped file, where the request size is smaller than the stripe unit. The same stripe unit will be re-read multiple times as the processors coordinate to access the file. Caching this unit can potentially prevent many unnecessary disk accesses. Prefetching the file sequentially can improve throughput.

While full disclosure of future accesses and when the data is required is ideal for scheduling input/output requests, in practice such detailed information is usually unavailable. However, certain access pattern characteristics are especially important for performance in a global context. We have identified four such characteristics.

The first is predictability of access. Even when complete future information is not available, if the pattern indicated by previous input/output requests is a good predictor of future accesses, it can guide caching and prefetching decisions. The second is knowing how much flexibility there is to reorder requests. Information about global synchronization points that occur before globally read data is accessed facilitates collective input/output operations. A third important access pattern characteristic is consistency. If multiple readers and writers access overlapping bytes of the same file, the file system must provide a mechanism to ensure that reads and writes are consistent. However, supporting atomic accesses is expensive and limits parallelism; it is better if it can be avoided. Finally, it is important to know if processors share data. This is useful information for selection of caching strategies.

A qualitative global access pattern classification is an indicator of these features in a simple, compact form. A probabilistic model of file accesses, such as a trained HMM, can potentially provide more specific information.

6.2 File System Design and Global Access Patterns

Current commercial and experimental file systems perform the optimizations described above to varying degrees. In lieu of using automatic classification, they provide application interfaces to express the global access pattern characteristics necessary for such optimizations. In this section, we examine the specific mechanisms used by current parallel file systems to optimize parallel input/output.

The input/output interface is extremely important to exploiting parallelism. Reordering and parallelization of input/output requests is impossible if they are artificially serialized within the file system (e.g., by file pointers). This is one of the primary reasons that the UNIX file system interface, even with extensions, is generally deemed unsuitable as a parallel file system API [14, 39], although it remains popular for compatibility reasons.

For example, a program that can utilize subsections of a file in any order cannot express this flexibility through a file system with a byte stream interface. Relaxing this constraint for applications can dramatically improve performance by allowing the file system to balance load and optimize throughput. Consider the Needleman, Wunch and Sellers (NWS) [58] dynamic programming algorithm for parallel genome matching, which is implemented by having each processor independently compare a test genome sequence against disjoint portions of the sequence database. Significant speedups can be obtained by structuring this database so that processors can access any record that has not yet been requested, without regard to byte ordering, and by exploiting this information before issuing physical input/output requests [31].

Although few experimental file systems currently support any form of "relaxed" file ordering, collective input/output interfaces are more commonplace. Disk directed input/output and two-phase input/output are examples of collective input/output strategies that use synchronization points specified by the application to batch input/output requests, servicing them in an efficient sorted order. These approaches significantly improve performance, exploiting input/output system bandwidth and using the most efficient access patterns for the file declustering strategy. Deep prefetching or some other means of access pattern disclosure in parallel file systems could help the file system to perform the same kinds of optimizations dynamically without explicit synchronization.

A major concern in parallel file system design, especially for systems providing a UNIX style interface, is maximizing concurrency while maintaining file consistency. If a file is written, or read and written by multiple processors, some coordination may be necessary to ensure that reads and writes proceed atomically. Unfortunately, this kind of guarantee usually restricts request concurrency and the overhead of synchronization can be expensive. Parallel file systems usually offer guaranteed atomicity as an option. For example, IBM PIOFS [4], the product version of Vesta[17][1], provides two modes for opening a subfile: NORMAL and CAUTIOUS – the latter manages overlapping reads writes by ensuring their serialization, and therefore performance is somewhat worse. Intel PFS[32] has analogous modes M_ASYNC and M_UNIX; in mode M_UNIX reads and writes to the same file are atomic, but mode M_ASYNC allows multiple readers and writers concurrent file access. MPI-IO [91], an attempt to define a high-level API, does not provide consistency by default, but provides a cautious mode if it is required.

In a parallel file system, file data must be distributed over disks so as to provide the best performance for the global file access pattern; this is a function of balancing disk load and exploiting input/output bandwidth. Most modern parallel file systems simply stripe data cyclicly across the disks [32], a strategy that usually results in balanced input/output loads. However, limiting the number and increasing the parallelism of input/output requests by matching the file declustering strategy to the access pattern can significantly improve performance. File systems vary greatly in the level of control they provide over declustering. The Vesta interface gives applications significant control over file declustering in order to maximize parallelism. Intel PFS provides allows applications to map a file to different stripe attributes when the file is created

[1] Henceforth we will refer exclusively to Vesta.

or temporarily for read-only files. With restricted control over the data distribution, in both of these systems the application programmer must ensure parallelism of access by making the access pattern conform to the file system. Although MPI-IO assumes physical data layout will be system-dependent, it provides hints to be supplied at file creation to improve input/output throughput given certain access patterns.

When the file system provides little control over physical data layout, it can improve performance by utilizing higher-level application-provided access pattern information. For example, while Vesta relies upon the application to maximize parallelism by matching the data layout to access pattern, Intel PFS allows the application to provide access pattern information by selecting one of six input/output modes. These modes, representing common access patterns, allow the file system to optimize performance without exposing internal file structure to the application. For example, the appropriate mode for a global sequential access pattern, mode M_GLOBAL, combines multiple input/output requests to identical file bytes into a single request. MPI-IO uses MPI [24] derived datatypes to express data partitioning between the logical file layout and the processors.

Server-side caching and prefetching are also important to performance, because they provide the ability to hide or reduce input/output latency without application intervention. Intel PFS allows the application to enable or disable server buffering on a per file basis, but it does no prefetching. At the other extreme, Vesta relies heavily upon buffer cache management to overcome temporal access pattern variations (e.g., aggregate read requests are recognized as sequential, enabling sequential prefetching, and out-of-order writes are buffered and written in larger segments) [22]. Caching and prefetching and dynamic recognition of global sequentiality are vital to Vesta performance.

Asynchronous input/output gives the application the ability to overlap computation with input/output latency, by issuing input/output requests before they are needed. Intel PFS, IBM Vesta and MPI-IO all provide system calls for asynchronous input/output.

To summarize, current file systems rely heavily upon the user interface to supply the global access pattern information critical to achieving good input/output performance. In Vesta, it is sufficient to describe spatial partitioning; input/output server caching and prefetching policies optimize temporal access patterns. Intel PFS requires a more complete global access pattern mode specification. Interfaces to other optimizations (e.g., asynchronous input/output) are provided separately. In general, control over file system performance is largely the application's responsibility.

6.3 Global Classification Methodology

It is clear from the preceeding discussion that global access pattern information is an important factor in parallel file system performance; we believe that global classification can supply this automatically in the same way that local classification provides local access pattern information.

Local access pattern classification is a small part of a larger classification problem. Local classifications are made per parallel program thread; however, the local access patterns within the parallel program merge during execution, creating a global access pattern. Global knowledge is especially important for tuning file system policies. For example, if every processor is accessing a whole file sequentially, one could potentially improve performance by employing a caching policy that does not evict a cached block until every processor has read it.

Global classification is difficult because it requires determining relationships between accesses from different processors within the file and throughout execution. Below we describe our framework for creating global classifications from local classifications and additional input/output trace information.

6.3.1 Global Classification Approach

There are many ways one might attempt global classification. One decentralized approach is to assume that only one data declustering strategy is used (e.g., data striping) and attempt to infer the global pattern from the subset of accesses directed to each disk. Although this is possible for a limited set of access patterns, it is highly dependent on the file declustering strategy. We prefer to separate classification from the file system architecture.

Another obvious approach is for a global classifier to observe every local file access, correlating the accesses according to their position in the file and identifying (through some pattern recognition tool) whether the processors contribute to a recognized global pattern. This is an architecture-independent strategy; unfortunately a direct implementation would be highly inefficient. Not only is it expensive to forward every access to a classifier, but identifying the global pattern can be difficult. The neural network used for local classification is insufficient for global classification because it does not represent how each processor contributes to the global pattern.

Our approach is an optimized version of the above strategy. We recognize that it is not necessary to monitor every access; we need know only the byte ranges accessed on each processor within some period of time. Furthermore, since global access patterns are interleaved local access patterns, we can describe a global pattern simply as a combination of local patterns, thereby avoiding a separate pattern recognition phase at the global level.

Our global classification infrastructure is based on an access pattern temporal algebra. We combine local classifications and other local information to make global classifications. The number of processors contributing to the global access pattern is called the *cardinality* of the classification. In our experiments, we attempt to make global classifications with cardinality p, where p is the number of processors involved in the global input/output. However, a global classification involving a subset of the total number of processors might still be useful for policy selection. A partial global classification may even be preferable, if it more accurately represents the temporal characteristics of the global access pattern.

Simply put, global classification identifies a snapshot (in time) of the evolving spatial access pattern as one of several known patterns. Assuming the pattern continues to emerge as predicted, classification can inform the file system of future accesses, data sharing, and concurrency of access.

6.3.2 Global Classifications in Time

Like local classifications, global classifications are valid for a specific time interval. 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. For example, if the local classification is based upon a single window of ten accesses, t_s is the timestamp of the first access and t_e is the timestamp plus the duration of the last access. A global classification is valid for $(max(t_s), min(t_e))$ over all 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 as shown in Figure 6.1a. It is difficult to predict in advance an appropriate duration of local classification windows that will ensure a global classification window, because it is dependent upon the dynamic input/output rates of

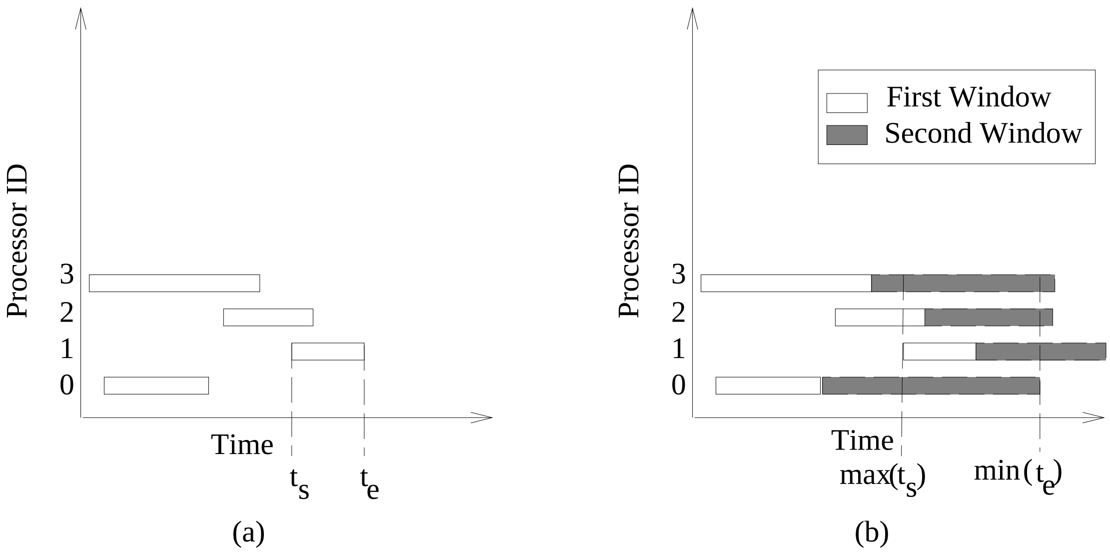


Figure 6.1: Nonoverlapping and overlapping local classification windows.

the individual processors. These rates change between and during program executions, varying with input data and system configuration.

When the local classification windows are too short to make a global classification, there are several options. We can make a global classification over the subset of processors whose windows overlap (a classification with cardinality $< p$), or we can extend the local window size until global classification is possible. The second approach is shown in Figure 6.1b. The four local classifications overlap in the time period $(max(t_s), min(t_e))$. Note that despite the fact that the local classifications overlap for only a small subset of the total time, information from the entire local classification periods is used to make a global classification. Another related approach is to increase the frequency of local classifications, increasing the probability that the windows will overlap. By increasing the frequency of local classifications, it is possible to maintain a hierarchy of global classifications with different window sizes, to capture the idea that certain global patterns might not be visible over all processors within smaller windows.

