The Alloc Stream Facility: A Redesign of Application-Level Stream I/O

Orran Krieger, Michael Stumm and Ron Unrau
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada, M5S 1A4

Abstract

This paper describes the design and implementation of a new application level I/O facility, called the *Alloc Stream Facility*. The *Alloc Stream Facility* has several key advantages. First, performance is substantially improved as a result of a) the structure of the facility that allows it to take advantage of system specific features like mapped files, and b) a reduction in data copying and the number of I/O system calls. Second, the facility is designed for multi-threaded applications running on multiprocessors and allows for a high degree of concurrency. Finally, the facility can support a variety of I/O interfaces, including *stdio*, *emulated Unix I/O*, *ASI*, and *C++ streams*, in a way that allows applications to freely intermix calls to the different interfaces, resulting in improved code reusability.

We show that on several Unix workstation platforms the performance of Unix applications using the *Alloc Stream Facility* can be substantially better that when the applications use the original I/O facilities.

1 Introduction

Part of the success of the Unix[1] operating system can be attributed to the design of its input/output (I/O) facility. The primary Unix I/O abstraction is a sequential byte stream, and is provided by an interface consisting of the system calls `open`, `read`, `write`, `seek`, `close`, and `ioctl` (See Figure 1). The Unix I/O facility is simple, easy to use, and has proven to be versatile in that it can be applied in a uniform way to a large variety of I/O services, including disk files, terminals, pipes, networking interfaces and other low-level devices.

Nevertheless, application programs running under Unix typically do not call the Unix I/O system calls directly, but instead use higher-level facilities implemented either by the programming language (e.g. Pascal, Fortran, and Ada) or by application level libraries associated with the language (e.g. Modula, C and C++.) I/O facilities at the application level have several advantages. Their interfaces can be made to match the programming languages, both in terms of 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 as well. Application level I/O facilities can also significantly improve application performance, primarily by buffering the input and output in order to translate multiple fine-grained application-level I/O operations into individual coarser-grained system-level operations. For example, an application may input one character at a time, each serviced from an application-level buffer without the need for a system call; a system-level read to fill the buffer is issued only when the application request cannot be serviced from the buffer, i.e. when it is empty.

open(filename, mode) : opens a stream
read(stream, buffer, nbytes) : reads `nbytes` from the current stream offset into `buffer` and increments the stream offset by `nbytes`
write(stream, buffer, nbytes) : writes `nbytes` from buffer to the current stream offset and increments the stream offset by `nbytes`
lseek(stream, offset, whence) : moves the stream offset to `offset` according to `whence`
fcntl(stream, cmd, arg) : manipulates or locks open stream
ioctl(stream, request, arg) : special control functions
close(stream) : closes `stream`

Figure 1: The Unix I/O system call interface.

The standard I/O library for the C programming language, *stdio*, is an example of a well known application-level I/O facility that is available on numerous operating systems running on almost all current hardware bases [Pla92]. The *stdio* interface (Fig. 2) provides functions that correspond to the Unix I/O interface (`fopen`, `fread`, `fwrite`, `fseek`, `fclose`), functions for character-based I/O (`getc`, `putc`), and functions for formatted I/O (`fprintf`, `fscanf`). The function `ungetc` pushes a byte that was previously read back onto the input

[1] Unix is a trademark of AT&T Bell Laboratories.

General I/O:

fopen(filename, mode) : opens a stream
fread(buf, size, nitems, stream) : reads `nitem` items of size `size` from the current stream offset into `buffer` and increments the stream offset
fwrite(buf, size, nitems, stream) : writes `nitem` items of size `size` from the `buffer` to the current stream offset and increments the stream offset
fseek(stream, offset, whence) : moves the stream offset to `offset` according to `whence`
fflush(stream) : flush buffers
fclose(stream) : close `stream`

String I/O:

fgets(string, nbytes, stream) : reads a line of at most `nbytes` into `string`
fputs(stream, string) : writes `string`

Character I/O:

getc(stream) : reads next character
ungetc(character, stream) : returns last character
putc(stream, character) : writes a character

Formatted I/O:

fprintf(stream, format, args...) : formats `args` according to `format`, writes to `stream`
fscanf(stream, format, args...) : reads from `stream`, formats `args` according to `format`

Figure 2: A subset of the *stdio* interface

stream so that in can be reread later, and is an example of functionality not directly available at the system level.

As is the case with most application level I/O facilities, the interface and implementation of *stdio* have remained largely unchanged since the late seventies. *Stdio* can be found in a rudimentary form in Unix Version 6, which was released in 1976, and is fully present in Unix Version 7, released in 1978. However, computer architecture, hardware technology and operating systems, which we collectively refer to as the *computing substrate*, has changed substantially over the last 10–20 years, and consequently application performance can be significantly improved by adapting both the implementation and interfaces of application level I/O facilities to these changes.

The validity of this claim is demonstrated in Table 1, which shows the execution times of three variants of `diff`, a popular Unix utility that compares the contents of two files [HS77]. The first version corresponds to the standard implementation of `diff`, using the original *stdio* libraries. The second version corresponds to the same implementation of `diff`, but linked to an optimized implementation of *stdio*; we describe this implementation later in the paper. The second version of `diff` runs between 1.2 and 2.1 times as fast as the original on the systems we considered. The third version of `diff` uses a new I/O interface, which we also present later. This version runs between 1.4 and 3.7 times as fast as the original.

The results clearly show that the implementation and the interface of an application-level I/O facility can have a significant effect on overall performance. The performance improvements achieved are primarily due to a reduction in data copying and a reduction in the number of system calls. Not visible in these numbers, is the extra degree of concurrency afforded by the new implementation which will benefit multithreaded and parallel applications.

system	orig. stdio	optimized stdio	(speedup)	new interface	(speedup)
AIX	0.55	0.26	(2.12)	0.15	(3.67)
SunOS	4.70	3.70	(1.27)	3.40	(1.38)
IRIX	1.80	1.25	(1.44)	0.85	(2.12)

Table 1: Measurements, in seconds of `diff` comparing two identical four megabyte files. The "AIX" system is an IBM R6000/350 system with 32 Megabytes of main memory running AIX Version 3.2. The "SunOS" system is a Sun 4/40 with 12 Megabytes of main memory running SunOS version 4.1.1. The "IRIX" system is a SGI Iris 4D/240S with 64 Megabytes of main memory running IRIX System V release 3.3.1.

