ELFS: Object-Oriented Extensible File Systems

Andrew S. Grimshaw
Edmond C. Loyot, Jr.

Computer Science Report No. TR-91-14
July 8, 1991

ELFS: Object-Oriented Extensible File Systems

Abstract

High performance scientific data analysis is plagued by chronically inadequate I/O performance. The situation is aggravated by ever improving processor performance. For high performance multicomputers, such as the Touchstone Delta that possess in excess of 500, 60 megaflops, processor I/O will be the bottleneck for many scientific applications.

This report describes ELFS (an *E*xtensib*L*e *F*ile *S*ystem). ELFS attacks the problems of 1) providing high bandwidth and low latency I/O to applications programs on high performance architectures, 2) reducing the cognitive burden faced by applications programmers when they attempt to optimize their I/O operations to fit existing file system models, and 3) seamlessly managing the proliferation of data formats and architectural differences. The ELFS solution consists of language and run-time system support that permits the specification of a hierarchy of file classes.

ELFS: Object-Oriented Extensible File Systems

1. Introduction

Contemporary high performance computer systems are becoming increasingly unbalanced. CPU speeds have increased dramatically over the last decade. At the same time I/O performance has improved only marginally. Thus, the performance of many scientific applications is bounded by the performance of their database[1]. They cannot get their data in and out of the machine fast enough. Evidence of this problem abounds. NRAO (the National Radio Astronomy Observatory), for example, has many database bound applications. One, a deconvolution algorithm, consumes 20 minutes of Cray CPU time, yet takes over 10 hours of wall clock time. The difference is due to database waits [20].

The advent of highly parallel architectures has made the problem even worse. For example, the Intel 128 node iPSC/860 has a peak performance of 7680 double precision mega-flops, and the recently released Delta has a peak rate almost four times that of the iPSC/860. Yet IO latency is still in the 10-20 millisecond range, and the aggregate bandwidth is only on the order of 3 MB/second for a 4 node IO system. What this means for scientific programmers is that their applications will be more I/O bound than ever before on the new machines; that they will not be able to read, modify, and write their scientific database fast enough unless new and better ways to manage scientific databases are found. Thus, they will be unable to fully exploit these new architectures to solve ever larger problems.

ELFS (an ExtensibLe File System) attacks the problems of 1) providing high bandwidth and low latency I/O to applications programs on high performance architectures, 2) reducing the cognitive burden faced by applications programmers when they attempt to optimize their I/O operations to fit existing file system models, and 3) seamlessly managing the proliferation of data formats and architectural differences. The ELFS solution consists of language and run-time system support that permits the specification of a hierarchy of file classes. Domain specific I/O operations can be specified for each class, reducing the cognitive burden on applications programmers that use the classes, as well as providing class specific optimization information to the implementation of the class. Prefetching and caching strategies can be specified on a class by class basis, enhancing performance. ELFS file objects may be partitioned and *striped* across multiple physical devices in a data domain sensitive fashion, increasing bandwidth over that available by straight

1. The term database has a different meaning for many scientific applications than is usually used by computer scientists. Computer scientists are used to thinking of relational, network, or hierarchial databases and database packages, e.g., accounting databases. A database to a computational scientist may be a collection of matrices, temporal data, or a set of grid points and their associated field values. The databases are treated more like files than databases in the computer science sense.

striping. Instances of the file classes are accessed in an asynchronous, possibly pipelined, manner, further improving performance.

File systems have changed very little over the past twenty years. While other areas of computer design, such as CPU architectures, programming languages and compilers, have experienced radical change, contemporary file systems differ little from those of twenty years ago. A typical example of a contemporary file system is the UNIX™ file system. The UNIX file system treats files as named sequences of bytes. It supports operations to create, open, close and delete files. Synchronous (blocking) operations seek to a particular location and to read and write blocks of data are also provided. This model has lasted so long because it provides sufficient functionality and adequate performance for those applications that fit its underlying assumptions.

There are two reasons why contemporary file system models are no longer adequate. First, I/O performance is not keeping up with CPU performance. CPU speeds have increased over one hundred fold in the last ten years, yet I/O latencies have improved only by a factor of about four and I/O bandwidths by a factor of about ten. The result is that I/O is increasingly a bottleneck [1, 12]. Even on a single processor IBM Risc System 6000, in the approximately 22ms it takes to satisfy a RANDOM I/O request, 1.84 million floating point operations can be performed. The problem is particularly acute for the new high performance parallel architectures. Thus it is more important than ever that applications spend a minimum amount of time waiting for I/O.

The second reason involves the file abstractions themselves. The programmer uses I/O operations to move data from permanent storage (files) to internal data structures. If the file abstractions available to the programmer do not map well to the application's internal data abstractions, the programmer must remap the data. This adds to the programmer's cognitive burden. The extra effort spent remapping the data reduces the amount of effort that can be spent on the application itself.

The I/O performance problem is compounded by poor I/O interfaces. Contemporary file system interfaces provide no mechanism for indicating how a file will be accessed. Therefore, in order to make performance optimizations, the file system must make assumptions about how the file will be used. For example, to reduce I/O latency and increase effective bandwidth, some file systems attempt to prefetch data that will be needed in the future. They also attempt to cache data that will be needed again. Usually, the prefetch and caching strategies are based on the assumptions that files will be accessed sequentially and that file access patterns exhibit locality. If the application's file access pattern fits these assumptions, prefetching and caching will greatly improve file system performance. Otherwise, prefetching and caching will not provide any performance improvement, and may, in fact, reduce performance. This may result in large latencies and low effective bandwidths for the application. Because of this behavior, the programmer who

wants the best performance must make the application fit the file system assumptions, often wasting effort finding a sequential access method, improving locality or performing local caching.

