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

Orran Krieger, Michael Stumm and Ron Unrau

Technical Report CSRI-275
January, 1993
Revised January, 1994

Computer Systems Research Institute
University of Toronto
Toronto, Canada M5S 1A4

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

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
okrieg@eecg.toronto.edu

Abstract

This paper introduces a new application level I/O facility called the *Alloc Stream Facility* (ASF). ASF 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 re-usability.

We show that on several Unix workstation platforms, I/O intensive applications perform substantially better when linked to ASF instead of the native facilities – in the best case, up to twice as good. Modifying the applications to use a new interface provided with ASF can improve performance even more.

1 Introduction

The success of the Unix operating system is partly attributable 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 [Che87]. 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++.)

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.

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.

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*

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

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 stream so that it 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 [Ste92]. However, computer architecture, hardware technology and operating systems, which we collectively refer to as the *computing substrate*, have 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.

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.

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 multi-threaded and parallel applications.

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]) 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 [RAA+88, ABB+86, Che88, ALBL91]. 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, 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:

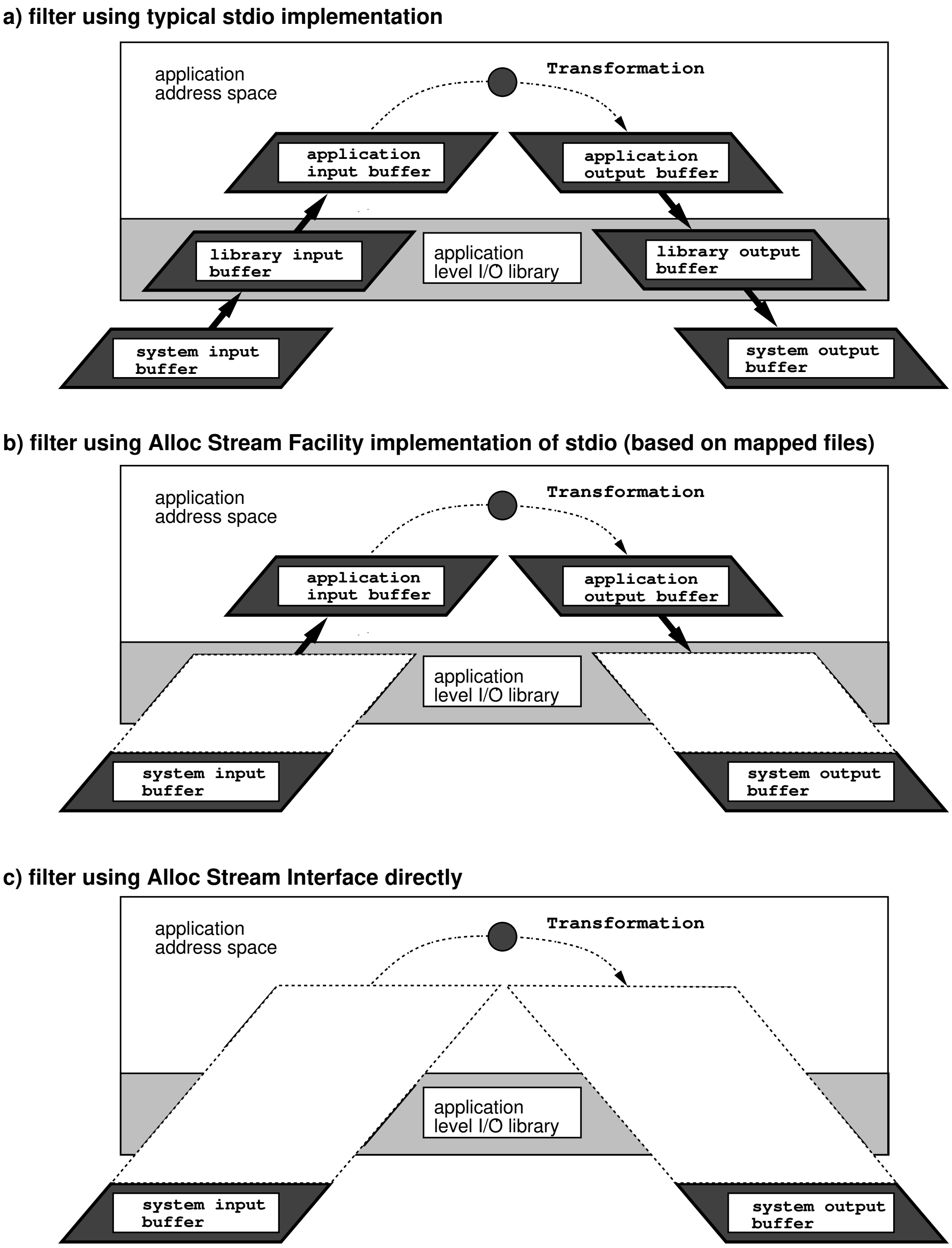

Figure 3: The flow of data for a simple Unix filter that a) uses a typical *stdio* implementation, b) uses the *Alloc Stream Facility* implementation of the *stdio* interface, and c) uses the *Alloc Stream Interface* also supported by the *Alloc Stream Facility*.

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 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.