Others have also recognized the need to modify both the interface and implementation of application level I/O facilities to adapt to changes in the computing substrate. For example, Korn and Vo's *sfio* library is a replacement library for *stdio* that provides a more consistent, powerful and natural interface than *stdio* and uses algorithms that are more efficient than those used by typical *stdio* implementations [KV91].

In this paper we introduce a new application level I/O facility called the *Alloc Stream Facility*. The new facility addresses three primary goals. First, it addresses recent computing substrate changes to improve performance, allowing applications to benefit from specific features like mapped files. Second, it is designed for parallel systems, maximizing concurrency in I/O, and reporting errors properly. Finally, it is modular and object oriented, so that it can easily support a variety of popular I/O interfaces (including *stdio*), and can be tuned to the performance behavior of the system, exploiting a system's strengths, while avoiding its weaknesses.

1.1 Changes in Computing Substrate

It is interesting to consider in more detail the recent changes in the computing substrate. First, the amount of available physical memory has increased by several orders of magnitude. It is not uncommon now to have 64 megabytes of main memory on a personal workstation, a number which can be expected to increase to hundreds of megabytes over the next several years. Because of the large memories, files, once accessed, can be expected to remain cached in memory, so that most operating system I/O calls no longer involve accesses to I/O devices. Many files in fact are created and deleted without ever being written to secondary storage [OD89]. For this

reason, an important component of the cost of I/O is the overhead that stems from copying data from one memory buffer to another and from the cost of making calls to the operating system. This contrasts sharply to, say, a PDP-11 system with 64 KBytes of memory, where only minimal file caching was possible, and the cost of I/O to secondary storage was high enough to dominate the data copying and system call overheads.

Second, processor speeds have improved much more dramatically in recent years than main memory speeds have, requiring the presence of large memory caches. This increases the cost of buffer copying relative to processor speeds, because copying either occurs through the cache, destroying the cache contents, or the copying occurs directly to and from (the relatively slow) memory. Moreover, in today's state-of-the-art computer systems, memory has become a critical contended resource. For example, researchers that have dramatically improved file I/O bandwidth (by introducing disk arrays [PGK88, FPD91]) have found that the memory bottleneck prevents them from effectively exploiting the increased I/O bandwidth [CK91]. The amount of buffer copying therefore has a large impact on performance.

Third, the cost of a system call has also been increasing relative to processor speeds [Ous90]. This is again partially due to the effects of slower relative memory speeds and due to the increased number of registers some modern processors need to save and restore on context switches. But it can also be due to new operating system structures that are becoming more widespread, where the system is controlled by a microkernel and a set of user-level servers. In these systems, the actual operating system (i.e. Unix) is implemented as a server running in an application address space provided by the microkernel [RAA+88, ABB+86, Che88, ALBL91], and a system call is translated into a message or remote procedure call from the invoking program to the server. As a result, it is important to minimize the number of calls to the operating system.

Fourth, many modern operating systems now support mapped files, where a file can be mapped into the application's virtual address space and accessed directly by accessing the region of memory into which the file was mapped. 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. Despite the fact that mapped files have been available for several years and that their use can improve performance, they have not found widespread use. We believe this is because mapped files have not been integrated into the standard I/O interfaces, and that they cannot provide a uniform interface for I/O (i.e. they can only be used for file I/O, but not, for example, for terminal I/O.) In this paper, we show how the performance advantages of mapped file I/O can be exploited, while still supporting a uniform I/O interface.

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 (e.g. event driven systems such as windowing systems). The current I/O interfaces, however, are grossly inadequate for multithreaded programs. An obvious inadequacy of the Unix I/O facility, for example, is the way in which errors are reported to applications: a single global variable *errno* is set by the I/O system whenever an I/O error is encountered [Jon91]. If multiple concurrent I/O operations incur errors, than having a single *errno* variable makes it impossible to distinguish the errors.

A second inadequacy of Unix I/O for multithreaded applications is the way random access I/O is supported. For random access file I/O, *lseek* (or *fseek* in *stdio*) operations are needed to position the *file offset* for subsequent reads or writes to begin from that point. To prevent concurrent *lseek* operations by multiple threads from interfering with each other, it is necessary for a thread to hold a lock from the time of the *lseek* operation until the corresponding I/O operation has completed, effectively serializing I/O operations.

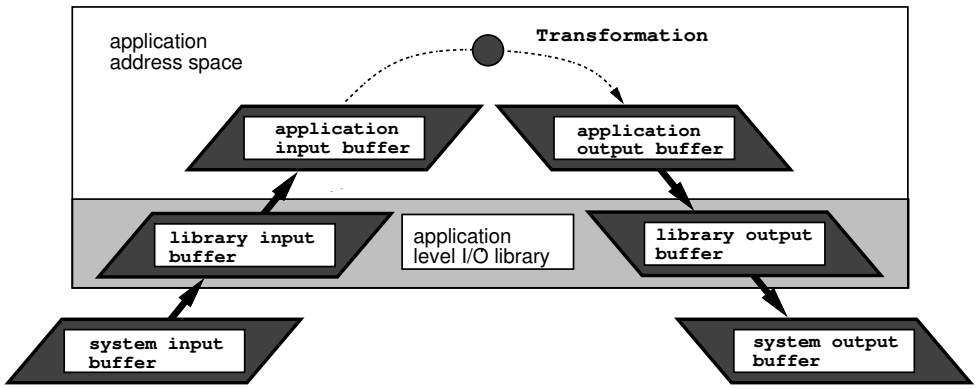
1.2 Reducing I/O Overhead

I/O overhead is an important factor in the performance of many applications. Consider, for example, a typical Unix filter that iteratively reads some input, performs a simple transformation on it, and writes some output [KP84]. Since the transformation is often simple, performance of this type of application is generally dominated by input and output. Figure 3a illustrates the flow of data in a filter implemented using the *stdio* I/O library. The filter is written as a loop, where for each iteration it: 1) calls `fread` to copy data from the *stdio* library input buffer to the application input buffer, 2) transforms the data between the application input buffer and the application output buffer, and then 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.

