SFIO: Safe/Fast String/File IO

David G. Korn
Kiem-Phong Vo

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes *Sfio*, a new input/output library, that can be used as a replacement for *Stdio*, the C language standard I/O library. *Sfio* is more complete, consistent, and efficient than *Stdio*. New facilities are provided for convenient, safe and efficient manipulation of data streams. An *Sfio* stream may be entirely memory resident or it may correspond to some actual file. Alternative I/O disciplines can be applied to a stream to customize its behavior with respect to data transformation and exception handling. Stream pools can be maintained to guarantee stream synchronization when switching streams for I/O operations. Separate streams can be stacked on one another to make new virtual streams. Both source and binary compatibility packages are provided allowing *Stdio*-based programs to benefit from the new library without recoding. *Sfio* has been used successfully in a number of applications including many rewritten standard system utilities. Benchmark timings show that *Sfio* performs very well against *Stdio* and standard utilities can gain substantial performance improvement when based completely on *Sfio*.

1. Introduction.

The C language does not have any input/output facilities. I/O operations are typically negotiated via system or function calls. Most C programs routinely use the standard I/O library, *Stdio*[KR]. The library was written by Dennis Ritchie in the mid 1970's to replace an earlier library called the portable I/O library. *Stdio* was a part of UNIX[1] Version 7 which is the ancestor of virtually all current UNIX systems. Central to *Stdio* is the idea of a buffered stream that models an input or output file. Application code can read or write arbitrary chunks of data with confidence that the underlying buffering scheme will perform efficiently. Beyond efficiency concerns, *Stdio* also provides other functions to read and write various types of data included formatted records.

Today the standard I/O library is ubiquitous in C programming environments. Features from diverse versions have been amalgamated in the ANSI C standard [AN]. Despite its success, there are recognized deficiencies in *Stdio* both in its design and various implementations. For example, the library does not provide an efficient way to transfer large amount of data between streams. This weakness causes many standard system utilities to revert to direct use of the underlying system calls. The poor performance of *Stdio* for line-oriented processing was addressed in the design of yet another I/O library, *Fio*, done by Andrew Hume for the 9th Edition UNIX system [Hu]. The interface of *Stdio* is incomplete, prompting a number of attempts over the years to extend it. The ability to perform both input and output on a stream opened for both read and write was added in the early 1980's by Andrew Koenig. A few years later, the ability to tune buffer size and set line buffering mode was added in the BSD UNIX version. In our own experiences, a frustrating problem with *Stdio* is the lack of any facility for graceful handling of exceptions. A bad case for most current versions of *Stdio* is when the `write()` system call is interrupted. Most of the time, this would cause a loss of data. Equally perplexing is the lack of any facility for synchronizing buffers of streams that share the same underlying file, e.g., `/dev/tty` and the streams `stdin`, `stdout` and `stderr`. Thus, there is no way to guarantee serialization of I/O operations short of flushing the buffer after each operation.

---

1. UNIX is a trademark of AT&T Unix System Laboratory

The *Sfio* library was written partly to overcome *Stdio*'s deficiencies. One may ask why not just rewrite *Stdio* and maintain if not binary compatibility, at least source compatibility? One of us, David Korn, tried this tack a number of years back. Numerous link editing problems occured because of name clashing. A major problem is the use of a fixed array of streams in *Stdio*. This makes it impossible to extend the stream data structure without violating binary compatibility. This limits the usefulness of a new implementation unless one can recompile the entire system and its host of applications. More important, preserving the *Stdio* interface means also preserving its various problems. We feel that a better interface could be designed that would avoid inconsistency, reduce redundancy, and be easily extensible. The good news is that by providing a completely new interface that occupies a different name space from that of *Stdio,* we are able to build both source and binary compatibility packages to *Stdio*. On a system with shared libraries, the compatibility packages can be installed to provide transparent access to *Sfio*.

The remainder of the paper will go into some details on the design and implementation deficiencies in *Stdio*. Then, the motivations behind *Sfio* and its main features are discussed. Benchmark data comparing the performance of *Sfio*, *Stdio* and *Fio* is presented. This includes data comparing standard versions of system utilities and our reimplementations which exploit the new abstractions provided in *Sfio*. Finally, the manual pages for *Sfio* are attached for detailed descriptions of the available functions.

2. The In-Problems with Stdio

There are many problems with the current design and implementations of *Stdio*. For reasons that should become clear soon, we call them the *in-problem*s. In the following discussion, the reader should keep in mind that there are many versions of *Stdio* and certain problems may apply to only particular versions.

- *Incorrectness:* A design defect in *Stdio* is that it does not check the type of a stream when I/O operations are performed. For example, it is perfectly ok to open a stream for read only, then write data into it. Here is an example:

  ```
  f = fopen("data_file","r");
  getc(f);
  putc('a',f);
  ```

  Of course, only the data in the internal buffer of the stream `f` is changed when `putc()` is issued, not the underlying file. However, interesting consequences can arise in the application code.

