Optimizing Input/Output Using Adaptive File System Policies*

Tara M. Madhyastha, Christopher L. Elford, Daniel A. Reed
Department of Computer Science
University of Illinois
Urbana, Illinois 61801
e-mail: tara@cs.ui.edu

Abstract

Parallel input/output characterization studies and experiments with flexible resource management algorithms indicate that adaptivity is crucial to file system performance. In this paper we propose an automatic technique for selecting and refining file system policies based on application access patterns and execution environment. An automatic classification framework allows the file system to select appropriate caching and prefetching policies, while performance sensors provide feedback used to tune policy parameters for the specific system environment. To illustrate the potential performance improvements possible using adaptive file system policies, we present results from experiments involving classification-based and performance-based steering.

1. Introduction

Input/output performance is the primary performance bottleneck of an important class of scientific applications (e.g., global climate modeling and satellite image processing). Moreover, input/output characterization studies such as Crandall [1] and Smirni [2] have revealed that parallel applications often have complex, irregular input/output access patterns for which existing file systems are not well optimized. Experience has shown that a few static file system policies are unlikely to bridge the growing gap between input/output and computation performance. In this paper we propose an automatic technique for selecting and refining file system policies based on application access patterns and execution environment. Knowledge of the input/output access pattern allows the file system to select appropriate caching and prefetching policies while the specific execution environment determines what policy refinements are necessary to further improve performance. For example, a sequential access pattern might benefit from sequential prefetching. The available memory and access latencies determine the quantity of data that should be prefetched. By being responsive to both application demands and system environment, this approach can provide better performance than a single static file system policy.

Adaptive file system policy controls rely on continuously monitoring access patterns and file system performance. We obtain a qualitative access pattern classification either through automatic analysis of the input/output request stream or via user-supplied hints. We also

* Supported in part by the National Science Foundation under grant NSF ASC 92-12369, by the National Aeronautics and Space Administration under NASA Contracts NGT-51023, NAG-1-613, and USRA 5555-22 and by the Advanced Research Projects Agency under ARPA contracts DAVT63-91-C-0029, DABT63-93-C-0040 and DABT63-94-C-0049.

continuously monitor file system performance sensors (e.g., cache hit ratios, access latencies, and request queue lengths). The values of these sensors, together with the access pattern, are used to select and tune specific file system policies. For example, the file system can enable prefetching when the access pattern is sequential, using the interaccess delays determine how much data to prefetch. Updated performance sensor values or changing access pattern classification may result in additional refinements to file system policies.

The remainder of this paper is organized as follows. In ß2 we give a high-level overview of the adaptive file system infrastructure. Validation of these concepts requires an experimental framework; we have implemented adaptive file system policies within a portable, user-level file system called the Portable Parallel File System (PPFS) Huber [3], described in ß3. Our system has two major components; in ß4 we discuss how one automatically classifies user access patterns and uses this information to select file system policies. In ß5 we describe how to use an input/output performance summary generated from sensor values to select file system policies and parameters that should be modified to improve performance. Finally, ß6-ß7 place this work in context, summarize our results, and outline directions for future research.

2. Adaptive Steering

Given the natural variation in input/output access patterns, it is unlikely that one, static, system-wide set of file system policies will suffice to provide good performance for a reasonable range of applications. Even in a configurable environment, *a priori* identification of effective file system policies is difficult because application access patterns are sometimes data dependent or simply unknown. Furthermore, input/output requirements are a complex function of the interaction between system software and executing applications and may change unpredictably during program execution. We believe that integration of dynamic performance instrumentation and automatic access pattern classification with configurable, malleable resource management algorithms provides a solution to this performance optimization conundrum. Below, we describe the two major components of this approach.

2.1. Classification-Based Policy Selection

Parallel file system research such as Patterson [4], Kotz [5], Krieger [6], and Grimshaw [7] has demonstrated the importance of tuning file system policies (e.g., caching, prefetching, writeback) to application access patterns. For example, access pattern information can be used to guide prefetching, small input/output requests can be aggregated and large requests can be streamed.

One intuitive way to provide the file system with access pattern information is via user supplied hints, or qualitative access pattern descriptions, for each parallel file. Unfortunately, this approach requires ongoing programmer effort to reconcile the hints with code evolution. Inaccurate hints can cause performance problems if the file system selects policies that are unsuitable for the actual access pattern.

Our solution to this dilemma is to automatically classify access patterns during program execution. This approach requires no programmer intervention and is robust enough to handle dynamically changing or data-dependent access patterns. A classifier module observes the application-level access stream and generates qualitative descriptions. These descriptions, combined with quantitative input/output statistics, are used to select and tune file system policies according to a system-dependent algorithm. Hints can be used in conjunction with this approach to provide access pattern information that cannot be intuited from the access stream (e.g., collective input/output).

2.2. Performance-Based Policy Selection

Although application access pattern information is a prerequisite for selecting appropriate file system policies, input/output performance ultimately determines the success of a particular policy choice. Extrinsic (external) input/output phases that occur when other applications compete for shared resources are equally important to file system policy selection, yet are not evident from application access patterns alone. Using a basic feedback system as a model, we can frame parallel file system policy optimization as a dynamic steering problem that tracks performance to refine file system policy selection. This type of computational steering framework has proven useful in other contexts (e.g. Vetter [8], Wood [9], Gergeleit [10], and Gu [11].)

In our dynamic steering framework, we monitor performance sensors that encapsulate the performance of critical file system features, consult access pattern dependent policy selectors that map changes in input/output performance to potential policy changes, and invoke system actuators to effect these policy changes. The resulting performance sensor metrics reflect the influence of our policy reconfiguration. When coupled with automatic access pattern detection, this closed loop steering infrastructure can adapt file system policies to match application access patterns and then tune these policies to the dynamic availability of system resources.