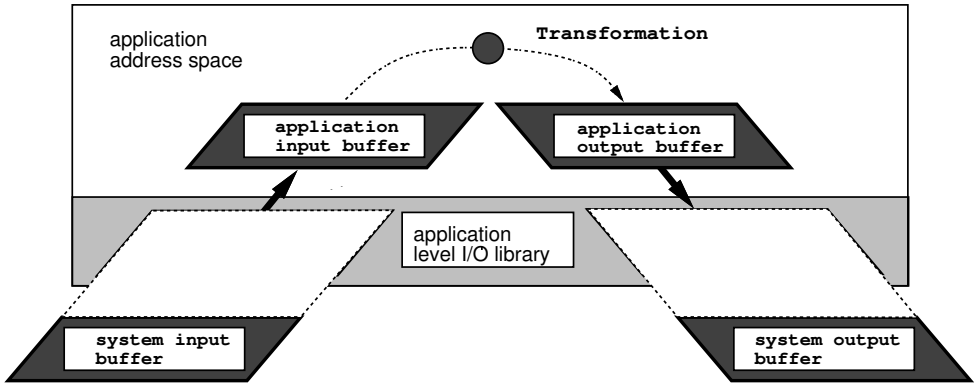
With this simple *stdio* filter, ignoring the transformation, each character in the input stream is copied four times: 1) from the system input buffer to the library input buffer, 2) from the library input buffer to the application input buffer, 3) from the application output buffer to the library output buffer, and 4) from the library output buffer to the system output buffer. Since memory sizes have increased, allowing for more system file buffers and thus less frequent disk I/O, and since data copying has become more expensive, data copying overhead is clearly not insignificant.

Figure 3b and 3c show how for the same application, the *Alloc Stream Facility* can reduce the number of times data

a) filter using typical stdio implementation



b) filter using Alloc Stream Facility implementation of stdio (based on mapped files)



c) filter using Alloc Stream Interface directly

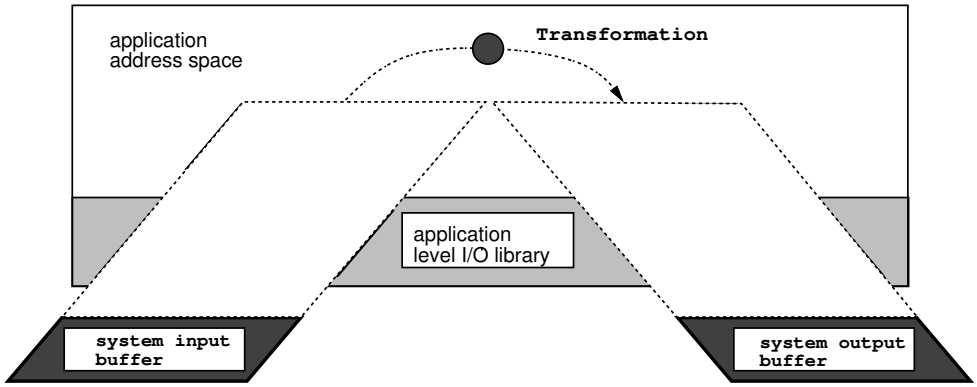


Figure 3: The flow of data for a simple Unix filter that a) uses a typical *stdio* implementation, b) uses the *Alloc Stream Facility* implementation of the *stdio* interface, and c) uses the *Alloc Stream Interface* also supported by the *Alloc Stream Facility*.

Memory allocation operations :

malloc(length) returns pointer : allocates `length` bytes of memory

free(ptr) returns error code : frees previously allocated region

ASI operations :

salloc(stream, length) returns pointer : allocates `length` bytes from `stream`

sfree(stream, ptr) returns error code : frees previously allocated region

Figure 4: Comparing The Standard C Memory Allocation Interface to *ASI*

is copied. Figure 3b illustrates the flow of data when the *Alloc Stream Facility* 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 then 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.

The *Alloc Stream Facility* also supports a new I/O interface called the *Alloc Stream Interface* (ASI). The interface is similar to the *Unix I/O* or *stdio* interfaces in that it is an interface to a character stream that can be used uniformly for all types of I/O, including to disk files, terminals, pipes, networking interfaces and other low-level devices. The key difference between the ASI, and the *Unix I/O* and *stdio* interfaces, is that the application is allowed direct access to the internal buffers of the I/O library instead of having the application specify a buffer into or from which the I/O data is copied.

If an application uses the system buffers directly, then copying can be further reduced. Figure 3c shows the flow of data when the filter is re-written to use *ASI* implemented on top of mapped files. With this version of the filter, the data copying occurs only when the application transforms the data between the system input and output buffer.

In the next section, we describe the *Alloc Stream Interface*. The following section describes the structure of the *Alloc Stream Facility*.

2 The Alloc Stream Interface

The *Alloc Stream Interface* (ASI) is used internally in the *Alloc Stream Facility*, but is also made available to the application programmer. It is modeled after the standard C memory allocation interface, and is thus for C programmers 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 the two memory allocation operations: `malloc`, which allocates a memory region and returns a pointer to that region, and `free`, which releases a previously allocated region (Fig. 4). The arguments to `salloc` and `sfree` differ from the corresponding memory allocation operations in that the *ASI* operations require a `stream` handle to identify the target stream.[2] For both `salloc` and `malloc`, the allocated region is located in the application address space and is managed by an application level library.

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

For a disk file in read mode, only data already in the file can be allocated. (If an application attempts to allocate past the end of file, the length parameter is set to the amount of remaining data and only that data is allocated.) The application should not modify data obtained by `salloc` in read mode.[3]

For a disk file in write mode, if data is allocated past the end of the file, the file length (from the perspective of the application) is automatically extended on `sfree`. This occurs on `sfree` rather than on `salloc` since with multi-threaded applications the data is in a non-deterministic state until the `sfree` has been performed and therefore should not be visible to other threads. The order of writes are guaranteed to be in the same order as the corresponding `salloc` operations even if the `sfree` operations occur in a different order.

Since `salloc` returns a pointer to the library buffers, the library never has to copy data between application and library buffers; instead the application just uses the buffers provided by the library. Also, since the library chooses the address of the data, it can ensure that the alignment allows for such

[2] In the case of `salloc`, the length parameter is passed by reference and on return either indicates the amount of allocated data or holds a negative error code.

[3] The type of error that occurs if the application attempts to modify data obtained by `salloc` in read mode depends on the particular stream module being used. For streams not implemented using mapped files, the modification will have no effect. For mapped file streams, modifications to the buffer results in a memory access protection violation that returns control to the library.

