*E*xtensible *F*i*L*e *S*ystems (*ELFS*) An Object-Oriented Approach to High Performance File I/O

John F. Karpovich
Andrew S. Grimshaw
James C. French

July 22, 1994

**Appeared in *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 191-204, October 1994.
Also available as University of Virginia, Department of Computer Science Technical Report CS-94-28 via the web @ ftp://ftp.cs.virginia.edu/pub/techreports/README.html.**

Work partially sponsored by NSF, NASA, CESDIS and NRAO.

*E*xtensib*L*e *F*ile *S*ystems (*ELFS*): An Object-Oriented Approach to High Performance File I/O

John F. Karpovich, Andrew S. Grimshaw, and James C. French
Department of Computer Science, University of Virginia
{jfk3w | grimshaw | french} @virginia.edu

Abstract

Scientific applications often manipulate very large sets of persistent data. Over the past decade, advances in disk storage device performance have consistently been outpaced by advances in the performance of the rest of the computer system. As a result, many scientific applications have become I/O-bound, i.e. their run-times are dominated by the time spent performing I/O operations. Consequently, the performance of I/O operations has become critical for high performance in these applications. The ELFS approach is designed to address the issue of high performance I/O by treating files as typed objects. Typed file objects can exploit knowledge about the file structure and type of data. Typed file objects can selectively apply techniques such as prefetching, parallel asynchronous file access, and caching to improve performance. Also, by typing objects, the interface to the user can be improved in two ways. First, the interface can be made easier to use by presenting file operations in a more natural manner to the user. Second, the interface can allow the user to provide an "oracle" about access patterns, that can allow the file object to improve performance. By combining these concepts with the object-oriented paradigm, the goal of the ELFS methodology is to create flexible, extensible file classes that are easy to use while achieving high performance. In this paper we present the ELFS approach and our experiences with the design and implementation of two file classes: a two dimensional dense matrix file class and a multidimensional range searching file class.[1]

1: Introduction

Many scientific applications require the use of some form of persistent data storage. We usually think of this persistent storage abstractly in terms of a set of files each of which contains a collection of data items. Programs may read data from files that existed before execution of the program, may write data to files that must remain after the program has finished executing, or both. What programmers of such applications would like is an environment that supports easy-to-use high performance access to stored data and supports code reuse and extensibility of existing code so that the effort needed to develop a new application or to modify an existing application is minimized. Ease of use, development, and maintenance have always been desirable goals, but it is becoming a more important issue as ever larger and more complex applications are created. In addition, many applications require high performance for file operations to avoid becoming I/O-bound, i.e. having their run-times dominated by the time spent performing file input and output. This last problem is exacerbated as CPU performance increases faster than persistent storage device performance. With the increase in CPU speeds, more data can be processed per unit time, requiring more data to be transferred to or from storage devices which are increasingly unable to meet this demand. This problem is further aggravated for parallel programs where the processing power of the application is effectively increased by using multiple processors.

The *E*xtensib*L*e *F*ile *S*ystem (*ELFS*) approach, first proposed by Grimshaw and Loyot [1], is based on the idea that in order to achieve high performance and

1. This work has been partially funded by grants NSF ASC-9201822, NASA CS-CESDIS 5908-93, and NASA NGT-50970.

```
class TwoDMatrixFile {
  public:
     int openFormatted(string fileName);
     DD_array* readRow(int rowNumber);
     DD_array* readColumn(int columnNumber);
     int writeRow(int rowNumber, DD_array* data);
     int writeColumn(int columnNumber, DD_array* data);
     void setStride(int newStride);
     int elementSize();
     int close();
}
```

Figure 1 - `TwoDMatrixFile` Class Interface

ease-of-use, files should be treated as *typed objects*. Thus the ELFS approach introduces the notion of applying the object-oriented paradigm to file systems. The philosophy of the ELFS approach is that the design of file objects can exploit type-specific knowledge to both support ease-of-use and provide better performance. In addition, features of the object-oriented paradigm, particularly encapsulation and inheritance, can be exploited to ease the tasks of design, development and maintenance of applications using file objects. To illustrate the concepts of the ELFS approach, we will use a two dimensional matrix file object as an example throughout the early sections of this paper. This serves as a good example because 1) it useful in many real-world applications; 2) the access patterns to such data are well known and easy to visualize; 3) many applications using such data, especially scientific applications, require high performance.

To support ease of use file object interfaces can be improved to allow the user to manipulate data items in a manner that is more natural than current file access methods available. For example, using our 2D matrix file, the interface should present data in terms of rows, columns or blocks, instead of bytes or records (partial interface for a 2D matrix file object is shown in Figure 1). Thus application programmers can express requests in a manner that matches their semantic model of the data. In addition, since the only access the user has to the file object is through the interface, the details of how the data is physically stored, retrieved, coerced, and converted is kept hidden. The underlying implementation manages these details, including possibly variable file formats and data representations.

To improve file performance, a typed file object can exploit type-specific knowledge about the kind of data to be stored and the likely access patterns to be employed. For example, the file object can use this information to organize the physical storage of the file efficiently and to implement high performance data retrieval techniques, including data caching, prefetching and asynchronous parallel data retrieval, sorting, and conversion. An important feature of our object-oriented approach is that all of the implementation details are hidden from end users - all they observe is the object interface and the performance of the operations when executed. Performance can be further improved by providing an interface that allows the user to specify traits about application access patterns that can be exploited by the implementation. Again using the 2D matrix file as an example, the user may know that accesses will be across columns with a certain stride. This information can then be used directly by the implementation to initiate prefetching of those columns needed next.

To support ease of development and maintenance, file objects can exploit encapsulation and inheritance mechanisms provided by the object-oriented paradigm. One of the goals of the ELFS approach is that by treating files as objects, programmers can exploit the mechanisms of the object-oriented paradigm to increase code reuse, extensibility, and modularity. We envision that the ELFS approach will be applied to create hierarchies of typed file classes. We want to stress that many of the concepts behind ELFS are familiar to the object-oriented community. Our contribution is the application of object-oriented mechanisms, particularly inheritance and encapsulation to the domain of high performance file systems for the scientific computing community.

