The Alloc Stream Facility

A Redesign of Application-Level Stream I/O

Orran Krieger and Michael Stumm, University of Toronto

Ron Unrau, IBM Canada

Many stdio and even Unix I/O applications run faster when linked to the ASF application-level library. Using the Alloc Stream Interface improves performance even more.

The success of the Unix operating system is partly attributable to the design of its input/output facility. The primary Unix I/O abstraction is a sequential byte stream provided by an interface of system calls: open, read, write, seek, close, and ioctl. The I/O facility is simple and versatile and can be uniformly applied to a variety of I/O services, including disk files, terminals, pipes, networking interfaces, and other low-level devices.[1] Nevertheless, application programs running under Unix typically do not use the Unix I/O system calls directly. Instead, they use higher level facilities implemented either by the programming language or its application-level libraries — for example, the stdio library for C or the iostream library for C++.

I/O facilities at the application level offer several advantages. The interfaces can be made to match the programming-language syntax and semantics, and they can provide functionality not available at the system level. They increase application portability because the I/O facility can be ported to run under other operating systems. Application-level I/O facilities can also significantly improve application performance, primarily by buffering the input and output to translate multiple fine-grained application-level I/O operations into individual coarser grained system-level operations. For example, if an application inputs one character at a time, each can be serviced from an application-level buffer without a system call; a system-level read must be issued only when the buffer is empty.

The interface and implementation of most application-level I/O facilities have changed little since the late 1970s.[2] However, the computing substrate — the computer architecture, hardware technology, and operating system — has changed substantially in the past 10 to 20 years, and we contend that application performance can be significantly improved by adapting both the implementation and interfaces of application-level I/O facilities to this change.

This article introduces an application-level I/O facility, the Alloc Stream Facility, that addresses three primary goals. First, ASF addresses recent computing substrate changes to improve performance, allowing applications to benefit from specific features such as mapped files. Second, it is designed for parallel systems, maximizing concurrency and reporting errors properly. Finally, its modular and object-oriented structure allows it to support a variety of popular I/O interfaces (including stdio and C++ stream I/O) and to be tuned to system behavior, exploiting a system's strengths while avoiding its weaknesses.

On a number of standard Unix systems, I/O-intensive applications perform substantially better when linked to our facility — in the best case, up to twice as good.

0018-9162/94/$4.00 © 1994 IEEE

Also, modifying applications to use a new interface provided by our facility can improve performance by another factor of two. These performance improvements are achieved primarily by reducing data copying and the number of system calls. Not visible in these improvements is the extra degree of concurrency our facility brings to multithreaded and parallel applications.

Changes in computing substrate

Recent changes in the computing substrate have significantly affected the cost of I/O and its composition.

First, the available physical memory has increased by several orders of magnitude. Many personal workstations now have 64-megabyte main memories, and that will probably increase to hundreds of megabytes over the next several years. Once accessed, files can often remain cached in memory, so many operating system I/O calls no longer involve accesses to I/O devices. In fact, many files are created and deleted without ever being written to secondary storage.[3] Thus, much I/O overhead stems from copying data from one memory buffer to another and from calls to the operating system.

Second, because processor speeds have improved much more dramatically than main memory speeds, large memory caches have become necessary. This increases the cost of buffer copying relative to processor speeds, since copying occurs either through the cache, destroying the cache contents, or directly to and from (the relatively slow) memory. Moreover, in today's parallel computer systems, memory has become a critical, contended resource. For example, researchers who have dramatically improved file I/O bandwidth (by introducing disk arrays) have found that the memory bottleneck makes it difficult to exploit this bandwidth.[4]

Third, the cost of a system call has been increasing relative to processor speeds.[5] Again, this is partially due to the effects of slower relative memory speeds and the increased number of registers modern processors need to save and restore on context switches. But it can also be due to new operating-system structures that use a microkernel and a set of user-level servers for system control. In these systems, the actual operating system is implemented as a set of servers running in application address spaces provided by the microkernel, and a system call is translated into a message or remote procedure call from the invoking program to a server. As a result, minimizing calls to the operating system has gained importance.

Fourth, many modern operating systems now support mapped files. A file can be mapped into the application's virtual address space and directly accessed in the mapped memory region. Mapped I/O requires no data copying between system space and application space; the data is usually made available to the application through page table manipulations alone. Although mapped files have been available for several years and can improve performance, they are not widely used. We believe this is because mapped files have not been integrated into standard I/O interfaces. Also, they cannot provide a uniform interface for I/O — that is, they can be used for file I/O but not, for example, for terminal I/O. In this article, we show how to exploit the performance advantages of mapped file I/O, while still supporting a uniform I/O interface.

Much I/O overhead stems from copying data from one memory buffer to another and from calls to the operating system.

Fifth, multithreaded programs are becoming more common, both because of the increasing availability of multiprocessor systems and because of their suitability as a structuring mechanism for some applications. Current I/O interfaces, however, are grossly inadequate for multithreaded programs. An obvious inadequacy of the Unix I/O facility, for example, is the reporting of errors to applications via a single global variable (errno).[6]

Reducing I/O overhead

I/O overhead is frequently an important factor in application performance. Consider, for example, a typical Unix filter that iteratively reads some input, performs a transformation, and writes some output. If the transformation is simple, the application's performance will be dominated by input and output.

