Exploiting the Advantages of Mapped Files for Stream I/O

Orran Krieger, Michael Stumm and Ron Unrau

Technical Report CSRI-267
November, 1991

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.

Exploiting the Advantages of Mapped Files for Stream I/O

Orran Krieger, Michael Stumm and Ron Unrau
Department of Electrical Engineering
University of Toronto
Toronto, Canada M5S 1A4
Tel: 416.978.5036, Fax: 416.978.7423
Email: `okrieg@eecg.toronto.edu`

Abstract

A new approach for providing user level support for fast stream I/O is motivated by four factors common to most modern systems: 1) the capability of the operating system to support mapped files, 2) the increasing number of applications that use threads, 3) the increasing discrepancy between processor speed and disk latency, and 4) the increasing amount of available main memory.

In this paper, we first describe the advantages and disadvantages of using mapped files to support stream access to files, and then describe a new interface, the Alloc Stream Interface (ASI), that allows for improved performance over existing stream interfaces. A library that supports ASI has been implemented on several systems (including IRIX and SunOS). In addition, the *Stdio* library has been re-implemented to use ASI. Significant performance advantages are demonstrated for *Stdio* applications that are linked to this new library and particularly for applications that are modified to use ASI directly. For example, on typical Unix platforms, some standard I/O intensive utilities are shown to run up to twice as fast when re-linked to use this library and up to three times as fast when converted to use ASI.

1 Introduction

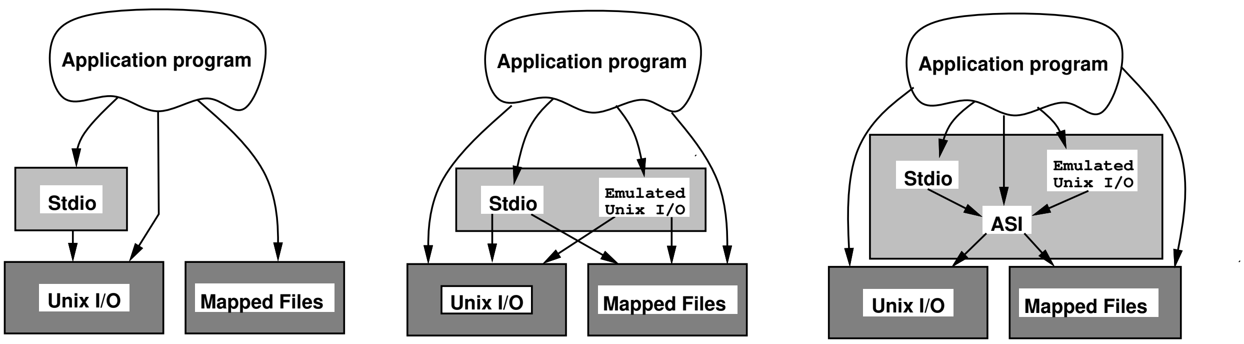
In Unix, applications access files by using the *Unix I/O* interface (i.e. `read`, `write`, ...). A major reason for the success of Unix is that the *Unix I/O* interface is used uniformly for all I/O. A uniform interface is important so that a program can be written to be independent of the type of data sources and sinks with which it is communicating. However, there are two main disadvantages with the *Unix I/O* interface: Firstly, as the interface definition now stands, an excessive number of system calls will result if a user process accesses a file with many small read and write operations. Secondly, the user supplies a private buffer into which data should be read, or from which data should be written. This can result in a large performance cost when data is copied to and from the system buffers.

To reduce the number of interactions with the operating system, the *Stdio* run-time library buffers data in the application's address space (refer to Figure 1.1) and thus amortizes the cost of interactions with the operating system over several application requests. However, buffering at the user level introduces yet another layer of copying, namely between the library and application buffers. Moreover, most implementations of *Stdio* are not re-entrant, and of the few that are re-entrant, we are not aware of any that allow more than one thread to concurrently access data from a single stream.

Since the speed of file I/O is so crucial to the performance of many applications, most modern operating systems support *mapped files*[1] where a file can be bound to a virtual address space such that a reference to memory is effectively a reference to the corresponding location in the file. As discussed in Section 2, mapped file I/O can result in less copying of data between different buffers, and can allow for different threads in an application to concurrently access different parts of the same file. However, the main disadvantage of mapped file I/O is that it cannot provide a uniform interface for all I/O; for example, it cannot be used for I/O to terminals or network connections (i.e. sockets).

To incorporate some of the advantages of mapped files while preserving the advantages of uniformity that a byte-oriented stream interface provides, the *Stdio* and *Unix I/O* interfaces can be implemented by a user level library that whenever possible uses mapped files, as shown in Figure 1.2. This approach is referred to as *Mapped Stream I/O* (MSIO) and is dis-

[1]For example, Mach [ABB+86], AIX [Mis90], Hurricane [SUK] ...



1.1 Typical Implementation: Stdio implemented using Unix I/O

1.2 The MSIO approach: Stdio and Emulated Unix I/O implemented using Mapped Files

1.3 The MSIO library: Stdio and Emulated Unix I/O implemented using ASI

Figure 1: *Layers of interaction between Stdio, Unix I/O and ASI. Lightly shaded blocks are user level libraries. Dark shaded boxes indicate interfaces supported by the operating system.*

cussed in Section 3. To differentiate between the *Unix I/O* interface supported by the operating system and a similar interface supported by a user level library, we refer to the latter as *Emulated Unix I/O*. *Emulated Unix I/O* has the disadvantage that the full *Unix I/O* semantics (e.g. shared file semantics) cannot be supported by a user level library. However, these semantics are not supported by most distributed file systems [LS90]. Moreover, significant performance gains can be obtained by having file servers support only block oriented operations, as discussed in [Che87].