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 error: changes mode of stream to `mode`

Mode variable operations: These operations differ from the corresponding basic operations in that a `mode` argument is used to explicitly specify that the operation is for reading or writing

Salloc(stream, mode, length)
SallocAt(stream, mode, length, offset, whence)

Unlocked operations: These operations have the same arguments as the corresponding basic operations, but differ in that they do not lock the stream.

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 5: The Alloc Stream Interface

optimizations as mapped files. The idea of having the I/O facility choose the location of the I/O data is in itself not new. For record I/O, Cobol and PL/I `read` both provide a *current record* the location of which is determined by the language implementation [Nic75]. Also, the `sfpeek` operation provided by the *sfio* library [KV91] allows applications access to the libraries internal buffers. However, with both the `read` defined by Cobol and PL/I and the `sfpeek` operation defined by *sfio*, the data is only available until the next I/O operation, which is not suitable for multi-threaded applications. In contrast, *ASI* data is available until it is explicitly freed.

ASI was designed from the start to support multi-threaded applications. All *ASI* operations return an error code directly so that it is always clear which thread incurred an error. Also, *ASI* provides the `sallocAt` operation, which atomically moves the stream offset to a particular location and performs a `salloc` at that location. Because this is done atomically, there will be no interference from other concurrently executing threads. Finally, *ASI* allows a high degree of concurrency when accessing a stream. Since data is not copied to or from user buffers, the stream only needs to be locked while the library's internal data structures are being modified, so the stream is locked only for a short period of time.

The full *ASI* interface is shown in Figure 5. `Sopen` and `sclose` are essentially the same as the *stdio* operations `fopen` and `fclose`. An *ASI* stream is always in one of two modes, namely read mode or write mode. If the stream is opened for read-only access then the mode of the stream defaults to read mode; otherwise it defaults to write mode. The mode of the stream can be changed using the *ASI* routine `set_stream_mode`. In addition to the basic *ASI* operations, there are also unlocked and mode variable operations (as shown in Fig. 5) that allow applications greater flexibility at the cost of additional program complexity.

3 The Structure of the Alloc Stream Facility

The *Alloc Stream Facility* is divided into three distinct layers (Fig. 6). At the top layer, *interface modules* implement particular I/O interfaces. These might include modules implementing *stdio*, *C++ streams*, or *emulated Unix I/O* (which provides the same interface as the *Unix I/O* system call interface). The ASI interface is also provided to the application by another interface module. Finally, specialized interface modules are also possible, for example for accessing file system directories.

At the bottom layer, interactions with the underlying operating system and all buffering is managed by stream modules. There typically will be many different stream modules, each optimized for a particular access behavior, stream type, and system.

The top and bottom layer are separated by a *backplane*. The interface provided by the backplane and used by the *interface modules* is *ASI*. The interface provided by the *stream modules* and used by the backplane is a subset of *ASI*, namely the unlocked *ASI* operations. Each stream module can, in addition to the required set of functions (i.e. the unlocked *ASI* subset), provide the backplane with access to a single buffer, called the *current buffer*, that is shared by the stream module and backplane using a simple producer/consumer protocol. When available, this buffer is used to minimize the number of function calls to the stream module; I/O requests from the interface layer are serviced by the backplane directly from the current buffer whenever possible without calling the stream modules. The backplane code is separated from the interface layer, because it would otherwise be common to all interface modules.

ASI was chosen for the backplane interfaces because it minimizes the amount of data copying, and hence allows us to construct our facility in a structured way (i.e. three different layers) without incurring a major performance penalty.

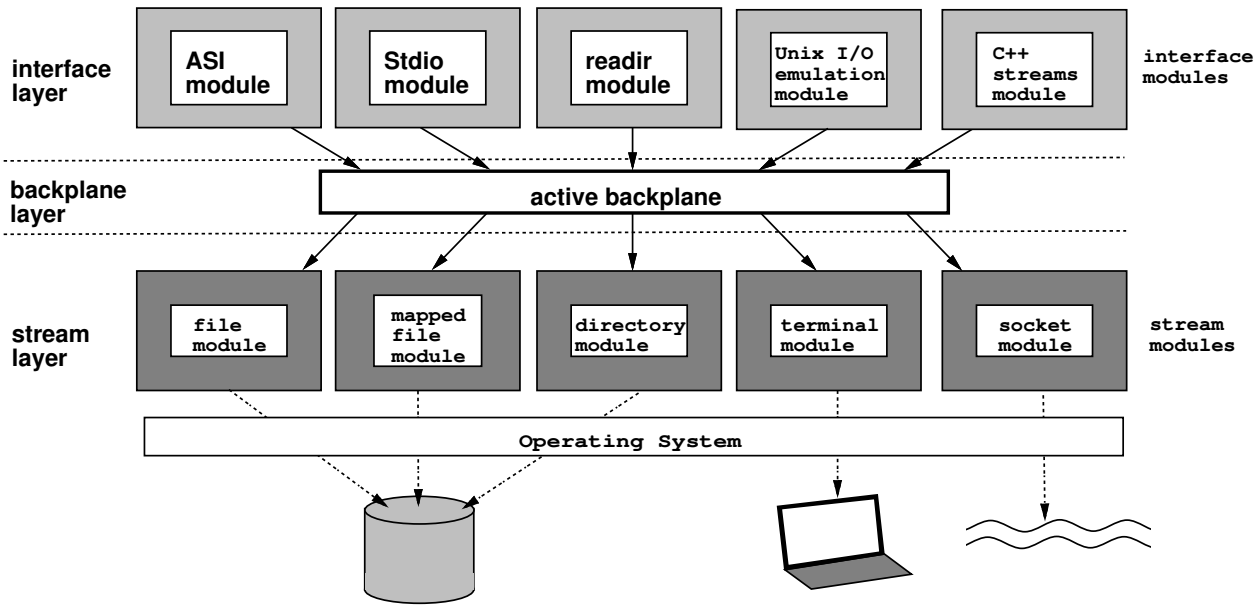


Figure 6: Structure of the Alloc Stream Facility

The modular, object oriented structure of the *Alloc Stream Facility* is important for several reasons. First, since each stream module supports only a single stream type, it can be tuned to support accesses to that stream type in an optimal manner. 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; substantially less checking must be performed in the former case, so the implementation of read-only files is simpler and faster than the implementation of read-write files. In contrast, typical implementations of *stdio* must check the type of stream[4] with an attendant degradation in performance.