3. Portable Parallel File System (PPFS)

PPFS is a portable input/output library designed as an extensible testbed for file system policies [3]. A rich interface for application control of data placement and file system policies makes it exceptionally well-suited for our experiments. Below we describe the PPFS design and extensions that facilitate adaptive file system policy experiments.

3.1. PPFS Design

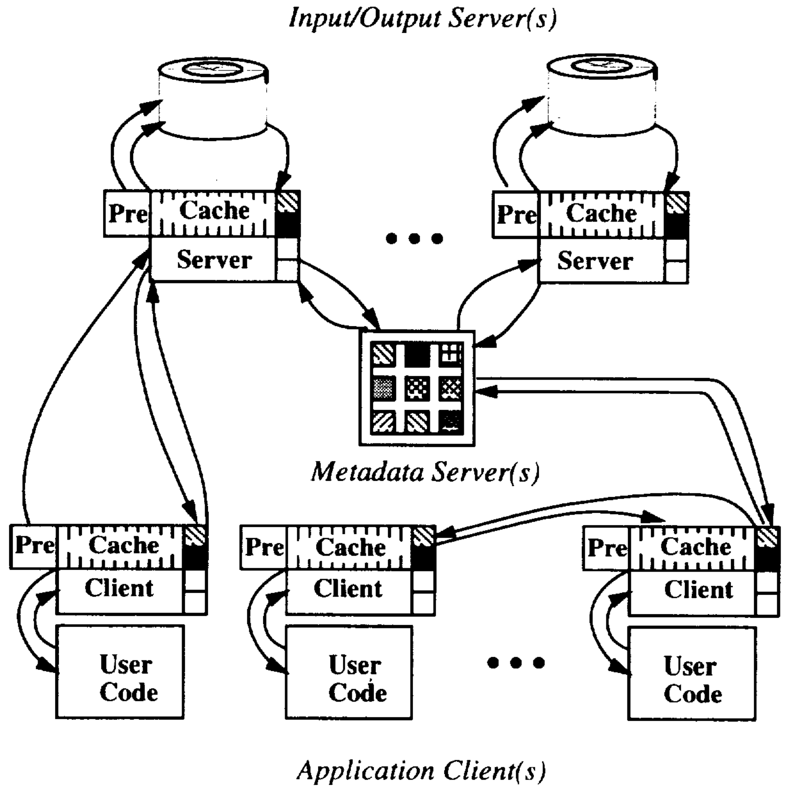


Figure 1: Basic PPFS Design

Figure 1 shows the PPFS components and their interactions. Application clients initiate input/output via invocation of PPFS interface functions. To open a file, the PPFS library first contacts the metadata server, which loads 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 policies for all levels of the system. Clients either satisfy the requests or forward them to servers (abstractions of input/output devices). Clients and servers each have their own caches and prefetch engines. All "physical" input/output is performed through underlying UNIX file systems on each PPFS server.

In the PPFS input/output model, files are accessed by 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.

3.2. 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 application access patterns, lessening the application programming burden and the likelihood of user misconfiguration. Dynamic performance data should be used to verify and refine these policy decisions. Through automatic access pattern classification, used to select file system policies, and performance-based policy refinement, we automate file system policy control. This has motivated two basic extensions to the base PPFS design: support for automatic access pattern classification and automatic policy refinement based on monitoring input/output performance.

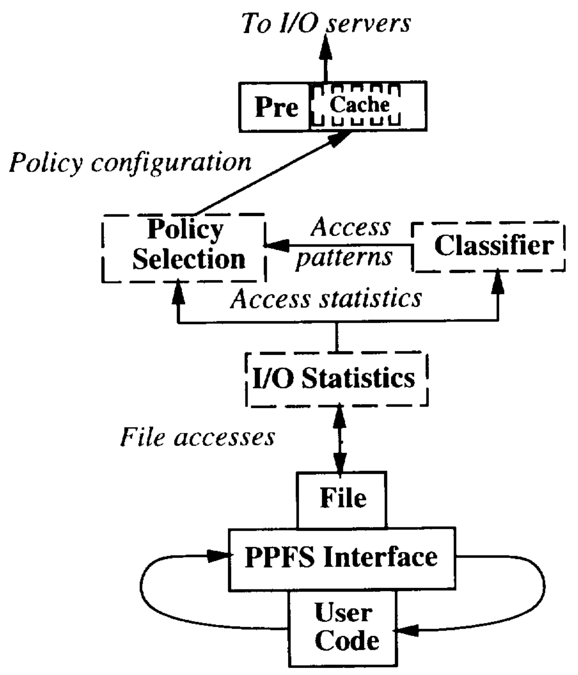


Figure 2: PPFS Classification and Policy Selection Extension

We have replaced manual 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 (each access is represented as a byte offset, read or write, and request size) and computes the statistics needed by the classifier module. PPFS uses the classification to select and tune prefetching and caching policies. Figure 2 illustrates the interaction of the classification extensions with the original PPFS components.