Figure 1a illustrates the data flow using a traditional implementation of the stdio I/O library. The filter iteratively (1) calls fread to copy data from the stdio library input buffer to the application input buffer, (2) transforms the data from the application input buffer to the application output buffer, and (3) calls fwrite to copy the data from the application output buffer to the library output buffer. Whenever the library input buffer is empty, stdio calls the Unix I/O read system call to copy data from the system input file buffer to the library buffer. Whenever the library output buffer is full, stdio calls the Unix I/O write system call to copy data from the library output buffer to the system output file buffer.

Ignoring the transformation, this simple stdio filter requires copying each input-stream character four times: (1) from the system input buffer to the library input buffer, (2) from the library input buffer to the application input buffer, (3) from the application output buffer to the library output buffer, and (4) from the library output buffer to the system output buffer.

ASF can reduce the number of times data is copied. Figure 1b illustrates dataflow when the ASF implementation of stdio uses mapped files. Rather than buffering data in the library, and using Unix I/O operations to copy data from the system buffers to the library buffers, the system buffers are mapped into the application address space by the virtual memory system and used directly by the library. The data is copied only twice: (1) from the mapped system input buffer to the application input buffer and (2) from the application output buffer to the mapped system-output buffer.

ASF also supports a new I/O interface called the Alloc Stream Interface (ASI). Like the Unix I/O and stdio interfaces, ASI is a character-stream interface that can be used uniformly for

all types of I/O. The key difference is that, instead of having the application specify a buffer for copying I/O data, ASI allows the application direct access to the I/O library's internal buffers.

ASI's direct use of system buffers further reduces copying. Figure 1c shows dataflow when the filter is rewritten to use ASI implemented on top of mapped files. Data copying occurs only when the application transforms the data between the system input and output buffers.

Alloc Stream Interface

The Alloc Stream Interface is used internally in the Alloc Stream Facility and is also available to the application programmer. It is modeled after the standard C memory-allocation interface; thus, C programmers find it natural and easy to use.

The two most important ASI operations are salloc, which allocates a region of an I/O library buffer, and sfree, which releases a previously allocated region. They correspond to two C library memory-allocation operations: malloc, which allocates a memory region and returns a pointer to that region, and free, which releases a previously allocated region. The arguments to salloc and sfree differ from the memory-allocation operations in their inclusion of a stream handle to identify the target stream. For both salloc and malloc, the allocated region is located in the application address space and managed by an application-level library.

Salloc is used with sfree for both input and output. For input, salloc returns a pointer to the memory region containing the requested data and advances the stream offset by the specified length. Sfree informs the library when the application has finished accessing the region, at which point the library can discard any associated state. For output, salloc returns a pointer to the memory region where the data should be placed and advances the stream offset. The application can then use this region as a buffer for data to be written to the stream. Sfree informs the library when the buffer modifications are complete.

Since salloc returns a pointer to the library buffers, the library never has to copy data between application and library buffers; the application simply uses the buffers provided by the library. Also, since the library chooses the address, it can ensure that the alignment allows for optimizations such as mapped files. The idea of having the I/O facility choose the I/O data location is not new. For record I/O, both Cobol and PL/I reads provide a current record, the location of which is determined by the language implementation. Also, the sfio library's sfpeek operation[7] allows application access to the library's internal buffers. However, with both the read defined by Cobol and PL/I and the sfpeek operation defined by sfio, the data is available only until the next I/O operation, which is not suitable for multithreaded applications. In contrast, ASI data is available until it is explicitly freed.

ASI was designed from the start to support multithreaded applications. Because all ASI operations return an error code directly, it's always clear which thread incurred an error. Also, ASI has an sallocAt operation, which atomically moves the stream offset to a

