SPDP Workshop on Modeling and Specification of I/O, October 1995.

Exploiting Mapped Files for Parallel I/O

Orran Krieger, Karen Reid and Michael Stumm
Department of Electrical and Computer Engineering
Department of Computer Science
University of Toronto
{okrieg,reid,stumm}@eecg.toronto.edu
http://www.eecg.toronto.edu/parallel/

Abstract

Harnessing the full I/O capabilities of a large-scale multiprocessor is difficult and requires a great deal of cooperation between the application programmer, the compiler and the operating (/file) system. Hence, the parallel I/O interface used by the application to communicate with the system is crucial in achieving good performance. We present a set of properties we believe that a good I/O interface should have and consider current parallel I/O interfaces from the perspective of these properties. We describe the advantages and disadvantages of mapped-file I/O and argue that if properly implemented it can be a good basis for a parallel I/O interface that can fulfill the suggested properties. To demonstrate that such an implementation is feasible, we describe methodology used in our previous work on the Hurricane operating system and in our current work on the Tornado operating system to implement mapped files.

1 Introduction

Harnessing the full I/O capabilities of a large-scale shared-memory multiprocessor or distributed-memory multicomputer (with many disks spread across the system) is difficult. Maximizing performance involves correctly choosing from the large set of policies for distributing file data across the disks, selecting the memory pages to be used for caching file data, determining when data should be read from disk, and determining when data should be ejected from the main memory cache. The best choice of policies depends on the resources of the system being used, how an application will access a file (which can change over time) and, in a multiprogrammed environment, how other applications are using system resources. We contend that to maximize I/O performance it is necessary for application programmers, compilers and the operating/file system to all cooperate. One of the greatest challenges facing developers of parallel I/O systems is to design interfaces that will facilitate this cooperation, will allow for implementations with high concurrency and low overhead, and will not unduly complicate the job of application programmers.

From a systems perspective, there are a number of levels of I/O interface, namely (1) the interface provided by the operating system, (2) the interfaces provided by run-time libraries, and (3) the I/O interface (if any) provided by the programming language. We argue that basing a system-level I/O interface on mapped file I/O is a good choice because it minimizes the policy decisions implicit in the accesses to file data, because it can deliver data to the application address space with lower overhead than other system-level I/O interfaces, and because it provides opportunities for performance optimizations that are not possible with other interfaces.

The next section presents a set of properties that we believe (and others have noted) are necessary for a good parallel I/O interface. We then describe some of the parallel I/O interfaces that have been developed and assess how well they support these properties. Section 4 presents our arguments for using mapped-file I/O as a basis for the system-level parallel I/O interface. Section 5 describes some of the problems with mapped-file I/O and solutions that overcome these problems. Finally, Section 6 describes techniques used to specify file system policies.

2 Interface properties

A good parallel I/O interface will have the following set of properties:

flexibility: The interface should be simple for novice programmers while still satisfying the performance requirements of expert programmers [14, 4, 13, 12]. The application should be able to choose how much, if any, policy related information it specifies to the system. In particular, it should be able to (1) delegate all policy decisions to the operating system, (2) specify (in some high level fashion) its access pattern, so that the operating system can use this information to optimize performance, (3) specify the policies that are to be implemented by the system on its behalf, or (4) take control over low level policy decisions, in effect implementing its own policies. As will be discussed in the next section, most current interfaces (implicitly) force novice users to make low level policy decisions (and hence constrain the optimizations that can be performed by the operating system), while still not giving sufficient control to expert programmers.

incremental control: A programmer should be able to write a functionally correct program and then incrementally optimize its I/O performance. That is, the programmer should be able to, with an incremental increase in complexity, provide additional information (or make more of its own policy decisions) in order to get better performance. Most current interfaces embed policy decisions in the operations used to access file data, forcing the applications to be rewritten when these policy decisions are changed.

dynamic policy choice: Applications can have multiple phases, each with a different file access pattern [14, 4, 27, 3]. The interface should therefore allow applications to dynamically change the policies used, be that by specifying a new access pattern, specifying a new policy, or making new policy decisions.

generality: The capabilities given to applications to specify policy should apply to both explicit I/O and implicit I/O due to faults on application memory. The same mechanisms for specifying policy should apply in both cases.

portability: The interface should be applicable to the full range of parallel systems, from distributed systems to multicomputers to shared memory multiprocessors [25, 22, 3, 12, 11]. An application ported from one platform to another should not have to be rewritten; it should only be necessary to change the policy related information used to optimize performance.