In practice, the approach we have taken is to logically extend the local classification windows by assuming that if two consecutive classifications are identical, the classification is valid for the entire time between classifications. This approximation is simple and effective; it limits the data sent to the global classifier and requires no adaptivity in the reclassification algorithm.

6.3.3 Creating Global Classifications from Local Classifications

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 3.1) 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: we define the set of local classifications as $L = l_1, l_2, \ldots, l_p$ and provide the requirements for identifying read/write global classifications below.

Read only Processors collectively read the file.

$$\otimes L = \text{read only}$$

In order for the global access pattern to be read only, each local access pattern must be read only.

Write only Processors collectively write the file.

$$\otimes L = \text{write only}$$

In order for the global access pattern to be write only, each local access pattern must be write only.

Read/write Processors collectively read and write the file.

$$\otimes L = \text{read/write}$$

As shown in Figure 3.2, the greatest lower bound of read only, write only, and read/write patterns is read/write. If the local access patterns collectively involve reading and writing, the strongest classification to be made at the global level is read/write.

Request size classifications are also straightforward. As indicated by Figure 3.2, when qualitative request patterns (e.g., variable or uniform) are combined with quantitative size information (small, medium, or large), this partitioning generates seven possible classifications. We simplify classification by identifying only rough request size categories; for example, we can make a global *Uniform small* classification when the request size on each individual processor is different, though small. If desired, we can enforce stricter categories by using information about the average request size. The requirements for three of these global classifications are shown below; the remaining four are similar.

Uniform small Processors collectively access the file with small, regular requests.

$$\otimes L = \text{uniform small}$$

All local access patterns must have uniform request sizes, and the average request size must be small. Global uniformity may be lost if we do not check that the average request size is the same across processors. Our proposed categories ensure only that the average request size is roughly the same (small) across processors.

Variable small Processors collectively access the file with small, variable requests.

$$\otimes L = \text{variable small}$$

Requests are globally variable small if all processors issue small requests, but not all local access patterns have uniform request sizes.

Variable irregular Processors collectively access the file with irregular request sizes.

The variable irregular classification is made when local access patterns neither all have uniform request sizes, nor do they have request sizes that are all either small, medium, or large.

$$\otimes L = \text{variable irregular}$$

As described above, global classifications involving sequentiality must be made using additional information about the bytes accessed by each client. Some global sequentiality classifications are formulated below, assuming a nonzero range $(max(t_s), min(t_e))$.

Global sequential Each processor independently accesses the entire file sequentially.

We make this classification under the following conditions:

- $max(t_s) < min(t_e)$
- $\otimes L$ = sequential
- Each local sequential classification has a start byte and an end byte that delimit the sequential access sequence; in more formal notation, (b_s, b_e), where b_s is the start byte and b_e is the end byte. To be globally sequential, there must exist a $(max(b_s), min(b_e))$ over all local classifications such that $min(b_e) > max(b_s)$.

Figures 6.2a and 6.2b show two snapshots of a global sequential access pattern. Each horizontal bar represents a local sequential access pattern that accesses the range of bytes shown. In 6.2a, $min(b_e) > max(b_s)$, so this pattern is global sequential over the 4 processors (cardinality 4). In 6.2b, although this pattern might actually be global sequential, in the timeframe of these local classifications processor 3 accesses no bytes common to those accessed by all the other processors. At best, the pattern is global sequential among processors $0, 1, 2$ or $0, 1, 3$ (cardinality 3).

We can still use a classification with low cardinality to tune policies in a variety of ways. A simple approach is to decide upon a threshold cardinality, above which global classifications are automatically extended to all processors. Another approach is to extend the local classification window size to enable global classification. Finally, global classifications with lower cardinalities may be accurate; for example, consider a producer-consumer application where half of the processors write a file and the other half read it.

Partitioned sequential Processors collectively access the entire file in disjoint, sequential segments.

Figure 6.3 illustrates a partitioned sequential access pattern; the shaded portions represent the sequential byte ranges accessed by each processor throughout the lifetime of the program, and the white portions represent the byte ranges accessed during the classification window.

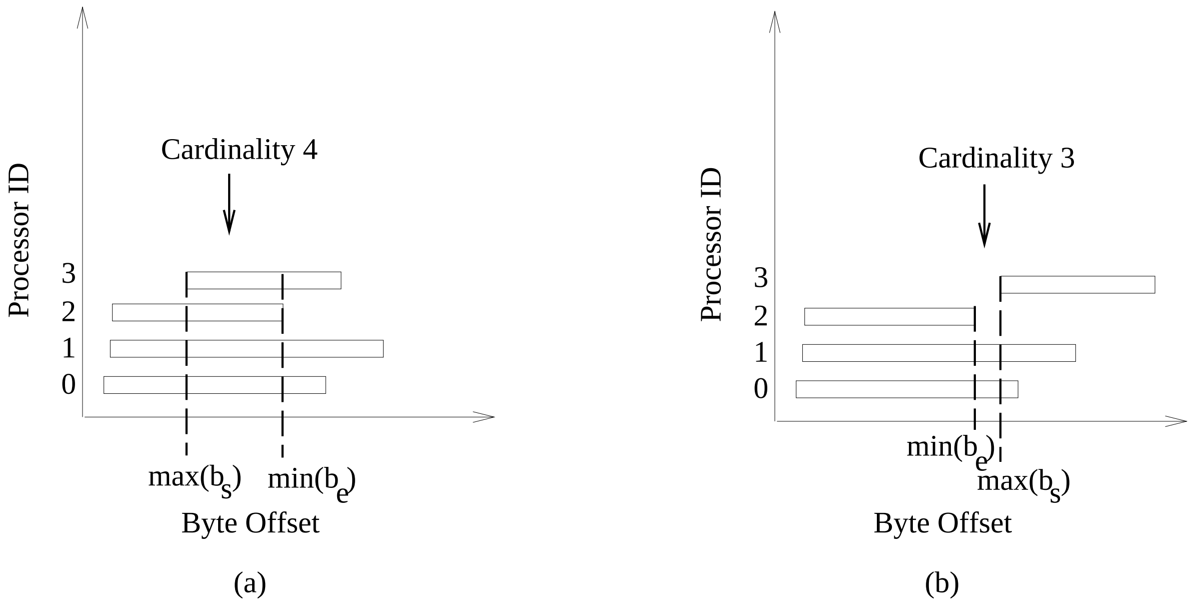


Figure 6.2: Global sequential classification with cardinality 4 and cardinality 3.

Given an arbitrary start time and finish time for our observation window, we cannot be certain that the locally sequential accesses are globally contiguous. However, we make this classification under the following conditions:

- $max(t_s) < min(t_e)$
- $\otimes L$ = sequential
- $\cap$ bytes accessed = $\emptyset$

In other words, all local access patterns must be sequential, and they must not access any overlapping bytes.

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

Figure 6.4 shows an interleaved sequential access pattern; 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

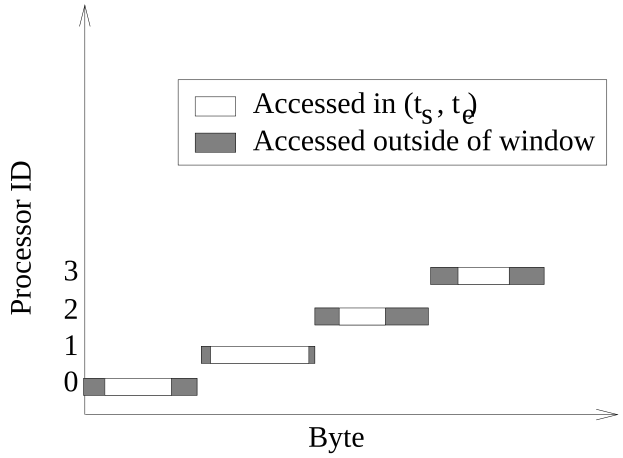


Figure 6.3: Global partitioned sequential classification.

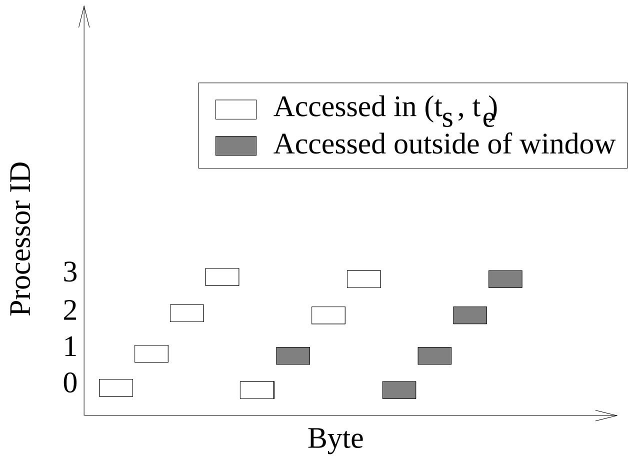


Figure 6.4: Global interleaved sequential classification.

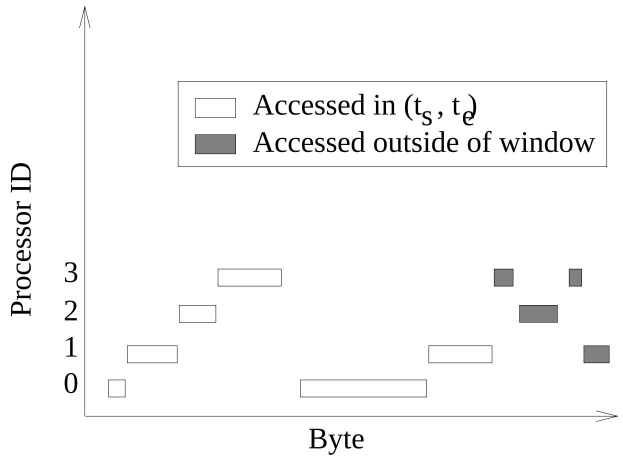


Figure 6.5: Self-scheduled global sequential classification.

- $\cap$ 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 (p is the number of processors involved in the classification). In Figure 6.4, the first 5 requests are contiguous, satisfying this requirement.

Self-scheduled global sequential Processors coordinate to access the entire file sequentially in disjoint, self-scheduled segments.

Figure 6.5 shows a self-scheduled global sequential classification. The white regions are accesses that have occurred within our observation window, and the shaded accesses will occur in the future, completing the pattern.

We make this classification under the following conditions:

- $max(t_s) < min(t_e)$
- $\otimes L =$ nondecreasing
- $\cap$ bytes accessed $= \emptyset$
- If we merge the input/output requests from the processors by file positions, there should be at least p contiguous requests. In Figure 6.5 the first 6 requests are contiguous, satisfying this requirement.

6.3.4 Global Classification Validation

The global classification methodology presented in §6.3 is general enough to classify a variety of global access patterns. In this section we demonstrate specifically that we can classify global access patterns observed in real applications from the Scalable Input/Output (SIO) suite.

We have examined applications from the areas of chemistry and engineering. These applications exhibit the following global input/output access patterns: sequential with cardinality one (one processor performs all input/output on behalf of the other nodes), sequential, partitioned sequential and interleaved sequential. The patterns are primarily read only or write only with either regular or irregular request sizes. Read/write and request size classifications are formed as the join of local classifications, which is straightforward to compute. Global sequentiality is calculated by merging access information and local sequentiality classifications. However, the success of global classification rests on the local classifications being loosely synchronous in time; classification windows must overlap to make a classification involving all processors. This is true for most global access patterns involving substantial input/output. Below we show some results from two typical applications from these problem domains; these applications were executed on a partition of the 512 node Intel Paragon XP/S at Caltech. This machine has 16 input/output nodes and unless otherwise specified was running OSF/1 1.3.