The ELFS approach is a methodology for how high performance file systems should be developed, i.e. through the development of file objects. The ELFS approach is designed to work in conjunction with new developments in file system technology, including hardware, software and organizational improvements. The advent of commercially available and affordable RAID systems is a perfect example. Improvements in retrieval algorithms and new or improved file structures are two more examples. These new technologies can and should be used when applicable in the implementation of ELFS file objects. The beauty of the file object model is that the implementation can be changed without disturbing end user code.

ELFS has three primary goals: ease of use, high performance, and ease of development and maintenance. Current techniques employed to support file operations are deficient in at least one of the these For example, a UNIX-like file system using primitives, such as `seek`, `read`, and `write`, does not support ease of use or development, while high performance can be achieved only with effort and care. The problems with UNIX-like file systems are 1) the primitives provided work at a very low level that of individual, unformatted bytes - all the effort of organizing and tracking higher-level data structures is left to the programmer; 2) there is no notion of type that can be exploited by the interface; 3) there is no support for encouraging reuse of this code and no guarantee that the code will be easy to extend or adapt to new uses.

The goal of this paper is to present the ELFS approach to file I/O in more detail, paying particular attention to the benefits derived from using the object-oriented paradigm. We also describe our experiences applying the approach to two classes of files, a two dimensional matrix file class and a multidimensional range searching file class designed for use at the National Radio Astronomy Observatory (NRAO). These implementations demonstrate that the ELFS approach can be translated from theory into practice and that ELFS file objects can achieve high performance in real-world applications. In fact, the performance of both implementations has been very encouraging.

In Section 2, we present a more detailed description of the current techniques most commonly used for implementing file I/O, discussing how well they accomplish each of the three goals we have set. Section 3 describes the ELFS approach in detail. In Section 4 we present the two file objects we have designed and implemented using the ELFS approach and discuss their respective interfaces, possible extensions and modifications that can be made to these file classes, and their performance. Section 5 presents some related work in this area and Section 6 discusses our conclusions and future plans for ELFS.

2: Current Methods

Application developers currently employ a wide range of approaches for implementing operations to store and retrieve data to/from persistent storage. Common methods include using the native file system of the target machine directly to develop file structures, using a database management system, or using library routines supporting a particular file structure. The remainder of this section discusses how well each of these approaches meets the requirements of application developers who need high performance for file operations.

Using the Native File System

Coding complex file structures using the native file system is usually a complicated task because the interface for the native system is at a low level. For example, in a Unix style file system, the programmer must manage all the details of the file layout and how to translate data access requirements into the primitive file system operations. Developers armed with only the native file system who desire high performance, often face a decision of whether to spend the effort to implement a more complex high performance file structure or to use a simpler, lower performance approach (e.g. a sequential file). Because of the ad hoc design and implementation of many of these applications, even when a high performance approach is used it is often difficult to adapt or reuse code for other applications. First, the implementation of file operations is often embedded and intertwined within the main application, making extraction and reuse of such code difficult and time consuming. Second, the implementation may make undocumented assumptions about the application and its access patterns. Reuse of such code is difficult because these assumptions must first be found and then the new application must match these assumptions or the code must be revised. Third, the file operations may directly update or access application data structures. Such code must be found and revised for any new application. Fourth, and most importantly, many implementations are simply not designed to be reusable. Part of the ELFS approach is a philosophy that files should be thought of as reusable objects and designed accordingly. The OO paradigm encourages such thinking and *supports* reuse via encapsulation and inheritance. Careful development of file objects can improve the situation. By first developing the functionality in base file classes and then extending the basic implementation by deriving new classes, the resulting code is more easily adaptable to new applications.

For many applications the native file system does not inherently provide the best performance. Though almost all file systems now employ caching and many use some form of prefetching, these mechanisms are fixed within the file system. These systems do not provide mechanisms to allow applications to interact with their environment and therefore, the caching and prefetching strategies cannot be changed or tailored to the needs of applications. In short, even if the application can provide an “oracle” for its access patterns, there is no way to convey this information.

Using a Database Management System (DBMS)

Database management systems have been developed partly in response to some of the problems described above in using the native file system directly. The goal of a DBMS is to provide a layer of software between the user and the native file system that provides data manipulation operations to the user in a fairly intuitive manner, and also handles the details of guaranteeing correct, consistent execution and fault-tolerance. A DBMS attempts to provide these services while still achieving as much performance as possible. There are a number of different approaches for how a DBMS should be developed, including hierarchical, relational and object-oriented approaches [2]. Each puts different emphasis on the structure of the DBMS implementation and interface, providing differing degrees of usability, maintenance, and performance. However, most DBMSs are designed for general purpose use and force users to adopt the models used by the DBMS, rather than allowing the user to employ a model that is more natural or suited to their needs. DBMSs generally dictate the data representation and data manipulation models that the programmer can use (OODBMSs are generally more flexible in these areas). DBMSs also usually enforce a single set of sharing semantics and a single set of recovery semantics, which are often not the best choices for many applications - they are either not adequate or they are overkill.

The generality of DBMS can often hurt performance. DBMS usually have a single underlying model for the structure of the physical storage of data. Since the physical representation is inflexible, the DBMS cannot provide the high performance we are interested in for many types of data and application access patterns. In addition, DBMS often require overhead to maintain the guarantee of data consistency among transactions from different asynchronous processes and the integrity of data in the face of site failures. This overhead is often unnecessary, for example in read-only applications or when it is known *a priori* that only one application at a time will access the data. These performance penalties often make a DBMS an unattractive choice for applications needing high performance file operations.

File Libraries

Libraries designed to provide high performance file operations for specific types of data or access patterns have the potential to provide both high performance and ease of use. For example, it is easy to imagine a library supporting matrix file operations such as those discussed in the interface shown in Figure 1. However, the extensibility of a library is heavily dependent on the design principles employed in the original implementation and the programming language used. Libraries developed in a language that supports the object-oriented paradigm are more likely to be easy to extend and maintain than those developed using less structured approaches. The file library approach using object-oriented techniques, effectively creating file objects, appears very promising in meeting the goals we have set. In fact, this is the basis of the ELFS approach: developing high performance file objects.