Many *stdio* filters either have the same data flow as the simple filter we have shown, or use formatted *stdio* operations (e.g. `printf`) to transform data between an application buffer and the *stdio* output buffer. In the latter case, re-writing the filter to use *ASI* still eliminates the data copy between the *stdio* input buffer and the application buffer. Examples of Unix filters that follow one or the other of these models include: `eqn`, `cut`, `wc`, `pr`, `sort`, `uniq`, `uudecode`, `uuencode` and `cat`. We have modified a number of filters to use *ASI*, and have found the modification to be generally trivial (i.e. only 2 to 10 lines needed to be changed in most cases), and performance generally improved.

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

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.[1] 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

[1] 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 (in which case the pointer returned by `salloc` is NULL).

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

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.[2]

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.[3]

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`.

`SallocAt` allows the application to allocate data from a particular location in a stream. The `whence` field indicates if the `offset` is absolute, relative to the current offset, or relative to the end of file.

`Srealloc` allows an application to shrink or grow the most recently allocated region. Library buffers are a convenient location to prepare data for writing and `srealloc` is used when the application does not know *a priori* the amount of data to be output: the application can `salloc` a large data space and then shrink the region back to the actual amount of data prepared. `Srealloc` repositions the offset in the stream. For example, if `salloc` allocated bytes 1-200 of a file, and `realloc` shrinks the region to bytes 1-20, then the stream offset for the next access will be at byte 21 of the file.

In addition to the basic *ASI* operations, there are also unlocked and mode variable operations that allow applications greater flexibility at the cost of additional program complexity. For example, the operations `u_salloc`, `u_sfree`, `u_srealloc` and `u_sallocAt` have the same parameters as the corresponding operations in the high level interface, but differ in that they do not lock the stream. The (capitalized) `Salloc`, `Srealloc` and `SallocAt` operations are mode variable variants of the corresponding (un-capitalized) high level operations in that the (read/write) mode is specified on each call, instead of switching between stream modes with `set_alloc_mode`.

Advantages of ASI

The most important advantage of ASI over other interfaces is that the buffer used for I/O is chosen by the library and not the application. This eliminates the need to copy buffers and lets the library exploit existing alignments, allowing for such optimizations as mapped files. The low overhead of ASI allows other I/O interfaces to be efficiently implemented above it. For example, we have implemented the *stdio* interface above ASI and have found that our implementation has bet-

[2] 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.

[3] The one exception is unbuffered read/write streams, where data is written in the order of `sfree` rather than `salloc` operations.

ter performance than most direct implementations of this interfaces.

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, for 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 multithreaded 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. (As described earlier, a major problem with the *Unix I/O* and *stdio* interfaces is that they provide no such function.) 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.

A final advantage of ASI over other interfaces is that it is optimized for uni-directional accesses (i.e. read-only or write-only accesses). From our experience, the vast majority of streams are used in a read-only or write-only fashion. (Also, even for streams open for read/write access, most applications do not frequently interleave read and write accesses.) ASF (and other application level libraries like *stdio*) must execute special code whenever the application switches between reading from and writing to a stream.[4] Hence, with interfaces like *stdio* or *Unix I/O*, where the mode of the access is specified by the function, (i.e. **read** is for read access, and **write** is for write access), the library must check if the previous operation was of the same mode. In contrast, if an application chooses to use the basic ASI operations (rather than the mode variable operations) it is agreeing to tell the library explicitly when it is changing the mode of a stream, and hence no checking is necessary.

[4]For example, depending on the type of the stream, it may have to modify the current offset in the buffer, invalidate some buffers, acquire a different set of locks, etc.

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* layer. This layer provides code that would otherwise be common to the different stream modules.

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.

The interface provided by the backplane and used by the *interface modules* is *ASI*. The majority of backplane code is implemented as "C" macros[5] that, whenever possible, use the current buffer provided by the stream modules to minimize the number of function calls to the stream modules.

Our approach is based on two observations. First, the most used stream modules buffer data. Second, code that is independent of the particular stream modules can be used for all performance critical streams[6] to access buffered data. Hence, we designed the *stream modules* to (optionally) export to the higher levels a single buffer, and implemented the stream independent code as macros in the backplane layer.

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.

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

[5]With a macro, the code is inserted by a compiler pre processor wherever the macro is called. This avoids the overhead of a procedure call and provides opportunities for compiler optimizations.

[6]In this context, performance critical means streams that can have high performance (e.g., file I/O and not terminal I/O).