  At the implementation level, the library also suffers from a number of defects. The case that caught our attention is when a system call gets interrupted by a signal. Here is an example:

  ```
  alarm(1);
  fwrite(buf,1,1000000,stdout);
  ```

  In this code fragment, the buffer being written out greatly exceeds the internal buffer of the standard output stream. Therefore, from time to time, a `write()` system call will be issued to flush data from the internal buffer. If the alarm signal arrives at an importune moment, the system call is interrupted and only a part of the buffer is written out. Most current *Stdio* implementations will note that an incomplete write had happened but nothing is done about the part of data not yet written out. The result is a loss of data.

- *Inefficiency*: *Stdio* enforces the notion that the application has to provide a buffer (different from the stream buffer) to transfer data in and out of streams. This causes unnecessary data copying in many common cases. In addition, it makes it hard to get data whose size may not be known a priori to a call, for example, a line of text. There are recent proprosals to add a new function `fgetline()` precisely for this purpose. Even if this is done, it is a kludge to get around the fundamental problem of how to safely access stream data without unnecessary data copying.

  In many versions of *Stdio*, an I/O operation entails reading or copying data into the internal stream buffer before transferring it to the final destination. For reading or writing an amount of data that exceeds the buffer size, this is wasteful. For example, a call like `fread(buf,10,BUFSIZ,stdin)` may imply multiple invocations of the `read()` system call.

- *Inadequacy:* Parts of *Stdio* simply fall short of what the interface promises. For example, the `fread()` and `fwrite()` functions promise I/O of objects which may have large sizes. Since the system calls `read()` and `write()` only promise I/O of byte streams, this is a promise that is hard to keep. For example, if a structure of size, say 100, is written out to a pipe but the `write()` system call is interrupted after the first 10 bytes is copied, `fwrite()` can only report that no structure has been written out. This is misinformation at best.

  Some implementations of *Stdio* are also inadequate resulting in unnecessary and complex interface requirement. For example, *Stdio* allows streams to be opened for both read and write. However, applications are not allowed to arbitrarily mix input and output operations. Instead, extra `fseek()` calls must be inserted before switching mode. This requirement results because current implementations of *Stdio* do not have a way to know what mode the stream is in. From an application's point of view, if a stream is passed as an argument to some function, since there is no way to know what mode it is in, the application may have to use a defensive measure causing many unnecessary calls to `fseek()`.

  Buffer synchronization for streams sharing the same file is a good source of problems for *Stdio* programmers. The following code fragment can cause unexpected results:

  ```
  printf("Please type something:");
  fgets(buf,sizeof(buf),stdin);
  printf("You just typed: %s",buf);
  ```

  Because of buffering, the user will not see the prompt until s/he has typed a line of input. Then both the prompt and the typed text will be printed. Strictly speaking, this is not an error on *Stdio* part. It just lacks a facility that would allow the above ‘‘natural’’ piece of code to work as expected (see section 3.4). A sharp reader may bring up `setbuf(stdout,NULL)` or `fflush(stsdout)` here but s/he should be forewarned at least of the efficiency cost incurred in the former case and the inconvenience of the latter. This is not to mention other more complex buffer synchronization situations such as two streams writing to the same file. Can the output be guaranteed to serialize logically?

- *Insecurity:* Parts of the interface of *Stdio* are notoriously insecure. For example, the *gets()* and *sprintf()* functions allow overwriting data space that follows a buffer. Below is an example of unsafe coding style possible with *Stdio*. The existence of such a coding style in standard system utilities was rumored to have been put to good use in at least a few system break-in techniques.

  ```
  char buf[1];
  sprintf(buf,"1234");
  ```

  More benign but equally troublesome for applications is the possibility of concurrent access of streams due to signal handling. In such cases, the behavior of the stream is unspecified since internal stream pointers may be in a bad state during a concurrent access.

- *Inconsistency:* An annoying problem with the interface design of *Stdio* is the positioning of the stream argument in a function call. For example, `fseek()` and `setbuf()` require the stream argument to come first while `fread()` and `fwrite()` require it to come last. Though this is not a fatal flaw, inconsistency of this type steepens the learning curve for programmers. Some of us, after many years of programming with *Stdio*, still often have to open the manual to find the correct function calling sequences.

- *Incompleteness:* This is an open area where everyone's wish list can be inserted. However, *Stdio* does lack a number of fundamental features. Most important is a lack of some way for applications to handle exceptions such as the end-of-file condition or interrupted system calls. More along the efficiency line, there is no safe method to directly access the internal buffer of a stream. The inefficiency resulting from this have forced many applications to invent their own buffering scheme.

  Except for `sprintf()` and `sscanf()`, the library is closedly tied to files. There is no capability to manipulate arbitrary data streams so the convenience of the I/O operations are lost for memory manipulation. This is a lack recognized by providers of the *C++ Iostream* package [St].