3: The ELFS Approach

The ELFS approach is designed to address three important issues surrounding the use of data files: 1) ease-of-use, 2) code maintainability, extensibility and reuse, and 3) performance. ELFS is based on the object-oriented paradigm and, in particular, exploits the idea that files can be thought of as objects. The ELFS approach is comprised of four key ideas: (1) design the user interface of a file to support ease-of-use and also to improve performance; (2) improve performance by matching the file structure to the access patterns of the application and the type of data; (3) selectively employ advanced I/O access techniques such as prefetching, caching and parallel retrieval, to improve performance; (4) encapsulate the implementation details of a file within file objects to enhance maintainability and exploit inheritance to encourage code reuse and extensibility. Each of these concepts are discussed in more detail below.

3.1: Interface

There are two main objectives in the design of a file object interface. First, the interface should closely match the user's semantic model of the data contained in the file. An interface that presents data and data manipulation operations in a manner that matches the way a program is likely to use them is much easier to use than an interface expressed in unnatural terms. Possible data representations include common computer language representations such as integers, floating point numbers and strings, as well as higher-level representations such as records, rows, columns, rectangles, lists, or any arbitrary structure that can be expressed in the implementation's language. The end user does not need to know how the data is physically stored or how the data is converted from the physical storage representation to the interface representation, but only what the interface representation means. This is true even if the file may be stored in many different formats or if the format changes over time: the implementation can manage these different formats and coerce or convert data as necessary. Development in an environment where the data representations of the file interface match the internal data representation of the application relieves the programmer of the tasks of data extraction and conversion, a potentially significant saving of effort.

Data representation is not the only interface design issue. The object behaviors should also match the functionality required by applications using the file object. For example, in a file containing various two dimensional objects (e.g. rectangles, circles, polygons, etc.) and their positions, the interface should provide functionality for operations needed to manipulate these objects. Besides typical operations such as retrieving, inserting or deleting a 2D object, the interface might provide object intersection functions, retrieval of all objects within a bounded rectangle, etc. Though different applications may require different behaviors, each behavior only needs to be implemented the first time it is needed. New applications can then use previously defined behaviors at little or no cost, leveraging off of the previous work.

One of the drawbacks mentioned about both DBMSs and native file systems is that they often define a single set of sharing and recovery semantics, often providing either too strong or too weak guarantees for particular applications. We feel that part of the definition of a file object is its sharing and recovery semantics. For example, some applications may need file objects that maintain the same sharing semantics as UNIX provides, while for other applications this is overkill. These semantics are defined for each file class and are exposed to the user through the class interface and the behavior of its member functions. The class implementation must then ensure that the defined semantics are guaranteed.

In addition to providing an intuitive means for manipulating file data, the interface should allow the application to declare knowledge about its usage of the file. This knowledge can then be used by the file object implementation to improve performance. For example, if an application using a matrix file knows that upcoming requests will have a particular pattern, say every *i*th row will be accessed, then the interface should provide a way for the program to express this. The underlying implementation can exploit this additional knowledge to achieve better performance by prefetching rows that are required in the near future. In the absence of this knowledge, most prefetching schemes would incorrectly retrieve the data in order, i.e. sequentially, with the corresponding loss of performance. Other examples of useful application knowledge include declaration that data will be used more than once within the program (allowing the implementation to attempt to cache data), declaration that the data file is read-only for a single reader (allowing the implementation to shut off consistency checks for the file object, if any are used), or declaration of queries ahead of time (allowing the query results to be retrieved and buffered before they are needed). We have observed that many applications often do know a great deal about their future access patterns and that this knowledge is currently not exploited. It is important to note that while the information declaration functions do increase the size of the interface, the user only needs to use these additional behaviors when performance is an issue. In the absence of any user knowledge, the file object will employ a default strategy.

3.2: File Structure

There has been a great deal of work over the past decades in developing file structures for various uses. The database literature contains many examples of creative file structures that are well suited for particular application needs or types of data. Examples include tree-based structures such as *k-d* trees [3,4] and *R* trees [5], partitioning-based structures like grid files [6] and Piecewise Linear Order-Preserving-hashing (or PLOP) files [7,8], primary files with secondary indexes, and many others. Each of these file structures has advantages and disadvantages under different access requirements and data attributes. For example, indexing schemes work well for single record retrieval when one of the indices can be used. Because of data locality, they also work well when a range of data is accessed along the key by which the primary file is sorted. However, a range retrieval along a key that is not the primary file's sort key will generally not perform well because the request will access data blocks scattered across the file.

For I/O performance to be maximized, it is crucial to choose file structures that best match an application's access requirements. The goal is to reduce the number of file accesses required and to reduce the cost of each access. A common method to reduce the number of accesses needed for a file operation is to arrange the physical storage of the file such that data that is likely to be used together is stored together, i.e. exploit data locality. In this manner, an entire group of data items can be read or written together in one file operation, instead of requiring several separate file operations. In addition, the cost of each operation can be reduced (i.e. reduced latency) by proper placement of data files. This is especially true in distributed systems that may have a wide range of point-to-point communication times. Placing data "close" to the process that requires it will reduce the overall latency of a file operation.

3.3: Advanced I/O Techniques

The third part of our approach, type specific access methods, encompasses a set of orthogonal methods that can be applied where appropriate to any file scheme to improve effective I/O bandwidth, latency, or both. These methods include selectively and intelligently prefetching data, caching data likely to be used again in the near future and parallelizing file operations and other I/O related activities such as sorting. Using an object-oriented class scheme allows the implementor to choose which of the above methods are appropriate for a given file type and how they will be implemented. Selection and application of access methods can be fine tuned in two important ways. First, an improved interface can allow the user to act as an oracle, directing a file object's use of different access techniques. Second, using inheritance from base file classes, a user can derive application-specific file classes that tailor the use of access techniques with potentially little effort.

Prefetching

Prefetching data involves guessing ahead and starting I/O operations for data that is likely to be requested in the near future. The simplest prefetching strategy is to read the next block of data in the file after the last one read (sequential prefetching). This method is employed in many database management systems and some native file systems and works well for retrievals where reads will be sequential. For access patterns that do not behave this way, prefetching can actually be detrimental as resources, such as buffer space, I/O device bandwidth, server CPU cycles and network bandwidth are all used for reading in poor guesses. The key is to guess correctly as often as possible. One promising approach to achieving better guesses is to allow the user to specify intentions for how the file is going to be accessed. These intentions can be used to determine whether prefetching should be employed and if so, which data is likely to be needed next. Examples of employing such a strategy include declaring the stride of accesses to a matrix file or pre-specifying a query that will be needed in the near future. In both cases, the file object knows what data will be needed and can prefetch effectively.