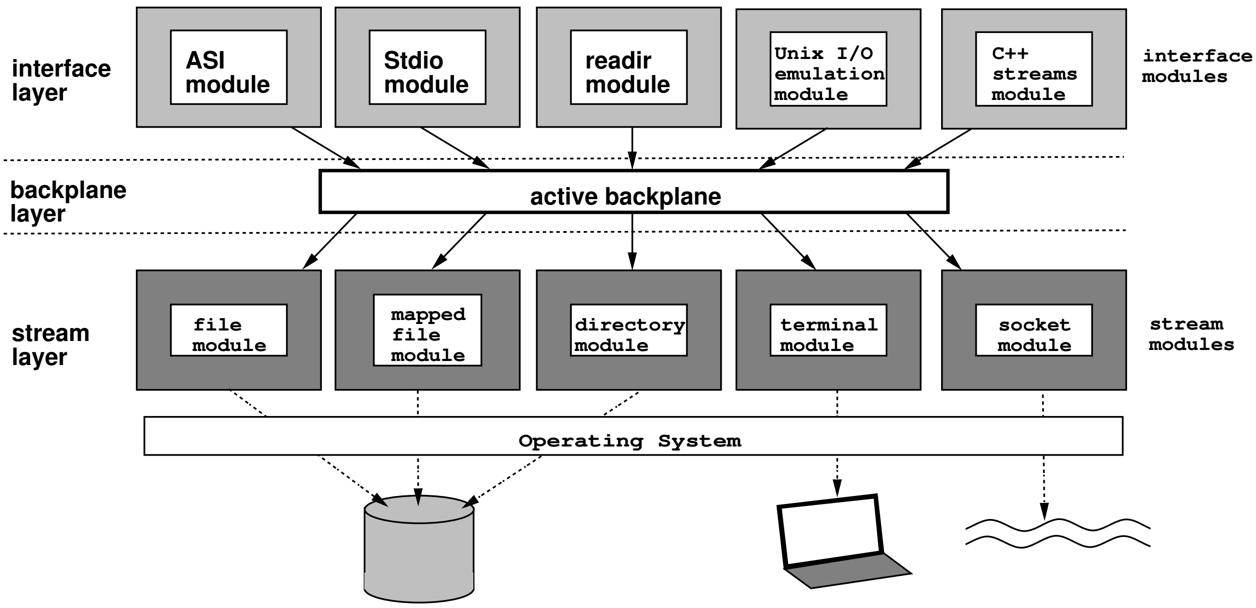

Figure 6: Structure of the Alloc Stream Facility

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[7] 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 [USK93]. 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.

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.

[7] 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.

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. This section concludes with a short description of the limitations of our current implementation.

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, 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

```
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

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` is shown in Figure 8. As this figure shows, 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).

In the common case, when the request can be satisfied by the *current buffer*[8], `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. The unlocked *ASI* operations (e.g. `u_salloc`) differ only in that they do not acquire locks. The mode variable operations (e.g. `Salloc`) differ in that (after acquiring the lock), they first check whether the buffer type corresponds to the type of the request and, if not, call `set_stream_mode` to change the stream mode before executing the *basic ASI* code.

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.

[8] 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.`

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 application 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.

An array of pointers to CIOS structures is maintained by the backplane, 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 source code compatible with other implementations. However, the application writer should take care

```
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`

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

We have implemented a large number of stream modules. For example, modules may be specific to: *i*) an access mode (i.e. read, write, and read/write), *ii*) a buffering policy, *iii*) a particular I/O service (e.g. disk files, pipes, and sockets), *iv*) a particular access pattern (e.g. sequential, or random) and *v*) a particular operating system and hardware platform. We have found that using stream modules optimized for both the application and system can result in substantial performance advantages.

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 is the fastest way to 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 (based on `shmget`) exists that performs better than `mmap`.

The following sections describe how file stream modules are implemented when they are based on *Unix I/O* system calls and how they are implemented when 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.[9] The stream 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.

On many systems, stream modules based on mapped file I/O deliver better performance than stream modules based on *Unix I/O*. The use of mapped file I/O reduces both copying and system call overhead. Copying overhead is reduced because the library, rather than using private buffers in the application address space, uses the system buffers. System call overhead is reduced because the library requests larger portions of a file when mapped file I/O is used, since with mapped file I/O no I/O cost is incurred until the data is accessed. In contrast, stream modules based on *Unix I/O* must be pessimistic, since the I/O cost of `read` is directly proportional to the amount of data requested.

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.

On some systems, Unix I/O can be more efficient than mapped file I/O for some types of access (e.g., for writing past the EOF). The appendix discusses the tradeoffs of using mapped file I/O on a number of systems.

4.4 Limitations and improvements

The current implementation of ASF for Unix is a proof of concept prototype.[10] We have not tried to make our implementation of *stdio* binary compatible with existing versions on the different Unix platforms. Also, for some operations (e.g., *setbuf*) more work must be done for full source code compatibility. Many optimizations are still possible, especially for the *Unix I/O* based stream modules. In this section we describe a number of problems with the existing implementation of ASF and ways it could be improved to address these problems.

A result that surprised us was that on some systems some applications perform worse when they use the ASI rather than the *stdio* interface of ASF, even though the *stdio* interface is implemented above the ASI interface and involves more over-

[9] As an optimization, if the current buffer is large enough, it is used as the new current buffer instead of being deallocated.

[10] The version of the ASF library that runs on the HURRICANE operating system is in daily use by all users of that system.

head. The reason is that *stdio* operations (e.g., `fread`) transfer data to or from a buffer with an alignment known by the compiler, (while `salloc` returns a pointer with an arbitrary alignment) and some compilers are less effective at performing optimizations (e.g., unrolling loops) if the alignment of a buffer is not known at compile time. We plan in the future to extend the ASI interface to address this problem. In particular, applications will make requests to a `salloc` variant that guarantees a particular alignment, and the library will do whatever copying is necessary to enforce this alignment. The overhead should be low as long as the buffer already has the requested alignment.