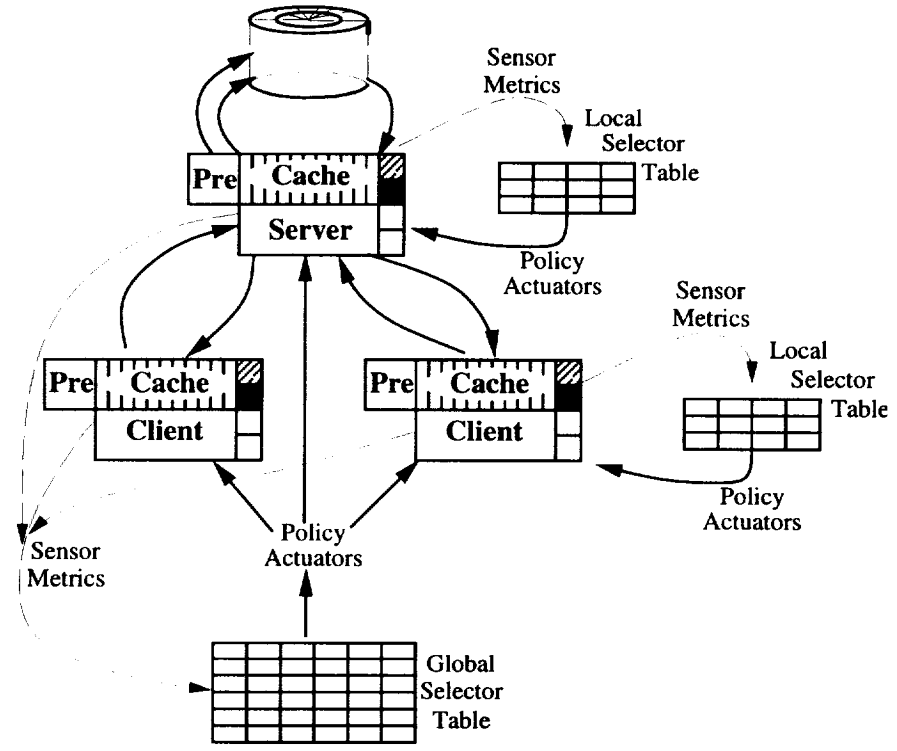


Figure 3: PPFS Performance Monitoring and Steering Extension

To refine policy selections using performance data, we instrumented the system components to periodically provide sensor metrics and created sensor-driven selector tables to automate invocation of the same native PPFS policy controls that a PPFS user could invoke manually. Figure 3 shows how our performance based policy selection extension interacts with the PPFS. Dynamically computed sensor metrics (e.g., input/output queue lengths, cache hit ratios, inter-request latencies) are routed to local and global policy selector tables, where they index appropriate file system policies and parameters for the system environment.

The local policy selector can only change local policies. For example, a client selector table may decide to increase the client file cache space and the number of records to prefetch ahead. It cannot change file system policies on other client nodes or on the PPFS servers. As shown in Figure 3, sensor metrics are also routed to a global selector mechanism that can select policy parameters for other nodes. For example, if the write throughput visible to client nodes for large writes drops below a certain threshold, the clients may elect to disable caching, and stream data directly to the PPFS servers. Rather than waiting for the individual server metrics and selector tables to disable server caching and stream data to disks, the global selector mechanism detects this input/output phase shift in the clients and invokes the policy change on the servers.

4. Automatic Classification and Policy Selection

As described in ß3, we have replaced the manual file system controls in PPFS with an adaptive access pattern classification and policy selection mechanism. Below we describe in greater detail our classification and policy control methodology.

A file access pattern classification is useful if it describes the input/output features that are most relevant to file system performance; it need not be perfectly accurate. For example, one might classify an input/output pattern as "sequential and write only" even if there are occasional small file seeks and reads – this would suffice to correctly choose a sequential prefetching policy. Such a qualitative description is difficult to obtain based on heuristics alone. Instead, one needs a general classification methodology capable of learning from examples.

As a first step toward adaptive file system policies, we have implemented automatic access classification to select file system policies, adapting to application requirements. This is only half of the complete system; after making policy selections we rely upon performance sensor data to refine policy parameters, adapting to the total system environment. Performance-based steering is the subject of ß5.

4.1. Classification Methodology

Within a parallel application, file input/output access patterns can be observed at two levels. 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 distributed 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 classifications are formed from local classifications and input/output statistics. In ß4.1.1 we describe our access pattern classification approach. In ß4.1.2 we illustrate how global classification works in a parallel application.

4.1.1. Access Pattern Classification

To accommodate a variety of underlying file structures and layouts, we describe access pattern classifications assuming a byte stream file representation. File accesses are made using UNIX style read, write, and seek operations, and file access patterns are determined from this representation. Thus, an input/output trace of file accesses may be represented as a stream of tuples of the form

$$\langle \textit{byte offset}, \textit{request size}, \textit{read} \,/\, \textit{write} \rangle$$

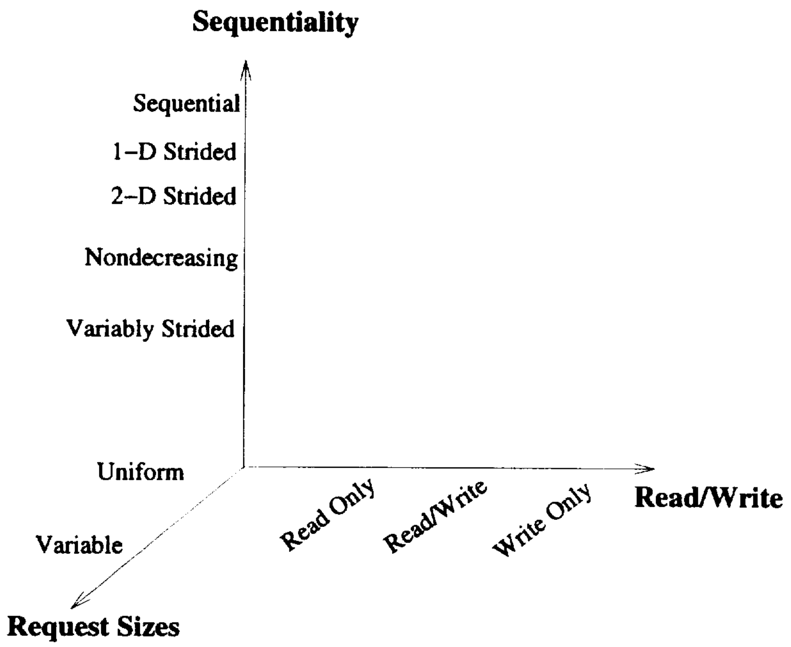


Figure 4: Access Pattern Space

Patterns observed in each of the time-varying values of the tuple components form a three dimensional access pattern space. Figure 4 shows certain categories along each axis that can be used to influence file system policy selection and label all points in the access space. Additional categories can be added as necessary to each axis to further refine the access pattern space.