low overhead: Since performance is the central goal for exploiting parallelism, the interface should enable a low overhead implementation [15]. For example, it should not be necessary to copy data between multiple buffers when servicing application requests. Similarly, the amount of inter-process communication (e.g., system calls) entailed by the interface should be minimized.

concurrency support: The interface must have well defined semantics when multiple threads access the same file, should impose no constraint on concurrency, and should support common synchronization requirements with minimal overhead [22, 14]. For example, if the threads of a parallel application are accessing a file as a shared stream of data, then the interface should be defined so that the cost to atomically update the shared file offset is minimal. On the other hand, it should not be necessary to synchronize on a common file offset when the application threads are randomly accessing the file.

compatibility: The interface must be compatible with traditional I/O interfaces, such as Unix I/O [9, 4]. Existing tools (e.g., editors, Unix filters, data visualization tools) should be able to access parallel files created using the parallel interface. Also, it should be possible to rewrite just the I/O intensive components of an existing application in order to exploit the advantages of a parallel I/O interface, without having to rewrite the entire application. This means that the application should be able to interleave its accesses to traditional interfaces (e.g., Unix I/O) and the parallel I/O interface.

Perhaps the most important implication of these properties is that a good parallel I/O interface should separate the operations used to access file data from the operations used to specify policy. That is, the operations used by the application to access file data should not tell the system, for example, when data should be read from disk, when data should be written to disk, and which memory modules should be used to cache file data; they should only say what data they are accessing. Decoupling these policy decisions from the operations to access file data is important, since the policies used may change as the programmer optimizes I/O performance or ports the application to a new platform with different I/O characteristics.

3 Parallel I/O interfaces

Previous research has examined parallel I/O at several levels. Some have developed complete parallel files systems [10, 19, 4, 8]. Others have developed servers or runtime libraries for optimizing I/O performance that run on multiple systems [25, 22, 27, 11, 3, 12]. Some research has concentrated on developing specific techniques to improve I/O performance [15, 7] that could be incorporated into larger systems. Research has also been carried out specifically on developing application interfaces [14, 3, 13] and compiler interfaces [26, 1]. All of these approaches to parallel I/O systems consider interface issues to a varying degree.

Most existing parallel I/O systems are based on a read/write interface. A read/write interface has two major drawbacks: (1) the application specifies a buffer that is the source or target of the file data and (2) the operations are synchronous, blocking until the system transfers data to or from the specified buffer. For large requests, the synchronous nature of reads and writes means that the application is implicitly making the low-level policy decision of when data should be transferred to and from the system disks. This is inefficient, since even if the data requested on a read is not all immediately required, the request will block until the entire buffer has been filled. Also, for the reasons stated earlier, such coupling of policy and file access is a bad idea for a parallel I/O interface. While the application could instead make many small requests, this can result in a large overhead.

Asynchronous read and write interfaces allow the application to overlap I/O and computation, but they tend to be difficult to use, since the application must check to see if the request has completed before it can (re)use the buffer [24]. Also, once an application has initiated an asynchronous request, it cannot use any part of the buffer until the entire request has completed. Hence, the application is still implicitly making the policy decision of the granularity of I/O requests to disk. To overcome this problem, applications may be forced to use small requests which result in increased overhead.

Another problem with read/write interfaces is that the application specifies to the system the buffer that should be used for I/O. Again, this dictates to the system low-level policy decisions that should not be embedded in accesses to file data. While with distributed-memory multicomputers there is little choice in the memory module that should be used, in a shared-memory multiprocessor the system has flexibility in choosing where to buffer file data, and it is possible that the buffer specified by the application may not be the best choice.

To avoid the limitations and performance problems of a simple independent read/write interface some researchers have turned to much higher-level interfaces where the programmer specifies I/O requests in terms of entire arrays or large portions of arrays, for example, and the underlying system can optimize each type of high-level request [15, 7, 11, 25]. The performance of such array based systems are impressive, and certainly interfaces tuned for arrays must be supported by any parallel I/O system that seeks to address the requirements of scientific applications. However, not all I/O intensive parallel applications are array based [5, 29], and the specialized nature of these interfaces makes them inappropriate for any other types of file access. Also, these interfaces typically still have the disadvantage that the application specifies the target buffer for an I/O request. Finally, from a flexibility perspective, they have the disadvantage that by being high level they limit the expert programmer's ability to further tune I/O performance.

There has been some work in defining interfaces that can specify to the system the policies it should use, especially in allowing applications to control how data is distributed across the system disks [4, 6]. These interfaces decouple the specification of policy from the accesses to file data, allowing an application to dictate how its data should be distributed across the system disks while hiding the distribution of the data from subsequent file accesses.