6.3.4.1 ESCAT

The Schwinger-Multichannel Electron Scattering (ESCAT) quantum chemistry application calculates low-energy electron-molecule collisions [93]. Figure 6.6 shows the entire execution input/output timeline for one version of the code, executed on 128 processors. File names are listed along the y axis, and accesses are shown along the x axis in time. From an input/output perspective, there are four main execution phases. In the first phase, the nodes read the problem definition and initial data from three input files (`tape5`, `tape7`, and `tape8`). In the second phase, all nodes participate in the calculation and storage of the quadrature data set; this phase is compute-intensive and is composed of a series of synchronized compute/write cycles to files `smndata` and `smndatb`. In the third phase, quadrature data is read from files `smndata` and `snmdatb` (at approximately 5709 and 5846 seconds, respectively). Finally, the application writes

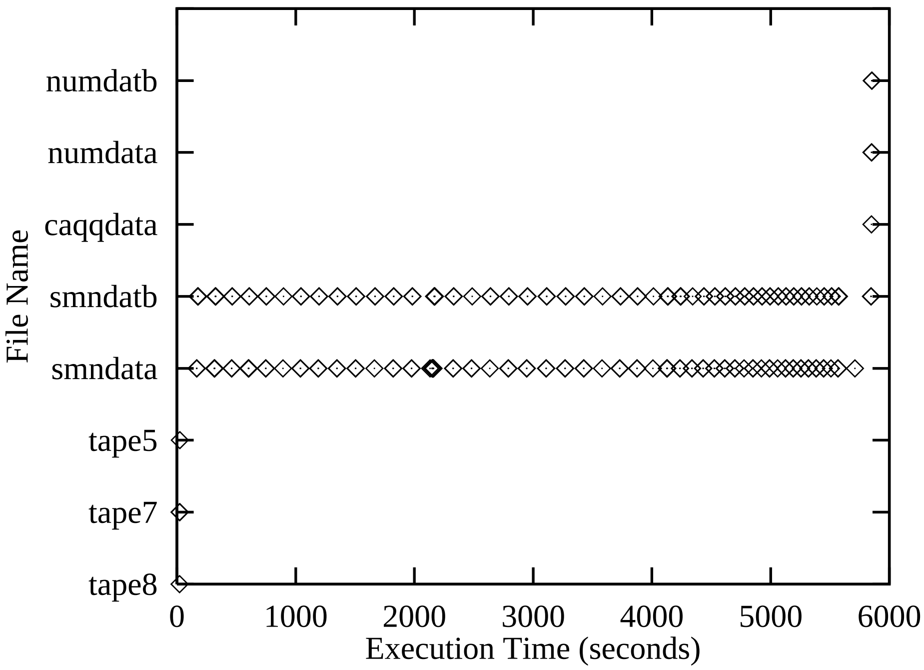


Figure 6.6: ESCAT: File access timeline.

the linear system matrices to disk (files **caqqdata**, **numdata** and **numdatb**) for later solution on another machine.

We have studied many versions of this code, exhibiting different input/output behaviors. Collectively, these versions exhibit three global access patterns: sequential with cardinality 1, sequential with cardinality p, and partitioned sequential. Figure 6.7 shows the first four classification windows for a partitioned sequential write only access pattern used to access the **smndata** file. Each horizontal bar represents a local classification over ten accesses, the length of the classification window. Space between bars is time between the end of the last access within one classification window and the beginning of the first access within the second window. Application synchronization ensures that no one processor races ahead of the others. A classification for all processors for access to this file can be made within the first time window, at approximately 1500 seconds.

6.3.4.2 PRISM

Computational fluid dynamics is another application area we have examined. PRISM is a parallel implementation of a 3-D numerical simulation of the Navier-Stokes equations [27, 28]. The

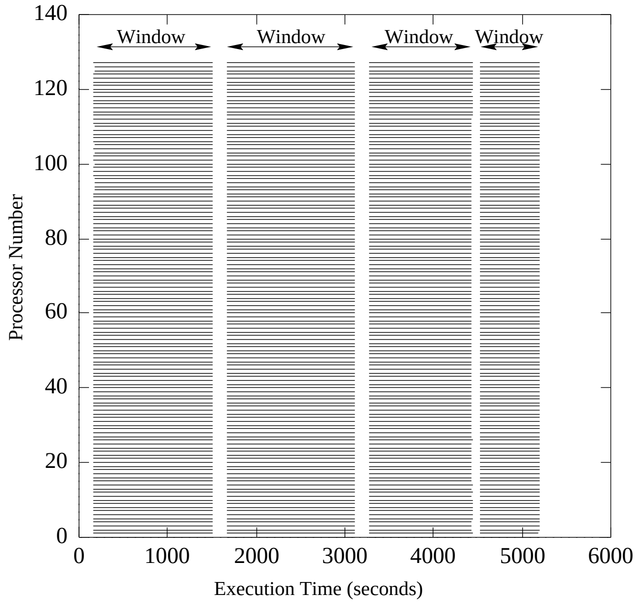
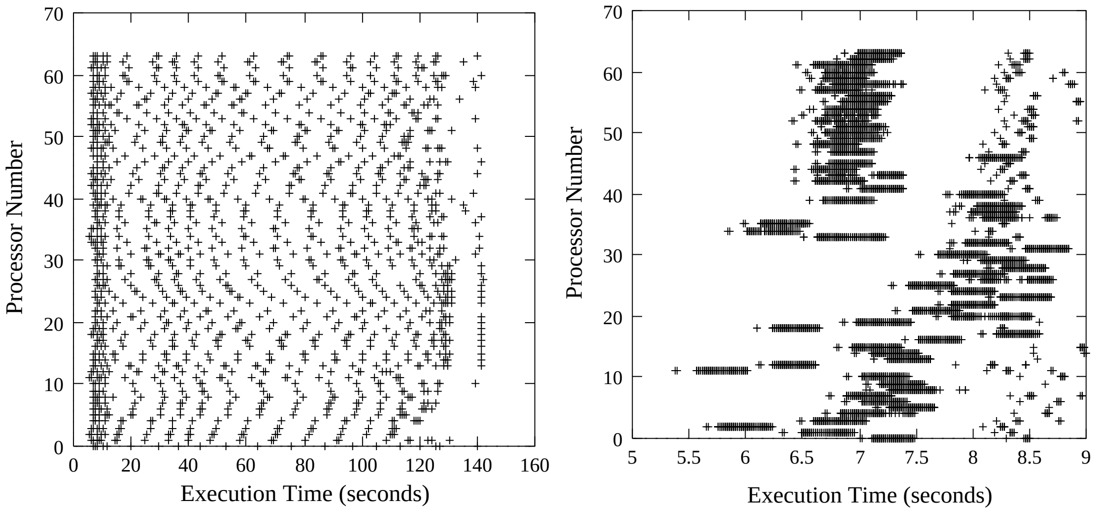


Figure 6.7: ESCAT: classification windows for a partitioned sequentially accessed file.

parallelization is by apportioning slides of the periodic domain to the nodes, with a combination of spectral elements and Fourier modes used to investigate the dynamics and transport properties of turbulent flow.

Figure 6.9 shows a file access timeline for PRISM. This code exhibits three distinct input/output phases. During the first phase, every processor reads three initialization files (`m16.rst`, `m16.rea` and `m16.mor`); each file is accessed with a global sequential access pattern. In the second phase node zero performs input/output on behalf of all the nodes, writing checkpoints and data (access to files `m16.Rstat`, `m16.Qstat`, `m16.Vstat`, `m16.mea` and `m16.his`. In the final phase, the result file is written to disk by all processors in an interleaved sequential access pattern `m16.fld`. Phases two and three occur iteratively throughout program execution.

Figures 6.8a and 6.8b show the endpoints for local classification windows for a globally sequentially accessed initialization file (`m16.rea`). This data was obtained under OSF/1 version 1.4. The reads are very small (most are < 50 bytes), and despite initial startup asynchronicity, there is a high degree of overlap in time, permitting an accurate global sequential classification to be made early in the read phase. Reading of this file continues until 137 seconds into program execution; however, a global classification with full cardinality can be made as soon as the last processor (number 31) begins reading the file, at 7.79 seconds.



(a) All classifications

(b) Detail

Figure 6.8: PRISM: Local classification points for global sequential access pattern.

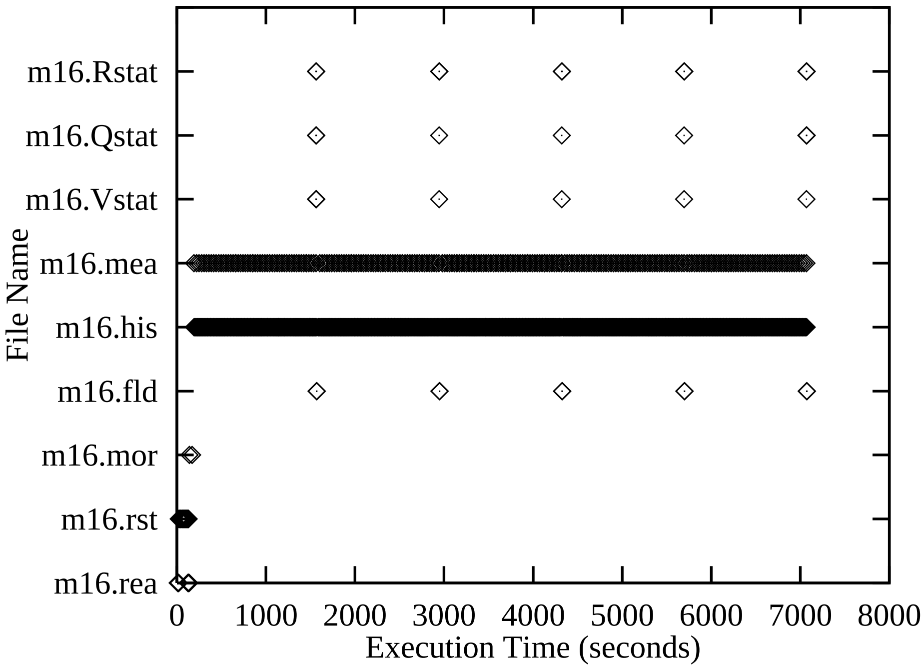


Figure 6.9: PRISM: File access timeline.

Figure 6.10 shows the first classification window for the output file **m16.fld**. Because the local classification for each window is 1-D strided and all of the bytes accessed are disjoint, the global classification is interleaved sequential. As Figure 6.9 indicates, access to the output file continues until the end of execution. Although the classification window length is large compared to the total execution time, it represents only ten input/output accesses on each processor. When compared to Figure 6.8, where classification window lengths are extremely short, it is obvious that the length of classification windows varies depending on the size of application-level input/output requests and system performance. We believe that global file access patterns can be detected early enough to positively influence file system performance. PRISM is not input/output intensive in the configuration tested; however, the input/output scalability of PRISM is crucial to solving larger problems. Larger problem sizes will result in longer access patterns, so that classification time will be a small percentage of overall execution time.

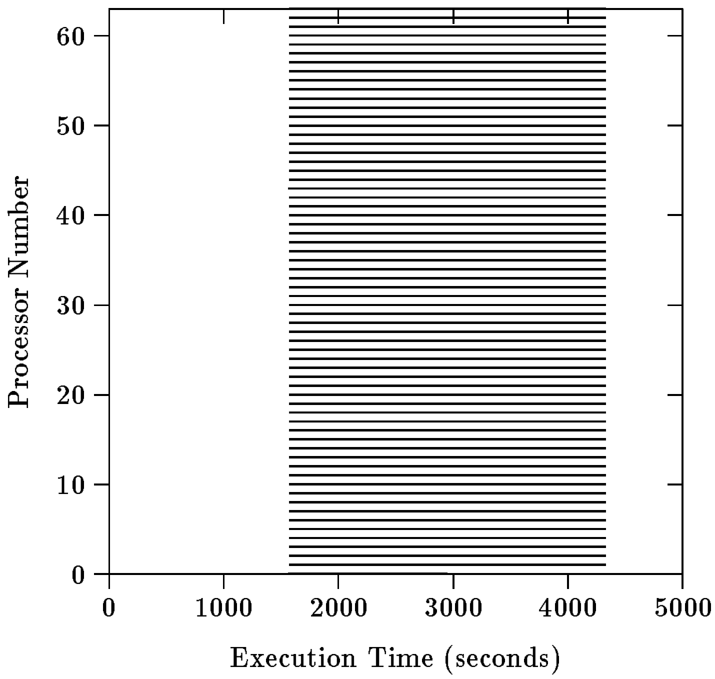


Figure 6.10: PRISM: classification window for interleaved sequential access pattern.

6.3.5 Global Classification and HMM Classification

We have described a general method for global classification that does not depend upon the source of the local classifications. However, when patterns do not change significantly between executions, the trained HMMs can be inspected at file open to construct an initial global classification.