As an example of this problem consider a common implementation of a 2D FFT on data that will not fit in memory. Typically the matrix is stored on disk by row or by column. Suppose that it is stored by row. The algorithm consists of three steps. In the first step a 1D FFT is performed on the matrix, reading and writing the matrix by rows. In the second step the matrix is *transposed on disk*, a very time consuming operation. The third step is to again perform a 1D FFT on the matrix by row. The transpose is necessary because the file system does not support efficient, type specific access methods, in this case, access by columns.

We believe the time has come to re-examine the file system interface. What is needed, and what ELFS will provide, is language and system support for a new set of file abstractions. These abstractions provide the following capabilities:

1) User specification of caching and prefetch strategies - This feature allows the user to exploit application domain knowledge about access patterns and locality to tailor the caching and prefetch strategies to the application.

2) Asynchronous I/O - ELFS permits the overlapping of I/O operations, including prefetching, with the application's computation.

3) Multiple outstanding I/O requests -ELFS allows the application to request data before it is actually needed. The application can then do some computation while the I/O is being processed. By the time the data is actually needed it is more likely to be available.

4) Data format heterogeneity - ELFS classes may be constructed so as to hide data format heterogeneity, automatically translating data as it is read and written. This can vastly simplify the applications code.

These new file abstractions are structured as a user-extensible class hierarchy with inheritance, thus providing the benefits of object-oriented programming to the application designer. The programmer can then extend basic file abstractions with type-specific operations. These operations can be used to specify access pattern information or other domain knowledge to the file system. A file interface that closely matches the application's internal data structures can be designed, thus reducing programming effort.

We are currently developing such a system at the University of Virginia. When complete, ELFS will consist of two parts: the definition of a file system class hierarchy, and compiler and run-time system services that support asynchronous objects and multiple outstanding requests without explicit programmer control. The system will be implemented using the parallel programming language MPL [5, 7], which is an extension of C++ [19].

The class hierarchy in our system contains the base class *unix_file* (see Figure 1).. This class supports the standard UNIX file operations, *creat, open, close, lseek*, *read*, and *write*. It

mimics the semantics of UNIX files, exhibits the same prefetching and caching features as UNIX files, and in fact is often just a call-through to the underlying file system. Our mechanism only becomes interesting when new classes are derived from *unix_file*.

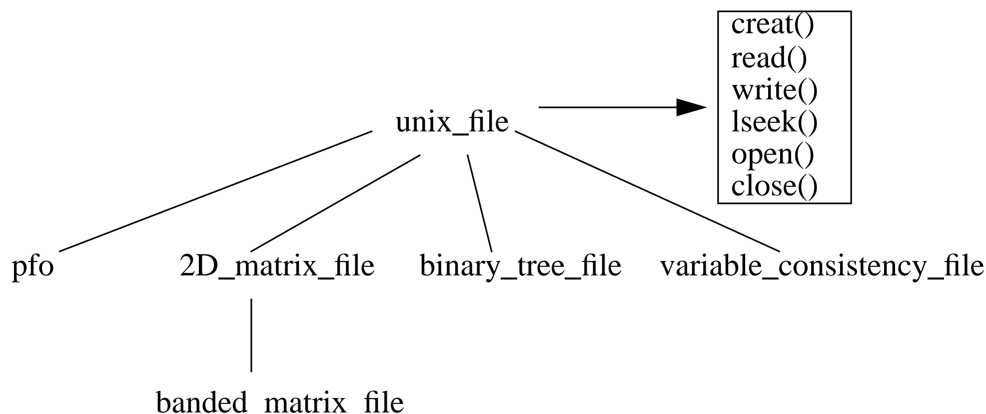


Figure 1. The ELFS Class Hierarchy.

For example, the class *2D_matrix_file* inherits all of the *unix_file* operations and provides additional member functions based on the semantics and structure of the underlying abstraction. For a two dimensional matrix these member functions allow the specification of the matrix dimensions (e.g., 100x100), the element size (e.g., 8 bytes for double precision) and the access method (e.g., by row, by column, or by block). Because the implementation of the class *knows* the underlying structure and is told (via member functions) the access pattern, it can prefetch and cache data more intelligently than the file system can without any such knowledge. Detailed information of a preliminary version of this class, including preliminary performance information, is presented later.

The remainder of this report is in four sections. In section 2 we discuss ELFS in more detail, including the presentation of a class hierarchy with several derived classes and their implementation rational. Section 3 describes the implementation environment. Section 4 compares ELFS to other approaches to the problem. Section 5 summarizes the report.

2. The ELFS Approach

This section describes our solution via the description of one possible class hierarchy. *Our objective is not to claim that these are the classes that one should choose, but rather to illustrate the power and flexibility of the approach*. We begin by briefly describing the object model that we are using, and then proceed with the class hierarchy, including descriptions of the interfaces and those implementation features germane to our claims.

2.1 Object Model

We use an object model that distinguishes between two "types" of objects, contained objects and independent objects.[2] Contained objects are objects contained in another object's address space. Instances of C++ classes, integers, structs and so on are contained objects. Independent objects possess a distinct address space, a system-wide unique name, and a thread of control. Independent objects are analogous to UNIX processes.