File Caching

File caching is another popular method for reducing I/O latency and increasing effective I/O bandwidth that relies on temporal locality of data references. File caching exploits this property by keeping recent requests in local memory so if they are needed again, the request can be satisfied from memory instead requiring an I/O operation. This approach is applied throughout the computer storage hierarchy and can be very successful if references have a high degree of temporal locality. However, in applications where data files or subsets of data files too large to fit into local memory are read, this scheme does not help and may waste memory and processor resources. Like prefetching, the user often knows the access patterns of the application best and so it seems reasonable that providing an interface to express the user's intentions will lead to better overall use of resources.

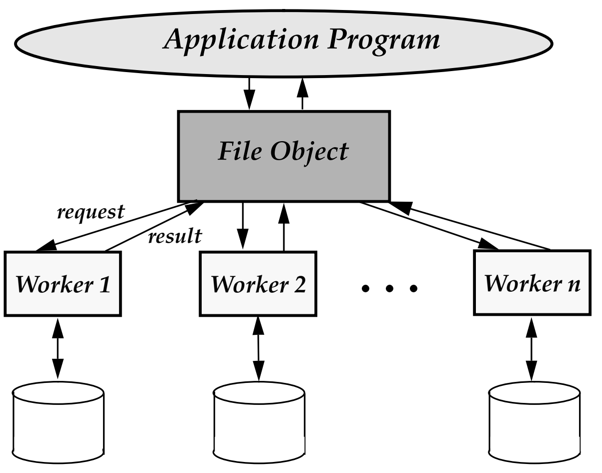


Figure 2 - Multiple I/O Servers for a Single Task

Multiple I/O Threads

Typical applications have one thread of control and therefore completely stop and wait when an I/O operation is pending. A better model is to try to overlap computation as much as possible with file I/O activity by splitting I/O operations and application code into separate threads of control that can execute in parallel (computer hardware is already built using this model). This idea can be extended by having multiple asynchronous I/O processes or threads that can overlap computation such as sorting and conversion and I/O requests. Of course, with a single I/O device for the entire file, the I/O device could create a bottleneck and limit performance. However, if the file is partitioned or replicated and placed on multiple devices, then separate asynchronous I/O processes can achieve true parallel retrieval and execution. Figure 2 shows the case of a partitioned or replicated file and multiple asynchronous I/O "workers". It is important to note that in the model shown, the end user still only sees the file object's normal interface, while the object's implementation coordinates the creation and use of the asynchronous workers.

3.4: Encapsulation Within File Objects

Treating files as objects and implementing them in a language that supports the object-oriented programming paradigm is intended to reduce the effort needed to develop and maintain applications that use files. This is accomplished by exploiting encapsulation and object inheritance. Encapsulation of the implementation of a file object limits how the object is used and which pieces of a program have access to hidden data and functions. Since all interactions with an object are defined by the object's interface, new functionality can be added via inheritance at any time to an existing class without invalidating any code already using the object. In this manner, the definition of an existing file object can easily be extended so that new requirements can be satisfied or the file object can be extended to support a new application.

4: Applying the ELFS Methodology

Thus far we have described the rationale behind the ELFS approach and given some examples where there are opportunities to exploit the approach. We next present our experience in applying the ELFS methodology to real-world problems. Specifically, we have created two file hierarchies, a two dimensional dense matrix file hierarchy and a multidimensional range searching file hierarchy. Both file hierarchies were implemented using Mentat, an object-oriented parallel processing system [9]. Mentat programs are written in the Mentat Programming Language (MPL), an extension of C++. The major addition to C++ in MPL is that classes can be tagged as being Mentat classes, which means that the member function invocations on the class are executed in parallel whenever possible. The Mentat system is a natural choice for implementing ELFS classes because it supports both the object-oriented programming paradigm and parallel execution, which is required for example in applying prefetching or parallel sorting techniques.

4.1: `TwoDMatrixFile` Class

Many applications, particularly scientific applications, use data in the form of a dense two dimensional matrix. Some examples include solution of dense systems of linear equations, image processing applications, grid-based modeling applications, and many others. Many of these applications require high performance from the file system because the problems and therefore the matrices are very large. If the matrix is too large to fit into memory, the program must move the data to and from persistent storage during execution. Even if this is not the case, the matrix usually must be initialized with data from a file. Poor file system performance can add significantly to the over-

Program 1:

```
int i, fd;
int m[maxR][maxC];

fd = open(filename);
for (i=0; i<numR; i++)
{
  // file position for row i,
  // column x
  seek(fd, position);
  read(fd, numBytes);
  // convert data
  m[i][x] = val;
}

        . . .

close(fd);
```

(a) Unix

Program 2:

```
TwoDMatrixFile    f;

f.openFormatted(filename);
maxR = f.numRows();
maxC = f.numColumns();

DD_intarray m(maxR,maxC);

m.column(2) =
        f.readColumn(2);

   . . .

f.close();
```

(b) `TwoDMatrixFile`

Figure 3 - Matrix Column Access: UNIX vs ELFS

all execution time of a program. The `TwoDMatrixFile` class is designed to alleviate the performance problems of these applications while simultaneously reducing the effort required to develop the file storage and retrieval operations.

Figure 3 demonstrates the power of an improved interface for matrix file operations. Both code fragments are designed to accomplish the same goal - read a specific column of integers from a matrix file. The sample code in 3*a* presents a possible implementation using UNIX style file operations; 3*b* presents an implementation using the `TwoDMatrixFile` and an auxiliary two dimensional data class, `DD_intarray` (`DD_intarray` is part of a hierarchy of 2D array classes derived from a base `DD_array` class). The `DD_intarray` class includes row and column operations in its definition. In 3*a* the application programmer needs to know all of the details about how the file has been physically stored. The programmer must know that the file is stored in row major format and therefore must iterate over the rows to retrieve each piece of the column. The programmer must also know the file organization and the data storage format. Is there a file header? If so, how is it organized? Is the data compressed or converted to some particular representation? For the programmer using the `TwoDMatrixFile`, this is not the case. The programmer only needs to know 1) there is a `TwoDMatrixFile` operation to return a matrix column in `DD_intarray` format and; 2) how to create and manipulate `DD_intarray` objects.