Many techniques can be used to classify and identify observed access patterns within the space shown in Figure 4. Our approach is to train a feed-forward artificial neural network as in Hinton [12] to classify patterns. Although neural networks are expensive to train initially, once training is complete, classification is very efficient. To train the neural network, we represent the access pattern in a compact, normalized form by computing input/output statistics on a small fixed number of accesses, called the classification window. For example, representative statistics might be the number of unique read request sizes, or a transition matrix of the probabilities that one type of request (read/write) will follow the other.

Table 1: Input/Output Trace Features

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

Table 1 shows the features recognized by our trained neural network. These features correspond directly to planes or regions within the space shown in Figure 4. The neural network selects one and only one feature within each category; for example, a set of accesses cannot be both read only and write only. Neural networks are inherently imprecise, allowing us to train a network to identify patterns that are "close" to a well-defined pattern in a more general way than specifying heuristics. For example, a pattern might be treated as read-only if there is only one small write among very large reads, but read/write if the single write is the same size as the reads. This allows us to train the file system to classify new access patterns.

4.1.2. Global Access Pattern Classification

Local access pattern classification is only part of a larger classification problem. Local classifications are made per parallel program thread; however, the local access patterns within a parallel program merge during execution, creating a global access pattern. Global knowledge is especially important for tuning file system policies. For example, if all processors access a single 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.

Our global classification infrastructure is based on an access pattern algebra. We combine local classifications and other local information to make global classifications. For example, if all local access patterns are read only, the global access pattern is read only. The number of processors contributing to the global access pattern is called the cardinality of the classification. Generally, 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 these processors is still 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.
Global access pattern classification cannot be useful for influencing file system policies unless we recognize common global access patterns in time to effect policy changes. To demonstrate that this is feasible, we have examined parallel applications from the Scalable Input/Output (SIO) application suite [1,2]. These applications exhibit a variety of global access patterns, including global sequential, partitioned sequential (processors sequentially access disjoint partitions), and interleaved sequential (individual strided access patterns are globally interleaved). The patterns are primarily read-only or write-only with regular and irregular request sizes. All of these patterns can be recognized by our classification infrastructure.

One specific application area we have examined is computational fluid dynamics. PRISM is a parallel implementation of a 3-D numerical simulation of the Navier-Stokes equations from Henderson [13,14]. The parallelization is implemented by apportioning slides of 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.

Figure 5 shows a file access timeline for PRISM on a 64 processor Intel Paragon XP/S running OSF/1 version 1.4. 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; m16.rst is also accessed with an interleaved sequential access pattern. In the second input/output 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.

When accesses are adjacent and very small, local classification windows (the time to make ten input/output accesses) are short, and we must observe more windows to detect overlap among processors and global behavior. For example, Figure 5a and Figure 5b show local classification times for a globally sequentially accessed initialization file (m16.rea). The reads are very small (most are less than 50 bytes) and we reclassify the pattern every ten accesses. We can make a global sequential classification when sequential access patterns with overlapping bytes have been detected on every processor. Despite initial startup asynchronicity, the slowest processor (number 31) completes its tenth access to this file at 7.79 seconds. Because this initialization input/output phase accounts for approximately 125 seconds of execution time, adapting file system policies to the access pattern is fundamental to improving performance.