The disjoint address space component of our model is an artifact of our implementation environment (described later) and is not critical to the following discussion. The independent thread of control is important.

The independent thread of control is critical because we want our file objects to be able to *asynchronously* perform prefetching and caching operations. Further, in our model, operations on independent objects are carried out in an asynchronous, possibly pipelined, fashion. Any necessary synchronization is managed by the compiler operating in conjunction with the run-time system. Thus, I/O operations can be carried out on the user's behalf while the user is performing other operations. Below we assume that the instances of our file objects are in essence separate processes, and that they can perform I/O and other functions independently from the user's thread.

2.2 The Class Hierarchy

There are four subclasses currently defined in our class hierarchy, *pfo's*, *2D_matrix_files*, *binary_tree's* and *variable_consistence_files*. Each is designed to illustrate how the shortcomings of existing file system interfaces can be cleanly avoided. *Pfo's* attack the problem of bandwidth by distributing a file across multiple devices. *2D_matrix_files* provide matrix I/O operations that are both conceptually close to the model the programmer wants, and are equally efficient for row *and* column operations. *Binary_tree_files* illustrate how aggressive prefetching for non-sequential data structures based on object structure, in this case a tree, can be performed, providing a performance improvement over traditional file systems. Finally, *variable_consistency_files* can be used to avoid the consistency semantics of NFS files [9].

2.2.1 Parallel File Objects - pfo's

Parallel file objects attack both the bandwidth and latency problems on parallel and distributed systems. A detailed description, including performance, can be found in [6]. In addition to the *unix_file* operators, *pfo's* provide operators that allow the user to:

---

2. The distinction between independent and contained objects is not unusual, and is driven by efficiency considerations.

1) specify the structure of the file, e.g. 1D array, 2D array, 3D array. This has the effect of imposing a logical structure on the file.

2) partition (decompose) the pfo into sub-pfo's. The decomposition is into subsets of the specified logical structure of the file. The partition is a true partition, i.e., the subsets do not overlap. Sub-pfo's may also be partitioned.

3) read and write logical blocks of data in the name space of the partitions, e.g., the user may read a row, or a column, or a block.

4) the location (processor and attached I/O device) of each partition may be specified. Thus, one can place file subsets close to the processors where they will be needed.

5) specify the access pattern, e.g., by row, by column, reverse order, etc. This provides information that enables the *pfo* to optimize based on the access pattern.

These operators combine to support the exploitation of file structure in order to optimize performance, engender high bandwidth by distributing the file to multiple devices with placement determined by the file structure, and provide very low access latencies by performing file operations asynchronously.

Very high bandwidth is achieved by scattering pieces of the *pfo* to multiple devices, each of which may be accessed in parallel (see Figure 2). This idea has been studied extensively [2, 8, 9, 13, 14, 15, 16]. What distinguishes our work from others is that the decomposition may be specified using *pfo*'s, e.g., row-wise, column-wise, in blocks, etc. This is in contrast to systems such as CFS [14]. CFS stripes the file across the disks in sequential 4K chunks. This decomposition will not be appropriate for all applications. By placing the data based upon its structure and who is going to use it, higher performance is possible.

2.2.2 Matrix Files

The purpose of the *2D_matrix_files* class hierarchy is twofold, to provide I/O operations that match the applications programmer's conceptual model of the data, enabling him to use appropriate abstractions, and to provide higher effective bandwidth and lower overall latencies than would be possible using the standard file operations. The *matrix_class* class hierarchy is shown in Figure 3. The first of the objectives is met by providing member functions to read and write rows and columns. A partial interface for the *2D_matrix_file* class is shown in Figure 3. The read and write operations have the expected semantics. Note though that neither returns a status code upon completion. If the user needs to know the status of the most recent operation the *int_status()* member is called. By not returning status information with every operation we permit additional asynchrony between the *2D_matrix_file* object and its client.

High performance is realized by exploiting knowledge of the underlying structure, making use of user supplied access pattern hints, performing prefetching and caching functions asynchro-