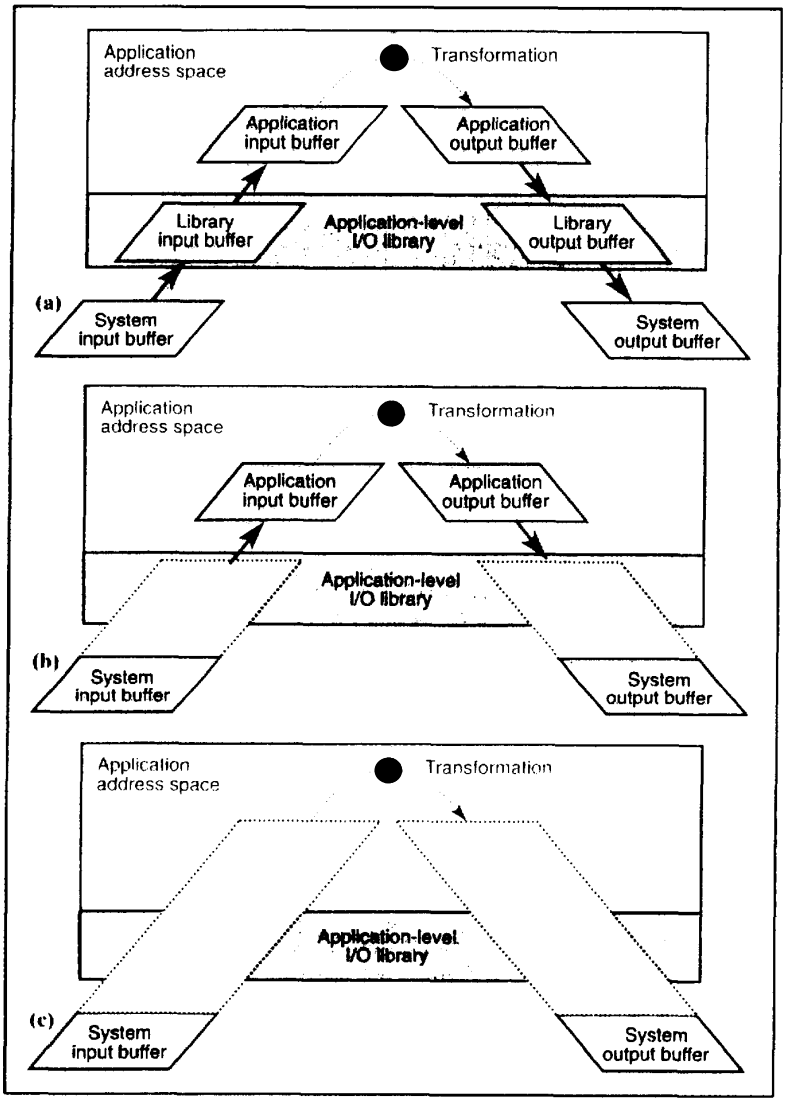


Figure 1. The flow of data for a simple Unix filter using a typical stdio implementation (a), the Alloc Stream Facility implementation of the stdio interface (b), and the Alloc Stream Interface supported by the Alloc Stream Facility (c).

particular location and performs an salloc at that location. Finally, ASI allows for a high degree of concurrency when accessing a stream. Because data is not copied to or from user buffers, the stream needs to be locked for only the short time needed to modify the library's internal data structures.

sopen(filename, flags, mode, error) returns handle:
opens a stream
sclose(stream) returns error: closes a stream

Basic operations:
salloc(stream, length) returns pointer: allocates **length** bytes from **stream**
sfree(stream, ptr) returns error code: frees previously allocated region
srealloc(stream, ptr, newlen) returns pointer: changes length of previously allocated region
sallocAt(stream, length, offset, whence) returns pointer: moves the stream offset to **offset** according to **whence** and allocates **length** bytes
sflush(stream) returns error code: flushes buffers
set_stream_mode(stream, mode) returns handle: changes mode of stream to **mode**

Mode variable operations: a **mode** argument specifies whether the operation is for reading or writing
Salloc(stream, mode, length)
SallocAt(stream, mode, length, offset, whence)

Unlocked operations:
u_salloc(stream, length)
u_sfree(stream, ptr)
u_srealloc(stream, ptr, newlen)
u_sallocAt(stream, length, offset, whence)
u_sflush(stream)
u_SallocAt(stream, mode, len, offst, whce)
u_Salloc(stream, mode, length)
u_set_stream_mode(stream, mode)
lockAsfStream(stream): locks stream
unlockAsfStream(stream): unlocks stream

Figure 2. The Alloc Stream Interface.

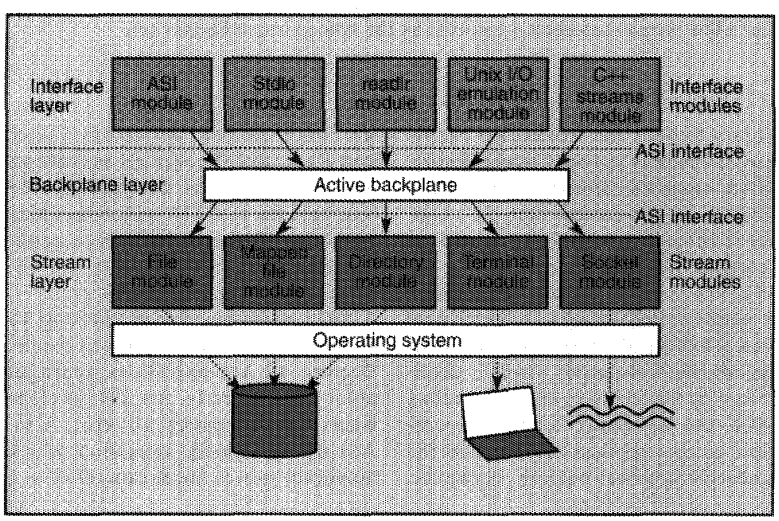


Figure 3. Structure of the Alloc Stream Facility.

Figure 2 shows the full ASI interface. An ASI stream is always in either read or write mode. The stream's mode can be changed with set_stream_mode. In addition to the basic operations, ASI has unlocked and mode-variable variants that allow greater application flexibility (at the cost of additional program complexity).