Figure 4 shows the interface for our current implementation of the `TwoDMatrixFile` class. As might be expected, the interface provides member functions

```
class TwoDMatrixFile {
   public:
      int createFormatted(string fileName,int rows,int cols,
            int size, int stride, int blockSize, int numBuffers);
      int openFormatted(string fileName);
      int openFormatted(string fileName, int stride,
            int numBuffers);
      DD_array* readRow(int rowNumber);
      DD_array* readColumn(int columnNumber);
      int writeRow(int rowNumber, DD_array* data);
      int writeColumn(int columnNumber,
            DD_array* data);
      void setStride(int newStride);
      int getStride();
      int numRows();
      int numColumns();
      int elementSize();
      int close();
}
```

Figure 4 - `TwoDMatrixFile` Class Interface

to create, open and close matrix files, to read and write rows and columns of data and to retrieve information about the matrix file - the number of rows and columns in the file and the element size. To potentially improve the performance of the file operations, the `setStride` function has been included to allow the user to change the stride that will be used for consecutive row or column operations (the `getStride` function returns the current value of the stride parameter). The underlying implementation is designed to exploit stride information by aggressively prefetching data at the specified stride. The user has no knowledge of when or how prefetching occurs. The user also has no knowledge of how the file is physically stored.

The `TwoDMatrixFile` class as presented supports only those data types for which there is an associated `DD_array` class. However, by deriving a new class from the `DD_array` hierarchy, the `TwoDMatrixFile` class can support new data types. In this manner, the `TwoDMatrixFile` class can be incorporated into many new applications with little or no effort[2].

The underlying implementation of the `TwoDMatrixFile` class exploits a file structure called block partitioning, that is designed to provide equal performance for row and column operations. Figure 5 demonstrates how block partitioning works - the matrix file is divided into blocks each containing a rectangular region of the matrix. The matrix data for each block is physically stored together in the file. To retrieve a row, several reads must be performed, one for

2. We realize that templates would be a better mechanism in this case. However, the MPL compiler does not currently support templates.

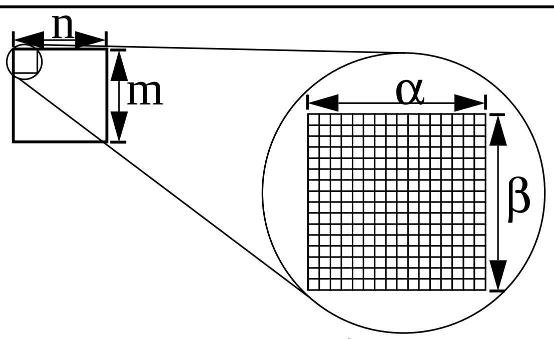


Figure 5 - Block Partitioning File Structure

each block intersected by the row. While this strategy may require more reads for row operations than if the file were stored in row major format, this performance penalty is more than made up with the performance gain for column accesses (if the entire file is read there is no such penalty). The performance degradation of row operations (or conversely column operations in a column-major format) is further offset by the fact that each row retrieval also reads in β−1 additional rows, which can be accessed at no additional cost. A possibility that we have not implemented is to support multiple file formats, row major, column major, and square partitioning, and to enhance the interface to allow the user to specify whether the file will be accessed by rows, columns or both. This enhanced file object would decide the best file format (a trivial choice in this case) and would manage access to the chosen file format, invisibly to the end user.

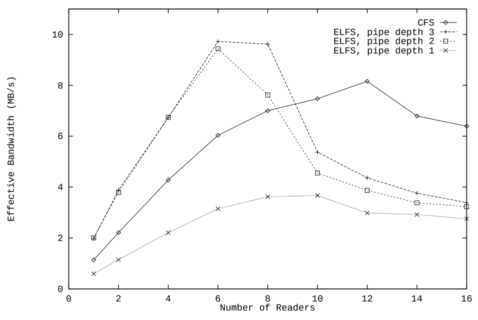
A sequential version (i.e. no prefetching was employed) of the `TwoDMatrixFile` class achieved comparable performance to using UNIX seeks and reads for row retrievals. The performance for column retrievals was about half that of row retrievals, due to less physical locality of the reads on the disk (the disk was not fragmented, therefore using a row major alignment of blocks places the pieces along rows closer together than the pieces for a column. We would expect increased file fragmentation to bring the row and column retrieval times closer together). The retrieval times for columns, however, were vastly superior to the UNIX column reads.

The parallel version, employing prefetching, also performed very well. This version was tested against the performance of the native Concurrent File System (CFS) on a 32 node Intel iPSC/2 hypercube, using a program that employed between 1 and 16 consumer processes. Since the performance of prefetching depends on the amount of computation a program performs in between successive reads, the parallel `TwoDMatrixFile` implementation was tested varying the number of floating point operations performed by the reader on each data element. Performance when executing only one floating point operation per element is close to CFS performance for row retrievals. For 10 FLOPs per element, the `TwoDMatrixFile` outperforms CFS by a significant margin for all numbers of reader processes. In all cases the `TwoDMatrixFile` is far superior for column retrievals. Figure 6 shows these results. Note that performance of CFS on column reads was so low that they are not included in the graphs. For more detailed discussion of the `TwoDMatrixFile` performance see [10].

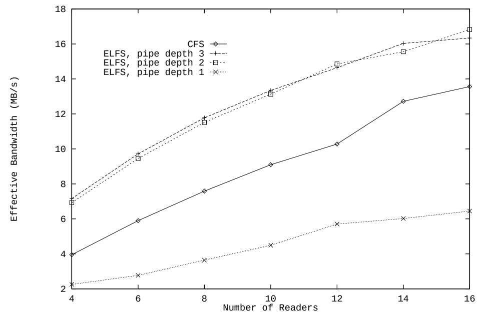
4.2: Multidimensional Range Searching (MRS) File Object

Multidimensional range searches appear in a wide range of applications. Such applications view a data set as an *n*-dimensional data space, where each dimension represents the values along a key field present in the data. The coordinates of each data record are its values for each of the *n* dimensions. Using this view, subvolumes of the data space can be defined by specifying a range of values for each dimension. For example, a data set containing a set of time indexed two dimensional images can be viewed as a three-dimensional data space *(time, x, y)*. Possible range searches for such a data set include retrieving a specified region of each image (a rectangle in *(x, y)*) for all time values, retrieving full images for a certain range of times, etc.