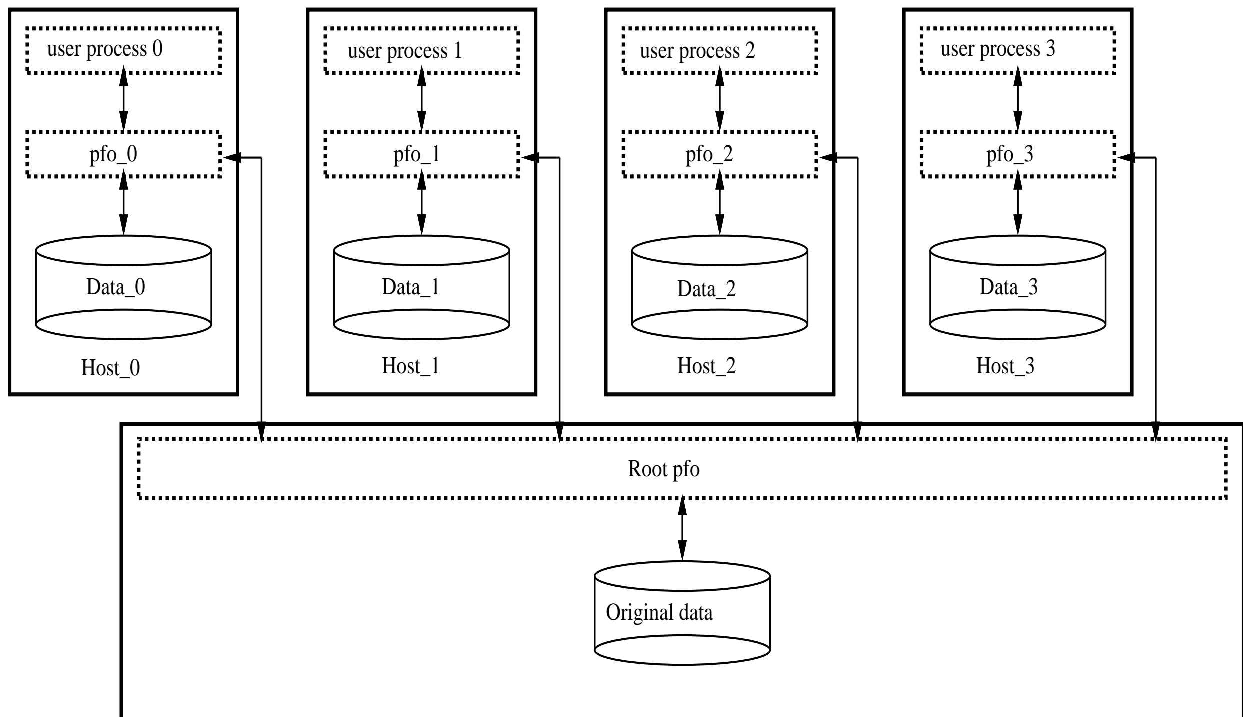


Figure 2. Bandwidth Enhancement via Parallel Access, Low Latency.

nously, and by pipelining I/O requests whenever possible.

To support both high performance row access and high performance column access we have used a “square partition” storage method (see Figure 4). Typically 2D matrices are stored either by row or by column. It is very inefficient to access a row of a column stored matrix or vice versa. When the matrix is stored by square blocks (Figure 4), row access and column access are equally efficient, and just as efficient as row access on a row stored matrix. The basic idea when reading by row is to fetch a row of square submatrices and then service the row requests from the buffer. While those read row requests are being satisfied, the *2D_matrix_file* object begins to pre-

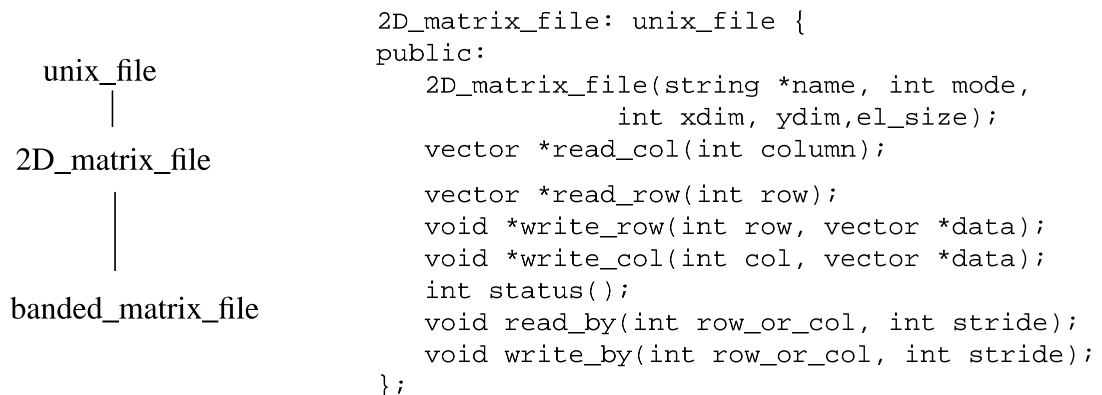


Figure 3. 2D_matrix_file Class Hierarchy and Interface.

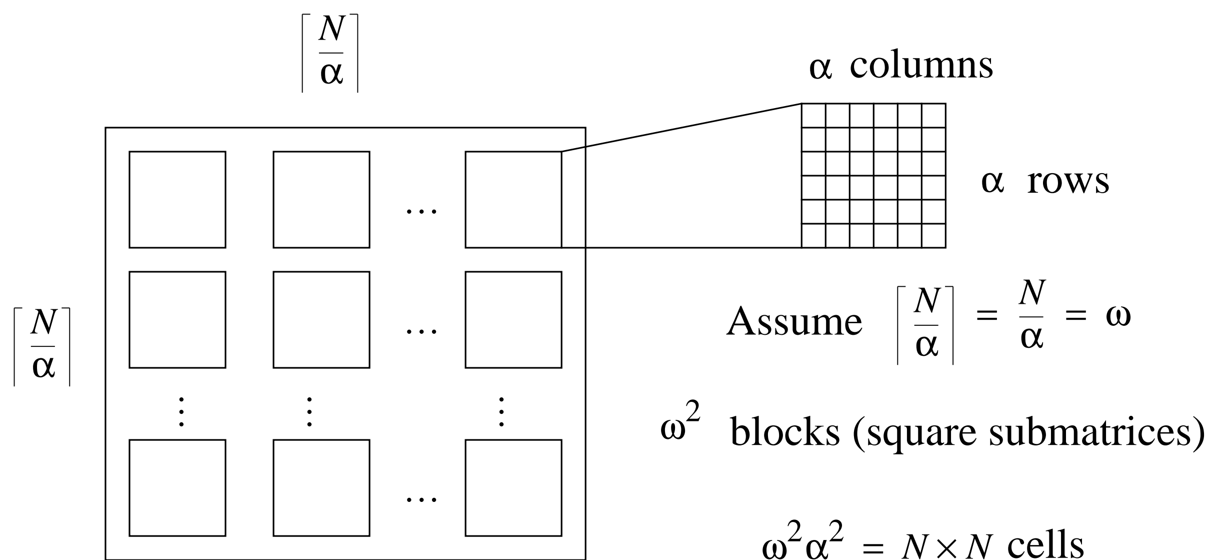


Figure 4. NxN matrix storage using “square partitions”.