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,[11] thus incorrectly identifying the amount of valid data in the file. To correctly handle files accessed concurrently by multiple applications, the mapped file stream module could be configured to use regular *Unix I/O* write operations to write data past EOF (our current implementation does not do this). 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 [USK93], 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`.[12]

Our emulated *Unix I/O* implementation uses a per application file offset, and hence does not support the sharing semantics of Unix. Also, `read` and `write` operations are not atomic (as is the case with the system call interface). In the Mach 3.0 operating system [DA92], *Unix I/O* is implemented by an emulation library where the library and the file system cooperate to: 1) handle file offsets shared by multiple applications, 2) ensure that `read` and `write` operations are atomic, and 3) handle the EOF problem described above. It would be interesting to port ASF to this system and explore providing this same level of compatibility for the *Unix I/O* system call interface.

It is often the case (even for parallel applications) that some streams are accessed by just a single thread, in which case locking is unnecessary. In our current implementation there is no way for the facility to recognize whether a stream is being accessed by a single thread or not. We have developed an extension to the facility that allows an application to indicate when a stream is being used exclusively by a single thread. With this extension, applications can use the basic ASI operations with only a little overhead[13] (over direct use of the unlocked ASI operations). More importantly, if an application calls the *emulated Unix I/O* or *stdio* interfaces (that are implemented using the basic ASI operations) it will not incur large locking overheads.

Another problem with the current ASF implementation on some Unix systems is that the best choice of stream module may depend on information not easily available to the library. For example, on SunOS, modifying a NFS remote mounted file results in worse performance when mapped file I/O is used than if *Unix I/O* is used (while mapped file I/O results in better performance than *Unix I/O* if the file is on a local disk). We expect this problem to become more extreme in the future as systems become available that transparently provide access to files stored remotely [CH93] and stored in tertiary storage [KSS93]. It will be important that the application level library be able to query the source of a file to be able to optimize performance.

Finally, our implementation of the *stdio* `putc` and `getc` macros is limited in that they only work correctly if the stream is already in the mode required for the operation (i.e., input or output). The reason behind this limitation is that the overhead to check the mode of the stream greatly increases the cost of these otherwise simple operations. This same limitation occurs in all other implementations of *stdio* that we have looked at. In the case of ASF, the implementation has similar properties to the basic ASI operations; the application dictates the mode of the stream external to the individual I/O requests. However, it is not consistent with the implementation of other *stdio* operations (e.g., `fread`) that internally ensure the stream is in the correct mode.

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

[11] 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.

[12] 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.

[13] The overhead to check if locking is necessary involves a cached read of a single word.

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.[14]

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.[15] The standard configuration on the *SunOS* system uses mapped files (based on `mmap`) for input and output. The standard configuration on the *IRIX* system uses mapped files (based on `mmap`) for input and uses *Unix I/O* operations for output. For comparison purposes, we also present the performance of each experiment when the facility is configured to use *Unix I/O* for both input and output.

Results for each program is 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. In these experiments all input and output is directed to files. The numbers we present indicate the expected speedup on a idle system with all file data available in the main memory file cache. To obtain these numbers we ran each experiment a large number of times and, in order to subtract away the impact of any disk I/O and other applications, we present the smallest numbers measured. (For all of these experiments, if a small number of outliers are ignored, the average of the many runs is within 10% of the minimum value presented.)

5.1 *Stdio* Applications

Figure 10 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, `diff` runs in less then half the time when linked to the *Alloc Stream Facility*. The major reason for the improved performance is the use of mapped files.

Many stdio-based programs show further improvements when they are modified to use ASI directly. 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.

	AIX		SunOS		IRIX	
appl.	Unix I/O	ASI	Unix I/O	ASI	Unix I/O	ASI
diff	1.00	1.53	0.96	1.31	0.99	1.26
compress	1.00	1.00	1.01	1.01	1.06	1.07
uncomp.	0.99	0.99	0.99	1.00	1.23	1.42
cut	0.87	0.91	0.90	1.03	0.92	1.07

Table 2: Speedup for stdio applications when ASF uses Unix I/O operations for both input and output.

Table 2 shows the speedup of *stdio* applications when ASF is configured to use Unix I/O for both input and output. In this case, unmodified *stdio* applications linked to our facility do not (in general) perform better than when linked to the installed facility. This behavior is expected, because when *Unix I/O* is used ASF implements the stdio interface in a fashion similar to the installed *stdio* facilities. However, if the applications are modified to use *ASI*, they again perform better in many cases.

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.

Figure 11 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, on the AIX system, `cp` is almost twice as fast when linked to the ASF. Thus, on some systems, our application level library implements the Unix I/O interface more efficiently than the operating system. Moreover, the *Unix I/O* applications improved further when modified to use the *Alloc Stream Interface*. For example, `cp` modified to use *ASI* ran two and a half times as fast as the original on the AIX system.

As expected, unmodified *Unix I/O* programs that use the *emulated Unix I/O* interface of the ASF perform significantly worse when the facility is configured to use *Unix I/O*, as is shown in Table 3. This is expected, because the *emulated Unix I/O* interface implemented by the application level library introduces extra overhead. However, we found it surprising that in most cases the Unix I/O applications (that is, the unmodified applications that use the low level *Unix I/O*

[14] 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.

[15] In the case of a file opened for write-only access, we first attempt to open it for read/write access in order to allow the file to be mapped into the application address space, and if this fails we open it for write-only access and use Unix I/O write operations to modify the file.