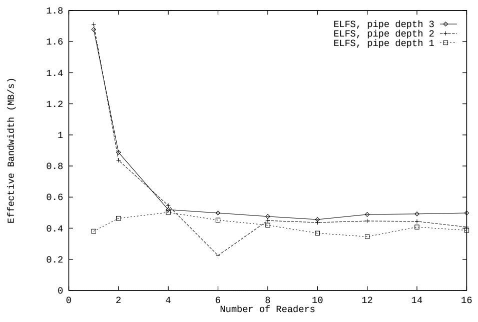
Our choice for implementing a file object for multidimensional range searching was motivated by a real-world problem. The National Radio Astronomy Observatory (NRAO) has many applications that view data exactly as described above. In particular, NRAO collects large volumes of interferometry data produced by radio telescopes which is then analyzed by scientists. These scientists view the data as a sparsely populated *n*-dimensional space and are interested in looking at particular subvolumes of the data (some of the dimensions for interferometry data include time, baseline, frequency, source, and polarization). Different scientists will need different views of the same data, depending on the type of analysis being performed. For example, one scientist may require all



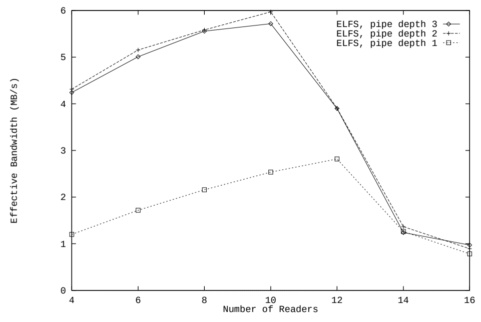
(a) Row Read Bandwidth -
1 Computation Per Row Item



(b) Row Read Bandwidth -
10 Computations Per Row Item



(c) Row Read Bandwidth -
1 Computation Per Column Item



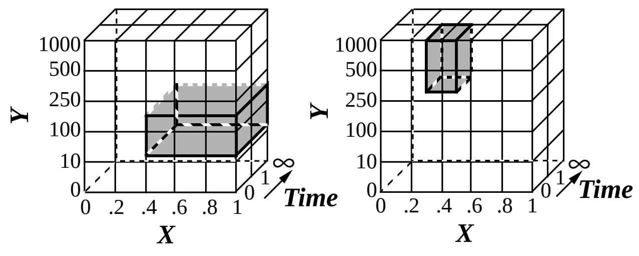
(d) Column Read Bandwidth -
10 Computations Per Column Item

Figure 6 - `TwoDMatrixFile` Effective Bandwidth vs CFS on 32 Node Intel iPSC/2

measurements for a time range for a specific region of the sky to study trends over time, while another scientist may need measurements for only one time frame and specific measurement frequencies but for a wider area of the sky. Performance of NRAO file I/O operations is currently unsatisfactory and many of their applications are now I/O-bound. Consequently, NRAO is studying methods to alleviate their I/O problem, including our multidimensional range searching file object. At the same time, NRAO is in the midst of re-engineering approximately 700,000 lines of code from FORTRAN and C to C++ to improve their efficiency in code development and maintenance. A more detailed description of our work with NRAO on the MRS file object is detailed in [11,12].

Our MRS file object implementation uses the PLOP file as the basic underlying file structure [7,8]. Though other file structures could be used for multidimensional range searches, it is our opinion that none of these candidates is clearly superior to PLOP files, while PLOP files have a relatively straightforward implementation. For a more in depth analysis of the choice of file structure see [11]. A PLOP file views a data set as a multidimensional data space. The data space is partitioned by splitting each dimension into a series of ranges called *slices*. The intersection of a slice from each dimension defines one logical data bucket. Data points are stored in the bucket that has corresponding values in each dimension. Therefore, within a bucket, the data points exhibit spatial locality in all dimensions. A tree structure for each dimension tracks the physical location of each bucket within the file, so that each bucket can be accessed very efficiently. This structure allows retrievals to eliminate parts of the file that do not correspond to values within the range search based on all dimensions, while quickly accessing those parts that may contain valid data.

We have implemented a sequential version of the PLOP file-based file object, (the `plopFile` class), and are currently working on the development of a parallel version that will allow us to distribute the



(a) All Times;
0.4 < X < 1.0;
15 < Y < 150

(b) 0 < Time < 1;
0.3 < X < 0.5;
300 < Y < 1000

Figure 7 - Sample Range Queries

PLOP file and apply prefetching and parallel retrieval, sorting, and conversion techniques. Here we will focus on the sequential version. The sequential implementation consists of a base class (`plopFile`) a derived class specifically designed for NRAO's interferometry data (`IFPlopFile`) and two further derived classes for specific types of interferometry data, "line spectrum" data and "continuum" data (`IFLinePlopFile` and `IFContPlopFile`, respectively). The resulting class hierarchy is shown in Figure 8 As would be expected, the base class provides general functionality for multidimensional range searching, while the derived classes add increasingly more specialized functionality. The interface revealed to the end user consists of a handful of member functions: a constructor and destructor, and `open`, `readHeader`, `writeHeader`, `addRecord`, `reportStats` and `createFromFits` functions. These few functions allow the user to create the PLOP file and add data to it. Our current version does not yet include update functions because they aren't needed for the current applications. In the future, these functions can easily be added to the existing interface if needed.