fetch the next row of sub-matrices. Column requests are satisfied in a similar manner. For large files the overhead of “priming the pump” (reading the first row of the sub matrices) is amortized over all row accesses. Note that asynchronous prefetching can begin as soon as the user specifies by row or by column access. Further, the stride parameters can be used, and we need read only those rows/columns that will actually be used.

2.2.3 Preliminary Results

During the Spring of 1991 we implemented a preliminary sequential, synchronous version of the *2D_matrix* class in *C++*. Performance results are quite encouraging. Figure 5 below presents a performance comparison between straight *C* calls to read rows from a Unix file, *2D_matrix read_row* calls, and *2D_matrix read_column* calls on a Sparc station 2 with attached disk. The horizontal axis is the dimension of the matrix (of integers), and the vertical axis is the time required to read the entire matrix in seconds. As can be seen there is almost no performance penalty when using the *2D_matrix* class for reading by row. In the current implementation it costs just over twice as much to read by column than by row using the *2D_matrix* class, whereas it is essentially impossible to read a the Unix file by column. We believe that the 2:1 ratio is an artifact of the non-fragmentation of the test disk. Under normal fragmentation we expect the ratio to approach 1:1. The point though is that by using “intelligent” file objects very good performance is

possible for application domain specific file operations, e.g., *read_column*.

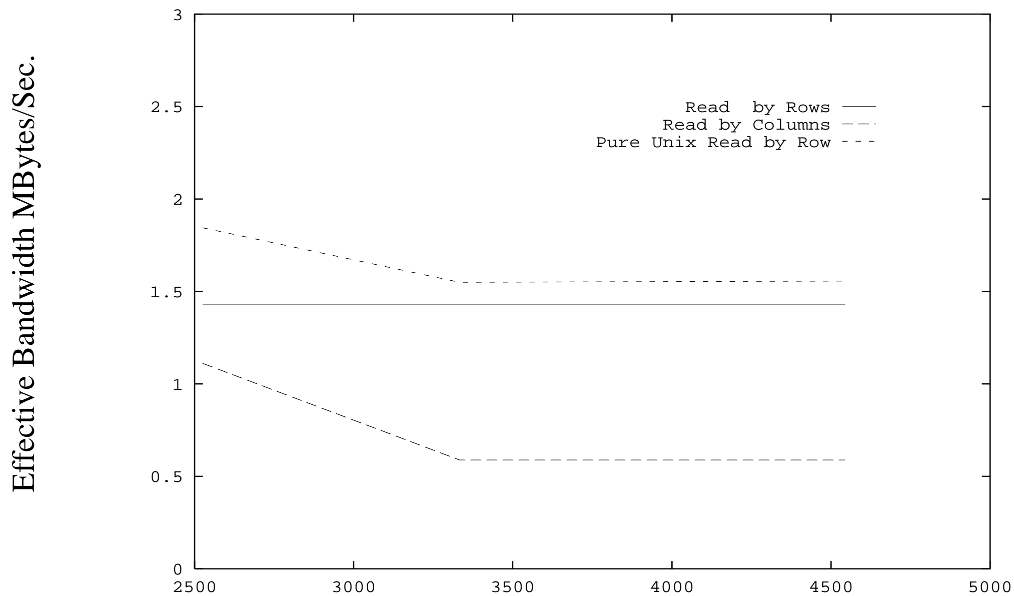
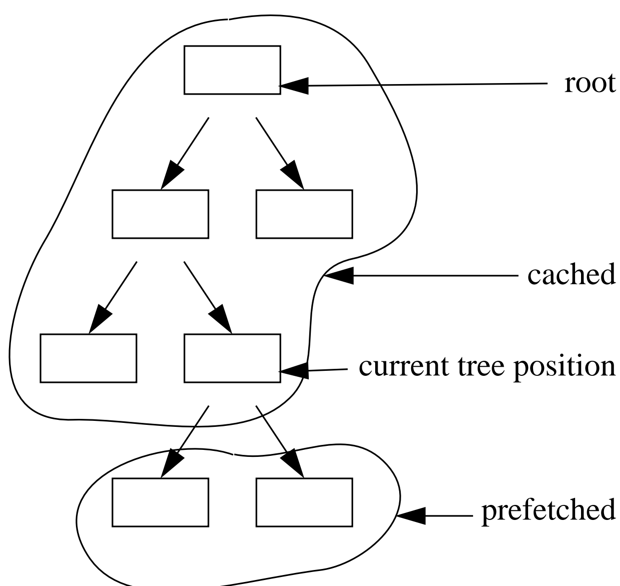