Structure of the Alloc Stream Facility

ASF is divided into three distinct layers (see Figure 3). At the top layer, *interface modules* implement particular I/O interfaces. These might include modules for stdio, C++ streams, or emulated Unix I/O (which provides the same interface as the Unix I/O system call interface). ASI is provided to the application by another interface module, and specialized interface modules — to access file system directories, for example — are also possible.

At the bottom layer, *stream modules* handle interactions with the underlying operating system and all buffering. Typically, there are many stream modules, each optimized for a particular access behavior, stream type, and system.

A *backplane* separates the top and bottom layers. ASI is the interface provided by the backplane and used by the interface modules. The interface provided by the stream modules and used by the backplane is a subset of ASI, namely the unlocked ASI operations. In addition, the stream modules often provide the backplane access to a single buffer, the *current buffer*, shared by the stream module and backplane through a simple producer/consumer protocol. I/O requests from the interface modules are serviced by the backplane from the current buffer whenever possible, thus significantly reducing the number of function calls to the stream modules. The backplane code is separated from the interface layer; otherwise, it would be common to all interface modules.

ASI was chosen for the backplane interfaces because it minimizes the amount of data copying and hence allows construction of a structured three-layer facility without a major performance penalty.

The modular, object-oriented struc-

ture of ASF offers several advantages. First, since each stream module supports only a single stream type, it can be optimally tuned to support accesses to that stream type. For this reason, we provide many specific modules instead of a few general ones. For example, read-only files are supported by a different stream module than read-write files. Because read-only files require substantially less checking, their implementation is simpler and faster than that of read-write files. In contrast, typical stdio implementations must check the type of stream, with an attendant degradation in performance. (In fact, to avoid slowing down all other stream types, most versions of stdio do not properly support disk files opened for both input and output. They require the application to insert fseek calls between input and output operations.)

Second, because each stream module exports only a small set of functions, writing a new stream module is simple. We've ported ASF to a variety of operating systems, including SunOS, IRIX, AIX, HPUX, and Hurricane. Although most of these systems support some variant of Unix, I/O performance is improved by adapting ASF to each system's particular characteristics.

Third, the interface modules are interoperable because they do not buffer data; only the stream modules buffer data. Operations from different interfaces can therefore be intermixed. For example, the application can use the stdio fread to read the first 10 bytes of a file and the emulated Unix I/O read to read the next five bytes. This allows using a library implemented with, for example, stdio even if the rest of the application uses Unix I/O. This improves code reusability and, more importantly, allows the programmer to exploit ASI's performance advantages by rewriting just the I/O-intensive parts of the application. Because the interfaces are interoperable, ASI appears to the programmer as an extension to the other (already existing) interfaces.

Implementation

We've implemented the ASF backplane and numerous interface and stream modules.

Backplane. The ASF backplane establishes and manages the communication between top-layer interface modules and bottom-layer stream modules. A client I/O state (CIOS) data structure (see Figure 4) is maintained for each open stream and shared by the backplane and the stream modules.

The backplane typically implements ASI operations as macros. Figure 4 shows the code executed by salloc and sfree. As the figure shows, the amount of code executed for salloc and sfree is very small. Despite the fact that they provide functionality equivalent to the stdio fread and fwrite functions, they have the simplicity of the stdio putc and getc macros (discounting the acquisition and release of the lock).

In the common case, when the request can be satisfied by the current buffer, salloc (1) acquires the lock for the CIOS structure, (2) checks that there is sufficient data (or space) in the buffer, (3) increments the number of references to the buffer, (4) decreases the amount of data (or space) remaining in the buffer, (5) advances the pointer for the next I/O operation, (6) releases the lock, and (7) returns a pointer to the allocated data. Sfree, in addition to acquiring/releasing the lock,

```
struct CIOS {
    slock lock ;                    /* lock for cios entry                          */

    /* stream specific function pointers */
    void * (*u_salloc)          () ;   int      (*u_sflush)     () ;
    int    (*u_sfree)           () ;   void     (*u_sclose)     () ;
    void * (*u_srealloc)        () ;   int      (*u_setmode)    () ;
    void * (*u_sallocAt)        () ;

    /* state of current buffer */
    int    mode ;                   /* 0 - read mode, 1 - write mode                */
    int    refcount ;               /* outstanding sallocs to buffer                */
    int    bufcnt ;                 /* characters/space in buffer                   */
    char *bufbase ;                 /* pointer to current buffer                    */
    char *bufptr ;                  /* pointer for next I/O op                      */

    void *sdata ;                     /* handle to stream specific data*/
} ;

    void *salloc( FILE *iop, int *lenptr )
      {
        void *ptr ;
        AcquireLock( iop->lock ) ;
        ptr = iop->bufptr ;
        if( iop->bufcnt >= *lenptr )
          {
            iop->refcnt++ ;
            iop->bufcnt -= *lenptr ;
            iop->bufptr += *lenptr ;
          }
        else
          ptr = iop->u_salloc( iop, lenptr ) ;
        ReleaseLock( iop->lock ) ;
        return ptr ;
      }

    int sfree( FILE *iop, void *ptr )
      {
        int rc = 0 ;
        AcquireLock( iop->lock ) ;
        if( (ptr <= iop->bufptr ) && (ptr >= iop->bufbase) )
          iop->refcnt— ;
        else
          rc = iop->u_sfree( iop, ptr ) ;
        ReleaseLock( iop->lock ) ;
        return rc ;
      }
```