Retrieval mechanisms for the PLOP files are implemented in a separate class, `queryWindow`. By packaging the retrieval portion separately from the file, the user can define multiple data windows simultaneously in the same application, each related to the same file or to different files. The interface for `queryWindow` allows the user to easily specify the types of queries required for multidimensional range searches as well as specification of a sort key. Setting up a query is done by declaring a `queryWindow` object and then setting the range or ranges to search for each dimension and specifying the dimension by

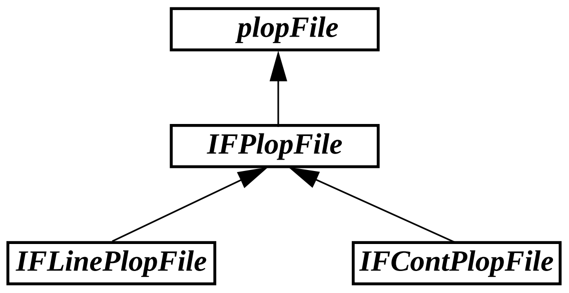


Figure 8 - PLOP File Class Hierarchy

```
public:
// constructor and destructor
queryWindow(IFPlopFile*);
~queryWindow();

// Reset functions
void resetAll();
void reset(int key);
void reset(char* keyName);

// sorting functions
void sortBy(int key, int sortDir);
void sortBy(char* keyName, int sortDir);

// do retrieval and collect statistics
void countPoints();

// set functions
void set(int key, int lower, int upper);
void set(char* keyName, int lower, int upper);
void set(int key, float lower, float upper);
void set(char* keyName, float lower, float upper);
void set(int key, Time lower, Time upper);
void set(char* keyName, Time lower, Time upper);
        . . .
```

Figure 9 - QueryWindow Class Interface

which to sort the resulting data. For example, to search for all data with times between 0.1 and 0.2, baselines 1 to 1000, and sorted by time, two `queryWindow` "set" calls are necessary - `set("Time", 0.1, 0.2)` and `set("Baseline", 1, 1000)` - and one call is necessary to set the primary sort dimension - `sortBy("Time", 1, ASCENDING)`. The `set` functions can all take multiple ranges for a single query. The result of such a query is that all data matching any of the ranges specified is retrieved. If the user desires the data for times (0.1,0.2), (0.5,0.75), and (0.9,1.0), a query can be made by using the `set()` function three times, one for each range. Single axis value queries are formed by specifying the same value for the upper and lower part of the range, such as (0.1,0.1). This interface was developed with help from NRAO scientists and we feel this interface is quite simple and intuitive for range queries.

Query Description	Total Records Retrieved	# Good Records	% Good Records	Total Data Retrieved (KBytes)	% Good Data (Accuracy)	Time (seconds)	Total BW (KBytes /sec)	Effective BW (KBytes/ sec)
All data	126,092	126,092	100%	63,232	78.7%	38.3	1,651	1,299
All data, sorted by TM	126,092	126,092	100%	63,232	78.7%	40.4	1,565	1,231
All data, sorted by BL	126,092	126,092	100%	63,232	78.7%	187.4	337	266
10% TM range	14,742	12,636	85.7%	7,584	65.7%	5.8	1,308	859
10% TM range, sorted by U	14,742	12,636	85.7%	7,584	65.7%	7.2	1,053	692
1 TM, 1 P	702	351	50%	316	43.8%	0.23	1,374	601
10% TM range, 1 P	7,371	6,318	85.7%	3,792	65.7%	2.33	1,627	1,079
50% TM range, 1 BL	752	186	24.7%	416	17.6%	1.45	287	51
50% TM, 1 A	15,026	4,836	32.2%	7,488	25.5%	14.5	516	132
1 BL, 1 P, sorted by TM	720	180	25%	400	17.7%	1.3	308	54

Figure 10 - Query Performance for Line-Spectrum PLOP File

To test the performance of our implementation we converted two small to medium sized NRAO data sets, one line spectrum file (~50 megabytes) and one continuum file (~270 megabytes). With NRAO scientists, we developed a suite of 10 queries representative of typical NRAO usage and used these to test the performance of each of the files. The performance of these queries for the line-spectrum file is shown in Figure 10. All tests were run on a SPARCStation IPX with 32 MB of RAM and an attached hard disk. The disk has an average rotational latency of 6.95 milliseconds and an average seek time of 9.8 milliseconds. The file we tested contains 6 keys: time (TM), baseline (BL), polarization (P), U, V, W and antenna (A). The query descriptions in Figure 10 describe those keys that were used to narrow the search; for any keys not listed, the entire range of values for that key are retrieved. In addition, the results were not sorted unless otherwise noted. Columns 2-6 describe the size of the requests and the accuracy of retrievals using the PLOP file structure. Column 4 (% Good Records) describes what percentage of all records retrieved satisfied the query. Column 6 (% Good Data) is also a measure of retrieval accuracy, but it also considers overhead. The last three columns describe our performance. Total Bandwidth measures the rate of data read by the file object, including overhead and data not satisfying the query. Effective bandwidth is the rate of *useful* data read per second.

We are currently attempting to gather similar performance statistics from existing NRAO applications. However, since the file operation code is deeply embedded within these applications it has been difficult to isolate code for file operations and to construct meaningful comparisons. For a more full description and analysis of our test results refer to [11,12].

5: Related Work

Many researchers have attacked the issue of high performance I/O from many different angles. Hardware designers are constantly trying to improve the performance of raw I/O devices. New organizational techniques, such as RAIDs [13], have been developed to better structure and use the devices currently available. Research has been done on better file organization [3-8] and there have been extensions made to programming languages to support high performance I/O, for example extensions to HPF Fortran detailed by Brezany, et al [14]. High performance file systems have been specially made for specific architectures, such as the Concurrent File System (CFS) for Intel supercomputers [15].

Other researchers have looked into user interface issues for manipulating persistent data. Examples include Kotz's work on multiprocessor file system interfaces [16], the Vesta Parallel File System introduced by Corbett, et al [17] and the many works in the OODBMS field.

These are just a few examples of related work in the field, but they do illustrate some of the variety of approaches used. All of these works are important pieces to providing an easy-to-use, high performance I/O system. One of the key advantages of the ELFS approach is that the encapsulation of files into objects simplifies the process of implementing new techniques as they emerge.

The idea of applying the object-oriented paradigm to database systems is not new [18,19]. What distinguishes our work is our objective. Rather than concentrate on flexibility, extensibility, and fidelity to the object-oriented paradigm, our work is targeted to the scientific programming community which places a premium on performance. Thus, we would be willing to sacrifice some flexibility for performance if it became necessary.