Second, because each stream module exports only a small set of functions, it is simple to write a new stream module to adapt to a particular system substrate. We have ported the *Alloc Stream Facility* to a variety of operating systems, including *SunOS* [KW88], *IRIX* [Sil], *AIX* [Mis90], *HP-UX* [WL84], and *Hurricane* [SUK92]. Although most of these systems support some variant of *Unix*, we have found that large improvements in performance can be obtained by adapting the facility to characteristics particular to each system. For example, much of the performance improvements obtained with the systems on which we tested was due to the fact that our implementation is capable of exploiting the particular variant of mapped files provided by each system.

Third, the interface modules are *interoperable* because they do not buffer data; only the stream modules buffer data. The application can therefore freely intermix operations from different interfaces. For example, the application can use the *stdio* operation `fread` to read the first ten bytes of a file and then the *emulated 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 re-writing just the I/O intensive parts of the application to use that interface. Because the different interfaces are interoperable, the *Alloc Stream Interface* appears to the programmer as an extension to the other (already existing) interfaces.

Fourth, because the interface modules do not buffer data, they are generally quite simple. This means that new interfaces can be easily added to the *Alloc Stream Facility*, and conversely this facility can be easily used by new programming languages.

4 Implementation

In this section we describe our implementation of the *Alloc Stream Facility* backplane and then describe some of the interface and stream modules we have provided to date.

4.1 The Backplane

The *Alloc Stream Facility* backplane establishes and manages the communication between top layer interface modules and bottom layer stream modules. A *Client I/O State* (CIOS) data structure (Fig. 7) is maintained for each open stream,

[4] 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.

```
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 */
  } ;
```

Figure 7: The Client I/O State structure

and is shared by the active backplane and the stream modules. In addition to a lock, this structure contains 1) function pointers to stream specific implementations of the unlocked ASI operations, 2) state used to manage the *current buffer* shared by the active backplane and the stream module, and 3) a pointer to stream specific data used only by the stream modules.

The *ASI* `sopen` operation allocates a CIOS entry, determines the stream type, and then calls a stream specific routine to initialize the CIOS structure. The handle returned by `sopen` is actually a pointer to the CIOS structure. `Sclose` calls the corresponding stream specific function to free any stream specific state and then de-allocates the CIOS structure.

Other than `sopen` and `sclose`, the backplane typically implements all *ASI* operations as macros. The code executed by `salloc` and `sfree` are shown in Figure 8.[5] In the common case, when the request can be satisfied by the *current buffer*[6], `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 that 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) simply decrements the reference count to this 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 available a new current buffer for subsequent `salloc` operations.

Other *basic ASI* operations are implemented in the active backplane in a similar fashion. They use the current buffer when possible and call the corresponding function provided by the stream module otherwise.

4.2 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 9 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.

The code shown in Figure 9 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 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: copying data from the library to the applica-

[5] As can be seen from Figure 8, the amount of code executed for `salloc` and `sfree` is very small in the common case. Despite the fact that they provide functionality equivalent to the *stdio* `fread` or `fwrite` functions, they have the complexity of the *stdio* `putc` and `getc` macros (discounting the acquisition and release of the lock).

[6] Special cases like unbuffered streams are supported by forcing the `bufcnt`, `bufptr` and `bufbase` fields to zero. Hence, each call to salloc results in a call to the corresponding stream specific function.

```
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 8: Code for salloc and sfree.

tion buffer is performed with the stream unlocked, allowing for a greater degree of concurrency than if the stream had to be locked for the entire read operation.

The CIOS structures in the backplane layer are organized as an array, so that *emulated Unix I/O* operations can refer to the same streams as *ASI* operations by indexing into the array. Functions are provided to translate CIOS pointers to array indexes and vice versa.

The implementation of *stdio* is very similar to that of *emulated Unix I/O*. The `stdio.h` file that is included with the library defines the `_iobuf` structure as a `CIOS` structure, so that a *stdio* `FILE` pointer is actually a pointer to the corresponding `CIOS` structure and no conversion between *ASI* and *stdio* streams is necessary. Our implementation of the *stdio* interface is fully source code compatible with other implementations. However, the application writer should take care when using *stdio* operations that constrain buffering such as `setbuf`. These operations are typically used to increase the performance of I/O, but with our implementation, `setbuf` actually hurts performance, because it increases the amount of copying.

4.3 Stream Modules

Although there are several types of streams, for brevity we will focus only on file streams. Depending on the system substrate, file streams can be implemented in different ways.

```
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 9: Read implemented using ASI

For example, we have three different implementations: one for systems where `mmap` is the fastest way of accessing a file, a second for systems that either do not support mapped files or where the `read` and `write` system calls are faster than access to mapped files, and a third for `AIX` systems where another mapped file facility exists that performs better than `mmap`. This variety in the implementation of stream types illustrates an important aspect of our facility: a new stream module can easily be added to take advantage of specific characteristics of a particular system without requiring changes to other part of the facility. Hence, the *Alloc Stream Facility* can provide a degree of performance portability not easily available in, for example, a traditional *stdio* implementation.

The following sections describe two implementations of a file stream module, one based on *Unix I/O* system calls and one based on mapped files.

4.3.1 Unix I/O-based stream modules

The file stream module implementation based on *Unix I/O* system calls is similar to traditional *stdio* implementations in that a single buffer (the current buffer) is used for the next access to the stream. However, in addition to this buffer, the stream module must keep a list of (old current) buffers for which there are still outstanding `salloc` operations. It associates with each of these buffers the file offset (for that buffer) and a reference count of outstanding `sallocs`.

The read-only stream specific implementation of `salloc`: 1) allocates a new buffer that is (at minimum) large enough to satisfy the request, 2) copies the portion of un-accessed data from the (old) current buffer to the new buffer, 3) performs a read operation to the operating system for the remainder of the data, and 4) updates the CIOS state accordingly. If the reference count for the (old) current buffer is zero then it is removed from the list of regions and deallocated.[7] The stream

[7] As an optimization, if the current buffer is large enough, it is used

specific implementation of `sfree` locates the buffer that contains the data being freed and decrements the reference count of that buffer. If the reference count is zero then the buffer is removed from the list of regions and deallocated.