Figure 4: The client I/O state structure and backplane code for salloc and sfree.

```
int read( int fd, char *buf, int length )
 {
   int error ;
   FILE *stream = streamptr( fd ) ;
   if( ptr = Salloc( stream, SA_READ, &length ) )
     {
       bcopy( ptr, buf, length ) ;
       if( ! (error = sfree( stream ) ) )
         return length ;
     }
   else  error = length ;
   RETURN_ERR( error ) ;
 }
```

Figure 5. Read implemented using ASI.

simply decrements the reference count to the current buffer. If salloc cannot be satisfied by the current buffer, the corresponding stream-specific function is called, a pointer to which is contained in the CIOS structure. In addition to satisfying the request, the stream-specific function will make a new current buffer available for subsequent salloc operations. (Special cases like unbuffered streams are supported by forcing the bufcnt, bufptr, and bufbase fields to zero so that each call to salloc results in a call to the corresponding stream-specific function.)

Other basic ASI operations are implemented in the backplane in a similar fashion. When possible, they use the current buffer; otherwise, they call the corresponding function provided by the stream module.

Interface modules. The simplest interface module is, of course, the ASI module. It simply exports the same interface as the backplane so that application programs can use it directly. Two other interfaces supported by our implementation are the stdio interface and the emulated Unix I/O interface. As an example, Figure 5 shows a slightly simplified version of the algorithm used to implement an emulated Unix I/O read. Read first calls salloc to allocate the data from the stream, then copies the data from the allocated region to the user-specified buffer, frees the allocated region, and finally returns to the application the amount of data read.

This code can be executed by different threads concurrently. Both the salloc and sfree operations acquire (and release) a lock to ensure that the CIOS structure is updated atomically. Also, because the stream offset is advanced by salloc, other threads concurrently calling salloc for the same stream will be given different areas of the buffer. This illustrates a major advantage of using ASI as the interface to the backplane: Data is copied from the library to the application buffer with the stream unlocked, allowing for greater concurrency than if the stream had to be locked for the entire read operation.

Stream modules. Stream modules can be specific to (1) an access mode (read, write, and read/write), (2) a buffering policy, (3) a particular I/O service (disk files, pipes, and sockets, for example), (4) a particular access pattern (sequential or random), and (5) a particular operating system and hardware platform. We've implemented many stream modules and have found that, when optimized for both the application and system, they substantially improve performance.

A good example of our approach is the way ASF supports file I/O on different Unix systems. Depending on the particular system, we use three different types of stream modules: one for systems where the Unix mmap operation provides the fastest file access, a second for systems that do not support mapped files or whose Unix I/O system calls provide faster access, and a third for AIX systems with the shmget mapped-file facility that outperforms mmap.

The stream modules based on Unix I/O system calls are similar to traditional stdio implementations. ASF does have some additional complexity because it must track buffers for which there are outstanding salloc operations, but each stream module is relatively simple because we use many specific modules instead of a single general one.

For stream modules based on mapped-file I/O, file regions of a fixed size are mapped into the application address space and managed as buffers. The Unix mapped-file interface, however, makes it difficult to implement Unix I/O end-of-file semantics in a mapped-file-based application-level library. Our solution to this problem is described in another work.[8]

Other researchers have also recognized the performance advantage of implementing stream I/O with mapped files. For example, Unix I/O can be implemented in the operating system using mapped files, allowing the system to exploit the virtual-memory hardware to improve the cache search time.[9] The Mach 3.0 operating system reflects Unix I/O calls back to the application level where they are serviced using mapped files, and the sfio library[7] uses mapped files whenever possible.

To date, we've concentrated on disk file I/O, but we believe that new service-specific stream modules can deliver improved performance for other I/O services such as pipes or network facilities. Related work by Maeda and Bershad[10] demonstrates substantial performance advantages from moving critical portions of network protocol processing into the application address space and modifying the networking interface to avoid unnecessary copying. With ASF, it is also possible to exploit specialized facilities for transferring data between address spaces, such as Govidan and Anderson's memory mapped stream facility[11] and Druschel and Peterson's Fbufs facility.[12]

Performance

We've compared ASF performance against the original stdio and Unix I/O facilities on three systems: an IBM RS6000/350 running AIX Version 3.2, a Sun 4/40 running SunOS version 4.1.1, and an SGI Iris 4D/240S running IRIX System V release 3.3.1.