A few parallel I/O systems have made portability a priority [25, 22, 11, 3, 12]. These systems have been built for distributed memory systems on top of native file systems and portable communication interfaces such as PVM or MPI. In general, the interfaces of these systems have not been designed so that the additional policy opportunities available on a shared-memory multiprocessor can be exploited. Hence, while the systems themselves might be portable, their interfaces make it difficult to maximize application performance on all platforms.

Other properties described in the introduction have also been addressed by many I/O systems. Most systems can dynamically change the access patterns by closing and reopening files with a different type, as in MPI-IO, or a different logical view, as in Vesta. All systems support concurrent file access, some relying on file types to define which parts of the file will be accessed independently [3, 14, 4], some by changing the semantics of file pointers [14]. PIOUS uses a transaction-based system to solve the synchronization problem and provide some fault tolerance[22]. In general, systems that implement new file types tend not to worry about compatibility with a traditional Unix interface. Vesta, however, provides a utility to convert parallel files to traditional ones that might be used in editors and visualization programs.

The following sections will show how mapped-file I/O can support all of the properties defined earlier, overcoming some of the limitations of existing systems, and how policies and techniques developed in other research can be applied to the mapped-file interface.

4 Advantages of Mapped-File I/O

Most modern operating systems now support mapped-file I/O, where a contiguous memory region of an application's address space can be mapped to a contiguous file region on secondary store. Once a mapping is established, accesses to the memory region behave as if they were accesses to the corresponding file region.

We believe that mapped-file I/O is the best basis for a system-level parallel I/O interface because 1) little policy related information is embedded in accesses to file data, 2) secondary storage is accessed in the same fashion as other layers in the memory hierarchy, 3) it has low overhead, and 4) all requests pass through the memory manager, allowing information available only in this layer of the system to be exploited to optimize performance. We describe each of these characteristics and the advantages that arise in turn.

4.1 A pure file access mechanism

One of the key advantages of mapped-file I/O is that the application accesses file data by simply accessing the corresponding region of its virtual address space — little or no policy information is implicit in this mechanism for accessing file data. In contrast, most I/O interfaces embed low-level policy decisions in the file access operations. For example, recall the discussion of the policies embedded in the read/write interface.

The lack of policy in mapped-file I/O makes it a good candidate for an interface with the properties described in Section 2. For example, consider the property of flexibility. As we will show in Section 6, a good implementation of mapped files can provide the expert programmer with more opportunities for optimization than current read/write interfaces (e.g., giving the expert programmer access to low-level memory manager information in making policy decisions). On the other hand, a novice programmer can write an application that uses mapped file I/O without making any policy decisions, delegating all such decisions to the operating system. In contrast, with a read/write interface even the novice programmer must specify policy decisions, and these decisions constrain the optimizations the operating system can perform.

As stated in the introduction, separating policy and interface is also important for both incremental optimization of programs and portability. Since mapped-file I/O embeds no policy information in file accesses, changing policy for optimizing performance or portability will require no changes to the portion of the program that accesses the file data.

4.2 A uniform memory interface

When a file is mapped into the application address space, I/O occurs as a side effect of memory accesses, and hence secondary storage can be viewed as just another layer in the memory hierarchy. This can simplify some applications because they require no special I/O operations to access secondary storage. Also, the use of mapped files makes it easier to address the generality property from Section 2 — any mechanisms developed to specify policies for mapped files can also be applied to regions of the application address space not associated with persistent files.

We have found that making secondary storage accessible as a layer in the memory hierarchy allows the techniques used to tolerate memory latency to be exploited for tolerating disk latency. The Hurricane memory manager [28] supports prefetch and poststore operations that allow the application to make asynchronous requests for memory-mapped pages to be fetched from or stored to disk. A compiler that automatically generates prefetch instructions for cache lines [21] was recently modified to generate prefetch requests to Hurricane mapped pages. Modifying the compiler involved less than two weeks effort, while modifying the compiler to generate asynchronous read requests would have been more difficult [20].

Even when using a system-level read/write interface, a sophisticated compiler can hide from the application the explicit read and write operations, giving the application an abstraction similar to mapped files. However, the compiler supported abstraction is specific to each application, and does not allow applications running in different address spaces to share access to the same physical memory pages. Hence, the compiler provided abstraction makes it difficult for different applications to concurrently access the same files. Also, as we will see in the next two sections, there are performance advantages in using mapped-file I/O, and therefore using mapped-file I/O as the system-layer interface is a good idea irrespective of the interface provided by the language.