The primary contribution of this paper is the definition of a new I/O interface, called the *Alloc Stream Interface* (ASI) that improves on the *Stdio* and *emulated Unix I/O* interfaces by significantly reducing the amount of data copying that is necessary. The *Stdio* and *Unix I/O* interfaces force the library (or operating system) to copy data into a buffer specified by the application. In contrast, ASI provides the application access to the internal buffers (mapped regions) of the library. With ASI, on a read operation, the underlying library returns to the application a pointer to a region of memory containing the requested data, and on a write operation, the library returns a pointer to a region where the data should be put. ASI has a number of advantages, especially when implemented using MSIO:

1. Overhead (i.e. cost not associated with disk I/O) is greatly reduced. Whenever an I/O request can be serviced directly from the file cache, with *Stdio* and *Unix I/O* most of the I/O overhead is due to copying data between various system, library and user buffers. Since MSIO uses mapped files, there is no copying of data between the system and library buffers. Moreover, since with ASI the application is given access to the mapped regions maintained by the library, there is no copying of data between library and application buffers. This advantage is becoming increasingly important, since more and more memory is being made available for file caching.

2. Threads in a multi-threaded application [Jon91] can concurrently access a stream. The behaviour of ASI operations are well defined regardless of the number of threads accessing a stream. Also, the stream is only locked when the library's internal data structures are being modified, since with ASI data does not have to be copied to or from user buffers. Hence, data is accessed by the application while the stream is unlocked, which allows the I/O requests of multiple threads to the same stream to be sent concurrently to the file system.

3. The interface is simple and easy to use. As we describe in Section 4, ASI is closely modeled on the Unix memory allocation interface (i.e. `malloc`, `realloc`, `free`), providing programmers an interface they are already familiar with. Our experience indicates that in most cases it is just as easy to write an application using ASI as it is using the *Unix I/O* or *Stdio* interfaces.

A library, called the *Mapped Stream I/O* (MSIO) library has been implemented on several systems including IRIX and SunOS. This library uses the MSIO approach to support the *Alloc Stream Interface*. As shown in Figure 1.3, the MSIO library also supports the *Stdio* and *emulated Unix I/O* interfaces, which are implemented on top of ASI. Applications that use this library can freely intersperse calls to all three interfaces even if they are directed to the same stream.

This allows application programmers to get the full performance advantage of ASI by rewriting just the I/O intensive parts of the application.

In related work, the *Sfio* library [KV91] is a replacement library for *Stdio*. As well as providing a much more consistent, powerful and natural interface than *Stdio*, *Sfio* uses algorithms that are more efficient than those used by typical *Stdio* implementations and uses mapped file I/O for reading from files. Moreover, *Sfio* provides an operation, called `sfpeek()`, that allows applications access to the internal buffers of the library.

Our work differs from *Sfio* in that: 1) it is better suited for multi-threaded environments, due to the reentrant nature of the library, and the short periods of time locks need to be held, 2) the MSIO library can make use of mapped files in more situations, and 3) it provides a more versatile interface for accessing library buffers.

This paper is structured as follows. In the following section, the advantages and disadvantages of mapped file I/O over *Unix I/O* are described. The section contains performance numbers as measured on several typical Unix systems. Section 3 describes the MSIO approach in detail, and Section 4 describes ASI. Section 5 describes the current implementation of the MSIO library. Finally, Section 5.4 compares the performance of applications linked to the standard system libraries against the performance of these same applications re-linked to use the MSIO library, and re-written to use ASI. We close with a discussion of future directions.

2 Mapped File I/O

In this section, we first present the result of some simple experiments on SunOS and IRIX systems, and then discuss the advantages and disadvantages of mapped file I/O over *Unix I/O* in the context of these numbers.

The `mmap/munmap` [JCF+83] interface is a typical interface for mapping files into an address space. `Mmap` takes as parameters the file number, protection flags, the length of the region to be mapped, and an offset into a file. It returns a pointer to the mapped region. `Munmap` accepts as parameters the address and length of the mapped region. This interface will be assumed throughout this paper.

Measured performance numbers for some simple programs on three different systems are shown in Table 1. Experiment `read` accesses a large file (3MBytes) by using `read` operations that read the entire page from a page aligned portion of the file into a page aligned buffer.[2] Experiment `mmapr` maps in an entire file, and then touches each page to cause a page fault. Experiment `mod`, for each page in the file, reads the page, seeks back to the beginning of the page and then writes it. Experiment `mmapw` maps in an entire file and then modifies one byte in each page. Experiment `write` writes each page in a file either by writing the entire page, or by writing one byte and then seeking to the next page. Note that these performance numbers were measured on very different hardware platforms (the SunOS 4.0.3 system is a Sun 4/280S, the SunOS 4.1.1 system is a Sun 4/40 and the IRIX system is a SGI Iris 4D/240S) and therefore only allow a comparison between different experiments on the *same* system. All experiments where performed many times, with the minimum number chosen. This reflects best the measure of the performance when all data being accessed is in the file cache.

As can be seen from the table, using mapped files for reading is nearly three times as fast as using Unix `read` operations on both the IRIX and SunOS 4.0 systems. Also, the system cost of modifying a page using mapped files is substantially less than if the page is read and written with *Unix I/O*. Finally, the system cost for modifying a mapped page is substantially less than writing the entire page. However, the performance improvements with mapped file I/O need not always be as significant. For example, on the Sun OS 4.1.1 system, mapped files are only 20% faster than `read` operations for reading from a file.[3] Also, if only a small amount of data is being written (and the application does not require the previous contents of the data) the combination of doing `write` and `lseek` operations seems very effective on all three systems.

2.1 Advantages of Mapped File I/O

Using mapped files has a number of advantages:

Copying Overhead: With mapped files, data need never be copied between system and user buffers. The processor time saved by not copying data is responsible for most of the performance advantage shown in Table 1. However, there are hidden advantages not evident in these numbers. As observed in [CK91], copying data can cause memory contention, which can be very important in the case of multiprocessor systems or if high

[2] It is fair to compare a `read` of an entire page to a *touch* of a mapped page, since these correspond to the system cost to get a page of data into the application address space.

[3] It is interesting to note that `read` performance on this system becomes much worse when files greater then 4MBytes are accessed, therefore, the relative performance of `mmap` looks better for these files.