Initial global classification using HMMs is exactly like normal classification, except that input/output statistics are unavailable. Therefore, only classifications of read/write and sequentiality characteristics are possible. Furthermore, accurate byte range information is necessary to identify overlapping accesses in global access patterns. For this approach to work for interleaved access patterns, the underlying HMM block size must correspond to the request size.

The cost of initial classification is the overhead of executing the application once or more to train the HMMs. This cost must be amortized over the performance improvements gained on 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 pattern). 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.

6.4 Global Classification in the PPFS Context

We use global classification in the context of PPFS to manipulate Intel PFS input/output modes and server cache buffering strategy. The chosen framework limits the potential input/output policies we can select; as described above there are a variety of ways in which global classification information can be used to improve performance. In this section we focus on the constraints imposed by the PPFS framework itself.

Our classification approach of viewing global access patterns through a UNIX style read/write interface ensures portability across a variety of interfaces, but eliminates the ability to perform optimizations that involve request reordering (e.g. collective operations) without higher-level information. The original PPFS interface is record-based, allowing input/output requests to be independent of each other (without the dependencies implied by a file pointer). In addition, PPFS supports file modes READ_ANY, allowing the application to request any record that has not yet been read by the application as a whole. This kind of optimization, which relaxes the ordered structure of records in a file, is impossible without additional application-supplied information.

Similarly, using an automatic classification that relaxes file system consistency can cause erroneous program behavior if the classification is incorrect or the access pattern changes. In PFS, improvements over M_UNIX obtained by selecting more appropriate modes result primarily from request parallelization or grouping, because this default mode serializes requests even when it is unnecessary (e.g., the file is opened for read only). If a file is opened for reading and writing, and atomic behavior is expected, a global classification is not sufficient to select policies that violate this guarantee. Strong consistency requirements must be specified by the application.

6.5 Summary

In this chapter we have described certain optimizations that any parallel file system should support, regardless of implementation, to provide high performance parallel input/output. We loosely classify these optimizations according to two major categories: those that exploit input/output request parallelism through collective operations or batching of requests, and those that hide input/output latency through caching and prefetching. Both of these approaches require knowledge of global input/output access patterns and file declustering strategy.

Commercial file systems support these optimizations through a variety of application programming interfaces designed to express global access pattern information and file layout. Ideally, automatic classification should supply a portion of this information that the application must currently specify.

We belive that global classification can provide the global access pattern information so important to performance. We have described our framework for global classification, and how our PPFS experimental framework uses global classification to select an appropriate PFS input/output mode.

Chapter 7

Global Access Pattern Classification Experiments

As motivated by the previous chapter, global access pattern information is critical to optimizing parallel input/output performance. In this chapter, we demonstrate how a flexible file system can use global classifications generated by our classification infrastructure to tune global policies and improve performance.

In 7.1 we describe our experimental environment. We layer PPFS over Intel PFS and use global classification to select file input/output modes. In 7.2 we test our experimental environment on simple global access pattern benchmarks. Finally, we examine performance of real applications in 7.3.

7.1 Experimental Environment

Our experimental platform is the Intel Paragon; we use PPFS to perform physical input/output using PFS [32] as the underlying file system. 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.

There are six PFS modes in OSF/1 Release 1.4, described in Table 7.1. The PFS modes we manipulate below are M_UNIX, M_RECORD, M_ASYNC, and M_GLOBAL. Mode M_UNIX is the default input/output mode; each processor has its own file pointer and UNIX style atomicity

Mode	File Pointer	Description
M_UNIX	individual	UNIX style atomicity guaranteed
M_LOG	shared	Global self-scheduled access, first come first served
M_SYNC	shared	Global synchronized self-scheduled access, node order
M_RECORD	individual	Interleaved sequential with fixed length requests
M_GLOBAL	shared	Global synchronized identical input/output
M_ASYNC	individual	Relaxes UNIX atomicity

Table 7.1: Intel PFS modes.

of access is guaranteed. M_RECORD corresponds to an interleaved sequential access pattern with fixed length requests. Processors using mode M_GLOBAL must all perform identical input/output operations. Finally, mode M_ASYNC provides individual file pointers but relaxes UNIX atomicity constraints to improve performance. 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 an optimal PFS input/output mode and buffering strategy for a file access pattern and compare performance to the default mode.

Note that PFS is implemented so that changing PFS modes not only affects performance, but can potentially change program semantics. However, our motivation for using it is to demonstrate the feasibility and importance of timely access pattern detection in conjunction with the well-defined set of policies provided by the PFS. As described in the previous chapter, the only information that must be assumed or provided for policy control based on global classification of these access patterns is that atomicity is not required.

7.2 Simple Global 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 pat-

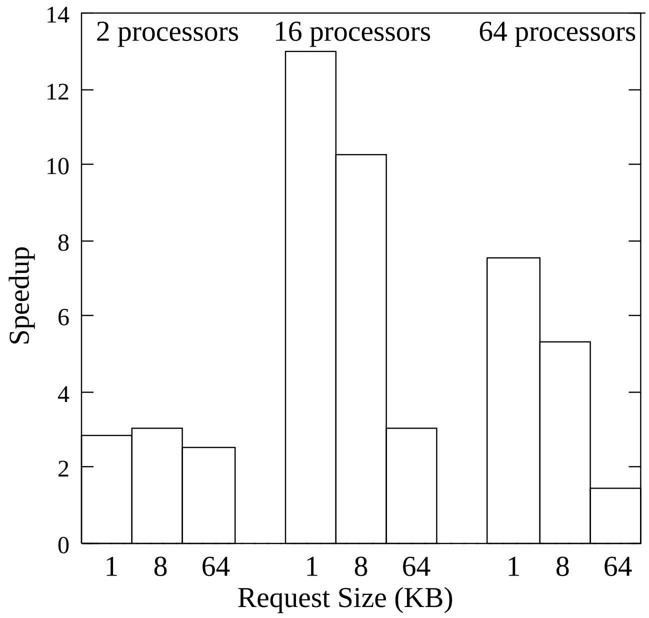


Figure 7.1: Global interleaved sequential speedups (automatic classification selects mode M_RECORD vs. M_UNIX).

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

Figures 7.1 and 7.2 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 patterns.

In general, when a file system knows 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. Performance using M_UNIX is generally much worse for smaller interleaved requests because of the serialized transfer overhead; consequently, M_RECORD speedup is larger for smaller requests.

Unfortunately, as the number of processors involved in an interleaved access pattern increases, the time necessary to classify the pattern increases as a percentage of total execution

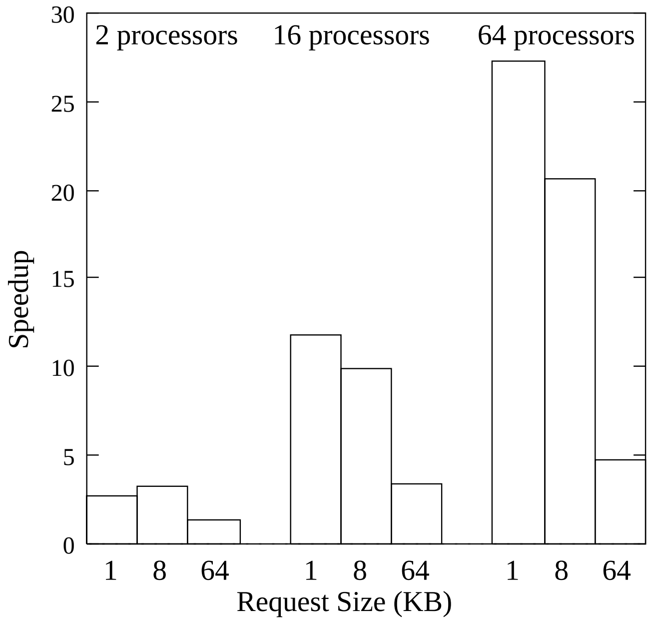


Figure 7.2: Global sequential speedups (automatic classification selects mode M_GLOBAL vs. M_UNIX).

time. This is because a fixed number of accesses are observed on each processor. Therefore, speedups obtained for 64 processors are poorer than those for 16 processors. Larger request sizes exacerbate this effect.

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

For a global sequential access pattern, classification occurs at the same point in relative execution time despite the number of processors. In contrast to an interleaved sequential access pattern, the input/output requirements scale with the number of processors. Therefore, speedups also scale with the number of processors. Small requests are also less efficient than large ones due to serialization of the additional overhead; therefore, the greatest speedups are from patterns with small request sizes.

Figure 7.3 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,

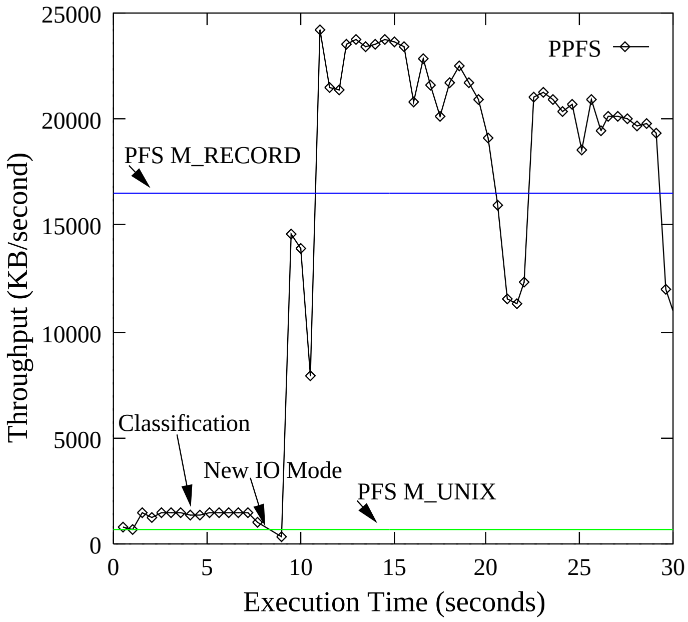


Figure 7.3: Global interleaved access pattern, 16 processors, 8 KB request size.

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 of native PFS less the overhead of PPFS library calls.

Speedups using the PPFS library are always smaller than those obtained by using the optimal mode from file open because of the time spent executing with default policies before making a global classification, and because of PPFS library overhead. In general, the larger the number of processors involved in input/output, the longer it takes to form a global classification, and the smaller the fraction of optimal speedup is obtained.

7.3 Parallel Applications

Having demonstrated that automatic classification can be used to improve input/output performance of simple benchmarks, we use this approach on parallel applications. Our selected applications are chosen from the Scalable Input/Output Initiative suite. Their input/output demands are characteristic of many scientific applications.

7.3.1 PRISM

PRISM, a computational fluid dynamics code, is a parallel implementation of a 3-D numerical simulation of the Navier-Stokes equations [27, 28]. 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 in even 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.

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.

Figure 7.4 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) and using HMM and ANN classification to select mode M_GLOBAL. Throughput using the default mode M_UNIX is very poor, because 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

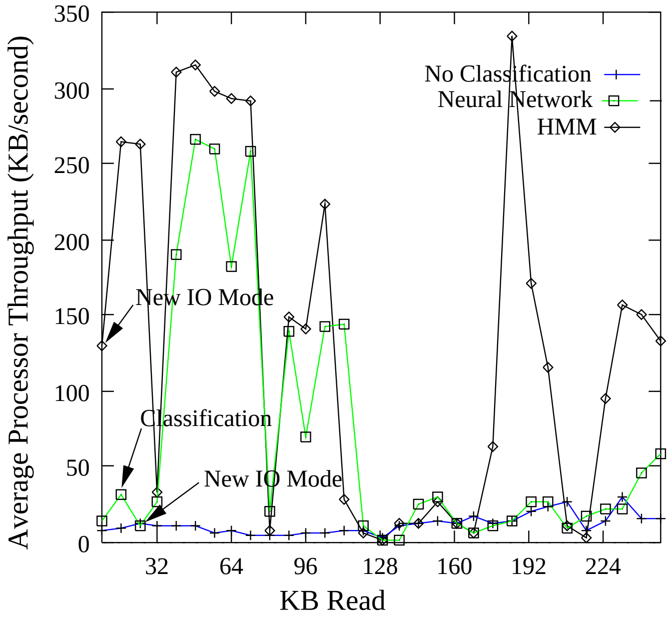


Figure 7.4: PRISM initialization file throughput.