To make this comparison, we measured the time to execute

- programs using stdio and Unix I/O linked to each system's originally

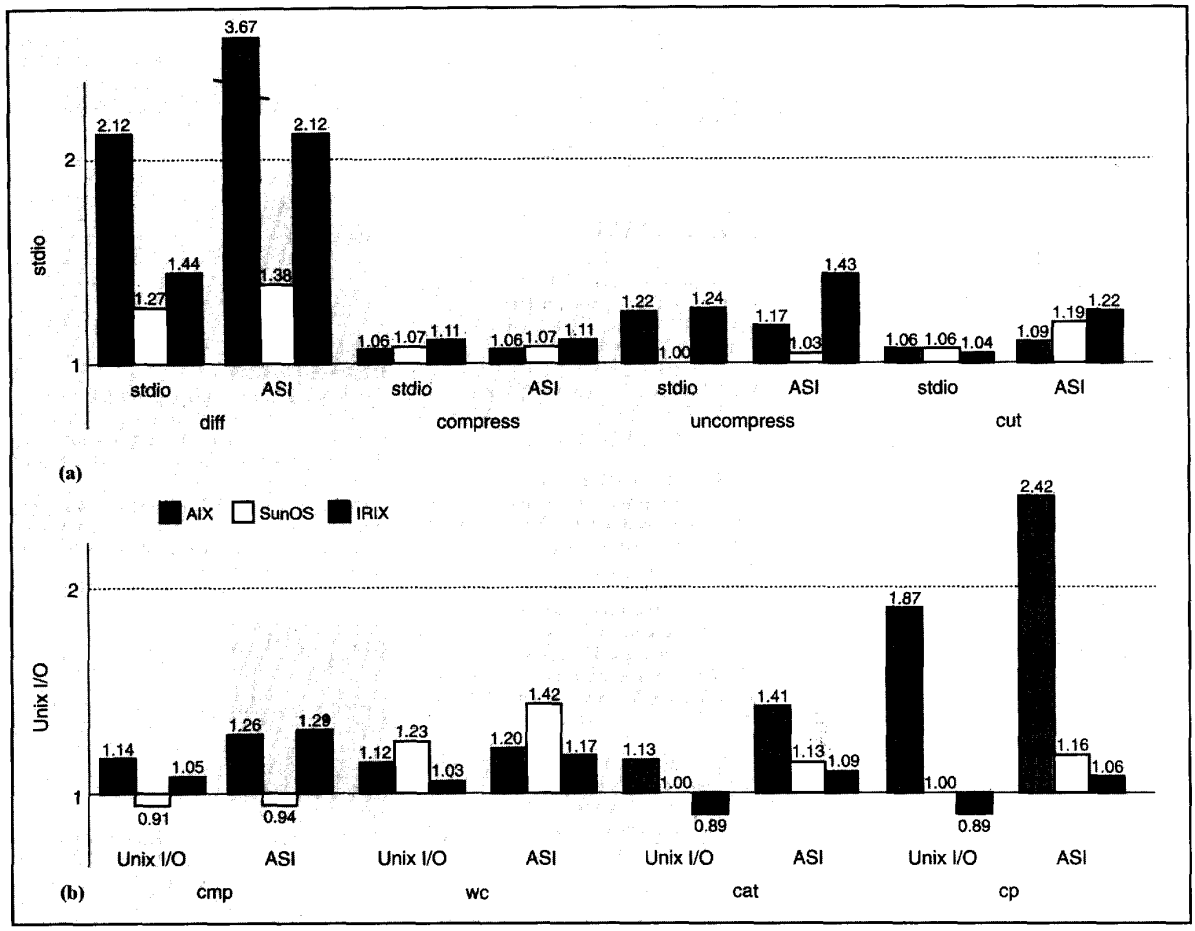


Figure 6. Speedup of stdio and Unix I/O applications that are (1) linked to the Alloc Stream Facility and (2) modified to use ASI. Diff compares the contents of two files (identical in our experiments); compress and uncompress use adaptive Lempel-Ziv coding to respectively compress and uncompress files; cut is a Unix filter that removes selected fields from the input file and writes the result to an output file; cmp compares the contents of two files; wc counts the number of characters, words, and lines in a file; cat copies the input file to the standard output; and cp copies one file to another.

installed facilities:

- the same programs, with no source code modifications, linked to ASF (so that they use the stdio and emulated Unix I/O interfaces); and
- the same programs, with source code modified to use the ASI interface directly. (Only minor changes to the programs, typically affecting fewer than 10 lines of code, were necessary to adapt them to ASI.)

Each system on which we performed these experiments has its own standard ASF configuration. The standard configuration on the AIX system uses mapped files (based on shmget) for both input and output. The standard configuration on the IRIX and SunOS systems uses mapped files (based on mmap) for input and Unix I/O for output.

Results are given in terms of speedup relative to the same program linked to the machine's installed facilities — that is, the time to run the program linked to the installed facilities divided by the time to run the program using ASF. For these experiments, all I/O is file based. The numbers indicate the expected speedup on an idle system with all file data available in the main memory file cache. (For more details on our experiments, see reference 8.)

Figure 6a, shows that several stdio programs linked to ASF perform at least as well as those linked to the installed system libraries and in some cases significantly better. For example, on the AIX system, ASF cuts diff runtime to less than half, primarily by using mapped files.