Our contribution is that we propose that I/O technologies should be exploited by selectively employing them when applicable and to combine them with the object-oriented paradigm to produce type-specific file objects. The result is extensible, ease-to-use, high performance file objects that meet the needs of scientific users.

6: Summary and Future Work

Our goals for designing the ELFS approach are to increase ease-of-use, improve file performance for those applications most in need of high performance, and to increase ease of development and maintenance. The object-oriented paradigm through object encapsulation and inheritance supports a programming style conducive to easing the burden of application development and modification. For these reasons, we feel that this model of programming should be applied to the development of file objects that are designed to manage access to persistent data. In particular, we feel that there is a need to apply the object-oriented paradigm to file objects designed for high performance. The reason for this is twofold. First, there is a growing need, especially in the scientific community, for high performance I/O, mainly due to inadequate improvements in storage device speeds. Second, high performance file objects, especially those employing parallel access techniques, are usually difficult to implement and will benefit greatly from OO techniques.

The ExtensibLe File System described in this paper, provides an approach for creating such file objects, exploiting the OO programming paradigm. We have demonstrated the feasibility of applying the ELFS approach to real-world problems by developing two useful file objects, a two dimensional matrix file and a multidimensional range searching file. So far, we are pleased with the performance of both objects. The success of these two objects leads us believe that the ELFS approach is a good model for high performance I/O support and can be applied successfully to a wide range of applications domains.

We are currently developing a new version of our multidimensional range searching file object with an emphasis on improving the interface and implementing a distributed file scheme and parallel access techniques. Planned improvements to the interface are to include data retrieval functions, to provide functions for specifying user knowledge about access patterns, such as the number of times data will be read, and to allow the user to pre-specify queries so that their retrieval can be performed in advance. We have developed a distributed PLOP file structure that provides several different distribution patterns, including patterns akin to striped and segmented strategies used in distributing array data in parallel languages like Fortran D. The new version will exploit the distributed file implementation, using parallel access to the file to improve performance. In addition, the new version will exploit parallelism by implementing selective prefetching and parallel sorting and conversion. It is our hope that this improved version will be a file object that meets all of the goals we wished to achieve.

Though our initial findings for the MRS file object have been very promising, we have tested it using only two closely related types of files in one application domain. We intend like to incorporate the MRS file object into other real-world applications. This will be useful in determining how easy it is tailor to an existing object for new applications. It would be particularly useful for a programmer outside of the ELFS project group to develop an application using one of our file objects, thus giving us an unbiased opinion about how easy it is to use the file object.

The objects developed so far have been developed basically in isolation from each other, each being developed mostly from scratch. Each implementation is a significant amount of work. It is useful to define which functionality is common across all or large classes of file object implementations and which functionality is unique. A long term goal of the ELFS project is to identify common functionality and to implement this functionality in a modular fashion. Programmers requiring a new file object can then use this base functionality in their implementation, avoiding the need to re-invent a portion of their code.

7: References

[1] A. S. Grimshaw and E. C. Loyot, Jr., "ELFS: Object-Oriented Extensible File Systems," University of Virginia, Computer Science TR 91-14, July 1991.

[2] C. J. Date, *An Introduction to Database Systems*, Volume I, Addison-Wesley, Reading, Mass., 1986.

[3] J.L. Bentley and J.H. Friedman, "Data Structures for Range Searching", *ACM Computing Surveys*, Vol. 11, No. 4, pp. 397-409, December 1979.

[4] J. T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes", *ACM SIGMOD Proceedings of Annual Meeting*, pp. 10-18, 1981.

[5] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proceedings of Annual Meeting, ACM SIGMOD Record*, Vol. 14, No. 2, pp. 47-57, 1984.

[6] J. Nievergelt and H. Hinterberger, "The Grid File: An Adaptable, Symmetric Multikey File Structure", *ACM Transactions on Database Systems*, Vol. 9, No. 1, pp. 38-71, March 1984.

[7] H. Kriegel and B. Seeger, "PLOP-Hashing: A Grid File without a Directory", *Proceedings of the Fourth International Conference on Data Engineering*, pp. 369-376, February 1988.

[8] H. Kriegel and B. Seeger, "Techniques for Design and Implementation of Efficient Spatial Access Methods", *Proceedings of the 14th VLDB Conference*, pp. 360-370, 1988.

[9] A. S. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat", *IEEE Computer*, pp. 39-51, May, 1993.

[10] B. Pane, "Efficient Manipulation of Out-of-Core Matrices", University of Virginia, Department of Computer Science.

[11] J. F. Karpovich, Andrew S. Grimshaw, James C. French, "Breaking the I/O Bottleneck at the National Radio Astronomy Observatory", University of Virginia, Computer Science, in progress.

[12] J. F. Karpovich, Andrew S. Grimshaw, James C. French, "High Performance Access to Radio Astronomy Data: A Case Study", to appear in *Proceedings of 7th International Working Conference on Scientific and Statistical Database Management*, September 1994.

[13] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proceedings of SIGMOD International Conference on Management of Data*, pp. 109-116, 1988.

[14] P. Brezany, M. Gerndt, P. Mehrotra, and H. Zima, "Concurrent File Operations in High Performance Fortran", ICASE Report No. 92-46, pp. 1-15, 1992.

[15] P. Pierce, "A Concurrent File System for a Highly Parallel Mass Storage Subsystem", *Proceedings of 4th Conference on Hypercubes, Concurrent Computers, and Applications*, vol I, pp. 155-160, 1989.

[16] D. Kotz, "Multiprocessor File System Interfaces", Dartmouth College, Dept. of Mathematics and Computer Science, technical report PCS-TR92-179, 1992.

[17] P. F. Corbett, D. G. Feitelson, J-P. Prost, and S. J. Baylor, "Parallel Access to Files in the Vesta Parallel File System", *Proceedings of Supercomputing '93*, pp. 472-481, 1993.

[18] S. B. Zdonik and D. Maier, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, Inc., San Mateo CA, 1990.

[19] F. Manola, S. Heiler, D. Georgakopoulos, M. Homick, and M. Brodie, "Distributed Object Management", *International Journal of Intelligent and Cooperative Information Systems*, Vol. 1, No. 1, June 1992.