		SunOS 4.1.1		SunOS 4.0		IRIX	
command	option	pages	KBytes	pages	KBytes	pages	KBytes
read		1.47	5.90	0.68	5.48	2.03	8.13
mmapr		1.89	7.55	2.22	17.78	5.62	22.47
mod		0.63	2.51	0.05	0.38	1.11	4.43
mmapw		1.91	7.64	2.08	16.67	4.25	16.88
write	page	1.19	4.77	0.06	0.45	2.34	9.39
	byte	5.46	21.83	4.17	33.33	5.13	20.51

Table 1: *Measured performance of mapped files versus Unix I/O on SunOS 4.1.1, SunOS 4.0 and IRIX. Performance numbers are in pages or KBytes per millisecond (i.e. the larger the number the better). The Sun OS 4.1.1 system and the IRIX system have 4KByte pages. The Sun OS 4.0 system has 8KByte pages.*

bandwidth I/O devices are concurrently trying to DMA data to the system. Also, when data is copied through the processor cache, a large portion of the cache context may be lost.

It is possible for a smart application and operating system implementation to obtain similar advantages using the Unix I/O interface. For example, if the application always uses a page aligned region of memory for accessing a page aligned portion of a file, and if the amount of data being accessed is a multiple of the page size, then the OS can bind the portion of the file into the application space copy-on-write. However, in practice, this entails an excessive increase in complexity for most user applications. Although system utilities and library packages (e.g. *Stdio*) could be written to use this approach, the resulting performance may still not be as good as is the case for mapped file I/O. For example, if a file is being modified rather than written in its entirety, with *Unix I/O* the application must do a `read` operation to get the data to be modified, modify the data (which will cause a page fault if the data is mapped copy-on-write) and then perform a `write` operation. Using mapped files, the application need only map in the data and then modify it.

Interactions with the Operating System: When mapped file I/O is used, applications generally map large portions of a file into their address space; if it turns out that the application accesses a small amount of the data, only pages actually touched will be read from the file system. In contrast, if Unix I/O is used, the application must be pessimistic about the amount of data it will require, since a `read` operation incurs the I/O cost when invoked. Therefore, with mapped files, the number of system calls is greatly reduced over if *Unix I/O* is used. However, the cost of a page fault is incurred every time a new mapped page is accessed.

The cost of a page fault is generally greater than the cost of a `read` or a `write` system call (not including the copy of data) on systems where all functions of the operating system are provided by a single monolithic kernel. This is shown in Table 1, since the cost of experiment `mmapw` is greater than the single byte `write` experiment. However, the cost of a page fault may be less than a `read` or a `write` operation on micro-kernel based operating systems, such as V [Che88], Mach [ABB+86] or Hurricane [SUK] [4]. For example, on Hurricane, page faults are handled by the micro-kernel with no communication to the file servers if the data is already in the file cache, resulting in a substantial performance gain.

Concurrency of Reads: With mapped files, the mapping operation is independent from the actual faulting for the data. Therefore, if several application threads access the same mapped region, each thread may independently cause a page fault that initiates an I/O operation to disk. This allows the application to exploit high bandwidth file systems (e.g. if disk I/O is spread across many disks [PGK88, FPD91]).

Reduced Memory Usage: When an application uses *Unix I/O*, there are often multiple copies of the same data resident in memory, since data is copied from the file cache to application specific buffers. This can result in paging or swapping activity. In contrast, if mapped file I/O is used,

[4] Hurricane is an experimental operating system designed to study scalability issues for large scale multiprocessors. It is the operating system used on the Hector multiprocessor [VSWL91]

no extra copies of the data are made, so the system memory is used more effectively. Moreover, in the case where there is insufficient main memory, if the data in the file cache is not modified it does not need to be paged out, since the data is already on disk, whereas with *Unix I/O*, the copy of the file data read in the application address space must be paged out to disk.

Shared file access: With mapped files, two programs accessing the same file will share the same physical memory pages. Therefore, programs can actively share file data and, for example, directly share synchronization variables in the file.

2.2 Disadvantages of Mapped File I/O

The specific implementation of the operating system or even the hardware may have a large impact on the performance of mapped file I/O relative to that for *Unix I/O*. For example, if the cost of a page fault is very expensive, the performance of mapped files will compare unfavorably to the performance of *Unix I/O*. One instance where page faults are expensive is on operating systems that require that the processor cache be flushed or invalidated on each page fault.

Page size may also have a dramatic impact on performance. Small page sizes increase the number of page faults that will occur when mapped files are used. On the other hand, large page sizes can result in superfluous I/O.

Another disadvantage of using mapped file I/O is that some operating systems will zero-fill the page on the first access to a new file block. Therefore, even in the case that the application will modify the the entire page, the cost of the zero-fill will be incurred. 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.

When an application uses *Unix I/O*, the operating system generally pre-fetches data on sequential reads to a file, thus reducing the user visible latency of the file system. Although some operating systems do not directly pre-fetch data in the case of mapped files, several operating systems do provide an interface to advise the operating system that mapped data will soon be accessed (e.g. SunOS has the `madvise()` system call). In effect, this is more powerful than having the operating system implicitly pre-fetch pages, since it can be applied to non-sequential accesses. Also, it allows the advantages of asynchronous reading [BJ91], without requiring a special mechanism to inform the application that the requested data is available. If the application attempts to access the data before it is available, a page fault will occur that allows the operating system to defer the access until the data is available.

Although support for mapped files is becoming a common feature of many operating systems, few applications have yet been written to use it. The most important reason for this is that mapped files cannot provide a uniform interface for all I/O. With *Unix I/O* or *Stdio*, applications can use the same operations whether the I/O is directed to a file, terminal or network connection. Mapped file I/O can only be used for I/O directed to a device which can be used to handle page faults, that is, a random access block oriented I/O devices such as a disk. In the next section, we introduce an approach, called MSIO, for supporting a uniform I/O interface while exploiting some of the performance advantages of mapped file I/O.