The write-only stream specific implementations of `salloc` and `sfree` differ from the read-only implementations in that no data is read into a buffer and that a buffer must first be written to a file before it is discarded. In order to ensure that modifications appear in the file in the same order as `salloc` operations, a buffer is not written to the file until all previous buffers have also been written. The file offset associated with each buffer is used to allow the buffer contents to be written out to the correct file location regardless of any intervening operations.

In the case of a read-write stream, a separate current buffer and a separate list of buffers is maintained for each mode. Whenever the mode is changed, the CIOS structure is modified to refer to the other current buffer. The implementation ensures that the write buffers and read buffers are kept consistent. This can be accomplished by invalidating any read buffers that contain modified file data when a region allocated for writing is freed. In the case of outstanding `salloc` operations (i.e. the read buffer has a reference count greater than zero), the invalidation is deferred until the corresponding `sfree` operations are called.

4.3.2 Mapped file-based stream modules

In this section, we describe stream modules based on the `mmap/munmap` mapped file interface [JCF+83], a variant of which is supported by many versions of Unix. The `mmap` system call takes as parameters the file number, protection flags, the length of the region to be mapped, and an offset into a file. It maps the specified file region into the requesters address space and returns a memory pointer to the mapped region. Memory accesses to the mapped region then directly access the corresponding file data.

The `mmap` stream module maps file regions into the application address space where they are treated in the same way as the Unix I/O file buffers described in the previous section. The `mmap` module, however, only uses a single list of regions for both input and output, and does not need to inform the operating system that a page has been modified, since the memory management system will detect this on its own in a system dependent way. These two differences make the mmap module much simpler than the Unix I/O modules.

A difficulty with the implementation of the mapped file module on many systems is that when a page past the end of file (EOF) is accessed, the system increases the file length to include the entire new page,[8] thus incorrectly identifying the amount of valid data in the file. To correctly handle files accessed concurrently by multiple applications, the mapped file module can be configured to use regular *Unix I/O* write operations to write data past EOF. Note, however, that the EOF problem is an artifact of current mapped file interfaces, and is not a generic problem with mapped files per se. For example, the file length and the number of pages in the file are independent in the Hurricane operating system [SUK92], so the file length need not be extended when the application uses `salloc` past the end of the file. Instead, the file length is extended after the application has completed the modification and has called `sfree`.[9]

Others have also recognized the potential 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 search time of the file cache [BRW89]. The Mach 3.0 operating system [DA92] reflects Unix I/O calls back to the application level where they are serviced using mapped files. Finally, the *sfio* library [KV91] uses mapped files whenever possible.

Whether to use mapped file-based stream modules or Unix I/O-based modules depends on the performance behavior of the system, and the mode of access being used. For example, using mapped files for accessing files in read mode is always more efficient on all systems we tested. The same is true for files accessed in read-write mode as long as the file is not extended. However, on some (but not all) systems, Unix I/O can be more efficient than writing to mapped regions if the file is being extended. The appendix presents the performance of basic system operations on a number of systems. On AIX systems the Alloc Stream Facility is typically configured to access files using a stream module based on the AIX `shmget` interface. With the AIX definition of `shmget`, the entire file can be mapped into a continuous portion of the application address space in such a way that no page faults occur when data already in the system file cache is accessed, with an attendant reduction in I/O overhead.

5 Performance

In this section, we compare the performance of the *Alloc Stream Facility* against the original *stdio* and *Unix I/O* facilities already available on three systems: an IBM R6000/350 running AIX, a Sun 4/40 running SunOS, and an SGI Iris

as the new current buffer instead of being deallocated.

[8] Both AIX and IRIX support options that causes the file to grow every time a page past the current EOF is modified. On other systems (e.g. SunOS), if a page past EOF is accessed, a segment fault results. On these systems, the mapped file module uses `ftruncate` to change the file length before a `salloc` past EOF can proceed.

[9] In our implementation, the file length is actually extended after a region is deallocated. This is a compromise between reducing the number of system calls and making the data available to other applications as soon as possible. If the implementation informs the file system that the file length has changed when `sfree` is called then other applications can see this data immediately. If the file system, on the other hand, is informed when the file is closed then the overhead is minimized, but the data does not become available until the file is closed.

	AIX		SunOS		IRIX	
appl.	stdio	ASI	stdio	ASI	stdio	ASI
diff	2.12	3.67	1.27	1.38	1.44	2.12
compress	1.06	1.06	1.07	1.07	1.11	1.11
uncomp.	1.22	1.17	1.00	1.03	1.24	1.43
cut	1.06	1.09	1.06	1.19	1.04	1.22

Table 2: Speedup of stdio 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.

4D/240S running *IRIX*. To make this comparison, we measured the time to execute:

1. some programs that use *stdio* operations and Unix system calls for I/O linked to the originally installed facilities on each system,
2. the same programs, with no source code modifications, linked to the *Alloc Stream Facility* (so they use the *stdio* and *emulated Unix I/O* interfaces), and
3. the same programs source modified to use the *ASI* interface directly.[10]

Each system on which we performed these experiments had its own *standard* configuration of the *Alloc Stream Facility*. 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 uses *Unix I/O* operations for output.

We present the results of all experiments in this section in terms of speedup relative to the same program linked to the installed facilities on the same machine. For these experiments all input and output is directed to files. The experiments were performed many times and, in order to subtract away the impact of disk I/O and other applications, the best measures obtained are presented.

5.1 *Stdio* Applications

Table 2 shows the performance of four *stdio*-based programs. The left side of each column shows that these programs linked to the *Alloc Stream Facility* performed at least as well as those linked to the installed system libraries, and in some cases significantly better. For example, on the AIX system, `diff` runs in less then half the time when linked to the *Alloc Stream Facility*. The major reason for this advantage is the use of mapped files.

[10] It is interesting to note that only minor changes to the programs were necessary to adapt them to ASI, typically affecting fewer than 10 lines of code.

	AIX		SunOS		IRIX	
appl.	Unix I/O	ASI	Unix I/O	ASI	Unix I/O	ASI
cmp	1.14	1.26	0.91	0.94	1.05	1.29
wc	1.12	1.20	1.23	1.42	1.03	1.17
cat	1.13	1.41	1.00	1.13	0.89	1.09
cp	1.87	2.42	1.00	1.16	0.89	1.06

Table 3: Speedup of Unix I/O applications that are 1) linked to the *Alloc Stream Facility* and 2) modified to use *ASI*. `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; `cp` copies one file to another.