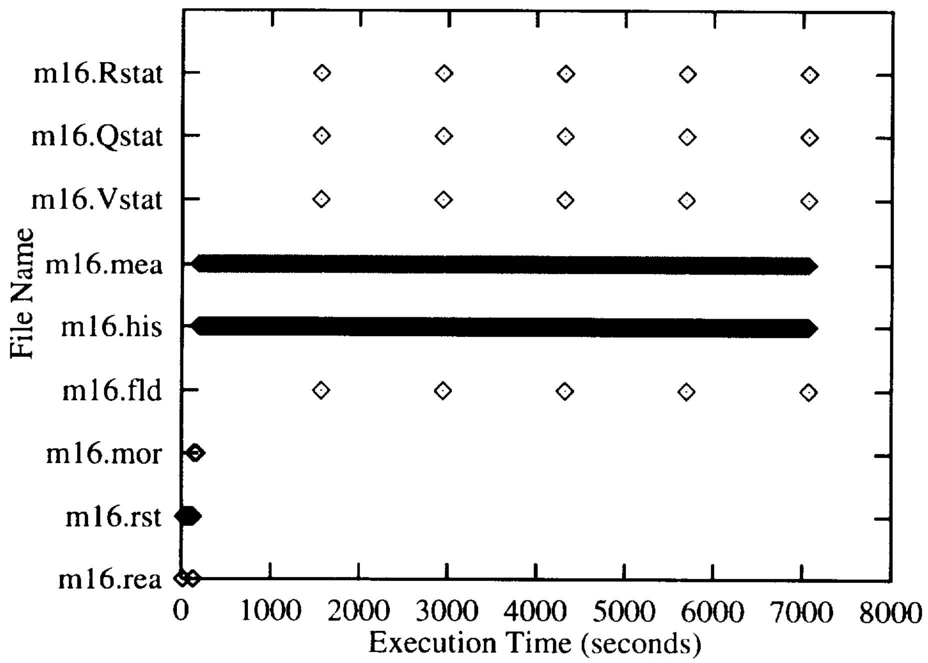


Figure 5: PRISM: File Access Timeline

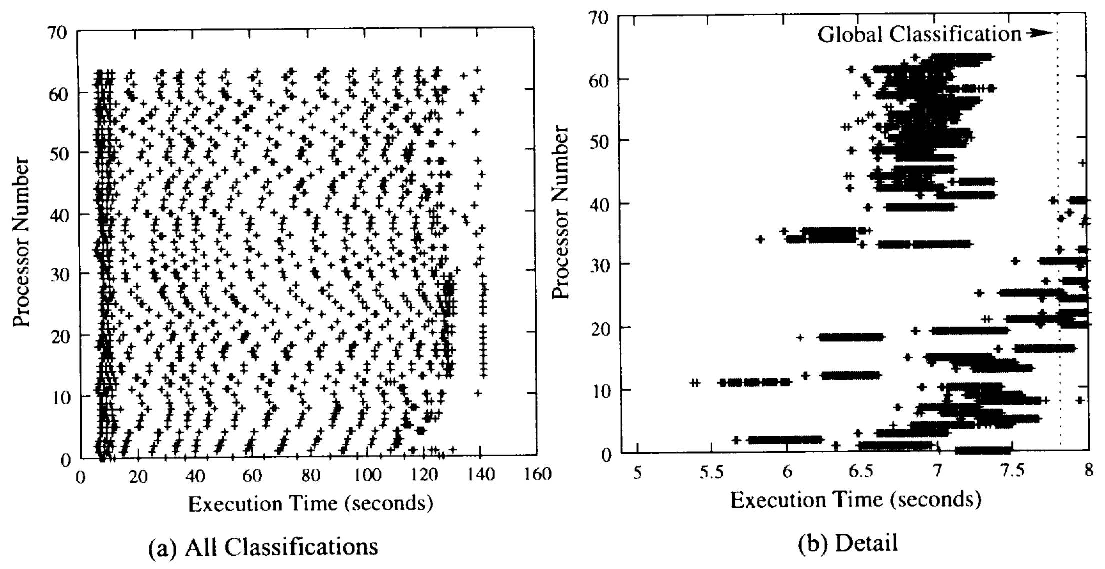


Figure 6: PRISM: Local Processor Classification Points for Global Sequential Access Pattern

4.2. Intelligent Policy Selection

A file access pattern classification as described above 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 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 should occur transparently, without application hints or user level optimizations.

Abstractly, PPFS continuously monitors and classifies the input/output request stream. This classification is passed to the file system policy suite for 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 adjust the cache size and prefetch anticipated blocks according to the access and stride sizes.

As described in ß4.1.2, we can combine local classifications to make global classifications, which we use to adjust policies at all system levels with global knowledge. For example, when all processors read the same file sequentially (global sequential) we can select a caching policy at input/output nodes that prefetches file blocks sequentially but does not flush cache blocks until every processor has accessed them. In contrast, if we detect an interleaved sequential global pattern, each input/output node could prefetch file blocks sequentially, retaining them only until each has been accessed in its entirety once.

Figure 7 shows a simple, parameterized example of a policy selection algorithm that selects PPFS policies for a uniprocessor UNIX workstation. Its default behavior is to favor small sequential reads, typical of UNIX workloads. However, when the classifier detects other access patterns, the algorithm adjusts policies to provide potential performance improvements. Quantitative values for the parameters of Figure 7 (e.g. LARGE_REQUEST) depend on the particular hardware configuration and must be determined experimentally.