3 Mapped Stream I/O

Mapped Stream I/O (MSIO) is an approach for supporting stream I/O interfaces such as *emulated Unix I/O* or *Stdio* with a user level run-time library that exploits the advantages of mapped files. With *MSIO* a portion of the file being accessed is mapped into the application address space. `Read` operations are handled by copying data from the mapped region to the buffer specified by the application. Similarly, for `write` operations, data is copied from the application buffer to the mapped region. The library keeps track of the current position in the file, modifying it every time the application performs a `read`, `write` or `seek` operation. When the region is exhausted, it is freed and a new region (usually the one directly following it) is mapped in.

In many ways, a stream I/O library based on mapped files is similar to the I/O libraries that buffer blocks, such as *Stdio*. For example, for reading, *Stdio* reads a block of data from the file and then services application reads from this block. However, in practice, supporting a stream interface using mapped files is substantially easier. First, the same code can be used for allocating a region for either reading or writing; with *Stdio*, the buffers for reading and writing have to be managed in different ways. Second, when a file is open for both read and write access, the same region can be used for both modes, making it much simpler for a stream to change modes (e.g. when doing a read after a write). In contrast, many implementations of *Stdio* require the application to perform explicit seek operations when changing modes [KV91].[5]

[5] As another example, with every implementation of *Stdio*

Finally, since the file offset is maintained by the library, seek operations can be supported without any communication to the file system.

It is possible to use a hybrid approach, where mapped files are used when the resulting performance is expected to be high and Unix I/O is used otherwise. For example, the *Sfio* library uses mapped files for reading only. The MSIO library described in Section 5 can be configured in three different ways, namely: (1) to use mapped files for all file I/O, (2) to use mapped files for modifying data before the end-of-file (EOF), and `write` operations for modifying data past the EOF, and (3) to use *Unix I/O* for all file operations.

There may be reasons other than performance for not using mapped files for all file I/O. For example, with mapped files, some operating systems ignore modifications to a file block past EOF. Therefore, to ensure that the data will not be lost, the application must first change EOF[6] before adding the new data. The disadvantage of this approach is that the EOF changes before the new data becomes available, which means that other programs cannot use EOF to determine how much valid data is contained in the file. In contrast, with *Unix I/O*, the application prepares the data for a `write` operation in a private buffer. When the `write` operation is performed, the file system copies the data from the private buffer to the system buffers and then (if necessary) updates the EOF.

The file system of the Hurricane operating system allows mapped files to be modified without forcing EOF to be extended. Applications explicitly set EOF when a modification is complete.[7] This approach has the advantage *Unix I/O* has in that data remains private to the application performing the write, while not having the disadvantage that data must be copied from a private buffer. This makes it possible to configure the MSIO library on Hurricane to use mapped files for all file I/O. On SunOS and IRIX, the MSIO library is configured to use `write` operations whenever the file size must be extended.

with which we have experimented, the library does not properly handle interspersed `getc` and `putc` operations, while correct behaviour can be trivially supported using mapped files.

[6] This can be accomplished by doing a `ftruncate` to the new file size, or a `lseek` and then a `write` operation. IRIX supports an optional flag to mmap that causes the file to grow every time a page past the current EOF is modified.

[7] When all open references to a file are complete, the file system garbage collects any blocks that have been written to disk past EOF.

4 The Alloc Stream Interface

In the course of developing a new user level I/O library to support the *emulated Unix I/O* and *Stdio* interfaces, we observed that a large proportion of the library time was being spent copying data between the library and application buffers. This motivated the design of a new interface called the *Alloc Stream Interface*, with the following goals:

1. Applications must be able to use the same interface for all I/O whether to files or stream devices such as terminals.
2. For performance, the interface should allow the application access to the internal buffers (mapped regions) of the library, rather than forcing the library to always copy data between the library and application buffers.
3. The interface should allow for implementations that can exploit mapped files.
4. Multi-threaded applications should be supported. This implies that the interface semantics must be clearly defined in the case where multiple threads are accessing the same stream, and it implies that the amount of time a stream needs to be locked should be minimized.
5. The interface should be easy to use, and preferably be similar to an interface programmers are already familiar with.
6. Because of the wide acceptance of current I/O interfaces, and in order to gain acceptance, the interface must in some way be compatible the other interfaces being used, such as *Stdio*.

4.1 The High Level ASI

The most important operations defined by *ASI* are depicted in Figure 4.1. `Sopen` opens the file named by *path* and, if the open succeeds, returns a *handle* to be used to identify the stream in subsequent accesses. The stream will always be in one of two modes, namely read mode or write mode. If the stream is opened for read-only or read-write access, the mode of the stream defaults to read mode; otherwise it defaults to write mode. (The mode of the stream can changed using the *set_alloc_flags* macro.) `Sclose` closes the named stream after unmapping any mapped files and flushing any buffered data. The remaining operations are modeled after the Unix memory allocation interface (i.e. `malloc`, `realloc`, `free`).

ASI considers only files to be *truly* read-write streams. If a non-file stream, such as a terminal or a

```
FILE *sopen( char *path, int access_flags, int creat_flags, int *rc ) ;
int  sclose( FILE *fl ) ;
void *salloc( FILE *fl, int *length ) ;
int sfree( FILE *fl, void *ptr ) ;
void *sallocAt( FILE *fl, int *length, int *offset, int whence ) ;
void *srealloc( FILE *fl, void *start, int oldlen, int *newlength ) ;
```

Figure 2: ASI operations

socket stream is opened for read-write access then it is handled as if there are two independent streams (i.e. a read and a write stream) addressed by the same handle. In other words, only with files will a write access change the data subsequent reads will obtain.

Reading from a Stream

`Salloc` is used together with `sfree` for accessing a stream. `Salloc` returns a pointer to a region of memory that contains the requested data, and advances the stream offset (i.e. the offset in the stream for the next access) by the specified length. `Sfree` tells the library that the application has finished accessing the data, at which point the library can discard any state associated with it. The length parameter to `salloc` is a value return parameter, initialized to the amount of requested data and modified on return to indicate the amount of allocated data. In the case of an error, a NULL pointer is returned and the length parameter is set to a negative error code.