The right side of each column shows that the performance of most of the programs improve further when modified to directly use the *ASI* interface. For example, on the AIX system, `diff` improved by a further 40% (from the unmodified program linked to our facility), to be 3.67 times as fast as the original program linked to the installed *stdio*. This performance gain is due to the fact that data does not have to be copied to (or from) application buffers.

5.2 *Unix I/O* Applications

The *Unix I/O* interface is specific to the Unix operating system and for portability reasons it is generally considered poor programming practice to use this interface directly. However, I/O intensive Unix programs that do large grain I/O often use the *Unix I/O* interface directly, because *stdio* entails an extra level of copying.

Table 3 shows the performance of four *Unix I/O*-based programs. The left side of each column shows that unmodified Unix I/O applications linked to the *Alloc Stream Facility* often perform better than when they use the *Unix I/O* system calls directly. For example, on the AIX system, `cp` almost doubled in speed when linked to the *Alloc Stream Facility*. This means that on some systems the Unix I/O interface is more efficiently implemented by our application level library rather than by the operating system, which is a surprising result. Moreover, the *Unix I/O* applications improved further when modified to use the *Alloc Stream Interface*, as shown in the right side of each column. For example, `cp` modified to use *ASI* ran two and a half times as fast as the original.

6 Conclusions

We have described the design and implementation of a new application level I/O facility, the *Alloc Stream Facility*. Its modular, object-oriented structure allows for much flexibility in providing a variety of I/O interfaces and in exploiting the performance potential of individual systems. Its new I/O interface, the *Alloc Stream Interface* (ASI), which is used internally but also made available for direct use by applications, substantially reduces copying overhead and maximizes con-

currency. As a result, the *Alloc Stream Facility* has three key advantages:

Performance: We have shown that many *stdio* and even *Unix I/O* applications demonstrate improved performance when they are linked to the *Alloc Stream Facility*. Application performance is further improved when they are modified to use the *ASI* interface directly. These performance improvements are a direct result of 1) the structure of the facility that allows it to take advantage of system specific features like mapped files, and 2) the reduction in I/O overhead implied by the definition of the *ASI* interface.

Concurrency: Both the *Alloc Stream Facility* and ASI were designed for multi-threaded applications running on multiprocessors. Concurrency is maximized by minimizing the amount of time streams are locked; the stream is unlocked while data is being copied. Moreover, all *ASI* operations return an error code directly so that it is always clear which thread incurred an error. Finally, the *ASI* `sallocAt` operation allows a thread to allocate data from a particular location in a file without interference from other threads.

Functionality: The structure of the *Alloc Stream Facility* allows it to support a variety of I/O interfaces (e.g. *stdio*, *emulated Unix I/O*, *ASI*, *C++ streams*, etc.). Applications can freely intermix calls to the different interfaces, resulting in improved code reusability. Moreover, it also means that programmers can exploit the performance advantages of *ASI* by modifying just the I/O intensive portions of their applications.

6.1 Future Work

As future work, we intend to explore the effectiveness of the *Alloc Stream Facility* on systems with parallel disks, for which we think it is particularly appropriate. Chervenak and Katz [CK91] found that the performance advantage of parallel disks was severely limited by the overhead of data copying in memory. We believe that the *Alloc Stream Facility* would allow better exploitation of parallel disks (especially on multiprocessor systems) not only because it reduces data copying substantially, but also because of the higher degree of concurrency *ASI* naturally allows.

We also are experimenting with a new implementation of pipes that exploits shared memory.[11] This shared memory is managed as a circular buffer, with a single producer using *ASI* operations to allocate portions of the buffer into which data can be placed, and a single consumer using *ASI* operations to allocate portions of the buffer containing valid data. We expect that the performance of this implementation will improve relative to implementations where the operating system buffers the pipe data, because less data copying and fewer systems calls are required. Because pipes implemented this way do not cause page faults, we expect these advantages to be even greater than those achieved by using mapped file I/O for disk file accesses.

Finally, we are extending the *Alloc Stream Facility* to allow *protocols* to be *pushed* onto existing streams, where a pushed protocol would effectively redefine the set of basic *ASI* functions (i.e.`u_salloc`, `u_sfree`, `u_srealloc` and `u_sallocAt`) for that particular stream. (This would be similar to the push and pop operations of the System V kernel streams [Rit84].) For example, an application could push a user defined decompression algorithm onto a compressed stream, so that the stream can subsequently be accessed with the regular I/O interfaces.

A To Map, or not to Map

Despite the advantages described earlier, in practice mapped file I/O does not always result in better performance than *Unix I/O*. For example, the first access to a new mapped page (e.g. past EOF) will cause a page fault that will return a zero filled page. If this page is zero-filled when the access occurs, then mapped file I/O typically performs worse than *Unix I/O*.[12]

To determine on what systems and under what conditions mapped files should be used, we developed a series of experiments (Tables 4 to 7) to measure basic system costs on the same systems described in Table 1. The AIX system, in addition to the `mmap` interface, provides the `shmget` interface for mapping files into the application address space. The AIX implementation of `shmget` results in no page fault costs when data already in the system file cache is accessed, and hence it can results in very high performance. For the AIX system, the performance of both `mmap` and `shmget` is compared to the performance of *Unix I/O*. For the other systems, the performance of `mmap` is compared to the performance of *Unix I/O*.

As can be seen from Table 5, the cost of accessing a page of a mapped file is on all three systems substantially less than the cost of reading a page size buffer. For example, on both the AIX and IRIX systems, the cost of reading a page is over 3.75 times the cost of touching a page of a file mapped with `mmap`. (The difference on SunOS, at 2.32, is also quite respectable.) When `shmget` is used on the AIX system, the cost of accessing a page is zero, since no page faults are incurred once the file is mapped in.

[11] Note, our implementation of pipes depends on the applications on both ends of the pipe being linked to our facility, hence it is an open question whether it can be made compatible with existing unix binaries.

[12] One way many operating systems reduce this expense is to maintain a list of pre-zeroed pages. This cost can be entirely eliminated if the operating system allows the application to map files such that pages are not pre-zeroed.