The algorithm of Figure 7 is but one simple possibility for policy control. Richer control structures can be built upon more accurate models of input/output costs. However, in ß4.3 we show that even this simple policy suite suffices to yield large performance increases over that possible with standard UNIX file policies. In ß5 we describe our methodology for tuning automatically selected policies in response to overall system performance, closing the classification and performance feedback loop.

```
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 7: Dynamic File Policy Selection (Example)

4.3. Experimental Results

As a validation of automatic behavioral classification and dynamic adaptation, we used the enhanced PPFS to improve the input/output performance of Pathfinder, a single processor satellite data processing code. Pathfinder is from the NOAA/NASA Pathfinder AVHRR (Advanced Very High Resolution Radiometer) data processing project described in Agbu

[15]. Pathfinder processing is typical of low-level satellite data processing applications – fourteen large files of AVHRR orbital data are processed to produce a large output data set. It is an extremely input/output intensive application; over seventy percent of Pathfinder execution time is spent in UNIX input/output system calls.

4.3.1. Pathfinder

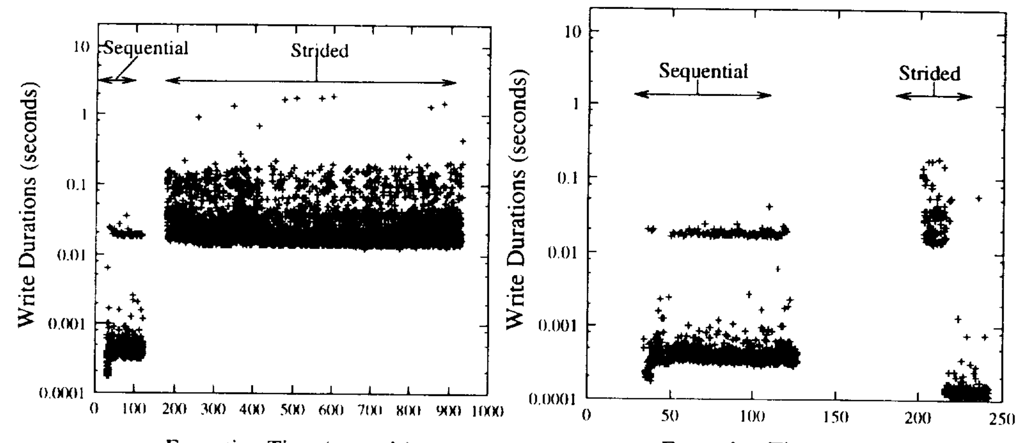
The goal of the Pathfinder project is to process existing data to create global, long-term time series remote-sensed data sets that can be used to study global climate change. There are four types of Pathfinder AVHRR Land data sets (daily, composite, climate, and browse images); we consider the creation of the 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 Hierarchical Data Format (HDF) from NCSA [16]. For simplicity, we examine the processing of a single orbital data file.

During Pathfinder execution, ancillary data files and the orbital data file are opened, and an orbit is processed 120 scans at a time. Although the orbit file is accessed sequentially, the access patterns for other ancillary data files range from sequential to irregularly strided. The result of this processing is written to a temporary output file using a combination of sequential and two-dimensionally strided accesses. Finally, the temporary file is re-written in HDF format to create three 8-bit and nine 16-bit layers.

Table 2 shows the relative execution times for Pathfinder using UNIX buffered input/output and PPFS with adaptive policies on a Sun SPARC 670. The dynamic adaptation of PPFS yields a speedup of approximately 1.87 with the policies Figure 7.[1] The PPFS automatic classifier could detect 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. Adapting to the first access pattern phase, PPFS selected an MRU cache block replacement policy. In the second phase it enlarged the cache, retaining the working set of blocks.

Figure 8a and Figure 8b illustrate the dramatic benefits of dynamic policy adaptation for Pathfinder's execution. Both graphs represent the same amount of input/output; however, in Figure 8a we use the same static policies for all access patterns. The first cluster of accesses in each graph 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. PPFS successfully retains the working set of blocks (the overall cache hit ratio exceeds 0.99), while UNIX buffered input/output forces a write of 8 KB for every one or two byte access.

[1] However, due to limited physical memory, we disabled caching for small, variably strided reads.



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

Figure 8: Pathfinder Write Durations (Beginning Phase)

Table 2: Pathfinder Execution Times (seconds)

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

5. Performance-Based Steering

Although file system policy selection is partially a function of application input/output access patterns, system performance ultimately determines the success of a particular policy choice. Performance sensors provide feedback on file system behavior that can be used to optimize the parameters of policy decisions.

Below, we describe a complement to qualitative access pattern classification: sensor based, closed loop policy selection and configuration. As described in ß2.2 and shown in Figure 3, our framework partitions the steering problem into three components. The sensor metrics in ß5.1 provide input for policy selectors of ß5.2 which, based on system and application performance history, select policy parameters and activate them via the policy actuators of ß5.3.

5.1. Performance Sensors

Table 3: PPFS Sensor Metrics

Dimension	Description
Operation Count	Total number of input/output requests

Operation Time	Mean operation service time
Read Count	Number of read requests
Read Byte Count	Number of bytes read
Read Time	Mean read service time
Write Count	Number of write requests
Write Time	Mean write service time
Cache Hits	Number of requests serviced by caches
Server Cache Hits	Number of requests serviced by offnode caches
Cache Check Time	Time to check local cache
Server Time	Time spend on input/output servers
Server Queue Time	Time spend in disk queue
Server Queue Lengths	Length of disk queue
Prefetch Byte Count	Number of bytes prefetched
Prefetch Cache Check Time	Time to scan cache on prefetch initiation
Prefetch Off Node Time	Time spent offnode for prefetch operations
Hit Miss Time	Time spent waiting for overlapped prefetch to complete

To capture input/output performance data, we augmented PPFS with a set of performance sensors that are periodically sampled using the Pablo instrumentation library of Reed [17]. Table 3 shows the current PPFS sensor metrics. We chose these particular metrics because they are inexpensive to calculate, and we believe they are broad enough to reflect the performance of malleable file system policies within PPFS. In practice, many metrics are strongly correlated with others, magnifying or validating trends detected via other metrics.

5.2. Policy Selectors

Table 4: Sample Sequential Access Selectors

Sensor Conditions	Policy Options
(poor_read_service_times) & (many_read_requests) & (managable_byte_throughput) & (NOT high_hit_ratio)	Increase Cache Size Increase Prefetch Amount
(NOT managable_byte_throughput) & (low_hit_ratio)	Decrease Cache Size Disable Prefetch

Given detailed performance sensor metrics and an access pattern classification, our framework tunes file system policies using the sensor metrics as the indices to a selector table containing policy parameters for that set of sensor metrics. The dashed lines of Figure 1 show the flow of sensor data from PPFS modules to the policy selectors. Table 4 shows some sample selectors that a system might provide, given a sequential access pattern classification. For example, if the sensor metrics indicate that relatively small read requests take a long time and the cache hit ratio is low, we might increase the cache size and the number of blocks prefetched to anticipate the request stream. If the sensors indicate that too much data is being requested to effectively cache and prefetch, we may disable caching and prefetching altogether to avoid thrashing the cache.