4.3 Low overhead

There are three reasons why mapped-file I/O results in less overhead than a read/write interface. First, with mapped-file I/O, rather than requesting a private copy of file data, the application is given direct access to the data in the file cache maintained by the memory manager. Hence, the use of mapped-file I/O eliminates both the processing and the memory bandwidth costs incurred to copy data.

Second, the system-call overhead is lower (relative to read/write interfaces) because applications tend to map large file regions into their address space; if it turns out that the application accesses a small amount of the data, only the pages actually accessed will be read from the file system. In contrast, the application must be pessimistic about the amount of data it requests when a read/write interface is used, since a read incurs I/O cost when invoked. The reduction in the number of system calls when mapped-file I/O is used may be offset by an increase in the number of soft page faults. However, some systems (e.g., AIX) do not incur any page faults when pages in the file cache are accessed. Also, the cost of a page fault is substantially less than the cost of a read system call on many systems [17].

Finally, mapped-file I/O places a lower storage demand on main memory. When an application uses a read/write interface, file data is buffered both in the cache of the memory manager and in application buffers. If mapped-file I/O is used, no extra copies of the data are made, so the system memory is used more effectively. If main memory is limited, the extra buffering of a read/write interface can result in paging activity, which adversely affects performance. This paging activity is aggravated by the memory manager's lack of information about the function of application buffers. The application buffers that cache data are considered by the memory manager as dirty pages (even if the data has not been modified) and hence must be paged out to disk. In contrast, when mapped pages in the file cache are not modified, they do not need to be paged out, since the data is already on disk.

4.4 Exploiting the memory manager

With mapped-file I/O, all requests to access file data must pass through the memory manager and the memory manager is responsible for all buffering of file data. This presents opportunities for policy optimizations not available when a read/write interface is used (and the application is responsible for its own buffer management).

The memory manager has access to low-level information, such as the occurrence of in-core page faults, not available to other layers of the system. Such information can be useful in dynamically detecting application access patterns in order to select policies that optimize for those patterns. For example, keeping track of page faults, the memory manager can detect that a process is accessing data sequentially, and on each page fault issue disk read requests for multiple pages. As another example, on a shared-memory multiprocessor the memory manager can use in-core page faults to determine which processes are using a particular page, and replicate or migrate that page for locality. Compilers can only optimize for access patterns that can be determined at compile time. Runtime libraries must instrument code in the path of file accesses in order to dynamically detect access patterns, and hence degrade performance in obtaining this information.

Consider again the prefetch and poststore operations described previously. These operations are similar to asynchronous read and write operations (Section 3), but since they pass through the memory manager they can be made simpler to use and more effective. Applications using prefetch operations do not need to check if data is valid before accessing it. If a page is accessed that has not yet been read from disk then a page fault occurs and the faulting process blocks until the data becomes available. Also, with mapped files the application can be optimistic about advising the operating system about which pages should be asynchronously written to disk. If it turns out that the application has not yet finished modifying the data, an access to the data will cause a page fault that removes the block from the disk queue.

The memory manager has available to it global information about the memory used by all applications running in the system. This information can be useful when implementing policies to optimize I/O performance. For example, the memory manager can ignore prefetch requests if demand for memory is high, while devoting a great deal of memory to prefetched data if memory demand is low. In contrast, an application that issues asynchronous read and write requests may make poor decisions in a multiprogrammed environment, asynchronously reading pages into buffers only to have the memory manager page them out because of a high demand for memory by other programs.

5 Addressing the problems of mapped-file I/O

While mapped-file I/O is supported by most current operating systems, there are a number of problems with both the interface and implementations of the interface that have limited its use. We describe several of these problems and the specific solutions that have been previously developed.

5.1 Interface compatibility

While support for mapped-file I/O has become a common feature on many operating systems, it tends to be used infrequently. The main disadvantage is that it is an interface for accessing only disk files. In contrast, read/write interfaces like Unix I/O allow applications to use the same operations whether the I/O is directed to a file, terminal or network connection. Such a *uniform* I/O interface allows a program to be independent of the type of data sources and sinks with which it communicates [2].

Another problem with the mapped-file I/O interface is that it is very different from more popular I/O interfaces like Unix I/O, and applications written to use these interfaces have to be rewritten to exploit the advantages of mapped-file I/O. Other parallel I/O interfaces are provided as extensions to Unix I/O, and only the I/O intensive portions of an applications need to be rewritten to exploit the advantages of the parallel interface.