(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 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 the processors.

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.

Table 7.2 shows the total read time for each of these experiments. Selecting mode M_GLOBAL during execution using ANN classification reduces total read time by approximately 90 seconds.

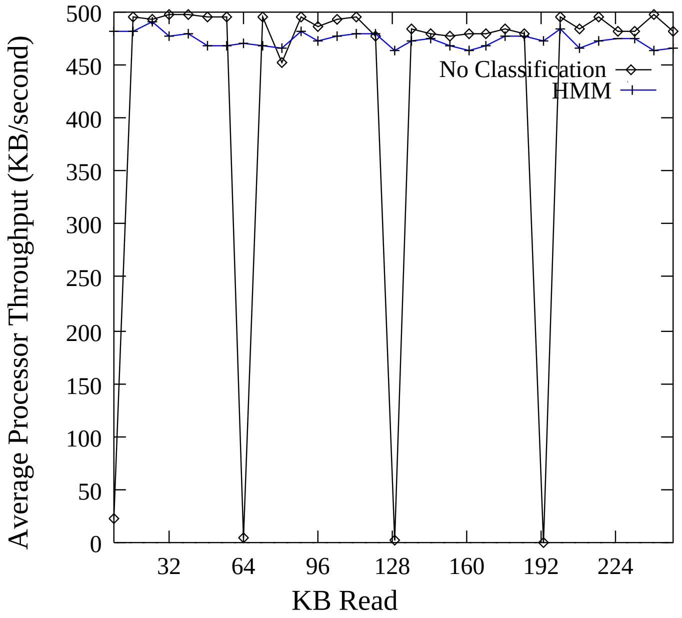


Figure 7.5: PRISM initialization file throughput with caching.

Experimental Environment	**Read Time**
PPFS (no classification)	430.24
PPFS (ANN)	339.892
PPFS (HMM)	263.363

Table 7.2: PRISM input/output times (seconds) for an initialization file.

Selecting this mode at startup, using HMM classification, reduces total read time by another 76 seconds.

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 7.5 shows the average per-processor throughput for the same initialization file using caching alone and this optimization using HMM classification. Local caching effectively

Experimental Environment	Open Time	Read Time
PPFS (no classification)	255.995	381.502
PPFS (HMM)	238.56	8.416

Table 7.3: PRISM input/output times (seconds) with caching enabled for an initialization file.

increases the file system read size from 8 KB to 64 KB. When individual processors access file blocks in local cache, throughput is equivalent. However, cache misses cause huge dips in throughput as all processors attempt to access the identical disk block. Table 7.3 shows the read and open times; user-level buffering of this file significantly reduces read time. Total open time is comparable for both experiments, implying that the cost of reading and broadcasting a small initialization file is little more expensive than the cost of synchronizing the processors to open the file.

7.3.2 QCRD

QCRD [94, 50] 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 can be described by five phases; we limit our analysis to the first two phases of QCRD, using a moderately sized data set.[1] In phase one, 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). In phase two, 64 processors coordinate to read these matrices with an interleaved access pattern. 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, each controlling a 4 GB Seagate disk.

[1] Qualitative performance of the remaining three phases is similar.

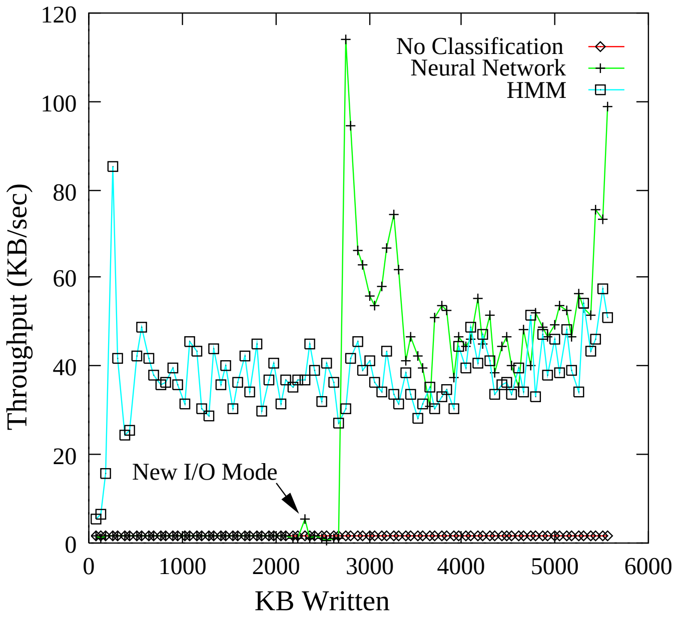


Figure 7.6: QCRD phase one throughput.

7.3.2.1 Interleaved Writes

In phase one of QCRD, processors write thirteen output matrices using global interleaved access patterns after node 0 writes a small header. When using the default mode M_UNIX, input/output operation time for phase one 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 7.6 shows the throughput, calculated every 64 KB, for one of the thirteen output matrices. Throughput using mode M_UNIX (no classification) is poor because in the PFS implementation of this mode, seeks are serialized and are very expensive. 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 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 7.4. The neural network classifier needs to observe some number

Experimental Environment	Write Time	Total I/O Time	Execution Time
PPFS (no classification)	46616.97	47286.69	7013.47
PPFS (ANN)	28579.68	30640.01	6953.44
PPFS (HMM)	3697.10	4545.88	6569.91

Table 7.4: QCRD phase one overall input/output times (seconds).

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 write time is a little 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. Early detection of global access patterns is important to overall performance, especially when input/output operations are large, many processors are involved, and accesses are disjoint.

7.3.2.2 Interleaved Reads

In phase two of QCRD, 64 processors collectively read 13 matrices (written in phase one). All processors read the first 2412 header 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 bytes), enabling I/O server buffering further improves performance.

Figure 7.7 shows the total throughput, calculated every 128 KB, for one of the input matrices that is read twice. Using the two classification methods described in Chapter 3, 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.

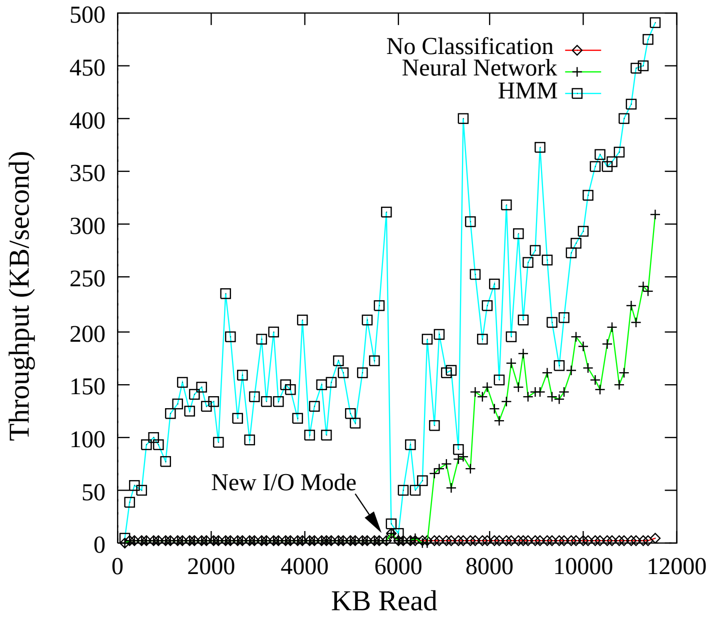


Figure 7.7: QCRD phase two throughput.

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	996.15
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 7.5: QCRD phase 2 input/output times (seconds).

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 obtains 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 7.5 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.

7.4 Complex Global Access Pattern Benchmarks

Our global classification framework emphasizes detection of global sequentiality. Although such access patterns are common in scientific codes, often this is because application requirements have been tuned to parallel file system implementations. Not all global access patterns are best optimized by recognition of sequential subsections. In particular, much research effort has been devoted to optimizing access to complex matrix decompositions through collective operations [13, 79, 7].

In this experiment we consider a benchmark based on access patterns exhibited by applications from Caltech for global climate modeling and modeling of the earth's interior [52]. These applications read and write data from a two or three dimensional grid, partitioning data in blocks of contiguous particles among the processors.

Figure 7.8 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 through our proposed global classification framework.

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

One 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.

Prefetching can improve performance by overlapping input/output latency with computation, but in this example there is no global sequentiality throughout the file; prefetching must correspond to the access pattern. Local classifications can identify each local pattern as two-dimensionally strided and potentially guide prefetching, but this approach is not generalizable to arbitrary decompositions. Because HMMs can be trained to recognize any arbitrary repeatable access patterns, they are most useful for access prediction.

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 7.8. 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 7.6 shows the input/output times for PFS operations for this benchmark. Prefetching one block ahead is not sufficient 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.

7.5 Summary

Results presented in this section demonstrate that timely global classifications can be made using our proposed framework, and that they can be used to select among a well-defined set

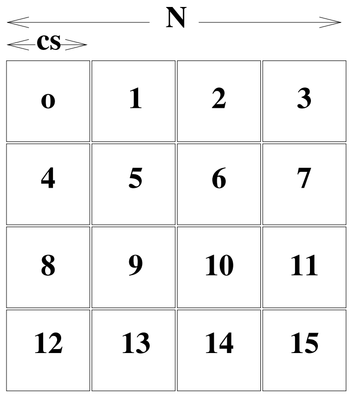


Figure 7.8: Block-block matrix partitioning.

of policies to improve performance of applications. If access pattern information from previous executions is available, it allows a better initial global classification. In parallel applications, a good initial strategy is particularly important because it can take longer to classify global access patterns than local access patterns.

Chapter 8

Access Pattern Classifications and Hints

We have demonstrated how to use global classification to select appropriate file modes within Intel PFS. However, as described in Chapter 6, file systems utilize access pattern information in a variety of ways. Automatic classification will benefit any file system interface that can use access pattern information to tune file system policies. We can view an access pattern classification as an automatically generated hint, used to select and tune any variety of file system policies.

In this chapter we describe how local and global classification could work in conjunction with the low level API (LL-API) [15] proposed by the Scalable I/O Initiative. This interface was developed by a consortium of universities, national laboratories, and industries involved in parallel input/output research. It includes functions that enable high performance to create a suitable foundation for writing higher-level application libraries. As such, it is ideal for exploring the interface between classification and policy control.

8.1 Low-Level Application Programming Interface

The SIO low-level API is an extension to the POSIX standard file interface to support high performance input/output. As described in Chapter 6, commercial parallel file systems each use access pattern information very differently at the user interface level. The low-level API attempts to include the functionality necessary for obtaining high performance from higher level user interfaces such as MPI-IO or PFS I/O modes.

Certain important design decisions were made to ensure that this foundation could efficiently support a variety of higher-level interfaces. To avoid reliance upon any particular message passing abstraction, the file system allows parallel applications to choose their own; it does not require clients to communicate with each other. Shared file pointers must be provided by higher-level libraries because the synchronization required to support them is expensive. Similarly, while caching may be supported, cache consistency is not required. Interfaces for scatter-gather and asynchronous input/output are provided. Vendors may use a flexible input/output control mechanism to allow the application to specify performance-related information (e.g., file layout and striping). Finally, the low-level API supports access pattern hints as a method for supplying access pattern information to improve performance. Suggested extensions (that need not be included in an implementation) include support for collective input/output.

There are two ways to specify access pattern information to the low-level API. The first is via hints that reveal anticipated access patterns to the file system. The second is through specific file system calls that encapsulate access pattern information, e.g., scatter-gather or collective input/output.

The low-level API specifies two classes of hints, ordered and unordered, but vendors may provide additional hint classes. Hints are valid for the specific task that issues them and for a particular file region. Ordered hints specify accesses that will occur, and the order they are issued corresponds to the order the hinted accesses will occur. This allows the task to interleave hints to multiple files. Unordered hints qualitatively describe future accesses with no explicit ordering; they can express randomness and reuse within file regions without specific access information. These hints remain in effect until cancelled.