	AIX	SunOS	IRIX
copy	0.09	0.56	0.44
copy to buf	0.07	0.59	0.30
copy from buf	0.07	0.59	0.30

Table 4: Measurements of the cost of copying data in milliseconds per page. These three tests repeatedly copy four megabytes of data. With the *copy* test, the data is copied between two four megabyte buffers. With the *copy to buf* test, the data is copied from a four megabyte buffer, in page size chunks, to a single page size buffer. With the *copy from buf* test, the data is copied from a page size buffer to a four megabyte buffer (in chunks of a page size).

	AIX		SunOS	IRIX
	shmget	mmap	mmap	mmap
page touch (r)	0.00	0.04	0.56	0.14
read page	0.15		1.30	0.53

Table 5: Measurements of input using mapped files and Unix I/O in milliseconds per page. The *page touch (r)* test repeatedly maps in a four megabyte file, touches each page in the file, and then un-maps the file. The *read page* test repeatedly uses `lseek` to move the file offset to the beginning of the file, and then reads each page of the file into a single page aligned buffer.

We compare the cost of `read` to the cost of touching a mapped page, because each brings a page into the application address space. They are not entirely equivalent, however, because the data (in the page) probably resides in the processor cache after the `read`. It is difficult to determine how great an impact this has on application performance, since it depends both on the particular machine architecture (e.g. the size of the cache) and the data access pattern of the application. However, on all three systems, the cost of a `read` operation is greater than the combined cost of touching a mapped page and copying a page sized buffer (Table 4). As a result, it is always cheaper to use mapped files for accessing files in read-only mode.

Table 6 shows that the cost of modifying a page using mapped files is substantially less than the cost of a `write` operation. In fact, on SunOS and AIX, using mapped files for modifying data already in a file results in an even greater performance advantage than using mapped files for input.

Table 7 shows that on the AIX system there is a significant advantage in using mapped files for adding new pages to a file. In fact, using mapped files in this case results in an even larger improvement than for the two previous cases. However, on the IRIX system using mapped files past the EOF is much more expensive than using `write` operations.

	AIX		SunOS	IRIX
	shmget	mmap	mmap	mmap
page touch (w)	0.00	0.04	0.54	0.19
write page	2.78		3.31	0.46

Table 6: Measurements of modifying a mapped file versus writing that file in milliseconds per page. The *page touch (w)* test repeatedly maps a four megabyte file, modifies a single character in each page, and then unmaps the file. The *write page* test repeatedly uses `lseek` to move the file offset to the beginning of the file, and then writes each page of the file from a single page aligned buffer.

	AIX		SunOS	IRIX
	shmget	mmap	mmap	mmap
page touch (w)	0.21	0.27	1.49	0.73
write page	2.93		3.00	0.68

Table 7: Measurements of adding new pages to a file using mapped files and using `write` operations in milliseconds per page. The *page touch (w)* and *write page* tests are similar to the previous experiment, except that before each iteration `ftruncate` is used to first change the length of the file to 0 (freeing all pages in the file) and then (except on the AIX system where the file grows automatically every time a page past the EOF is touched) to change the file length to four megabytes.

References

[ABB+86] M. Accenta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for Unix development. In *Proceedings of the USENIX 1986 Summer Conference*, pages 93–112, 1986.

[ALBL91] T. Anderson, H. Levy, B. Bershad, and E. Lazowska. The Interaction of Architecture and Operating System Design. In *Fourth International confernece on Architectural Support for Programming Languages and Operating Systems*, pages 108–119, 1991.

[BRW89] A. Braunstein, M. Riley, and J. Wilkes. Improving the efficiency of UNIX file buffer caches. In *Proceedings of the twelfth ACM Symposium on Operating System Principles*, 1989.

[Che88] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[CK91] Ann L. Chervenak and Randy H. Katz. Performance of a Disk Array Prototype. In *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 188–197, 1991.

[DA92] R. Dean and F. Armand. Data Movement in Kernelized Systems. In *Usenix Workshop on Microkernels and Other Kernel Architectures*, 1992.

[FPD91] J. French, T. Pratt, and M. Das. Performance Measurement of a Parallel Input/Output System for the Intel iPSC/2 Hypercube. In *ACM Sigmetrics Conference*, pages 178–187, 1991.

[HS77] J. Hunt and T. Szymanski. A flast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[JCF+83] W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher. 4.2BSD System Manual. 1983.

[Jon91] M. Jones. Bringing the C Libraries With Us into a Multi-Threaded Future. In *USENIX-Winter 91*, pages 81–91, 1991.

[KP84] B. Kernighan and R. Pike. *The Unix programming environment.* Prentice-Hall, 1984.

[KV91] D. Korn and K.-Phong Vo. SFIO: Safe/Fast String/File I/O. In *USENIX-Summer'91-Nashville, TN*, 1991.

[KW88] S. Kleiman and D. Williams. SunOS on SPARC. *Sun Technology*, Summer 1988.

[Mis90] M. Misra, editor. *IBM RISC System/6000 Technology*, volume SA23-2619. IBM, 1990.

[Nic75] J. Nicholls. *The Structure and Design of Programming Languages*, chapter 11, pages 443–446. Addison-Wesley, 1975.

[OD89] J. Ousterhout and F. Douglis. Beating the I/O Bottleneck: A Case for Log-Structured File Systems. *ACM press, Operating Systems Review*, 23(1):11–28, 1989.

[Ous90] J. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer USENIX Conference*, pages 247–256, June 1990.

[PGK88] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAIDS). In *ACM SIGMOD Conference*, pages 109–116, Chicago, Illinois, June 1988.

[Pla92] P.J. Plauger. *The Standard C Library.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[RAA+88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. CHORUS Distributed Operating Systems. *The Usenix Association Computing Systems Journal*, 1(4):305–370, December 1988.

[Rit84] D. Ritchie. A Stream Intput-Output System. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[Sil] Silicon Graphics, Inc., Mountain View, California. *IRIX Programmer's Reference Manual.*

[SUK92] M. Stumm, R. Unrau, and O. Krieger. Designing a Scalable Operating System for Shared Memory Multiprocessors. In *Usenix Workshop on Microkernels and Other Kernel Architectures*, 1992.

[WL84] S. Wang and J. Lindberg. HP-UX: implementation of Unix on the HP 9000 series 500 computer systems. *Hewlett-Packard Journal*, 35(3):7–15, March 1984.