We have developed an application level I/O library, called the Alloc Stream Facility (ASF) [18], which addresses these problems. ASF provides an interface, called the Alloc Stream Interface (ASI), which preserves the advantages of mapped-file I/O while still allowing uniform access for all types of I/O (e.g., terminals, pipes, and network connections). In the case of file I/O, ASF typically maps the file into the application address space and translates ASI requests into accesses to the mapped regions. In the case of an I/O service that supports a read/write interface, ASF buffers data in the application address space and translates ASI requests into accesses to these buffers (filling and flushing the buffers using read and write requests).

The Alloc Stream Interface preserves the advantages of mapped file I/O by avoiding copying or buffering overhead. The key ASI operations differ from read/write operations in that, rather than copying data into an application-specified buffer, they return a pointer to the internal buffers or mapped regions of the library. Hence, ASI does not have either of the two disadvantages of read/write interfaces: First, the system rather than the application specifies the buffer to be used for I/O. Second, in the case of a mapped file, ASI is not synchronous. The application can access the buffer returned without having to wait for all the data to be read from disk (accesses to pages not yet in memory will be blocked by the memory manager).

In addition to ASI, ASF supports a number of other I/O interfaces (implemented in a layer above ASI) including Unix I/O and stdio. These interfaces are implemented so that an application can intermix requests to any of the different interfaces. For example, the application can use the stdio operation, `fread`, to read the first ten bytes of a file and then the Unix I/O operation, `read`, to read the next five bytes. This allows an application to use a library implemented with, for example, stdio even if the rest of the application was written to use Unix I/O, improving code re-usability. More importantly, it also allows the application programmer to exploit the performance advantages of the Alloc Stream Interface by rewriting just the I/O intensive parts of the application to use ASI. Because the different interfaces are interoperable, the Alloc Stream Interface appears to the programmer as an extension to the other supported interfaces.

5.2 Support for concurrency

Mapped-file I/O imposes no constraints on concurrency when file data is accessed. While this is generally a good thing, applications may want to have synchronization or locking implicit in their I/O accesses in order to guarantee that a particular process or application is exclusively accessing a portion of the file.

The Alloc Stream Facility supports common synchronization requirements with minimum overhead. Since data is not copied to or from user buffers in ASI, the stream only needs to be locked while the library's internal data structures are being modified, so the stream is only locked for a short period of time. Also, since all accesses to file data are performed with locks released, application threads may be concurrently accessing different pages in the mapped region, and hence will independently cause page faults. For a system with multiple disks, the page faults can potentially be satisfied concurrently at different disks.

ASF is implemented using the building-block composition technique described in Section 6. This technique allows an application to select the library objects that implement its streams, making it possible for the implicit synchronization performed by the library to be tuned to the requirements of the application. For example, different processes may use the same object and hence share a common file offset, or they may use independent objects and pay no synchronization overhead to update a common file offset. In the former case, processes may use an object that (at a performance cost) implicitly locks the data being accessed, or they may use an object that just atomically updates the file offset without acquiring any locks on data.

5.3 Overhead

Under some conditions, mapped-file I/O can result in more overhead than read/write interfaces. Two such cases are writing a large amount of data past the end-of-file (EOF), and modifying entire pages when the data is not in the file cache. In the former case, mapped-file I/O will cause the page fault handler to zero-fill each new page accessed past EOF. In a read/write interface, zero-filling is unnecessary because the system is aware that entire pages are being modified. In the latter case, mapped-file I/O will cause the page fault handler to first read from disk the old version of the file data. Again, this does not have to be the case with Unix I/O.

While it was a problem in the past, zero-filling pages does not introduce any processing overhead on current systems. In fact, on most current systems zero-filling a page prior to modifying its data can actually improve performance. Most modern processors are capable of zero-filling cache lines without first loading them from memory. With such hardware support, zero-filling the page saves the cost otherwise incurred to load the data being modified from memory.

The latter problem is easily solved by having the application (or I/O library) notify the memory manager whenever large amounts of data are to be modified. The Hurricane memory manager provides a system call for this purpose. This operation marks any affected in-core pages as dirty, pre-allocates zero-filled page frames for any full pages that have not been read from disk, and initializes the page table of the requesting process with the new pages in order to avoid subsequent page faults.

5.4 Random small accesses

The minimum granularity of a mapped-file I/O operation is a memory page. That is, data is always read from disk, written to disk, and transferred into the application address space using some multiple of the system page size. This would seem to be a disadvantage compared to read and write operations where data can be transferred at a much smaller granularity.