Figure 5. Preliminary Performance for Sequential 2D_matrix

2.2.4 Binary_tree_files

The *binary_tree_file* class uses prefetch and caching strategies based on the tree structure in Figure 6. The binary tree case described can be generalized to k-ary trees. The file maintains a




```
binary_tree_file : unix_file {
public:
   int open(string *name, int mode
            int data_size);
   tree_element *get_left_child();
   tree_element *get_right_child();
   void insert_left_child(
            tree_element *child);
   void insert_right_child(
            tree_element *child);
   tree_element *get_parent();
   void go_left();
   void go_right();
   void go_up();
   int status();
};
```

Figure 6. Binary_tree Caching and Interface.

current position in the tree, analogous to the current position of a standard file. The prefetching strategy is to aggressively begin prefetching the left and right children of a node whenever the

current_tree position is modified by a *go_left()*, *go_right()*, *get_left*(), *get_right*() or *go_up()*. Thus, when the next user request arrives, the requested node will have already been fetched, or the I/O operation will have at least begun.

The caching strategy is to cache all nodes from the *current_tree_position* to the *root*, including the current children and all of the children of the nodes on the path to the root. When the maximum cache size in *tree_element* nodes has been exceeded, nodes closest to the root are discarded as new nodes are added. Of course, the class could permit different caching strategies to be specified, e.g., assuming a breadth first cache instead of a depth first cache.

The prefetched cache strategies described above should provide better hit rates than the standard file system strategies of sequential prefetch and block based LRU. However, the point is not whether these are the best strategies, but rather that they are object specific strategies and can be changed on an application by application basis.

2.2.5 Variable_consistency_files

Variable_consistency_files are motivated by the desire to tune consistency on an application-by-application basis rather than for a whole system. Recall first the consistency semantics of UNIX files, where writes to a file by a process are *immediately* visible by other processes. Two distributed file systems, NFS [18] and Sprite [11], take different approaches to consistency [9]. NFS does not support UNIX semantics [9]. In order to improve performance, NFS caches data blocks at the client's machine, improving performance at the expense of consistency. The designers of NFS made this trade-off because they felt performance was paramount. Sprite on the other hand, preserves UNIX semantics no matter the cost. Sprite caches to improve performance until a server detects that a file is opened by more than one client, at least one of whom is writing. When this condition is detected, caching is disabled for the file, guaranteeing consistency and a performance penalty.

We can see that there is a trade-off between performance and consistency because of caching effects. The designers of the NFS and Sprite file systems have made the decision as to which of these two attributes will be sacrificed for the other. Unfortunately, all applications running on these systems must operate within the constraints imposed by their decisions. We feel that the application's designer is best able to make the decision on how and where to trade consistency for performance. Thus the application writers should be able to *specify* the consistency semantics that they need on a file by file basis. Further, since the consistency requirements of a file may vary during the course of an application's execution, the consistency should be variable.

Figure 7 below illustrates a possible interface for a variable consistency file that provides just such a capability. The class is derived from *unix_file*. Thus all of the *unix_file* operations are

```
class variable_consistency_file: unix_file {
public:
   set_consistency_window(int seconds = 0);
};
```

Figure 7. Interface for *variable_consistency_files*.

inherited. Their implementations, however, have been overloaded. The one new member function, *set_consistency_window*(), permits the user to vary the consistency of the file. A consistency value of 0 seconds implies that the UNIX consistency semantics are to be used for the file, i.e., all reads return the last value written. A value of 5 seconds means that the value returned by a read is no more than 5 seconds old. When more than one instance of a particular file is open, the smallest of the specified values is used.

When using a *variable_consistency_file* an application has complete control of the consistency semantics of the files used, and may vary the semantics over time to achieve the desired trade-off of performance and consistency.

3. Mentat Programming Environment

We have begun implementation of ELFS using the Mentat parallel programming system [4, 5, 7]. Mentat is an object-oriented, parallel computation system designed to provide easy-to-use parallelism for parallel and distributed systems. Mentat alleviates most of the burden of explicit parallelization that message passing systems typically place on the programmer. Further, Mentat programs block only if specific data dependencies require blocking, thereby greatly increasing the degree of parallelism attainable over that from RPC systems.

The Mentat Programming Language, an extension of C++, simplifies writing parallel programs by extending the encapsulation provided by objects to the encapsulation of parallelism. Users of Mentat objects are unaware of whether member functions are carried out sequentially or in parallel. In addition, member function invocation is asynchronous (non-blocking); the caller does not wait for the result. It is the responsibility of the compiler, in conjunction with the run-time system, to manage all aspects of communication and synchronization. The underlying assumption is that the programmer can make better granularity and partitioning decisions, while the compiler and run-time system can correctly manage communication and synchronization. By splitting the responsibility between the compiler and the programmer we exploit the strengths of each, and avoid their weaknesses.

Mentat has been implemented on three architectures that span the MIMD spectrum, a network of Sun workstations (loosely coupled), the Intel iPSC/2 (tightly coupled), and the BBN Butterfly (shared memory). Mentat programs are source compatible between supported architectures.

There are three important aspects of Mentat with regard to ELFS. The first is that instances of Mentat classes, called Mentat objects, possess an independent thread of control. Thus, ELFS file objects, when implemented as Mentat objects, operate asynchronously from their clients, performing prefetching and suffering the I/O wait, while the client performs other operations. Second, a client may have multiple outstanding requests to a Mentat object. Thus, several I/O requests may be pipelined, and I/O requests to different file objects may be performed in parallel. Third, the management of communications and synchronization between clients and Mentat objects is transparently managed for the programmer by the MPL compiler and the Mentat run-time system.