8.2 LL-API Examples

The SIO low-level API is not intended as a user-level programming interface, rather, it is a performance-oriented API for writing higher level libraries and interfaces with different input/output models. However, in this section we show an example of code to write a file sequentially, providing unordered hints.

```
typedef long sio_offset_t;
typedef long sio_size_t;
typedef unsigned long sio_count_t;
typedef unsigned long sio_transfer_len_t;
typedef unsigned short sio_hint_class_t;

typedef struct
{
  sio_hint_flags_t flag;
  sio_file_io_list_t * io_list;
  sio_count_t list_len;
  void * arg;
  sio_size_t arg_len;
} sio_hint_t;

typedef struct
{
  void * addr;
  sio_size_t size;
  sio_size_t stride;
  sio_count_t element_cnt;
} sio_mem_io_list_t;

typedef struct {
   sio_offset_t offset;
   sio_size_t size;
   sio_size_t stride;
   sio_count_t element_cnt;
} sio_file_io_list_t;
```

Figure 8.1: SIO type definitions.

The SIO LL-API requires that certain types be defined. Definitions of data types used in our example are shown in Figure 8.1. File descriptors are integers, for compatibility with UNIX. Memory addresses are of type `void *`. The basic structures defined in Figure 8.1 are `sio_hint_t`, to store hint information, and `sio_file_io_list_t` and `sio_mem_io_list_t`, which describe a collection of regions in file or memory space, respectively.

Because strided accesses corresponding to matrix decompositions are common in scientific codes, the `sio_file_io_list_t` structure efficiently describes a strided access pattern beginning `offset` bytes into the file, consisting of `element_cnt` requests of size `size`, each request starting `stride` bytes after the start of the preceeding request. The corresponding structure

```
int fd;
sio_return_t status;

sio_file_io_list_t iorequest;
sio_mem_io_list_t  buf, wholefile;
sio_transfer_len_t total;
sio_hint_t         hint;

int nrecords = 0;
int length = MAXRECORDS;

/* open the file */
status = sio_open(&fd,                          /* file descriptor pointer */
                  "myOutputFile",               /* file name */
                  SIO_MODE_CREATE|SIO_MODE_WRITE,/* file mode */
                  0,                            /* control ops */
                  0);                           /* control op count */
```

Figure 8.2: Opening a file for output using the SIO LL-API.

sio_mem_io_list_t describes the placement of file data in memory. Together these structures can be used to compactly describe very complex file input/output operations.

The hint structure, **sio_hint_t** stores information describing a hint. The **flag** field describes the access pattern being hinted, and **io_list** field contains the file regions for which a hint is valid.

Figure 8.2 shows a fragment of code that uses the SIO low level API to open the file called "myOutputFile". The LL-API open call is similar to the UNIX open, except that it does not check file permissions and allows specification of control operations, which allow properties of files and file descriptors (e.g., preallocated file size) to be set and retrieved. Control operations may be specified at file open or later during the file lifetime.

Figure 8.3 shows how to hint that our newly opened file will be written sequentially. The **sio_hint** call takes as arguments the file descriptor, the hint class, and an array of access pattern hints with its corresponding length. We have specified in the **hint** and **wholefile** structures that the pattern is write only and sequential, consisting of **MAXRECORDS** segments of size **RECSIZE**.

```
/* below is the sio_file_io_list_t specification for the whole file,
   to be provided to the hint call */

wholefile.offset =0;
wholefile.size = RECSIZE;
wholefile.stride = RECSIZE;
wholefile.element_cnt = MAXRECORDS;

/* hint specification */

hint.flag = HINT_WRITE && SIO_HINT_SEQUENTIAL;
hint.io_list = &wholefile; /* hint is valid for the whole file */
hint.list_len = 1;
hint.arg = NULL;
hint.arg_len = 0;


/* hint sequential access */

status = sio_hint(fd,                         /* file descriptor */
                  SIO_HINT_CLASS_UNORDERED,/* hint class */
                  &hint,                      /* array of access pattern hints */
                  1);                         /* number of access pattern hints */
```

Figure 8.3: Hinting sequential file access using the SIO LL-API.

Figure 8.4 shows the code that actually writes the file with this access pattern. We iterate once for each record, for a total of `MAXRECORDS` iterations. At each iteration we initialize the `iorequest` structure with the parameters corresponding to the record that will be written, using the `nrecords` variable to calculate the correct file offset. The application-defined call to `fillOutputBuffer` copies the output data to the `addr` field of `buf`. The `sio_sg_write` call writes the record to its place in the file. Finally, we close the file with `sio_close`.

8.3 Automatically Specifying Local Hints

The SIO LL-API has a rich, extensible interface for local hints that the file system can use to make policy decisions. A local access pattern classification can be recast as an automatically generated SIO ordered or unordered hint. Qualitative, neural network classifications are anal-

```
/* initialize buffer */
buf.offset = 0;
buf.size = RECSIZE;
buf.element_cnt = 1;

while (nrecords < MAXRECORDS) {
     total = 0;

     /* request is for one segment of size RECSIZE */
     iorequest.offset = nrecords * RECSIZE;
     iorequest.size = RECSIZE;
     iorequest.stride = 0;
     iorequest.element_cnt = 1;

     fillOutputBuffer(buf.addr);        /* generate output */


     result = sio_sg_write(fd,          /* file descriptor */
                          &iorequest,   /* file list */
                          1,            /* file list length */
                          &buf,         /* memory list */
                          1,            /* memory list length */
                          &total);      /* total bytes transferred */

     nrecords++;

}

/* close the file */
status = sio_close(fd);
```

Figure 8.4: Writing a file sequentially using the SIO LL-API.

ogous to unordered hints, while HMM classifications can be used to supply both ordered and unordered hints.

8.3.1 Neural Network Classification and Hints

All SIO hints are valid for a particular file region, specified in the arguments to the hint file. When providing hints from ANN classifications, we can either assume the classification is valid for the entire file, or for the subset of the file that is likely to be accessed before reclassification.

Sequentiality	Read/Write	Request Size	SIO Hint
*	Read Only	*	SIO_HINT_READ
*	Write Only	*	SIO_HINT_WRITE
*	Read/Write	*	SIO_HINT_READ,SIO_HINT_WRITE
Sequential	*	*	SIO_HINT_SEQUENTIAL
Random	*	*	SIO_HINT_RANDOM_PARTIAL

Table 8.1: Mapping local classifications to unordered hints.

Table 8.1 shows the mapping of local classifications to unordered hints. SIO_HINT_READ and the corresponding SIO_HINT_WRITE hint that the file is to be read or written; if both are specified together the access pattern is read-write. SIO_HINT_SEQUENTIAL specifies that access within the relevant file region is nonoverlapping and monotonically increasing. There is a corresponding hint for reverse sequential access, SIO_HINT_REVERSE, a pattern not recognized by the current neural network classifier. This is an example of a pattern that the neural network could be easily retrained to recognize if it occurred frequently. Finally, there are two hints for specifying random access to a region, SIO_HINT_RANDOM_COMPLETE and SIO_HINT_RANDOM_PARTIAL; the former specifies that the entire region will be accessed randomly and the latter indicates that the entire region may not be accessed.

For a random access pattern classification, the corresponding hint is SIO_HINT_RANDOM_PARTIAL, because it is impossible to accurately distinguish complete and partial random access patterns without user input. The small window of accesses used to make an ANN classification simply does not contain enough information to predict this feature. At best, as we will describe below, an HMM classification can provide a good guess based on behavior of previous executions.

The same limitation prevents us from being able to hint about future use; SIO_HINT_WILL_USE and SIO_HINT_NO_FURTHER_USE indicate whether all bytes in the specified file regions will be accessed in the future, information unavailable from dynamic classification using a small window of accesses.

8.3.2 HMM Classification and Hints

HMM classification can supply more specific hints than ANN classification because a trained HMM specifies the probabilities of transitions (reads or writes) between file regions. At any time, the HMM can be used to compute ordered or unordered hints for patterns or block accesses expected to occur with high probability within a specific file region.

For example, reconsidering unordered hints, using HMM classification we can determine whether any region starting from the current position in the file will be accessed reverse sequentially by calculating the probability of transition sequences to monotonically decreasing consecutive blocks. The access pattern is reverse sequential for the region in which this probability remains high. It is also possible to hint about use and reuse of file regions by computing the probability of reaccessing blocks within those regions. If at any time the application access pattern differs from the hinted access pattern, any previously supplied hints can be cancelled.

The interface for specifying ordered hints is simpler. The application hints about forthcoming reads and writes in the order they are expected to occur, and can cancel inaccurate hints. A process can simply use a trained HMM to hint continuously for some number of accesses in advance of the access stream. This will work well if only one file is being accessed by the hinting thread; unfortunately neither of our classification methodologies take time into account, so ordering hints correctly among files is a potential problem.

8.4 Using Global Access Pattern Information

All hints described in the SIO LL-API are local to the task that specifies them; there is no way for one task to hint about a global access pattern. Any coordination of local hints to glean and utilize global pattern information must occur at a higher level. Part of the reason for this is the mandate that the LL-API not require clients to communicate among themselves. Global coordination is expensive, so the improvement from any optimization requiring it must offset the additional synchronization and communication overhead. As shown in Chapter 7, we obtained application speedups despite PFS synchronization overhead; improvements from correct global policy decisions can far outweigh the cost.

Therefore, global classification can be implemented above the LL-API in the same way PPFS is implemented over Intel PFS in our experimental framework. Instead of using global classifications to select appropriate access modes, the file system can use them to tune local vendor-specific controls, select appropriate caching policies, or manipulate underlying collective input/output operations. The benefit of any particular policy selection depends upon the underlying file system architecture and the implementation of the low-level API.

For example, if the global classifier identifies a pattern as global interleaved, it may be efficient to issue "rounds" of input/output calls (one or more requests per processor, forming a single large sequential input/output) as a single collective operation. This gives the file system the flexibility to reorder the strided requests so it can perform a single sequential input/output.

The LL-API specifies file system controls that might also be useful upon detection of partitioned access patterns or nested strided patterns. If the access pattern does not match the default data layout, it might be more efficient to change it either at file open or during execution. [1]

Some access-pattern specific optimizations do not depend upon low-level file system functionality and can be implemented at a higher level. For example, in Chapter 7 we described a user-level optimization for small, read-only files accessed with a global sequential access pattern; one processor reads the file and broadcasts it to the others. With HMM classification we can identify this pattern at file open and initiate this read/broadcast strategy.

Other natural optimizations are highly dependent upon file system structure. For example, knowledge of the global access pattern is extremely useful for guiding prefetching at the input/output server level. If clients propagate local hints to the input/output servers, each server can construct its own view of the global access pattern (comprised of the intersection of all hinted bytes and the portion of the file it controls). With this information it can choose which blocks to cache and prefetch, considering the global access stream as a whole. A global classifier centralizes this process and simply updates input/output server policies directly.

8.5 Summary

Global classification can benefit any file system that provides a mechanism for using access pattern information in policy decisions. Having previously demonstrated experimentally its utility using the well-defined set of policies provided by the Intel PFS, in this chapter we

[1] Maintaining redundant storage formats can make dynamic selection of data layout an efficient operation.

describe how it can be used in conjunction with the SIO proposed low-level API. As a design for a file system interface intended as a building block for higher-level libraries, the LL-API represents functionality deemed most important for high performance. Local classification maps directly into hints in the LL-API, and global classification can be supported above the interface to tune vendor-specific controls and modify higher-level global policies.

Chapter 9

Future Work

In previous chapters, we introduced a new methodology to automatically classify input/output access patterns, both locally and globally. File systems can use access pattern information to select policies that optimize performance.

Experimental data from sequential and parallel benchmarks and applications demonstrate the potential of this approach. However, this research may be extended in many interesting directions that we describe in this chapter.

9.1 Policy Selection Issues

We have described simple methods for using local or global classifications to make either local or global policy decisions. However, many unexplored issues remain regarding the interaction of local and global policy selections, and policy control for the file system as a whole. This section outlines some directions for further research involving the use of classifications to select file system policies.

9.1.1 Local and Global Policy Control

Approaches described in this thesis have considered using classification either for local or global policy control. However, it is clear that local and global prefetching and caching policies are interrelated and must be considered together.