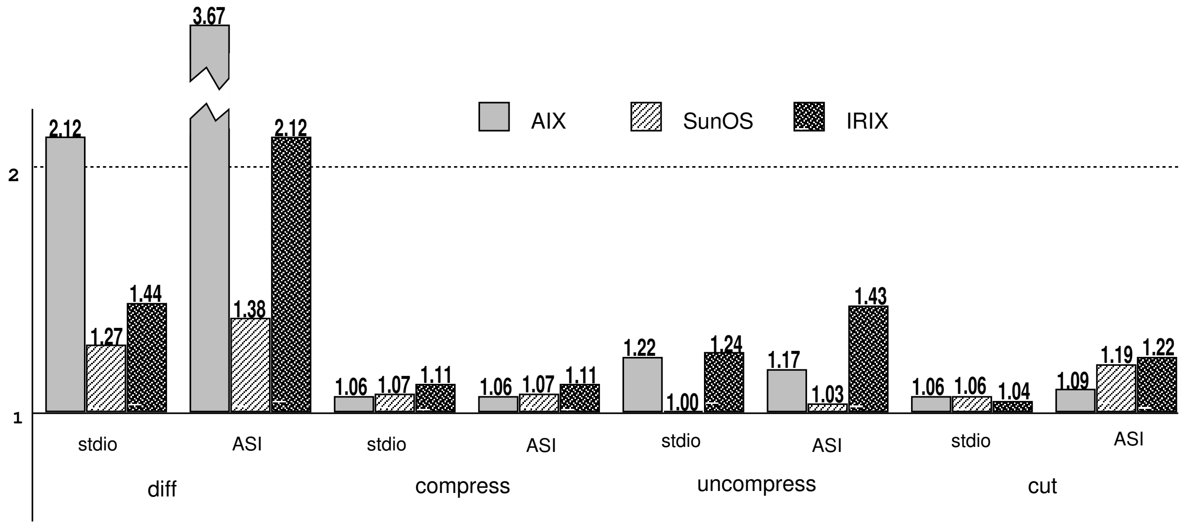

Figure 10: Speedup of stdio applications that are 1) linked to the *Alloc Stream Facility* and 2) modified to use *ASI*. `Diff` compares the contents of two files (identical in our experiments); `compress` and `uncompress` use adaptive Lempel-Ziv coding to respectively compress and uncompress files; `cut` is a Unix filter that removes selected fields from the input file and writes the result to an output file.

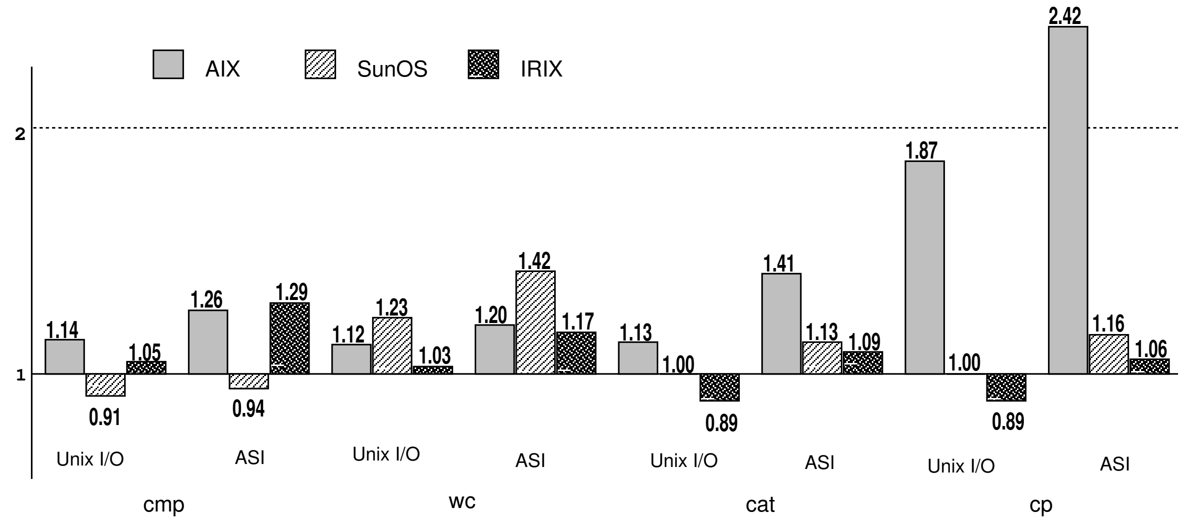

Figure 11: Speedup of Unix I/O applications that are 1) linked to the *Alloc Stream Facility* and 2) modified to use *ASI*. `Cmp` compares the contents of two files; `wc` counts the number of characters, words and lines in a file; `cat` copies the input file to the standard output; `cp` copies one file to another.

appl.	AIX Unix I/O	AIX ASI	SunOS Unix I/O	SunOS ASI	IRIX Unix I/O	IRIX ASI
cmp	0.88	0.97	0.80	0.97	0.84	0.99
wc	0.92	0.97	0.87	1.00	0.89	0.99
cat	0.85	1.00	1.08	1.13	0.73	0.85
cp	0.83	0.95	0.97	0.99	0.71	0.83

Table 3: Speedup of Unix I/O applications when ASF is configured to use Unix I/O for both input and output

system calls directly) could be modified to use *ASI* (a high level interface) with almost no performance penalty.