In Mentat, when a Mentat object member function is encountered, the arguments are marshalled and sent to the callee but the caller does not block waiting for the result. Consider the statement.

x = mentat_object.member_function().

The run-time system monitors (with code provided by the compiler) where *x* is used. If *x* is later used as an argument to a second or third Mentat object invocation, then arrangements are made to send *x* directly to the second and third member function invocations. If *x* is used locally in a strict operation, e.g., *y=x+1;*, then the run-time system will automatically block the caller and wait for the value of *x* to be computed and returned. Note, though, that if *x* is never used locally (except as an argument to a Mentat object member function invocation) then the caller never blocks and waits for *x*. Indeed *x* might never be sent to the caller, *x* might only be sent to the Mentat object member functions for which *x* is a parameter. It is important to note that these decisions, as well as all communication and synchronization, are handled completely by the compiler and run-time system. The programmer is left free to concentrate on the application, not on the details. Two examples illustrate the power of the Mentat approach.

Example 1: Consider the code fragment below that uses an opened instance, *m_file*, of the Mentat class *2D_matrix_file*.

```
1: for(i = 0;, i < n; i++)
2:          data[i] = m_file.read_row(i);
3: // Process row 0.
4: for (j = 0; j < row_size; j++) // Block if data not available.
5:          sum = sum + mpy_factn* data[0][j]
6: // Process row 1.
7: for (j = 0; j < row_size; j++) // Block if data not available.
8:          sum = sum + mpy_factn* data[1][j]
```

Execution of the above results in *n read_row()* requests being sent to *m_file*. The execution of lines 4 and 5 will be delayed until the 0th row has arrived. While lines 4 and 5 are executed, *m_file* continues to service the remaining requests, fetching the rows and delivering them to the

client. If the processing of row 0 takes longer than the servicing of the row 1 request, then execution will not block on lines 7 and 8. The key point is that the user is not responsible for managing the synchronization.

Example 2: This example illustrates the construction of a simple pipeline process. For this example we define the Mentat class

```
REGULAR MENTAT class data_processor
public:
   data_block* filter_one(data_block*);
   data_block* filter_two(data_block*);
};
```

The member functions *filter_one()* and *filter_two()* are filters that process blocks of data. The exact function is not important. Consider the code fragment in Figure 8.

```
m_file in_file,out_file;
data_processor dp;
in_file.create();out_file.create();
int i,x; x = in_file.open((string*)"input_file",1);
x = out_file.open((string*)"output_file",3);
data_block *res;
for (i=0;i<MAX_BLOCKS,i++) {
     res = in_file.read((i*BLK_SIZE),BLK_SIZE);
     res = dp.filter_one(res);
     res = dp.filter_two(res);
     out_file.write((i*BLK_SIZE),BLK_SIZE,res);
}
```

Figure 8. A Pipelined Data Processor.

This code fragment sequentially reads MAX_BLOCKS data blocks from the file "input_file", processes them through filters one and two, and writes them to "output_file".

In a traditional RPC system this fragment would execute sequentially. When coded using Mentat, the program execution graph of Figure 9 results. Each of the four operations can be executed on a separate processor. Further, the executions would be pipelined, with communication and synchronization being fully overlapped.

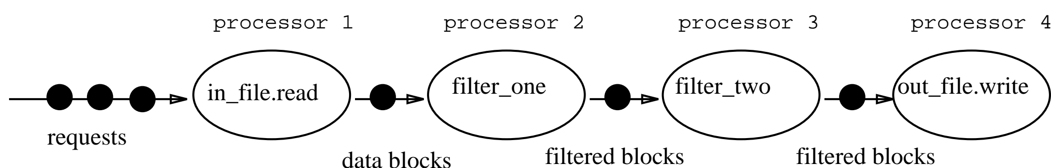


Figure 9. Program Execution Graph for Example2.

4. Related Work

Other researchers have identified the potential I/O bottleneck in high performance computers [8, 12]. Most of the proposed solutions involve caching, logging, prefetching or parallel disk hardware [8, 12, 14]. These solutions all require special hardware or operating system modifications and focus strictly on I/O performance. The solution presented in this report is broader in scope. By providing a framework for developing application-specific file abstractions, ELFS reduces the overall effort involved in programming high performance computers as well as addressing the I/O performance problem.

The memory management and file systems of the Choices [10, 17] operating system provide some of the same functionality as the file abstraction framework presented in this report. ELFS differs from the CHOICES file system in four significant ways. First, ELFS enables the specification of file behavior at the application level, not at the operating system level. Thus, defining, and using, new abstractions is very simple and requires no operating system changes. Second, ELFS is designed to facilitate the exploitation of application domain knowledge and structure, as well as user provided access pattern hints, to more intelligently cache and prefetch data. Third, ELFS explicitly encourages the decomposition of data based on object semantics. This better utilizes available bandwidth than straight striping. Finally, ELFS exploits compiler techniques to *transparently* perform I/O in a parallel *and* pipelined fashion.

5. Summary and Future Work

ELFS is being developed to address both the IO performance requirements of current and future generations of parallel and sequential architectures, and to alleviate some of the cognitive burden placed on the programmer when he attempts to get the best possible performance out of traditional file systems. The basic ideas of ELFS are to 1) provide an *extensible* file system class hierarchy that permits optimizations (e.g., prefetching and caching) based upon the structure and semantics of the underlying file, and 2) allow operations on instances of the class hierarchy to be performed in an asynchronous, possibly pipelined, fashion whenever possible. This combination will result in file objects with very high observed bandwidths and very low observed latencies, while at the same time reducing the burden on the programmer.