Most stdio-based programs show further improvement when they're modified to directly use ASI. For example, modifying diff improved its performance on the AIX system an additional 40 percent, making it 3.67 times faster than the original program linked to the installed stdio. The additional gain is due to the fact that data need not be copied to (or from) application buffers.

The Unix I/O interface is specific to the Unix operating system, and for portability reasons, its direct use is generally considered poor programming practice. However, because stdio entails an extra level of copying, I/O-intensive

programs with large-grain I/O often use the Unix I/O interface directly. Figure 6b shows the performance of four Unix I/O-based programs. Surprisingly, unmodified ASF-linked applications often perform better than with direct use of the Unix I/O system calls. For example, cp is almost twice as fast when linked to ASF on the AIX system. Thus, on some systems, our application-level library implements the Unix I/O interface more efficiently than the operating system. Using ASI provides even greater improvement. For example, cp modified to use ASI ran two-and-a-half times faster than the original on the AIX system.

ASF's modular, object-oriented structure allows much flexibility in providing I/O interfaces and exploiting different systems' performance potential, and its new I/O interface substantially reduces copying overhead and maximizes concurrency. Overall, ASF offers distinct advantages in performance, concurrency, and functionality.

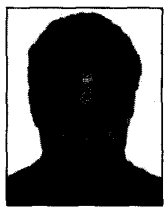
- *Performance.* Many stdio and even Unix I/O applications run faster when linked to ASF. Performance improves further when applications are modified to use the ASI interface directly. The improvements are a direct result of the facility's structure, which takes advantage of system-specific features like mapped files, and the reduction in I/O overhead implied by the interface's definition.
- *Concurrency.* Both ASF and ASI are designed for multithreaded applications running on multiprocessors. They maximize concurrency by keeping the stream unlocked while copying data. ASI operations also return an error code directly, so it's always clear which thread incurred an error. Finally, the ASI sallocAt operation allows location-specific data allocation without interference from other threads.
- *Functionality.* ASF supports a variety of I/O interfaces and permits intermixed calls to the different interfaces. These features improve code reusability and allow programmers to improve performance by modifying just the I/O-intensive portions of applications to use ASI. ■

Acknowledgments

Thanks go to Benjamin Gamsa for his contributions to this article, and to Zvonko Varensic, Ken Sevcik, and K.-Phong Vo for their useful comments.

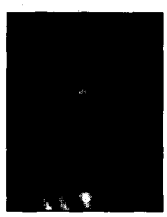
References

1. D. Cheriton, "UIO: A Uniform I/O System Interface for Distributed Systems," *ACM Trans. Computer Systems*, Vol. 5, No. 1, Feb. 1987, pp. 12-46.
2. W.R. Stevens, *Advanced Programming in the Unix Environment*, Addison Wesley, New York, 1992.
3. M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *Proc. 13th Symp. Operating System Principles*, ACM, New York, 1991.
4. A.L. Chervenak and R.H. Katz, "Performance of a Disk Array Prototype," *Proc. ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems*, ACM, New York, 1991, pp. 188-197.
5. J. Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware?" *Proc. Summer Usenix Conf.*, Usenix Association, Berkeley, Calif., 1990, pp. 247-256.
6. M. Jones, "Bringing the C Libraries with Us into a Multithreaded Future," *Proc. Usenix Winter Conf.*, Usenix Association, Berkeley, Calif., 1991, pp. 81-91.
7. D. Korn and K.-Phong Vo, "SFIO: Safe/Fast String/File I/O," *Proc. Usenix Summer Conf.*, Usenix Association, Berkeley, Calif., 1991, pp. 235-255.
8. O. Krieger, M. Stumm, and R. Unrau, "The Alloc Stream Facility: A Redesign of Application-Level Stream I/O," Tech. Report 275, Computer Systems Research Inst., Univ. of Toronto, 1993.
9. A. Braunstein, M. Riley, and J. Wilkes, "Improving the Efficiency of Unix File Buffer Caches," *Proc. 12th ACM Symp. Operating System Principles*, ACM, New York, 1989, pp. 71-82.
10. C. Maeda and B.N. Bershad, "Protocol Service Decomposition for High-Performance Networking," *Proc. 14th ACM Symp. Operating System Principles*, ACM, New York, 1993, pp. 244-255.
11. R. Govidan and D.P. Anderson, "Scheduling and IPC Mechanisms for Continuous Media," *Proc. 13th ACM Symp. Operating System Principles*, ACM, New York, 1991, pp. 68-109.
12. P. Druschel and L.L. Peterson, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," *Proc. 14th ACM Symp. Operating System Principles*, ACM, New York, 1993, pp. 189-202.



Orran Krieger is completing his PhD in electrical and computer engineering at the University of Toronto. His research interests include operating systems, file systems, and multiprocessors.

Krieger received a BASc from the University of Ottawa in 1985 and an MASc from the University of Toronto in 1989, both in electrical engineering.



Michael Stumm is an associate professor in the Departments of Electrical and Computer Engineering and Computer Science at the University of Toronto.