The name `salloc` was chosen to indicate that data is *allocated* from a stream for use by a particular thread. That is, the library considers data requested by a `salloc` operation private until a `sfree` operation indicates that the thread has finished using the data. In no case will two salloc operations return pointers to the same data. If two threads wish to concurrently access the same data, then the one that allocated the data must pass the pointer to the other thread.

For a file in read mode, only data already in the file can be allocated. If an application attempts to allocate past the EOF, a length of 0 will be returned. The application should never modify data obtained from `salloc` in read mode. If the stream is not associated with a file, then the modifications will not be visible on the other side of the stream (e.g. the screen of a terminal). If a file is opened read-only, modifications could cause the program to segment fault.

Note that because `salloc` returns a pointer to the data, it can ensure that the alignment allows for memory mapping optimizations.

Writing to a Stream

In write mode, `salloc` returns a pointer to a region of memory where the data should be placed, and advances the stream offset by the specified length. The application can then use this region as a private buffer in which to place data to be written to the stream. `Sfree` tells the library that the application has finished putting data in that region, at which point the library can write out the modified data. If the stream is to a file, the length of the file is automatically extended on `sfree` if the allocated data is past EOF. For a non-file stream, 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.

Repositioning in a stream

`SallocAt` can only be used for files. It causes the stream offset to move to a particular location in the file and performs a `salloc` at that location. It is equivalent to a *Stdio* `fseek` followed by a `salloc`. For parallel applications, the *Stdio* and *Unix I/O* stream interfaces have the disadvantage that if a thread performs a seek operation in order to access data at a particular location in the file, it is possible for another thread to (indirectly) modify the file offset before the first thread is able to access the data. To resolve this problem, `sallocAt` causes the stream to be locked until the requested data has been allocated. The offset parameter to `sallocAt` is a value return parameter, which on return indicates the position of the file offset in the file.

When a thread allocates data from a stream, the library ensures no other thread can allocate the same data by having `sallocAt` block until all allocated regions are freed if it causes the file offset to move backwards in the file. This handling of `sallocAt` is stricter than necessary, but has the advantage that it reduces the amount of state that needs to be maintained for `salloc` and `sfree`. A less restrictive policy would allow for more concurrency, but would reduce the performance for these common operations.

Changing the amount of data allocated

If more data than necessary was allocated, it should be possible to return data so that it can be allocated later (possibly by another thread). Also, since library buffers are a convenient location to prepare data for writing, the application may want to `salloc` a large data space, and then shrink the region back to the actual amount of data prepared.

`Srealloc` allows the application to shrink or grow a previously allocated region. In the case of the last allocated region, `srealloc` will reposition 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 moved to byte 21 of the file. Moreover, if the file size would change with the corresponding `free` operation, then it changes according to the `srealloc` and not the original `salloc` operation. Using the previous example, if the file size was originally 0 bytes, then the file size is set on sfree to be 20 bytes and not 200 bytes.

Error Codes

All *ASI* operations return full error codes, which always have a negative value. `Sopen` returns the error code in rc, and `salloc`, `sallocAt` and `srealloc` return (negative) error codes in the length parameter. This is in contrast to *Unix I/O* and *Stdio* that return the error code in the global variable `errno`, which is not suitable for multi-threaded applications.

4.2 The Low Level ASI

The high level interface described above supports simultaneous access to a stream by multiple application threads. Therefore, all operations require the stream to be locked when various critical data structures of the library are being accessed. Also, the mode of the file (i.e. read or write) always determines whether a `salloc` operation is for reading or writing. There is a lower-level interface to ASI that includes less restrictive operations for accessing a stream.

unlocked operations: 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. These operations are useful in two cases: 1) if the application is sequential and does not want to incur the overhead of locking, 2) if in a parallel application a particular thread wishes to acquire a lock for the stream while performing a number of operations to that stream. `SLock( stream )` and `SUnlock( stream )` operations can be used by the application to explicitly lock a stream when the unlocked operations are being used.

mode variable operations: The (capitalized) `Salloc`, `Srealloc` and `SallocAt` operations differ from the corresponding (un-capitalized) high level operations in that the mode is specified on each call. For example, in

```
void *Salloc( FILE *fl, int flags,
                        int *length ) ;
```

the flags parameter can either be set to `SA_READ` or `SA_WRITE` to indicate whether the thread is allocating for reading or for writing. This interface is useful if different threads are concurrently accessing the same stream, some for reading and some for writing.

macro operations: With ASI, the actual amount of code executed to access a stream is very small in the common case (i.e. when the data to be allocated is already in a buffer or mapped region), so the overhead of a procedure call is significant. Therefore, to minimize the cost for performance critical portions of a program a set of *fast* macros that correspond to both the high and low level procedures are provided. For example, `f_Salloc`, `f_Srealloc` and `f_SallocAt`, have the same parameters as the `Salloc Srealloc` and `SallocAt` calls, but are implemented as macros.

5 The Mapped Stream I/O Library

The Mapped Stream I/O (MSIO) Library is an implementation of the ideas presented in this paper. It supports: *emulated Unix I/O*, *Stdio*, and the *Alloc Stream Interface* (ASI). The *emulated Unix I/O* and *Stdio* interfaces are supported by a layer of software above ASI. Implementations of the MSIO library exist on the Hurricane, IRIX and SunOS operating systems.