In reality, we seldom expect this to be a problem. There is such a large overhead to initiate a disk request that making the minimum unit of transfer to and from the disk a full page introduces only a small extra overhead. However, in distributed and multicomputer systems, the time to transfer the extra data across the network may adversely affect performance, especially if the source is the file cache of an I/O node rather than a system disk. If this overhead proves to be a problem, the application can use ASF to access the file, and ASF can be configured to make read and write requests for file data in the same fashion as it makes read and write requests to handle I/O for terminals, pipes, and network connections.

5.5 Application controlled policy

In many current systems, achieving high I/O rates when reading data from disk is difficult if mapped-file I/O is used. The basic problem is that while read/write interfaces give the application a mechanism for (low-level) control of file system policies, no corresponding mechanism is generally available for mapped-file I/O.

Consider the problem of keeping the disks on the system busy performing useful I/O. Read and write requests can affect a large number of blocks in a single request. Hence, an (expert) application programmer can keep all the disks in the system busy, instructing the file system when to read data from disk and when to write data back to disk. In contrast, with mapped-file I/O disk-read requests result indirectly from page faults, hence each process will typically have only one request outstanding at a time.

In our previous work, we have solved the limitations of mapped-file I/O by giving applications low-level capabilities of making policy decisions, similar to those implicit in read/write interfaces. For example, the prefetch and post-store operations described previously provide a solution to the problem of keeping the disks busy. In a single request, the application can cause an arbitrarily large number of pages to be asynchronously read from or stored to disk. As another example, it would be simple to add a system call to Hurricane to allow applications to explicitly specify which memory modules should be used to cache particular file blocks.

Operations like prefetch give the application low-level control similar to that of read/write interfaces. However, such low-level control is less natural when mapped-file I/O is used. In the next section, we describe how higher level interfaces can be used to specify policy without requiring the application to make individual policy decisions.

6 Specifying policy

We have shown how mapped file I/O can be used as the system-level interface for accessing file data, but have only peripherally discussed how policy information can be specified by the application. In Section 2, we suggested that applications should be able to control policy specification at four different levels: delegating all policy decisions to the operating system, specifying access patterns so that the operating system can use this information, choosing the policies that are implemented by the operating system on behalf of the application, and controlling the low-level implementation of its own policies.

We briefly described in Section 4.4 how mapped-file I/O gives more opportunities to system software to automatically adjust policies to application requirements, i.e., efficiently handling the case where the application delegates policy decisions to the operating system. In Section 5.5 we also described how applications can control policy at a low level. In this section, we first discuss how interfaces developed by others, that allow the programmer to specify access patterns and policies, can be adapted to mapped-file I/O. Then we discuss a new interface that we have developed that gives the expert user more control over specifying the operating system policies used to optimize application performance.

6.1 Adapting policy interfaces to mapped file I/O

Much of the recent work on efficient support for parallel I/O concentrates on the requirements of scientific applications, and in particular on efficient access to matrices. A common characteristic of recently developed interfaces is that the application can specify per-processor views of file data, where non-contiguous portions of the file appear logically contiguous to the requesting processor [4, 27, 25]. These interfaces give the application a great deal of flexibility in dictating how its matrix should be distributed across the system disks.

Another advantage of providing multiple logical views of a file is that applications can easily change their logical access patterns. For example, an application can read columns from a file stored in a row-major format without having to do a large number of small read and seek operations. To efficiently handle such requests, several systems support collective I/O, where all the processes of an application cooperate to make a single request to the file system. This enables the system to handle all requests for a single file block at the same time, avoiding multiple reads of the same block from disk. It also makes it possible to use techniques such as disk-directed I/O [15, 7] that allow the layout of the data on disk to be taken into account to minimize disk seeks.

The interfaces for supporting processor specific views and collective I/O are all built on read/write interfaces for accessing the file data. Each processor passes to system software (i.e., an application-level library or system server) a buffer that it is the source or target of their data, and the system software performs the mapping between the application buffer and the file data in some (hopefully) optimal fashion.

Processor specific views and collective I/O could be provided by an application-level library above a system-level mapped-file interface in the same fashion that the Passion runtime library [27]provides these facilities above a system-level read/write interface. The prefetch and post-store operations we described previously would allow such an implementation to be at least as efficient as when a read/write interface is used.