The sensor rules shown in Table 4 are qualitative rather than quantitative. We quantify the selector table rules when we calibrate them with the specific sensor metrics for a given platform. For example, on an IBM SP/2 with 128 MB of memory per node manageable_byte_thruput may calibrate to (Read_Byte_Count[2] < 100 MB/second). Similarly on an Intel Paragon with only 32 MB of memory per input/output node, the calibration may be (Read_Byte_Count < 25 MB/second).

To create selector tables for a given access pattern, we need to know how different file system policies perform for this access pattern. By executing access pattern benchmarks with a variety of policies and under a variety of load conditions, we can develop a set of selector rules such as those shown in Table 4. We calibrate the qualitative rules on a given platform by storing the quantitative performance sensors with the qualitative rules. Our portable, dynamic steering infrastructure can then adapt to a system's resource constraints by simply loading selector tables calibrated for that system.

5.3. Policy Actuators

After the policy selector mechanism determines what file system policy parameters should be used, actuators provide the mechanism to instantiate policies and configure parameters. Currently, PPFS supports actuators that allow dynamic reconfiguration of cache sizes, replacement policies, and prefetch and write behind parameters on each client and server node. These actuators provide a rich variety of controls to our dynamic steering infrastructure. We have tested these controls by interactively steering application behavior based on a virtual reality display of the sensor metrics as in Reed [18].

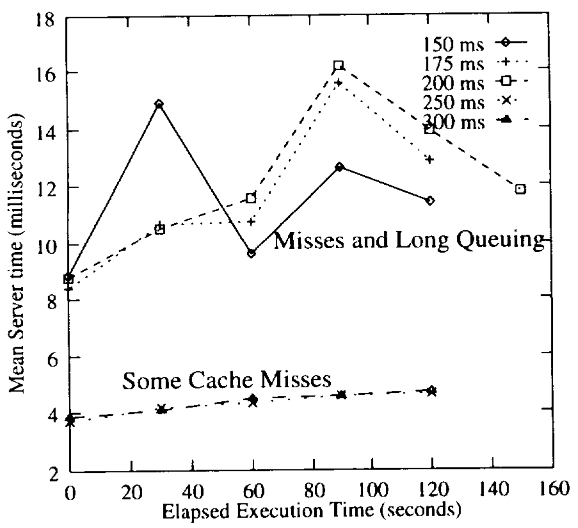
5.4. Experimental Results

To demonstrate the efficacy of sensor-based adaptive control when coupled with behavioral assertions, we used an input/output benchmark to conduct a set of simple experiments on several parallel architectures. We had several fundamental goals for the benchmark study. First, we wanted to verify that sensor metrics help us make improved PPFS policy decisions. We also wanted to determine how long we have to wait between policy changes to allow the sensor metrics to settle to their new steady state values.

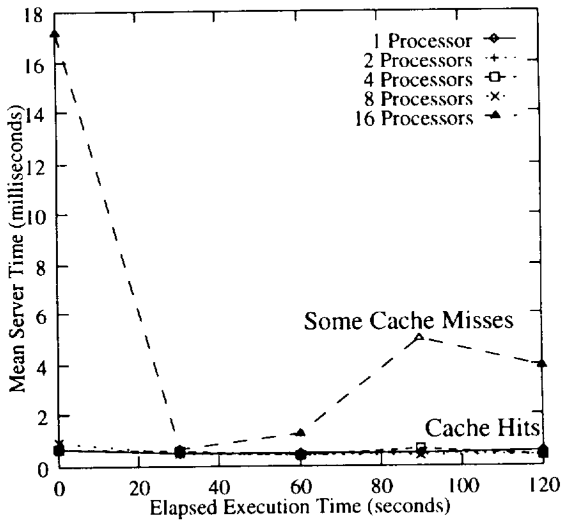
In our benchmark, a group of tasks reads disjoint interleaved portions of a shared file. Task i reads all blocks, i modulo the number of tasks (e.g., task 0 of p tasks reads file

[2] Note that Read_Byte_Count is a sensor metric from Table 3.

blocks 0, $2p$, p, ...) Between accesses, a processor computes for a uniform random interval with a parametric mean. We executed this benchmark on several parallel architectures with a variety of request sizes, prefetching options, and computation overheads for varying numbers of reader tasks.



a) Changing Interaccess Latency



b) Changing Processor Count

Figure 9: Sensor Variation for Different Workloads

Figure 9 shows the effect on server request overhead[3] of varying the inter-access computation interval and the number of nodes reading a file. This experiment was performed on an Intel Paragon XP/S using a single input/output server controlling a RAID-3 disk array with a throughput of 3.1 MB/second. In Figure 9a, eight processors read the file and the PPFS server prefetches only sixteen KB ahead of the access stream. In Figure 9b, on the other hand, the PPFS server prefetches 256 KB ahead and clients wait on average 175 milliseconds in between each access. The PPFS server performance depends on the number of requests arriving at the server each second. In Figure 9a, the arrival rate varies from 27 to 54 requests per second. Similarly, in Figure 9b, the request arrival rate varies from 6 to 92 requests per second.

The sensors values in Figure 9 fall into three basic categories. As shown in the top of Figure 9a, most of the requests could result in cache misses coupled with long queuing delays where the server time exceeds ten milliseconds. A substantial increase in the amount of prefetching is required to alleviate this problem. When some of the requests result in cache misses, we see that the server time is between four and six milliseconds.[4] A

[3] Server request overhead is the time that a request spends on the PPFS server node. It includes cache check time, buffer copy overhead, and disk queuing times if the request is not in the server cache.

[4] In Figure 9b, the startup transient lasts about sixty seconds before these cache misses occur regularly.

moderate increase in the number of blocks prefetched should result in improved performance. Finally, at the bottom of Figure 9b, we see that when all of the requests can be serviced from the cache, the mean time spent on the PPFS server is less than one millisecond.

Table 5: Benchmark Selector Rules

Sensor Conditions	Policy Options
Quantitative Rules	
(large_server_times) & (many_read_requests)	Substantially Increase Prefetch Amount
(moderate_server_times) & (many_read_requests)	Moderately Increase Prefetch Amount
Quantitative Calibration	
(MEAN_SERVER_TIME > 8 MS) & (READ_REQUEST_COUNT >40)	Substantially Increase Prefetch Amount
(READ_REQUEST_COUNT > 40 & (MEAN_SERVER_TIME > 2 MS) & (MEAN_SERVER_TIME < 8 MS)	Moderately Increase Prefetch Amount

Based on the figure, we can develop the two simple selector rules shown in Table 5 for this benchmark access pattern. One rule detects when the prefetch parameters should be increased considerably while the other detects when the prefetch parameters should be increased slightly. To calibrate these rules for the Intel Paragon with a single RAID-3 disk array, we simply augment the selector table with the appropriate sensor values as shown at the bottom of the Table 5. When the calibrated selector table is used for an application that exhibits this access pattern, the steering infrastructure can detect poor PPFS server performance and increase the prefetch parameters appropriately.[5]

6. Related Work

Current work in parallel file systems 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.

[5] The rules in Table 5 are examples of a subset of the needed rules for this benchmark. A complete set of rules could also reduce the amount of prefetching performed when the sensors indicate that resources were being wasted.