In this section, we consider the data structures of the MSIO library and the way these structures are used for typical stream accesses. As shown in Figure 3, there is one Client I/O State structure (CIOS) associated with each stream and either one or two sorted lists of regions, where a region is either a portion of the address space mapped to a file, or a buffer holding data associated with that stream. In the case of a mapped file, there is only one region list used

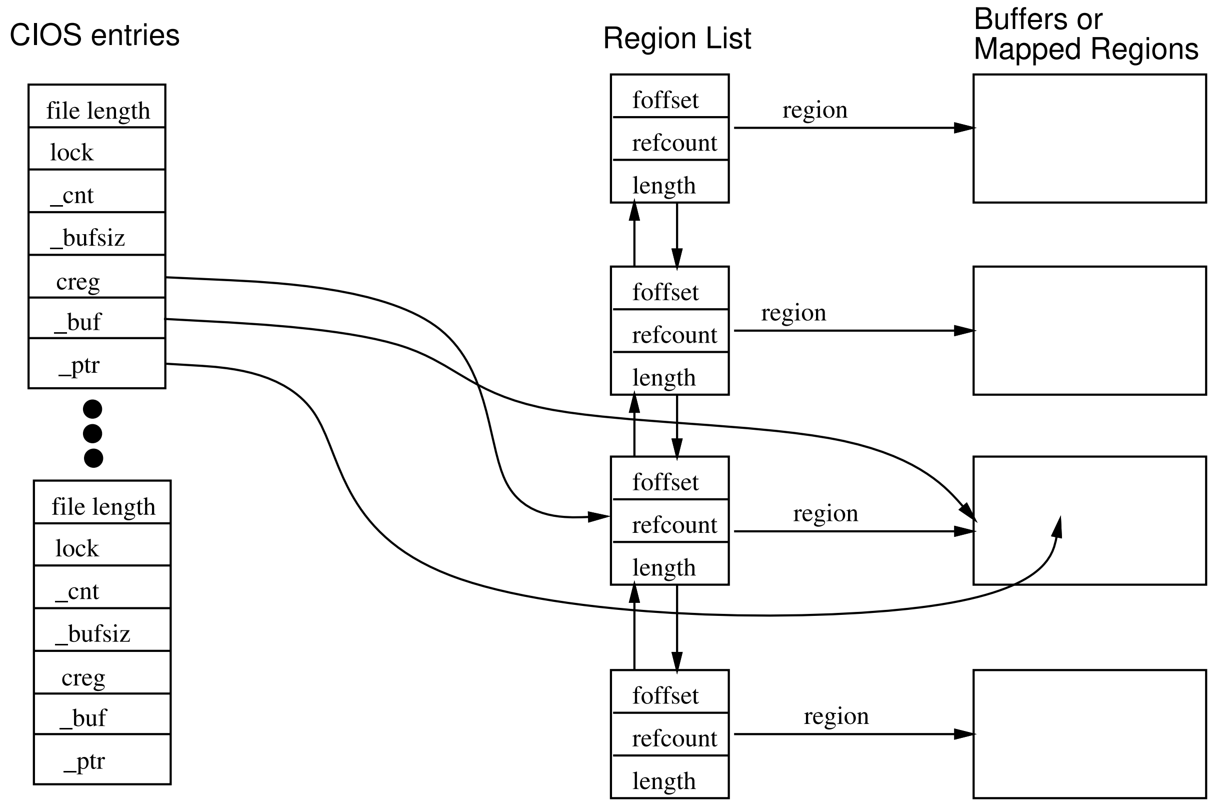


Figure 3: Client I/O State

for both read and write accesses. Otherwise, if the stream is open for read-write access, there are two region lists; one for read accesses and one for write accesses. The CIOS structure contains the following important fields:

`_buf` points to the *current region*, the region that will be used for the next access to the stream,

`creg` points to the region list element corresponding to the current region,

`_bufsiz` is the size of the current region,

`_ptr` points to the data to be used for servicing the next access to the stream,

`_cnt` indicates the number of bytes remaining to be allocated in the current region.

For each region in the region list, the MSIO library maintains the offset into the file, a reference count for the number of outstanding references to the region (i.e. the number of `sallocs` performed without a corresponding `sfree`), and the length of the region. The current region always has a reference count of at least one.

5.1 The Common Case

The most common ASI operations are `salloc` and `sfree`. In this section, we describe the algorithms used for these operations assuming the region being accessed is the current region, which is the common case.

The algorithm for `salloc` is shown in Figure 4. On `salloc`, the library first locks the CIOS structure to ensure that no other thread is currently accessing it. It then checks whether the operation is allowed on that region and ascertains that the `salloc` can be satisfied by the current region[8]. The `_cnt` is then reduced by the size of the access and `_ptr` is increased accordingly, the `refcount` of the region is incremented by `salloc` to indicate that the region is actively being accessed, the CIOS is unlocked, and the original version of _ptr is returned. At this point, `_ptr` points to the next byte to be allocated from the stream.

For `sfree`, the library locks the CIOS, searches the region list to find the region that includes the data

[8] That is, either the `_cnt` is greater than the amount of requested data, or the `salloc` is for writing and there is sufficient room in the buffer.

```
void *salloc( FILE *cios, int *length )
  {
    acquire( cios->lock ) ;
    if( (mode of region and
                   operation match) &&
        (cios->_cnt >= *length) ||
        ( (operation for write) &&
          (room in buffer) ) )
      {
        char *tmpptr = cios->_ptr ;
        cios->_cnt -= *length ;
        cios->_ptr += *length ;
        cios->creg->refcount++ ;
        release( cios->lock ) ;
        return tmpptr ;
      }
    add a new region and
    allocate the data from that ;
  }

int sfree( FILE *cios, void *ptr )
  {
    acquire( cios->lock ) ;
    if( operation to creg )
      {
        cios->refcount-- ;
        release( cios->lock ) ;
        return OK ;
      }
      free region from region list ;
  }
```

Figure 4: `Code for salloc and sfree`

being freed, and decrements the reference count of that region. If the reference count of the region is not zero, `sfree` returns to the application. Otherwise, `sfree` deallocates the region as described below. In the common case, the data being freed is in the current region, so the code can be optimized as shown in Figure 4, since the refcount is guaranteed to always be greater than zero.

5.2 Mapped Files

When `salloc` can not be satisfied by the current region and the operation is to a mapped file, then the MSIO library: 1) maps a new region that includes the requested data into its address space, 2) initializes a new region structure to point to the region and inserts it into the region list, 3) initializes the `_buf`, `_bufsiz`, `_ptr`, and `_cnt` variables, and finally 4) changes `creg` to point to the new region structure.

When the reference count is zero, on a `sfree`, the library discards the mapped region. This entails removing the region from the region list, unmapping the region and freeing the structures associated with it. In the case where data has been modified, the library need not inform the operating system that the page has been modified, since the memory manager will detect this on its own through the dirty bit associated with the page.