Stumm received a diploma in mathematics and a PhD in computer science from the University of Zurich in 1980 and 1984, respectively. He is a member of ACM and the IEEE Computer Society.

Ron Unrau is working on advanced compiler optimizations at the IBM Software Solutions Lab in Toronto, Canada. His research interests include parallel operating systems and compilers.

Unrau received his BASc in computer engineering from the University of Alberta in 1984 and his MASc in biomedical engineering and PhD in electrical and computer engineering, both from the University of Toronto, in 1989 and 1993.

Readers may contact Orran Krieger at the University of Toronto, Dept. of Electrical and Computer Engineering, Toronto, Canada M5S 1A4. His e-mail address is okrieg@eecg.toronto.edu.

optimizations such as direct memory access, outboard packet buffering, and programmed I/O to increase throughput on high-speed networks.

The architectural features and tradeoffs presented in this article define a design space that should help system architects systematically evaluate design choices.

High-Performance I/O for Massively Parallel Computers, pp. 59-68

Juan Miguel del Rosario and Alok N. Choudhary

This article presents an overview of the many issues related to high-performance I/O in parallel computing environments. The authors discuss I/O requirements for Grand Challenge applications and relevant issues in performance characterization, I/O architecture alternatives, operating and file systems, compiler and runtime support, checkpointing, network I/O, and so on. They present a status report on current practice and research in these areas, discuss outstanding problems, and describe some alternative solutions.

The increasing use of parallel computers has been accompanied by an increased demand for I/O systems support. Data movement to temporary storage, archival storage, visualization systems, or across the network to other computing resources has become a necessity in high-performance computing. Still, research and development of I/O systems for this type of environment are at an early stage of evolution.

Although I/O systems research is not new, only recently have efforts been made to comprehensively characterize the I/O problem encompassing various perspectives (for instance, I/O in parallel machines, distributed computing, and mass storage).

I/O Issues in a Multimedia System, pp. 69-74

A.L. Narasimha Reddy and James C. Wyllie

In a multimedia server, disk requests can require constant data rates and guaranteed service. The authors discuss the impact of the real-time nature of I/O requests on various I/O system components as well as the impact of disk scheduling algorithms on the performance of a multimedia system.

This article describes a hybrid scheduling algorithm, Scan-EDF (earliest deadline first), which combines a real-time policy such as EDF with a seek-optimizing policy such as CScan (circular Scan). It then shows how Scan EDF can support a larger number of real-time streams and simultaneously provide better response times to aperiodic requests.

The authors also investigate the impact of buffer space on the maximum number of video streams that can be supported. Then they show that even more streams can be supported by using delayed deadlines and larger requests. Of the two techniques, they prefer delayed deadlines, which provide better response times to aperiodic requests.

When multiple disks are connected to the system through a single bus such as SCSI (for Small Computer Systems Interface), SCSI bus scheduling can add extra delays to individual requests. The authors examine the impact of priority-driven arbitration of a SCSI bus on disk throughput. They then show that deadline extension helps to increase system throughput when multiple disks are connected on a single SCSI bus.

The Alloc Stream Facility: A Redesign of Application-Level Stream I/O, pp. 75-82

Orran Krieger, Michael Stumm, and Ron Unrau

Although the Unix I/O facility is simple and versatile, application programs running under Unix typically do not use its I/O system calls directly. Instead, they use higher level facilities implemented by the programming language or its application-level libraries. Using application-level I/O facilities improves functionality and portability and can also significantly improve application performance.

This articles introduces a new application-level I/O facility called the Alloc Stream Facility. ASF addresses recent computing substrate changes to improve performance, allowing applications to benefit from specific features such as mapped files. It's also designed for parallel systems, maximizing concurrency and reporting errors properly. Because it's modular and object oriented, it supports a variety of popular existing I/O interfaces and can be tuned to a system's behavior, exploiting its strengths while avoiding its weaknesses.

The authors' experiments demonstrate that on a number of standard Unix systems, I/O-intensive applications perform substantially better when linked to ASF instead of the facilities provided — in the best case, up to twice as well. Modifying the applications to use a new interface provided with ASF can improve performance even more.

Container Shipping: Operating System Support for I/O-Intensive Applications, pp. 85-93

Joseph Pasquale, Eric Anderson, and P. Keith Muller

New I/O devices with data rates ranging from 10 to 100 Mbytes per second are becoming available for personal computers and workstations. These include human-interaction devices for video capture and display (and audio record and playback), high-capacity storage devices, and high-speed network communication devices. These devices have enabled I/O-intensive applications for desktop computing that require input, processing, and output of very large amounts of data. This article focuses on an important aspect of operating system support for these applications: efficient transfer of large data objects between the protection domains in which processes and devices reside.

Many operating systems are inefficient in transferring large amounts of data between domains. Most of them, even when they try to avoid physical copying, offer a data-transfer model that assumes a need for complete accessibility to all transferred data. This assumption leads to overheads that can otherwise be avoided. The authors' design for an interdomain transfer facility (which was inspired by the "container-shipping" solution from the cargo-transportation industry) is based on virtual transfers and avoids all unnecessary physical copying.