Characterization studies have revealed a large natural variation in input/output access patterns. During the past two years, our group and others have used the Pablo input/output analysis software to study the behavior of a wide variety of parallel applications on the Intel Paragon XP/S [1,2] and IBM SP/2. We have determined from these application studies that high performance 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 access, and a variety of temporal variations.

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 [5,6,7]. 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 approaches to application policy control; these can be grouped into systems that offer explicit policy control (e.g. SPIN from Bershad [19], exokernel from Engler [20], the Hurricane File System from Krieger [21], and Galley from Nieuwejaar [22]), and implicit policy control, via hints [4], expressive user interfaces (e.g., ELFS [7] and collective input/output as in del Rosario [23] and Kotz [24]), or intelligent modeling of file access (e.g., Fido from Palmer [25] and knowlege based caching from Korner [26]). Fido is an example of a predictive cache that prefetches by using an associative memory to recognize access patterns over time. Knowledge based caching has been proposed to enhance cache performance of remote file servers.

The second component of our research, dynamic performance based steering, has been used successfully in many contexts. A natural analog to explicit policy control is interactive steering, where the steering infrastructure extracts run time sensor information from an application, presents this information to the user who selects system or application policies, and actuates these policies to change application behavior. Falcon as in Gu [27] and SciChem from Parker [28] are two representative examples of this interactive approach.

In contrast to interactive steering environments, automatic steering environments do not require continuing user involvement. Instead, steering decisions are made automatically without user intervention. DIRECT [10], Falcon [29,30] and the Meta Toolkit [9] all provide automatic steering interfaces. DIRECT targets real time applications, a domain where the primary concern is validating that the system meets real-time constraints. This goal is different from run-time performance improvement, but the steering infrastructure is similar. Automated run-time steering is used in Falcon to select different mutual exclusion lock configurations based on the number of threads blocked on the lock [30]. The Meta Toolkit provides a framework for performing dynamic steering and provides special guards that help to maintain mutual exclusion of critical state variables [9] that may be changed during actuator execution. When an actuator is invoked, the appropriate guards are executed before the system module is modified.

7. Conclusions

The wide variety of irregular access patterns displayed by important input/output bound scientific applications suggests that optimizing application performance requires a judicious match of resource management policies to resource request patterns. Because the interactions between dynamic, irregular applications and system software change during application execution, we believe that the solution to this performance problem is adaptive file system policies that are controlled by user-level access patterns and by system-level performance metrics.

In this paper, we described a prototype of an adaptive file system and presented the results of experiments demonstrating the viability of this approach. This prototype, built upon on our PPFS user-level parallel file system. selects and configures file caching and prefetching policies using both qualitative classifications of access patterns and performance sensor data on file system responses.

In the coming months, we plan to more tightly couple automatic access pattern classification with performance steering. We are currently rounding out the prototype by extending PPFS to perform run time global access pattern classification and enhancing the performance-driven steering infrastructure.

References

[1] 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).

[2] 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).

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] VETTER, J., AND SCHWAN, K. Models for Computational Steering. *In Proceedings International Conference on Configurable Distributed Systems* Annapolis (May 1996).

[9] WOOD, M.D. *Fault-Tolerant Management of Distributed Applications Using the Reactive System Architecture.* PhD thesis, Cornell University, January 1992. Available as technical report TR91-1252.

[10] GERGELEIT, M., KAISER, J., AND STREICH, H. Direct: Towards a Distributed Object-Oriented Real-Time Control System. In *Workshop on Concurrent Object-based Systems* (Oct. 1994).

[11] GU, W., VETTER, J., AND SCHWAN, K. An Annotated Bibliography of Interactive Program Steering. Tech. Rep. GIT-CC-94-15, College of Computing, Georgia Institute of Technology, 1994.

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

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

[14] 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.

[15] AGBU, P.A., AND JAMES, M.E. *The NOAA/NASA Pathfinder AVHRR Land Data Set User's Manual.* Goddard Distributed Active Archive Center, NASA, Goddard Space Flight Center, Greenbelt, 1994.

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

[17] 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.

[18] REED, D.A., SHIELDS, K.A., TAVERA, L.F., SCULLIN, W.H., AND ELFORD, C.L. Virtual Reality and Parallel Systems Performance Analysis. *IEEE Computer* (Nov. 1995), 57-67.

[19] 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).

[20] 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).

[21] KRIEGER, O. *HFS: A Flexible File System for Shared-Memory Multiprocessors.* PhD thesis, University of Toronto, October 1994.

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

[23] 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.

[24] 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.

[25] 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.

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

[27] GU, W., EISENHAUER, G., KRAEMER, E., SCHWAN, K., STASKO, J., AND VETTER, J. Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs. Tech. Rep. GIT-CC-94-21, College of Computing, Georgia Institute of Technology, 1994.

[28] PARKER, S.G., AND JOHNSON, C.R. SciRun: A Scientific Programming Environment for Computational Steering. In *Proceedings of Supercomputing '95* (December 1995).

[29] GHEITH, A., MUKHERJEE, B., SILVA, D., AND SCHWAN, K. Ktk: Kernel Support for Configurable Objects and Invocations. Tech. Rep. GIT-CC-94-11, College of Computing, Georgia Institute of Technology, Feb. 1994.

[30] MUKHERJEE, B., AND SCHWAN, K. Improving Performance by Use of Adaptive Objects: Experimentation with a Configurable Multiprocessor Thread Package. Tech. Rep. GIT-CC-93-17, College of Computing, Georgia Institute of Technology, Feb. 1993.