5.3 Caveats

Locking was disabled for all the experiments performed on Unix systems. This is fair on the AIX and SunOS systems, where the installed versions of *stdio* do not do locking. However, the comparison of ASF to the installed *stdio* on IRIX, where locking is performed, may not be fair. An important example of where the comparison to the IRIX system is not fair is the compress and uncompress programs, where the overhead of locking in the installed version of the `putc` and `getc` macros is probably quite high [Jon91].

We have generally made every effort to ensure that the comparisons to installed facilities are fair, but it is difficult to be certain they are. For example, we do not know what size buffer is used by the installed versions of *stdio* that we compare against. The default buffer size used by ASF for accessing a file is the buffer size returned by `lstat` if available, or otherwise, the BUFSIZ specified in the installed `stdio.h` file.

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 concurrency. 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 re-usability. 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

While we concentrate in this paper on I/O to disk files, we believe that with the development of new service specific stream modules, ASF can deliver improved performance for other I/O services such as pipes or network facilities. Related work by Maeda and Bershad [MB93] demonstrates that substantial performance advantages result from moving critical portions of network protocol processing into the application address space and by 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 [GA91] and Druschel and Peterson's *Fbufs* facility [DP93].

We are currently experimenting with a new implementation of pipes that exploits shared memory.[16] 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

[16]Note that 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.

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.

The *Alloc Stream Facility* was originally developed as the application level library for the HURRICANE file system, a file system being developed for shared memory multiprocessors with disks distributed across the multiprocessor [KS93]. As part of this project, we are investigating different mechanisms for locking and managing the CIOS structures. Many of the file system policies, like pre-fetching and compression are implemented by ASF in stream specific modules. Currently, ASF is being modified to: 1) allow the application to select the stream modules that will be used (our approach is similar to that used by the ELFS file system [GL91]), and 2) push and pop stream modules on top of existing stream modules (this is similar to the push and pop operations of the System V kernel streams [Rit84].)

A To Map, or not to Map

We examine basic costs on a number of systems to determine under what conditions mapped file I/O should be used. The four systems we consider are: an IBM R6000/350 running AIX, a Sun 4/40 running SunOS, an SGI Iris 4D/240S running *IRIX*, the HECTOR prototype [VSWL91] running HURRICANE. The AIX system has a 4-way set associative physical write back cache with a cache line size of 64 bytes. The Sun system has a direct mapped virtual write through cache with a 16 bytes cache line. The SGI system has a direct mapped physical write back cache with a 16 byte cache line[17]. HECTOR has a 4-way set associative physical cache with a 16 byte cache line. One HECTOR, the write through or write back policy can be specified on a per-page basis.

A.1 Copying overhead

It has been argued by others that the central goal of our facility, avoiding copying, is of little value. The proponents of this view believe that any advantages obtained, for example, by using mapped file I/O are attributable to poor implementations of Unix `read` and `write` rather than from avoiding the cost of copying. They believe that with large fast caches, if data is being copied to or from relatively small buffers, the cost of copying the data in the cache is irrelevant compared to the time to transfer the data to or from the memory.

To examine the cost of copying data, we ran four simple experiments that copy data from one buffer to another (Figure 4). The *copy* test copies data between two four megabyte memory regions. The *copy to buf* test copies data from a four megabyte memory region, in page size chunks, to a single page size buffer. This second test emulates the copying overhead of page size read operations that transfer data from the file cache to an application buffer. The *copy from buf* test repeatedly copies data from a page size buffer to a four megabyte memory region (in chunks of a page size). This experiment emulates the copying overhead of `write` operations that transfer from an application buffer to the file cache. The *copy buf* test repeatedly copies data between two page sized buffers. This test emulates the copying that occurs between library and application buffers as would occur, for example, in *stdio*.

To avoid any memory management overhead, we pre-initialized the buffers and memory regions used in the experiments. On all systems, we did various optimizations (e.g., hand unrolling the copy routine) to try to optimize performance, hence the numbers presented can be viewed as upper bounds on performance (for non-hardware assisted) copying on these systems. The `copy from buf` and `copy to buf` were run with the buffers aligned to cache line and to page boundaries. On HECTOR, we ran the copy tests with: 1) caching enabled for both the memory region and the buffer (both write through and write back), 2) caching enabled for the buffer but not the memory region, and 3) caching disabled for both. In all cases, the large memory regions were distributed (at page boundaries) round robin across 8 memory modules, and the buffer was allocated from local memory.

We can see from these experiments that the relative costs of different types of copy operations is very system dependant. For example, the alignment of the data being copied affects performance on some systems and not on others. For some of these experiments write-back caching performs best, while for others it is better to either not cache the data or use a write-through policy.

The most important comparison to made from Table 4 is the relative performance of the *copy buf* test and the different *copy to buf* and *copy from buf* tests. This comparison shows that on the systems evaluated a substantial portion of the time spent copying data is attributable to the processor (and cache) rather than the memory. For example, on the AIX system the performance of the *copy buf* experiment is only 25% better than the *copy from buf* experiment. The main reason for this is that the memory bandwidth on this machine is very high and the cache lines are long. Hence, even though processor cycles are shorter than memory cycles, the processor can be the bottleneck since it transfers data on a word by word basis rather than on a cache line basis. Even on a system like HECTOR, where remote memory accesses can be very slow, the *copy buf* experiment takes nearly 25% of the time of the *copy to buf* experiment.