A much more interesting alternative is to have the memory manager directly support these facilities, replacing the per-processor buffers required by the read/write interface with mapped regions. Providing this support in the memory manager could result in a large improvement in performance. Consider Kotz's disk-directed I/O [15] modified to use mapped file I/O, and assume that the memory manager makes each page available to the application process as soon as all the I/O nodes have completed accessing it. Such an implementation would allow application processes to access their mapped region while the collective I/O operation is still being serviced by the I/O nodes. If the I/O to a page has not yet completed, the process accessing that page will fault and be blocked by the memory manager until the I/O has completed. In contrast, with Kotz's implementation using a read/write interface, processes are blocked in a barrier until the entire collective I/O operation has completed. Hence, the use of mapped file I/O for disk-directed I/O both avoids the overhead of a global barrier and allows processors to perform useful work while the I/O operation is proceeding.

6.2 Building-block composition

In the previous section, we described how current matrix-based interfaces can be supported on a mapped-file based system. While these interfaces are necessary, their high-level nature makes it impossible for expert users to further optimize performance. Also, these interfaces are specialized for matrix-based I/O, ignoring other classes of I/O intensive applications. For example, many multiprocessors are designed to support both general purpose Unix applications and such specialized I/O intensive applications as databases in addition to scientific applications. For other examples, we refer to a paper by Cormen and Kotz, where they describe a number of I/O intensive algorithms that are not matrix based [5].

In this section we briefly describe building-block composition, a low-level technique for specifying policy that we employ in the Tornado operating system [23]. While allowing matrix-based interfaces to be implemented in a layer above it, building-block composition allows the expert user much greater control over operating system policy. Also, application level libraries, such as ELFS [13] and ASF [18], can exploit the power of building-block composition, while hiding the low-level details from the application programmer.

Building-block composition can be considered both a technique for structuring flexible system software (that can support many policies) and a technique for giving applications the ability to control operating system policies. The basic structuring idea is that each instance of a virtual resource (e.g., a particular file, open file instance, memory region) is implemented by combining together a set of what we call *building blocks*. Each building block encapsulates a particular abstraction that might (1) manage some part of the virtual resource, (2) manage some of the physical resources backing the virtual resource, or (3) manage the flow of control through the building blocks. The particular composition used (i.e., the set of objects and the way they are connected) determines the behavior and performance of the resource.

We give policy control to the application by allowing it to dictate the composition of building blocks used to implement its virtual resources.[1] The building blocks, once instantiated, verify that each referenced object is of the correct type and that any other required constraints are met. Hence, if some object requires that a particular file block size be supported, it verifies that all objects it references can in fact support that block size. This type of checking makes it safe for untrusted users to customize the building-block compositions.

As a simple example, Figure 1 shows four building-block objects that might implement some part of a file and how they are connected. Object B contains references to C and D, and in turn is referenced by object A. Object C and D may each store data on a different disk, object B might be a distribution object that distributes the file data to C and D, and object A might be a compression/decompression object that de-compresses data read from B and compresses data being written to B.

We have used building-block composition in the Hurri-

[1] The composition is dynamic and can, in principle, be changed repeatedly by the application.

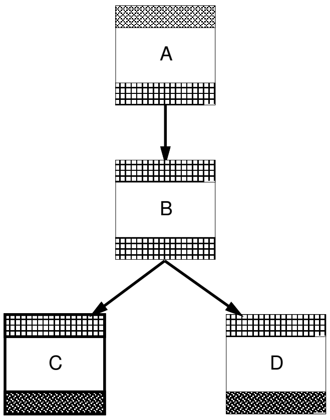

Figure 1: *Building blocks implementing some virtual resource, such as a file. Objects* C *and* D *may each store data on a single disk, object* B *might be a distribution object that distributes the file data to* C *and* D*, and object* A *might be a compression/decompression object that de-compresses data read from* B *and compresses data being written to* B.

cane file system [16] (of which the Alloc Stream Facility is one layer). Each file (and open file instance) is implemented by a different building block composition, where each of the building blocks may define a portion of the file's structure or implement a simple set of policies. For example, different types of building blocks exist to store file data on a single disk, distribute file data to other building blocks, replicate file data to other building blocks, store file data with redundancy (for fault tolerance), prefetch file data into main memory, enforce security, manage locks, and interact with the memory manager to manage the cache of file data.

We found that building block composition added low (and in fact negligible) overhead to the implementation of the file system. The use of building blocks gave us a great deal of flexibility, allowing the implementation of files to be highly tuned to particular access patterns. File structures can be defined in HFS that optimize for sequential or random access, read-only, write-only or read/write access, sparse or dense data, large or small file sizes, and different degrees of application concurrency. Policies can be defined on a per-file or per-open instance basis, including locking policies, prefetching policies, and compression/decompression policies.