For example, consider local and global prefetching policies for a simple global interleaved pattern. Exploiting this global information to prefetch the file sequentially may be less effective than having each processor prefetch its required blocks if the individual processor request rates vary significantly throughout program execution. Simply put, does the improved throughput obtained from global sequential prefetching outweigh the cost incurred by making processors that might normally proceed more quickly wait for the sequential prefetches to complete? When there is sufficient disk bandwidth to satisfy the global input/output demands, client prefetching successfully hides input/output latency even if processors are not synchronized. When disks become heavily loaded, or there are load imbalances at the servers, maximizing throughput by prefetching the interleaved pattern sequentially is more important.

An analogous situation occurs with writebehind policies, although application stall time is not a critical factor. Consider a global interleaved write-only access pattern. Assuming clients do not share data, they can cache blocks locally or write them through to the server cache level. Ideally the servers should make large, contiguous writes, which is possible if the processors are roughly synchronized. The ideal writebehind sequence is the reverse of the best prefetch sequence. Local (client) and global (server) caching policies are clearly interrelated in a fundamental way.

Expected interaccess latency of processors in a global access pattern is crucial to this model of prefetching and writebehind. We believe that a promising research direction involves extending a trained HMM to remember estimates of interaccess latencies. These estimates could provide the file system with dynamic global load estimates to weigh local vs. global caching decisions.

9.1.2 Global Classification in a Multi-application Context

Up until now we have considered input/output performance per file, without regard to the system environment as a whole. Although examining how classification can tune per-file performance is a necessary precursor to optimizing overall file system performance, the second problem is more challenging and difficult.

We could attempt to extend our interpretation of global classification to be the interleaving of the input/output access patterns of all system tasks. Unfortunately, information is lost in such an expansive global classification; it is unlikely that it would be able to describe system input/output demands accurately enough to be useful for tuning policies.

In an environment where multiple applications compete for input/output resources, file system policies cannot operate autonomously. There must be a mechanism for mediating resource requests from competing applications. We raised this issue in our analysis of the Pathfinder code; even in the single processor, single application case, cache space partitioning is challenging. Using trained HMMs we can more accurately estimate space requirements, but this is only a small step towards solving the problem. Interaccess latency estimates, as described above, would further help to sort out resource allocation in a multi-file, multi-application environment. However, the policy selection framework must be capable of incorporating such information. The next section describes more sophisticated integration of classification and policy control in greater detail.

9.2 Integrating Classification and Policy Selection

In Chapter 3 we describe our underlying classification and policy selection methodology: architecture-independent qualitative classifications form a classification space that can be mapped onto system-dependent file system policy decisions.

Although this approach of disengaging classification from policy control is highly portable, information is lost in the translation of the access stream to a qualitative classification. For example, a classification of 1-dimensionally strided is not very useful to a prefetcher unless the average request size and stride size are also known. It does not matter what the access pattern is actually called as long as it can be used to improve performance.

Hidden Markov models are a step toward this realization; by constructing probabilistic models of access patterns they retain important information that is lost in the act of making a qualitative classification. However, they are still architecture independent; architecture dependent information (e.g. interaccess latencies) is removed.

We believe a promising research direction lies in eliminating the step of qualitative classification altogether and linking access pattern statistics, probabilistic access pattern models and file system performance metrics directly to system-dependent policy controls. The input to the decision mechanism is the multidimensional space of access pattern and performance metrics, and the output is the optimal file system policies. This approach utilizes all available information about access pattern and system state while encapsulating the decision mechanism in an extensible, system-dependent module.

A neural network might be used to implement this decision mechanism as a natural extension of our classification work. The network would be trained on the same access pattern statistics that the classification network currently uses, in addition to metrics that provide information about the file system state. Instead of a qualitative classification, training output is a description of file system policies that provide the best performance for these benchmarks. These policies would be determined in advance, experimentally. When applications run that exhibit input/output behavior similar to the benchmarks, the file system will adapt accordingly.

Our suggestion of using a neural network reflects the assumption that the mapping of access pattern and performance information may be too complex to encode in a simple rule database. If it is not, or if most of the performance improvements made possible by dynamic adaptation can be expressed by simple rules, the information should be codified in simple ruleset. One method we are currently investigating for writing such a ruleset is fuzzy logic [96, 34]. Fuzzy variables and sets allow a ruleset to encapsulate approximate rules. For example, in the following rule "If accesses are *small* and *sequential* prefetch *aggressively*," the italicized concepts are fuzzy variables. Additional variables and rules can be added as necessary to refine policy selection on the basis of the overall system state.

Our policy selection methodology has treated caching and prefetching policies as separate entities; however, they are closely related to each other and are often fully integrated in current research literature that investigates prefetching for multiple disks [35, 92]. By using access pattern hints to determine the value of allocating prefetch buffers compared to cache buffers, prefetching decisions are tightly integrated with cache replacement policies.

When considering cache allocation and prefetching in a system-wide context, policy controls as simple as "MRU" or "disable caching" are unlikely to suffice. The final step in integrating classification and policy control would be to use learned probabilities of future accesses in combination with interaccess latencies and input/output performance statistics to dynamically rank cache blocks, perhaps using a set of fuzzy logic rules, throughout application execution. The caching/prefetching strategy would utilize these rankings to choose blocks for prefetching and replacement. For example, a block from a sequentially read file that will be never be reaccessed is a much better candidate for replacement than one that will be reused multiple times in the near future.

Chapter 10

Conclusions

We have described how to automatically classify access patterns, using this information to dynamically select and tune file system policies. Below, we briefly summarize the contributions of this thesis before describing them in greater detail.

- We presented two complementary methodologies for classification of local access patterns. The first uses a trained neural net to generate a qualitative classification from statistics computed from a small window of input/output accesses. The second method constructs a probabilistic model of file accesses from previous execution traces using training hidden Markov models. These classification methods are completely novel.

- Because parallel input/output performance is a function of the global access pattern, we defined a way to combine local classifications to make global ones. Like local classifications, global classifications can be mapped to system-dependent file system policies to improve overall parallel program performance. Other global classification approaches are limited to recognizing sequentiality in the global request stream. By combining local classifications, we can identify a much wider range of global behavior.

- We proposed a method for dynamically tuning performance by mapping regions in the delimited space of access patterns to suitable, system-dependent, policies. We demonstrated that automatically generated local and global classifications can be used to make file system policy decisions that improved performance of uniprocessor and parallel input/output intensive applications. This approach frees the programmer from supplying this information.

10.1 Local Classification

Many studies have shown that matching file system policies to access patterns is a key factor in input/output performance. Rather than requiring the programmer to conform to static system-wide policies or manually specify access pattern information to an adaptive file system, we derive this information automatically.

Our approach to local classification, or classification of a single execution thread, is to transform an evolving, quantitative input/output trace into a qualitative description that can be used (in conjunction with quantitative statistics) for prediction of future accesses. Neural networks are highly efficient classifiers, though they are limited in the complexity of patterns they can identify. Nevertheless, even simple qualitative classifications correspond to user-supplied hints in other systems.

Frequently an application will exhibit a file access pattern that defies simple qualitative classification but has a definite, predictable structure. To classify such patterns we use hidden Markov models to construct probabilistic representations of file accesses from traces of previous executions. The larger access pattern context afforded by previous execution data makes HMM classification better able to detect certain large-scale access pattern features. The tradeoff is the expense of the additional training.

Although learning methods have been used to classify access patterns in different contexts, both of these approaches are completely novel. Furthermore, the use of hidden Markov models rather than simple Markovian predictors allows modeling of more complicated behavior such as multiple access patterns. However, even a simple degenerate HMM predictor is sufficient to accurately model complicated access patterns.

10.2 Global Classification

Local classification is but a small facet of the larger problem of global access pattern classification. When multiple threads of a parallel program access a file that is declustered across several input/output nodes, recognition and exploitation of the overall global pattern is crucial to obtaining high performance.

We propose a method of combining local classifications to make global classifications that is both responsive and efficient. We have implemented this classification framework within a portable parallel file system testbed and find that it is capable of recognizing a variety of important global patterns.

This approach of combining local classifications to make global classifications is also novel. Other approaches to global classification are limited to detection of sequentiality in the merged request stream. By combining local classifications we can recognize a larger range of patterns and tune file system policies accordingly.

Our experimental results demonstrate the importance of early classification, particularly early global classification. We have found that a history-based classification can be used to select efficient initial policies.

10.3 Intelligent File System Policy Selection

To test the efficacy of our classifications to select and tune file system policies, we have implemented our classification framework within the portable parallel file system (PPFS) testbed. A mapping between access pattern classifications and file system policies selects appropriate policies dynamically throughout execution.

Even on a uniprocessor UNIX workstation, tailoring the user-level cache size and cache block size and cache to meet the application requirements dramatically improves performance of input/output intensive satellite data processing codes.

PPFS uses global classifications on the Intel Paragon to select among the well-defined set of policies corresponding to certain access patterns. We have tested this approach on codes from the SIO application suite and found that timely global classification is extremely effective as a means to provide the file system with the access pattern information that is crucial to high performance.

10.4 Summary

Access pattern information is important for optimizing input/output. Inferring such information automatically reduces programmer burden and increases portability.

In this thesis, we have proposed an approach to optimizing parallel input/output where automatic, system-independent file access pattern classifications are used to select and tune local and global file system policies. Our proposed classification methods are accurate and responsive; it is possible to extract much of the access pattern information that is critical to performance automatically. Experimental results demonstrate the utility and promise of the research.

Bibliography

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

[2] ANDERSON, T. The Case for Application-Specific Operating Systems. In *the Third Workshop on Workstation Operating Systems* (1992).

[3] BAKER, M. G., HARTMAN, J. H., KUPFER, M. D., SHIRRIFF, K. W., AND OUSTERHOUT, J. K. Measurements of a Distributed File System. In *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Oct. 1991), Association for Computing Machinery SIGOPS, pp. 198–212.

[4] BASSOW, F. E. Installing, Managing, and Using the IBM AIX Parallel I/O File System. IBM Document Number SH34-6065-00, February 1995. IBM Kingston, NY.

[5] BAYLOR, S. J., AND WU, C. E. Parallel I/O Workload Characteristics Using Vesta. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems* (April 1995), pp. 16–29.

[6] BAYLOR, S. J., AND WU, C. E. Parallel I/O Workload Characteristics Using Vesta. In *Input/Output in Parallel and Distributed Computer Systems*, R. Jain, J. Werth, and J. C. Browne, Eds. Kluwer Academic Publishers, 1996, ch. 7, pp. 167–185.

[7] BENNETT, R., BRYANT, K., SUSSMAN, A., DAS, R., AND SALTZ, J. Jovian: A Framework for Optimizing Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference* (October 1994), IEEE Computer Society Press, pp. 10–20.

[8] 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).

[9] CAO, P., FELTEN, E. W., AND LI, K. Application-Controlled File Caching Policies. In *Proceedings of the 1994 Summer USENIX* (June 1994).

[10] CAO, P., FELTEN, E. W., AND LI, K. Implementation and Performance of Application-Controlled File Caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation* (Nov. 1994).

[11] CHARNIAK, E. *Statistical Language Learning.* The MIT Press, 1993.

[12] CHEN, Y., FOSTER, I., NIEPLOCHA, J., AND WINSLETT, M. Optimizing Collective I/O Performance on Parallel Computers: A Multisystem Study. In *Proceedings of the 11th ACM International Conference on Supercomputing* (July 1997), ACM Press.

[13] CHOUDHARY, A., BORDAWEKAR, R., HARRY, M., KRISHNAIYER, R., PONNUSAMY, R., SINGH, T., AND THAKUR, R. PASSION: Parallel And Scalable Software for Input-Output. Tech. Rep. SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[14] CORBETT, P., FEITELSON, D., FINEBERG, S., HSU, Y., NITZBERG, B., PROST, J.-P., SNIR, M., TRAVERSAT, B., AND WONG, P. Overview of the MPI-IO Parallel I/O Interface. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems* (April 1995), pp. 1–15.

[15] 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`.

[16] CORBETT, P. F., BAYLOR, S. J., AND FEITELSON, D. G. Overview of the Vesta Parallel File System. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems* (1993), pp. 1–16. Also published in Computer Architecture News 21(5), December 1993, pages 7–14.