We do not believe that copying overhead will become irrelevant in future systems, even with the growing gap between

[17] This specification, although obtained from the SGI technical support group in Toronto, may be suspect given the results obtained later.

experiment (alignment)	AIX	SunOS	IRIX	HURRICANE cached wr b	wr t	buf cached wr b	wr t	unca
copy	0.086	0.57	0.43	2.31	1.77	—	—	2.04
copy from buf (pg)	0.069	0.48	0.28	1.83	1.29	0.86	—	1.49
copy from buf (cl)	0.069	0.43	0.28	1.83	1.29	0.86	—	1.49
copy to buf (pg)	0.070	0.59	0.27	0.59	1.21	1.25	1.87	1.87
copy to buf (cl)	0.070	0.56	0.27	0.59	1.21	1.25	1.87	1.87
copy buf	0.053	0.42	0.16	0.14				

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 memory regions. With the *copy to buf* test, the data is copied from a four megabyte region, 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 memory region (in chunks of a page size). We ran the *copy to buf* and *copy from buf* with the buffer aligned to a page (pg) and to a cache line (cl). The *copy buf* experiment repeatedly copies data between two page sized buffers.

processor speeds and memory latency. With relatively small on chip first level caches, copying data through the cache will destroy important cache context. Moreover, direct mapped second-level caches can perform poorly if the alignment is unfortunate (e.g., if a page is copied to a second page where both are mapped to the same cache lines). Finally, copying results in a large amount of data being accessed with no processing time between the memory accesses. If the application accesses data without copying it, than it may be possible to overlap the time to process the data with the time to transfer it to or from the memory (e.g., if prefetching is used for input data [MG91]).

It is interesting to note that the SPARCserver 400 series machines implement hardware accelerators to reduce the kernel cost of copying data. This accelerator performs memory to memory copies at the full memory bandwidth, bypassing the cache. This indicates that the designers of this machine do not believe caches will make copying costs irrelevant. Since memory bandwidth is a crucial resource in current machines (especially with shared memory multi-processors, or systems with parallel disks and high I/O requirements), we feel that software solutions that can avoid copying are better than a hardware solution that reduces the processor cost but does nothing to reduce the memory bandwidth used.

A.2 Reading data in the file cache

Table 5 compares the cost of accessing a page of a mapped file relative to the cost of Unix I/O `read` and `lseek` operations. On the AIX system, both `shmget` and `mmap` are used to map the file into the application address space. On the other Unix systems `mmap` is used. For comparison purposes, the performance of mapped file I/O on HURRICANE is shown, in which case the HURRICANE `BindRegion` call is used. (We do not show `read` and `write` times on HURRICANE because these operations are emulated by the ASF library rather than implemented by an operating system server or kernel.)

We will consider the time for `lseek` to be a lower bound for a non-trivial system call. The three Unix systems used for evaluation are monolithic systems. Therefore, the basic cost of a system call can be expected to be small (in contrast to some micro-kernel operating systems). This is demonstrated by the low cost of the `lseek` operations on the three systems.

	AIX shmget	AIX mmap	SunOS mmap	IRIX mmap	Hurricane BindRegion
page touch (r)	0.00	0.04	0.56	0.14	0.16
read file 4meg	0.137		1.19	0.53	
read file 2meg	0.137		0.67	0.53	
read page	0.138		0.55	0.43	
read byte	0.084		0.17	0.27	
read 10 pages	0.973		5.39	3.23	
lseek	0.011		0.035	0.038	

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 file, touches each page in the file, and then un-maps the file. The *read file* 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. The **`read page`**, **`read byte`**, and **`read 10 pages`** tests repeatedly read the first portion of the file and then use `lseek` to reposition the file offset to the beginning of the file. The `lseek` test shows the cost of Unix I/O `lseek` operations.

When mapped file I/O is used (assuming all data is in the file cache), the greatest cost incurred is that to handle page faults. On the three Unix systems, the cost to handle a read page fault relative to a `lseek` operation varies dramatically. On the AIX system, the page fault time is quite close to the `lseek` time, while with the SunOS and IRIX systems the difference between the two is larger. Without knowing the details of the operating systems and hardware, it is difficult to explain this. However, we believe that it may be possible to improve the performance of read page faults on these system, since better performance is achieved under HURRICANE, despite the fact that it runs on slower hardware and does not cache its data.

On all three Unix systems it is more efficient to access large files using mapped file I/O rather than using `read` operations. 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 high. When `shmget` is used on the AIX system, the cost of accessing a mapped page is zero. AIX exploits the segments of the RS6000 architecture, so that all processes accessing the same file (mapped using `shmget`) share the same segment and data in the file cache can be accessed with no page faults.

A surprising result in Table 5 is that on SunOS much better performance is obtained if `read` operations are used to (repeatedly) access a 2 Megabyte file rather than a 4 Megabyte file (even though the amount of main memory is sufficiently large to in both cases to keep all data in the file cache). SunOS does better for small files than large files because only a subset of the file cache is mapped into the kernel address space at a given time. If a page in this subset is accessed, then the kernel can service the read request quickly; otherwise, the kernel incurs the cost of a page fault to service the request [Sha92].