We are involved in an effort to develop a new operating system, called Tornado, for a new shared memory multiprocessor. Building-block compositions will be supported by all components of the new operating system, including the memory manager. We have defined different memory management building blocks for prefetching, locking, redirecting faults for application handling, page replacement, page selection, compression, page replication, page migration, and interacting with different file servers. We are at a very initial stage in our implementation, but believe strongly that the same advantages that we found for the file system will also apply to the memory manager.

7 Concluding remarks

We presented a list of the properties we believe a good parallel I/O interface should have. One of the key implications of this list is that the interface should separate between the specification of policy and the accesses to file data. We argued that mapped file I/O is a good choice for a system-level interface because it (1) minimizes the policy decisions implicit in the accesses to file data, (2) can deliver data to the application address space with lower overhead than other system-level I/O interfaces, and (3) provides opportunities for optimizing policy that are not possible with other interfaces. The performance and interface problems of mapped file I/O were described along with solutions that have been developed for addressing these problems. Finally, we describe how current techniques for specifying policy can be applied to mapped file I/O, and described the building-block composition approach which we have developed to give applications finer low-level control over operating system policy.

References

[1] Rajesh Bordawekar, Alok Choudhary, Ken Kennedy, Charles Koelbel, and Michael Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, July 1995. Also available as the following technical reports: NPAC Technical Report SCCS-0696, CRPC Technical Report CRPC-TR94507-S, SIO Technical Report CACR SIO-104.

[2] D. Cheriton. UIO: A Uniform I/O system interface for distributed systems. *ACM Transactions on Computer Systems*, 5(1):12–46, February 1987.

[3] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–15, April 1995.

[4] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Sandra Johnson Baylor. Parallel access to files in the Vesta file system. In *Proceedings of Supercomputing '93*, pages 472–481, 1993.

[5] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[6] Erik P. DeBenedictis and Juan Miguel del Rosario. Modular scalable I/O. *Journal of Parallel and Distributed Computing*, 17(1–2):122–128, January and February 1993.

[7] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[8] Peter Dibble, Michael Scott, and Carla Ellis. Bridge: A high-performance file system for parallel processors. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, June 1988.

[9] Dror G. Feitelson, Peter F. Corbett, Sandra Johnson Baylor, and Yarsun Hsu. Parallel I/O subsystems in massively parallel supercomputers. *IEEE Parallel and Distributed Technology*, pages 33–47, Fall 1995.

[10] James C. French, Terrence W. Pratt, and Mriganka Das. Performance measurement of a parallel input/output system for the Intel iPSC/2 hypercube. In *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 178–187, 1991.

[11] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471, 1993.

[12] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995.

[13] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Extensible file systems ELFS: An object-oriented approach to high performance file I/O. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 191–204, October 1994.

[14] David Kotz. Multiprocessor file system interfaces. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 194–201, 1993.

[15] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

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

[17] Orran Krieger, Michael Stumm, and Ronald Unrau. The Alloc Stream Facility: A redesign of application-level stream I/O. Technical Report CSRI-275, Computer Systems Research Institute, University of Toronto, Toronto, Canada, M5S 1A1, October 1992.

[18] Orran Krieger, Michael Stumm, and Ronald Unrau. The Alloc Stream Facility: A redesign of application-level stream I/O. *IEEE Computer*, 27(3):75–83, March 1994.

[19] Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Conference*, pages 291–305, 1993.

[20] Todd C. Mowry and Angela Demke. Information on modifying a prefetching compiler to prefetch file data. personal communication, 1995.

[21] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 62–73, October 1992. Published as SIGPLAN Notices, volume 27, number 9.

[22] Steven A. Moyer and V. S. Sunderam. A parallel I/O system for high-performance distributed computing. In *Proceedings of the IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems*, 1994.

[23] Eric Parsons, Ben Gamsa, Orran Krieger, and Michael Stumm. (de-)clustering objects for multiprocessor system software. In *Proceedings of the 1995 International Workshop on Object Orientation in Operating Systems*, 1995.

[24] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. Informed prefetching: Converting high throughput to low latency. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 41–55, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[25] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995. To appear.

[26] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and Runtime Support for Out-of-Core HPF Programs. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 382–391, July 1994.

[27] Rajeev Thakur, Rajesh Bordawekar, Alok Choudhary, Ravi Ponnusamy, and Tarvinder Singh. PASSION runtime library for parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.

[28] Ronald C. Unrau. *Scalable Memory Management through Hierarchical Symmetric Multiprocessing*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada, January 1993.

[29] Darren Erik Vengroff and Jeffrey Scott Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of the 1995 IEEE Symposium on Parallel and Distributed Processing*, October 1995. To appear.