As discussed in Section 3, data allocated past the end of file must be handled in a special manner. The MSIO library can be configured to either handle writes past EOF as mapped files or by doing regular *Unix I/O* write operations to extend the file. The latter case is described in the next section. For the former, the MSIO library must interact with the file system to indicate that the file length has changed. Whether the file length has increased is checked whenever `creg` is changed (for example, when a new *current region* is mapped in). This can be determined from the information in the CIOS structure and region list element.[9] If the file length has changed the new file length is recorded in the CIOS structure. Whenever a region is discarded, the library informs the file system that the file length has changed to include this new region.

The library could also inform the file system that the file length has changed: 1) when `sfree` is called, and 2) when the file is closed. The approach we chose is a compromise between reducing the number of library/OS interactions and making the data available to other applications as soon as possible.

5.3 Non-mapped I/O

If a non-mapped stream is being accessed for both read and write operations, the MSIO library maintains a separate region list for reading and writing. If the mode of the current region does not correspond to the mode of the operation, it is necessary to change the mode of the stream, and with it the region list being used and the corresponding variables in the CIOS structure.

When `salloc` cannot be satisfied by the current region, the library: 1) allocates a new buffer that is (at minimum) large enough to satisfy the request, 2) copies the portion of the data in the current region to the new buffer and, 3) if the salloc is for a read, performs a read operation to the operating system for the remainder of the data. After this point, the

[9] Special care must be taken when `sallocAt` is called, however, this is beyond the scope of this discussion.

reference is handled in the same manner as if it was to a file.

As with mapped files, a region is not discarded until the reference count is zero. At this point, if the buffers are for reading, they are discarded by removing the buffers from the region list, and by freeing both the buffers and the associated structures. A write buffer cannot be discarded until all previous regions have also been discarded. This is important to ensure the order of modifications according to the order of `salloc` operations. When a region can be discarded, the data in that region and all subsequent regions with zero reference counts are written out before discarding the regions.

5.4 Conversion between ASI, emulated Unix I/O and Stdio

ASI is more general than the *emulated Unix I/O* or *Stdio* interfaces because ASI can be used to implement both of the other interfaces with little or no overhead. In the MSIO library, *emulated Unix I/O* and *Stdio* interfaces are supported by a layer of software above the ASI layer. A simplified version of the algorithm used to implement an *emulated Unix I/O* read with (mode variable) ASI operations is shown in Figure 5. `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.

One major advantage of using ASI for implementing other interfaces is that the copying of data from the library to the application is performed with the stream unlocked. This restricts the concurrency to a particular stream far less than if the stream had to be locked for the entire read operation.

An application can freely intersperse calls to all three interfaces. To support this, the CIOS contains a superset of the information used in the `_iobuf` structure of the Stdio library. This allows for source code compatibility to programs that assume the structure used by Stdio. The `stdio.h` file that is included with the library defines the `_iobuf` structure as a `CIOS` structure. Therefore, no conversion between a ASI stream and a Stdio stream is necessary; that is, they are pointers to the same data structure type. The CIOS structures are organized into an array, so that by indexing into the array *emulated Unix I/O* calls can refer to the same stream. If a stream was created using an *emulated Unix I/O* call, the application can use ASI commands on that stream after converting the file descriptor to a pointer to a CIOS structure using the `fileptr` macro defined in `stdio.h`. Similarly, the application can convert the CIOS structure pointer to a file descriptor using the `fileno` macro.

Although the current implementation does not support this, *Stdio* object code compatibility may be important, for example, in the case that the operating system supports dynamically linked libraries. Since the *Stdio* interface is fully supported, the only considerations are the data structures and macros provided by *Stdio*. The macros defined by *Stdio* work with the subset of the CIOS that is compatible with the `_iobuf`, therefore, the macros are not a problem. In order to be object code compatible, rather than having all information in the CIOS in a single table, it would have to be split into a table of `_iobuf` structures, and a parallel table that contained the remainder of the information in the CIOS structure.

Although the *Stdio* interface is fully supported by the MSIO library, the application writer should take care in using *Stdio* operations that constrain buffering, for example, by calling `setbuf`. These operations are typically used to increase the performance of I/O, but with the MSIO library, `setbuf` actually hurts performance, since data must be copied to and from the specified buffer on each call to the *Stdio* interface.

6 Measurements

The MSIO library was originally written for the Hurricane operating system, and is used on a daily basis by users of that system. In order to allow MSIO to be compared to *Unix I/O* and other implementations of *Stdio*, it was ported to SunOS and IRIX.

The measurements shown in Table 2 are in seconds of time as obtained by the Unix time program. For utilities that use *Stdio*, there are three versions of the code: the original version (under the heading Stdio/Unix I/O) linked to the IRIX version of the Stdio library, the original version linked to the MSIO library, and the version modified to use ASI. The dataset used in these tests was either one or two 3 million byte files.

The performance numbers for applications using the MSIO library on machines using IRIX or SunOS does not match the results obtained on Hurricane, an operating system designed with mapped files in mind. However, significant performance advantages have been demonstrated. In particular:

1. Stdio applications that are re-linked to use the MSIO library uniformly perform better. For example, as can be seen in Table 2, `diff` takes 40% less time and `cut` takes 14% less time. Even `compress` and `uncompress`, which are computationally intensive, show some improvement.

```
int read( int fd, char *buf, int length )
 {
   int rc ;
   ptr = Salloc( stream, &length, SA_READ ) ;
   if( length < 0 ) RETURN_ERR( length ) ;
   bcopy( ptr, buf, length ) ;
   if( (rc = sfree( stream )) < 0 )
     RETURN_ERR( rc ) ;
   return length ;
 }
```

Figure 5: `Read implemented using ASI`

command	Stdio/Unix I/O	msio lib, Stdio	msio lib, ASI
cmp	0.90		0.60
wc	2.05		2.00
diff	1.40	0.90	0.55
compress	28.20	27.10	27.00
uncompress	8.45	8.35	7.50
cut	4.00	3.45	2.75

Table 2: *Measurements, in seconds, on IRIX of a number of BSD4.3 Reno release system utilities.*

2. Stdio applications that are re-written to use ASI directly perform significantly better than the Stdio version linked to the MSIO library. For example, `diff` improves by a further 40% (or close to 3 times as fast as the original), and `cut` improves by a further 20%.