We have compared the cost of a page `read` to the cost of a page fault (i.e., 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` but not if the page is only touched. 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 Unix systems, the cost of a page `read` is greater than the combined cost of touching a mapped page and copying a page sized buffer (i.e., the *copy to buf* experiment in Table 4). As a result, it is always more efficient to use mapped files for accessing large files in read-only mode on these systems.

If data is read in large blocks (e.g., the `read 10 pages` experiment) on the AIX and IRIX systems, and if a small file is repeatedly read on SunOS, then (considering the *copy to buf* experiment in Table 4) copying overhead is more than 70% of the cost of read operations on all systems. However, with single page reads on AIX and IRIX, and with reads to a large file on SunOS, copying overhead makes up less than half the cost of the `read` operations.

If buffers are page aligned, then an alternative to copying data on a read operation is for the operating system to map the page copy-on-write. However, this could lead to worse performance if the buffer is later modified. Moreover, we believe that requiring the application to ensure the alignment of its buffers (rather than having the operating system choose the alignment as is done with mapped file I/O) entails an excessive increase in complexity for many user applications.

	AIX		SunOS	IRIX	Hurricane
	shmget	mmap	mmap	mmap	BindRegion
page touch (w)	0.00	0.04	0.54	0.19	0.53
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 un-maps 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	Hurricane
	shmget	mmap	mmap	mmap	BindRegion
page touch (w)	0.21	0.27	1.49	0.73	3.33
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.

A.3 Modifying data in the file cache

Tables 6 and 7 show the cost of modifying data in the file cache. The *page touch (w)* test repeatedly maps a four megabyte file, modifies a single character in each page, and then un-maps 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. Table 6 shows the performance of these experiments when the data being modified is already in the file cache, and Table 7 shows the performance when new pages are being added to the file.

Table 6 shows that the cost of modifying a page using mapped files is substantially less than the cost of a `write` operation for all Unix systems. In fact, on SunOS and AIX, using mapped files for modifying data already in a file improves performance even more than it did when using mapped files for input.

Table 7 shows that on the AIX system there is a significant advantage in using mapped files appending data to a file. However, on the IRIX system using mapped files past the EOF is much more expensive than using `write` operations.

Adding a new page to a file can be expensive, since the first fault to a new page often incurs the cost of zero filling a page of memory. However, depending on the hardware, it is possible that this cost can be very low. For example, with the RS6000 (as well as the MIPS 4400) cache controllers allow the processor to request that a cache line be zero filled. In this case, zero filling a page that is being modified may actually improve performance, since zeroing the cache lines may be less expensive than the cost that is otherwise incurred to load them from memory. As another example, SPARCserver 400 series machines implement hardware accelerators to zero memory.[18]

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 *4th Intl. Conf. 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 *Proc. 12th ACM Symp. on Operating System Principles*, pages 71–82, 1989.

[CH93] Robert A. Coyne and Harry Hulen. The high performance storage system. In *Proceeding of Supercomputing*, pages 83–92, Portland, Oregon, Nov. 1993.

[Che87] D. Cheriton. UIO: A Uniform I/O System Interface for Distributed Systems. *ACM Transactions on Computer Systems*, 5(1):12–46, February 1987.

[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.

[DP93] P. Druschel and L.L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. 14th ACM Symp. on Operating System Principles*, pages 189–202, 1993.

[18] These accelerators zero the memory rather than the cache, and hence consume memory bandwidth.

[GA91] R. Govidan and D.P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proc. 13th ACM Symp. on Operating System Principles*, pages 68–109, 1991.

[GL91] Andrew S. Grimshaw and Edmond C. Loyot, Jr. ELFS: object-oriented extensible file systems. Technical Report TR-91-14, Univ. of Virginia Computer Science Department, July 1991.

[HS77] J. Hunt and T. Szymanski. A fast 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.

[KS93] Orran Krieger and Michael Stumm. HFS: a flexible file system for large-scale multiprocessors. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 6–14, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[KSS93] J. Kohl, C. Staelin, and M. Stonebraker. Highlight: Using a log-structured file system for tertiary storage management. In *USENIX Winter Conference*, pages 435–447. USENIX Association, Jan 1993.

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

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

[MB93] C. Maeda and B.N. Bershad. Protocol service decomposition for high-performance networking. In *Proc. 14th ACM Symp. on Operating System Principles*, pages 244–255, 1993.

[MG91] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12:87–106, June 1991.

[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 *Proc. of the Summer USENIX Conference*, pages 247–256, June 1990.

[PGK88] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, 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.

[Sha92] Bill Shannon. Implementation information for unix i/o on sunos. personal communication, 1992.

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

[Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Professional Computing Series. Addison Wesley, One Jacob Way, Reading, Massachusetts 01867, 1992.

[USK93] Ron Unrau, Michael Stumm, and Orran Krieger. Hierarchical Clustering: A structure for scalable muliprocessor operating system design. Technical Report 268, Computer Systems Research Institute, University of Toronto, 1993.

[VSWL91] Zvonko G. Vranesic, Michael Stumm, Ron White, and David Lewis. "The Hector Multiprocessor". *Computer*, 24(1), January 1991.

[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.