5.1 Future Work

We are currently investigating several issues with regard to extensible file systems such as ELFS. First, are user defined file systems a win? There are two dimensions to this question, performance and ease-of-use. The second primary question to be investigated is what are the semantics of multiple inheritance when applied to file abstractions?

The performance question involves answering at least three sub-questions. First, how often can I/O be pipelined in user applications? If there is little opportunity for I/O pipelining then I/O will be essentially synchronous, implying that an application must pay the full I/O latency. This has serious implications for high-performance computing, and will necessitate increased research on low latency devices. On the other hand, if I/O can be pipelined, then current device technology will be sufficient. A related question is: 'what are typical computation ratios (computation time to I/O requirements)?' Once again, if the ratios are small then there is little hope for high-performance computation without better I/O device technology. Finally, does striping based upon object semantics really make a performance difference? If it does not, then why bother?

The ease-of-use question is more difficult to directly address. The question is whether an abstract data type that hides the implementation details of how to obtain high performance is easier to use than the naked file system. Proponents of object-oriented design would argue that this is the case. We are monitoring, and collecting comments from applications domain users of ELFS objects with the aim of answering this question.

Finally, what about multiple inheritance in file systems? This is the least understood aspect of ELFS. Typically when multiple inheritance is used, the member functions (behavior) and data of the parent classes are disjoint. This is convenient because it allows straightforward implementations. Unfortunately this is not the case in an extensible file system. While parent classes may have their own private run-time member variables, the underlying data structure, the file, is the same for both parents. Further, it will often be the case that what you really want in a derived class are member functions that share some of the behaviors of all of the parent classes, e.g., a *variable_consistency_2D_matrix* class. How do we reason about semantics in this case? How do we specify the merger of semantics, or the dominance of one over the other? We do not yet have the answers to these questions.

6. References

[1] H. Boral and D. J. Dewitt. Database Machines: An Idea Whose Time has Passed. In *Proceedings of the 1983 International Workshop on Database Machines*, pp. 166-187. 1983.

[2] T. W. Crockett. File Concepts for Parallel I/O. *Proceedings Supercomputing '89*, pp. 574-579, November, 1989.

[3] J. J. Dongarra. Performance of Various Computers Using Standard Linear Equation Software. Technical report no. CS-89-85, Computer Science Department, University of Tennessee, Knoxville, TN, November 1990.

[4] A. S. Grimshaw. The Mentat Run-Time System: Support for Medium Grain Parallel Computation. *Proceedings of the Fifth Distributed Memory Computing Conference*, pp. 1064-1073. Charleston, SC., April, 1990.

[5] A. S. Grimshaw. 'An Introduction to Parallel Object-Oriented Parallel Programming with

Mentat,’ Computer Science Report No. TR-91-07, University of Virginia.

[6] A. S. Grimshaw, and J. Prem, ‘High Performance Parallel File Objects,’ To appear in *6th Distributed Memory Computing Conference*, Portland, OR., April 1991.

[7] A. S. Grimshaw and Jane W. S. Liu. Mentat: An Object-Oriented Macro Data Flow System, *Proceedings of OOPSLA ‘87*. ACM, October 1987.

[8] D. F. Kotz and C. S. Ellis. Prefetching in File Systems for MIMD Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 218-230, April 1990.

[9] E. Levy and A. Silbershatz. Distributed File Systems: Concepts and Examples. *ACM Computing Surveys*, vol. 22, no. 4, pp. 321-374, December 1990.

[10] P. W. Madany, R. H. Campbell, V. F. Russo, and D. E. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. TR no. UIUCDCS-R-89-1507, Department of Computer Science, University of Illinois at Urbana-Champaign, April 1989.

[11] M. Nelson, B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, vol. 6, no. 1. February, 1988.

[12] J. Osterhout and F. Douglas. Beating the I/O Bottleneck: A Case for Log-Structured File Systems. *Operating Systems Review*, vol. 23, no. 1, pp. 11-28, January, 1989.

[13] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). *Proceedings of the International Conference on Management of Data*, pp. 108-116, June, 1988.

[14] P. Pierce. A Concurrent File System for a Highly Parallel Mass Storage Subsystem. *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, pp. 155-160, March, 1989.

[15] A. L. N. Reddy and P. Banerjee. An Evolution of Multiple-Disk I/O Systems. *IEEE Transactions on Computers,* vol. 38, no. 12, pp. 1680-1690, December, 1989.

[16] A. L. N. Reddy and P. Banerjee. Design, Analysis, and Simulation of I/O Architectures for Hypercube Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp.140-151, April,1990.

[17] V. F. Russo and R. H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems Using Object-Oriented Design Techniques. *Proceedings of OOPSLA ‘89.* ACM 1989.

[18] R. Sandberg, D Goldberg, S. Kleiman, D. Walsh, and B. Lyone. Design and Implementation of the Sun Network File System. In *Proceedings of Usenix 1985 Summer Conference*, pp. 119-130, June, 1985.

[19] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1986.

[20] B. Duquet, and T. Cornwell. Deconvolution on the Digital Production Cray X-MP, p. 10, NRAO Newsletter, no. 24, July 1, 1985.