3. Applications that use the Unix I/O interface for reading also get significant performance advantages when converted to ASI directly. For example, `cmp` is 35% faster.[10]

7 Conclusions

File I/O is increasingly becoming a performance bottleneck on many systems, prompting researchers to address this problem through hardware and software techniques. In this paper we have explored software based techniques that reduce the overhead of file I/O by, for example, reducing data copying and the number of system calls as well as improving the use of memory. Our techniques exploit memory mapped files as much as possible, leading to substantial performance gains as shown in Section 2. For a further reduction in overhead, we found it necessary to define a new interface for I/O we call ASI. This new interface was prompted not only by performance arguments, but by the inadequacy of *Stdio* and *Unix I/O* for multi-threaded applications. ASI has the following advantages:

Generality: *ASI* is a byte-oriented stream interface, which allows an application to access an arbitrary number of bytes without specifying a particular offset (e.g. location in a file). This type of an interface can be uniformly used for all types of I/O.

High Performance: 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. Moreover, since the library chooses the address of the data, it can ensure that the alignment allows for such optimizations as memory mapping.

Support for multi-threaded applications: In order to be re-entrant, an interface must have a well defined behaviour when multiple threads are accessing the same stream. Allowing the application access to the internal buffers of the library can have significant performance advantages. However, it is then necessary for the ap-

[10] `Wc` is a special case, where even though the I/O time improved substantially certain compiler optimizations where not possible with ASI, leading to performance that was about the same for the Unix and ASI versions.

plication to inform the library when it has completed using allocated data, so that the buffer can be re-used. With ASI, `sfree` is used for this purpose. In contrast, *Sfio* has no such operation and their `sfpeek` operation (which corresponds to `salloc`) can therefore only be used by single threaded applications (where any operation to the stream implicitly frees the data allocated by a previous `sfpeek`).

Another way in which ASI supports multi-threaded applications is that it allows for a high degree of concurrency in accesses to a particular stream. Since with ASI data is not copied to or from user buffers, the stream is locked only while the library's internal data structures are being modified. Hence, the stream is locked only for a short period of time, and, since access to data occur while the stream is unlocked, multiple threads can page fault on different pages in the same mapped region and thus have their I/O performed in parallel.

All operations defined by ASI return an error code. In contrast, most *Stdio* implementations return error codes in a single `errno` variable that is used for all streams. This is not suitable for multi-threaded applications.

Finally, with `sallocAt`, a thread can allocate data from a particular location in a file without interference from other threads. With most implementations of re-entrant *Stdio*, the application must explicitly lock the stream if it wants to read data from a particular location in a file.

Ease of use: ASI closely parallels the Unix memory allocation interface (i.e. `malloc`, `realloc`, `free`) providing programmers with an interface they are already familiar with. Our experience indicates that in most cases, it is just as easy to write an application using ASI as it is using the *Unix I/O* or *Stdio* interfaces. In fact, for some applications ASI is easier to use, since buffer management and I/O are combined into a single set of calls (i.e. the application need not do a malloc and then a read or write).

We have implemented a library that supports the ASI interface, together with the *emulated Unix I/O* and *Stdio* interfaces. The *emulated Unix I/O* and *Stdio* interfaces are implemented using ASI for all I/O, proving the generality of the ASI interface. It is important to note that the performance advantages of ASI can be exploited by re-writing just the I/O critical portions of a program; calls to all three interfaces can be freely interspersed. We have ported the library to a number of platforms, including SunOS, IRIX and Hurricane, and have shown (in Section 5.4) that the performance of applications is consistently better when linked to our library rather then the original system library.

In future work, we intend to explore the effectiveness of ASI on systems with parallel disks, for which we think ASI 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 ASI would allow for better exploitation of parallel disks (especially on multiprocessor systems) not only because ASI reduces data copying substantially, but also because of the higher degree of concurrency ASI naturally allows.

Another project related to this work is a transaction management system that uses mapped file I/O for accessing the data base. With this system, transaction servers communicate by accessing shared mapped files, and synchronize using locks placed directly in data base files.

Acknowledgements

We would like to thank K.-Phong Vo, Ken Lalonde, Dave Galloway and Mark Moraes, for their help and useful comments. Bill Shannon and Dean Kemp from Sun were very useful in advising us about difficulties we had in porting our package to SunOS. Finally, Benjamin Gamsa and Jonathan Hanna contributed in improving the presentation of this paper.

References

[ABB+86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *1986 Summer USENIX Conference*, 1986.

[BJ91] A. Lester Buck and Robert A. Coyne Jr. An Experimental Implementation of Draft POSIX Asynchronous I/O. In *USENIX-Winter 91*, pages 289–306, 1991.

[Che87] David R. 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 *ACM Sigmetrics Conference*, 1991.

[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.

[JCF+83] William Joy, Eric Cooper, Robert Fabry, Samuel Leffler, Kirk McKusick, and David Mosher. 4.2BSD System Manual. 1983.

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

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

[LS90] Eliezer Levy and Abraham Silberschatz. Distributed File Systems: Concepts and Examples. *ACM Computing Surveys*, 22(4):323–374, December 1990.

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

[SUK] M. Stumm, R. Unrau, and O. Krieger. Hurricane, A Shared-Memory Multiprocessor Operating System Structured for Scalability. submitted for publication, 1991.

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

[Zho88] S. Zhou. A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Trans. on Softw. Eng.*, 14(9):1327–1341, September 1988.

Availability

The MSIO library, as well as all programs used in our tests are available on request from the authors: Orran Krieger (okrieg@eecg.toronto.edu), Michael Stumm (stumm@eecg.toronto.edu) and Ron Unrau (unrau@eecg.toronto.edu). Mail can be sent to:

```
Orran Krieger
Graduate Office
Department of Electrical Engineering
University of Toronto
Toronto, Canada, M5S-1A4
```