[17] Corbett, P. F., and Feitelson, D. G. Design and Implementation of the Vesta Parallel File System. In *Proceedings of the Scalable High-Performance Computing Conference* (1994), pp. 63–70.

[18] 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).

[19] Crockett, T. W. File Concepts for Parallel I/O. In *Proceedings of Supercomputing '89* (1989), pp. 574–579.

[20] del Rosario, J. M., Bordawekar, R., and Choudhary, A. Improved Parallel I/O via a Two-Phase Run-time Access Strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems* (1993), pp. 56–70. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[21] 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).

[22] Feitelson, D. G., Corbett, P. F., and Prost, J.-P. Performance of the Vesta Parallel File System. In *Proceedings of the Ninth International Parallel Processing Symposium* (April 1995), pp. 150–158.

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

[24] Forum, M. P. I. MPI: A Message-Passing Interface Standard, 1995. Available at`http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html`.

[25] 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.

[26] 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.

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

[28] 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.

[29] HIGH PERFORMANCE COMPUTATIONAL CHEMISTRY GROUP, PACIFIC NORTHWEST NATIONAL LABORATORY. *NWChem, A Computational Chemistry Package for Parallel Compu ters, Version 1.1.* Richland, Washington, 99352, USA, 1995.

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

[31] 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.

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

[33] JOSEPH, D., AND GRUNWALD, D. Prefetching using Markov Predictors. In *Proceedings of the International Symposium on Computer Architecture* (June 1997), pp. 252–263.

[34] KASABOV, N. K. *Foundations of Neural Networks, Fuzzy Systems, and Knowledge Engineering.* The MIT Press, 1996.

[35] KIMBREL, T., TOMKINS, A., PATTERSON, R. H., BERSHAD, B., CAO, P., FELTEN, E. W., GIBSON, G. A., KARLIN, A. R., AND LI, K. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *Proceedings of the USENIX Association Second Symposium on Operating Systems Design and Implementation* (1996), pp. 19–34.

[36] KLEINROCK, L. *Queueing Systems, Vol. 1, Theory.* John Wiley, 1975.

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

[38] KOTZ, D. *Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors.* PhD thesis, Duke University, April 1991. Available as technical report CS-1991-016.

[39] Kotz, D. Multiprocessor File System Interfaces. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems* (1993), pp. 194–201.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] Kotz, D., and Nieuwejaar, N. Dynamic File-Access Characteristics of a Production Parallel Scientific Workload. In *Proceedings of Supercomputing '94* (November 1994), pp. 640–649.

[45] Krieger, O. *HFS: A flexible file system for shared-memory multiprocessors.* PhD thesis, University of Toronto, October 1994.

[46] 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.

[47] Krieger, O., and Stumm, M. HFS: A Performance-Oriented Flexible File System Based on Building-Block Compositions. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems* (Philadelphia, May 1996), pp. 95–108.

[48] 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.

[49] KUENNING, G. H. The Design of the SEER Predictive Caching System. In *Proceedings of Workshop on Mobile Computing Systems and Applications* (Dec. 1994), pp. 37–43.

[50] 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.

[51] 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.

[52] 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`.

[53] LI, P., CURKENDALL, D., DUQUETTE, W., AND HENRY, H. Remote Interactive Visualization and Analysis (RIVA) Using Parallel Supercomputers. In *Proceedings of the 1995 Parallel Rendering Symposium* (October 1995).

[54] MICHAEL BERRY, E. A. The Perfect Club Benchmarks: Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications 3*, 3 (1989), 5–40. Fall.

[55] MILLER, E. L., AND KATZ, R. H. Input/Output Behavior of Supercomputer Applications. In *Proceedings of Supercomputing '91* (November 1991), pp. 567–576.

[56] MOWRY, T. C., DEMKE, A. K., AND KRIEGER, O. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation* (October 1996), USENIX Association, pp. 3–17.

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

[58] NEEDLEMAN, S. B., AND WUNSCH, C. D. An Efficient Method Applicable to the Search for Similarities in the Amino Acid Sequences of Two Proteins. *Journal of Molecular Biology 48* (1970), 444–453.

[59] NIEUWEJAAR, N., AND KOTZ, D. The Galley Parallel File System. In *Proceedings of the 10th ACM International Conference on Supercomputing* (May 1996). To appear.

[60] Nieuwejaar, N., Kotz, D., Purakayastha, A., Ellis, C. S., and Best, M. File-Access Characteristics of Parallel Scientific Workloads. Tech. Rep. PCS-TR95-263, Dept. of Computer Science, Dartmouth College, August 1995. To appear in IEEE TPDS.

[61] Ousterhout, J., and DaCosta, H. A Trace -Driven Analysis of the Unix 4.2BSD File System. *ACM Operating Systems Review 19*, 5 (December 1985), 15–24.

[62] 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.

[63] Park, Y., Scott, R., and Sechrest, S. Virtual Memory Versus File Interfaces for Large, Memory-intensive Scientific Applications. In *Proceedings of Supercomputing '96* (November 1996), ACM Press and IEEE Computer Society Press. Also available as UH Department of Computer Science Research Report UH-CH-96-7.

[64] Pasquale, B. K., and Polyzos, G. C. A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload. In *Proceedings of Supercomputing '93* (1993), pp. 388–397.

[65] Pasquale, B. K., and Polyzos, G. C. Dynamic I/O Characterization of I/O Intensive Scientific Applications. In *Proceedings of Supercomputing '94* (November 1994), pp. 660–669.

[66] Patterson, and Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers, Inc., 1994.

[67] Patterson, and Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1996.

[68] 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.

[69] Poole, J. T. Preliminary Survey of I/O Intensive Applications. Tech. Rep. CCSF-38, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[70] Prost, J.-P., Snir, M., Corbett, P., and Feitelson, D. MPI-IO, A Message-Passing Interface for Concurrent I/O. Tech. Rep. RC 19712 (87394), IBM T.J. Watson Research Center, August 1994.

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

[72] Reddy, A. L. N., and Banerjee, P. A Study of I/O Behavior of Perfect Benchmarks on a Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (1990), pp. 312–321.

[73] Reed, D. A. Scalable Performance Environments for Parallel Systems. In *Proceedings of the Sixth Distributed Memory Computing Conference, to appear* (Apr. 1991).

[74] Reed, D. A., Aydt, R. A., Noe, R. J., Roth, P. C., Shields, K. A., Schwartz, B. W., and Tavera, L. F. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of the Scalable Parallel Libraries Conference*, A. Skjellum, Ed. IEEE Computer Society, 1993, pp. 104–113.

[75] Rew, R. K. *netCDF User's Guide, Version 1.0.* Unidata Program Center, University Corporation for Atmospheric Research, Apr. 1989.

[76] Ruemmler, C., and Wilkes, J. Modelling Disks. Tech. Rep. HPL-93-68, Hewlett Packard Laboratories, July 1993.

[77] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning Internal Representations by Error Propagation. In *Parallel Distributed Processing*, D. E. Rumelhart and J. L. McClelland, Eds., vol. 1. MIT Press, Cambridge, MA, 1986.

[78] Rumelhart, D. E., and McClelland, J. *Parallel Distributed Processing.* MIT Press, 1986.

[79] Seamons, K. E., Chen, Y., Jones, P., Jozwiak, J., and Winslett, M. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing '95* (December 1995).

[80] Simitci, H., and Reed, D. A. A Comparison of Logical and Physical Parallel I/O Patterns. In *submitted for publication* (June 1997).

[81] 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.

[82] Smirni, E., and Reed, D. A. Workload Characterization of Input/Output Intensive Parallel Applications. In *Modelling Techniques and Tools for Computer Performance Evaluation* (June 1997).

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

[84] Song, I., and Cho, Y. Page Prefetching Based on Fault History. In *USENIX MACH III Symposium* (Santa Fe, NM, 1993), pp. 203–213.

[85] Steere, D. C. Using Dynamic Sets to Speed Search in World Wide Information Systems. Tech. Rep. CMU-CS-95-174, School of Computer Science, Carnegie Mellon University, March 1995.

[86] Thakur, R. *Runtime Support for In-Core and Out-of-Core Data-Parallel Programs*. PhD thesis, Department of Electrical and Computer Engineering, Syracuse University, May 1995.

[87] Thakur, R., Bordawekar, R., Choudhary, A., Ponnusamy, R., and Singh, T. PASSION Runtime Library for Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference* (October 1994), pp. 119–128.

[88] Thakur, R., and Choudhary, A. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. Tech. Rep. CACR-103, Scalable I/O Initiative, Center for Advanced Computing Research, Caltech, June 1995. Revised November 1995.

[89] Thakur, R., Gropp, W., and Lusk, E. An Experimental Evaluation of the Parallel I/O Systems of the IBM SP and Intel Paragon Using a Production Application. Tech. Rep. MCS-P569–0296, Argonne National Laboratory, February 1996.

[90] Thakur, R., Lusk, E., and Gropp, W. I/O Characterization of a Portable Astrophysics Application on the IBM SP and Intel Paragon. Tech. Rep. MCS-P534-0895, Argonne National Laboratory, August 1995. Revised October 1995.

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

[92] 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).

[93] Winstead, C., and McKoy, V. Studies of Electron-Molecule Collisions on Massively Parallel Computers. In *Modern Electronic Structure Theory*, D. R. Yarkony, Ed., vol. 2. World Scientific, 1994.

[94] 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.

[95] Xu, S. Personal communication. NCSA, June 1995.

[96] Zadeh, L. A. Fuzzy Sets. *Information and Control 8*, 3 (June 1965), 338–353.

Vita

Tara Madhyastha was born and raised in New Jersey (last exit North on the Garden State Parkway). She received a Rutgers Presidential Scholarship to attend Rutgers University, where she majored in computer science and English.

In 1990 she graduated with Henry Rutgers Honors and left the vast suburbs and shopping malls of her home state to discover culture shock at the University of Illinois at Urbana-Champaign. She completed her M.S. thesis on data sonification under the direction of Professor Daniel Reed in May 1992.

In summer 1993, she visited NASA Goddard Space Flight Center to characterize the input/output behavior of satellite data processing codes, which was the early inspiration for her Ph.D. research. Her work in parallel input/output was supported by a NASA/JPL Graduate Student Research Program fellowship from 1995-1997.

Tara earned her Ph.D. in computer science from the University of Illinois in 1997. She is currently supported by a NSF Postdoctoral Associateship at Carnegie Mellon University.

1 MADHYASTHA, T. M., ELFORD, C. L., AND REED, D. A. Optimizing Input/Output Using Adaptive File System Policies. In *Fifth NASA Goddard Conference on Mass Storage Systems and Technologies* (September 1996).

2 MADHYASTHA, T. M., AND REED, D. A. A Framework for Sonification Design. In *Auditory Display: Sonification, Audification and Auditory Interfaces* (1992), Addison-Wesley.

3 MADHYASTHA, T. M., AND REED, D. A. Data Sonification: Do You See What I Hear? *IEEE Software* (March 1995).

4 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.

5 MADHYASTHA, T. M., AND REED, D. A. Exploiting Global Access Pattern Classification. In *Proceedings of SC'97* (November 1997).

6 MADHYASTHA, T. M., AND REED, D. A. Input/Output Access Pattern Classification Using Hidden Markov Models. In *Workshop on Input/Output in Parallel and Distributed Systems* (November 1997).

7 REED, D. A., ELFORD, C. L., MADHYASTHA, T., SCULLIN, W. H., AYDT, R. A., AND SMIRNI, E. I/O, Performance Analysis, and Performance Data Immersion. In *Proceedings of MASCOTS '96* (Feb. 1996), pp. 1–12.

8 REED, D. A., ELFORD, C. L., MADHYASTHA, T., SMIRNI, E., AND LAMM, S. L. The Next Frontier: Interactive and Closed Loop Performance Steering. In *Proceedings of the 1996 International Conference on Parallel Processing Workshop* (August 1996), pp. 20–31.

9 REED, D. A., OLSON, R. D., AYDT, R. A., MADHYASTHA, T. M., BIRKETT, T., JENSEN, D. W., NAZIEF, B. A. A., AND TOTTY, B. K. Scalable Performance Environments for Parallel Systems. In *Proceedings of the Sixth Distributed Memory Computing Conference* (1991), IEEE Computer Society Press, pp. 562 – 569.