3. The Sfio Library

3.1 Design Considerations

The *Sfio* library was built to correct the problems of *Stdio* and to simplify our life as programmers. Interface design considerations concluded for us that a rewrite of *Stdio* just wouldn't do so we started fresh. Below are some of the considerations that went into the design of *Sfio*.

- *Efficiency*: A package as basic as *Sfio* must be efficient in its implementation. The I/O primitives in *Sfio* are designed to avoid any unnecessary data copying. New and efficient algorithms are used when appropriate. For example, the *Sfio* `sfprintf()` and `sfscanf()` functions use new data conversion algorithms that are much faster than their counterparts in *Stdio*.

  In modern UNIX systems, there are often multiple ways to do I/O with different trade-offs. For example, on SUN OS and System V Release 4, it is possible to use memory mapping to access file data. *Sfio* takes advantage of these facilities where appropriate to speed up thoughput.

  Beyond internal efficiency, an I/O package like *Sfio* must also provide primitives that allow an application to be efficient in its I/O manipulations. An example is `sfreserve()` which allows an application safe access to data blocks in a stream without having to allocate separate buffers. We have rewritten many standard system utilities based on this primitive with good performance improvement. It is worth noting that because of efficiency problems with *Stdio*, popular tools such as *cat* or *cp* were often rewritten using raw system calls. However, by going to lower level system calls, these implementations cannot take advantage of new efficient features such as memory mapping without further recoding. Our implementations of these tools based on *Sfio* achieve the same or better efficiency.

- *Consistency*: All *Sfio* constants begin with the prefix `SF_` and all functions begin with the prefix `sf`. Where applicable, *Sfio* functions simply emulate their system call counterparts. For example, the `sfread()` function is called as `sfread(stream,buf,size)`. This example also shows that the stream argument always positions first in any function call. Then, if required, the second argument is a buffer and the third would be the buffer size.

  All streams are handled in the same way by the application. There is no difference between string (memory only) and file streams. This is used by `sftmp()`, a function to create streams for storing and retrieving temporary data. For efficiency, such a temporary stream starts out being a string stream with a buffer of some size decidable by the application. A stream discipline is set up so that if this memory area is exhausted, a true temporary file will be created. From the application's point of view such a temporary stream simply appears as if it is always a file.

- *Orthogonality:* All stream-applying primitives in *Sfio* are orthogonal so they can be used in arbitrary orders. For example, the `sfsetbuf()` functions can be called any time to change the buffer of a stream, not only when the stream is just opened. This is important for *Sfio* since it supports string streams that correspond to in-memory data buffers. In the same vein, there is no restriction on the order of I/O operations on streams that are opened for both read and write.

- *Integrity:* Except for convoluted escape hatches (e.g., `longjmp()`), the library guarantees that internal stream buffers maintain their integrity when exceptions occur. For example, if a `write()` system call is interrupted, no data will be lost (unless, of course, if `write()` itself loses it). An application can also set alternative disciplines to handle such exceptions. Streams that are opened for read only or write only will disallow the opposite operations. Finally, during an operation that modifies a stream, the stream is locked to prevent other concurrent accesses to the same stream.

- *Robustness:* We have mentioned the use of locks to prevent concurrent stream accesses. The library also prevents most unsafe operations that may cause buffer overflow. Except for string scanning patterns such as `%s` or `%[]` in `sfscanf()`, there is no way to specify a buffer without an accompanying size. The only reason for saving the current semantics of these patterns despite their insecurity is a concession for portability of the legion of existing applications. We do provide alternative patterns, `%S`, `%C` and `%{}`, which require the accompanying buffer sizes. Unbounded lines or records delineated by some arbitrary character can be read via `sfgetr()` which either uses the internal stream buffer or allocates buffer space as necessary to store the input.

- *Minimality:* As a general rule, we avoid providing functions unless they do something that cannot be done outside of *Sfio* without some loss. For example, there are no functions corresponding to the `getchar()`, `putchar()` family since they only provide short-hand notations to functions applying to the standard output stream. There are border cases where this rule is relaxed somewhat. For example, `sfsprintf()` and `sfsscanf()` are provided even though the respective input/output string could be opened for I/O as a string stream. Their popularity and the saving of creating a stream structure is worth their existence.

  At the implementation level, we try to minimize the amount of code that gets pulled in with normal compilation. For example, even though stacked streams can only be closed as a unit, the code for stack manipulation is never pulled in unless stacking is used in the application.

- *Extensibility:* No library provider can anticipate all possible needs that an application may want from the package. Thus, we have made a stream parametrizable via the use of disciplines. A discipline redefines the system calls `read()`, `write()`, and `lseek()`, and provides a function to handle exceptions. Different disciplines can be stacked on one another to make a sequence of filters. For example, the *pack* and *unpack* utilities have been rewritten based on a discipline that provides a virtual plaintext view of writing and reading packed files. The beauty in such a scheme is that the same discipline can be reused in other applications for accessing the same type of data.

3.2 Creating and Deleting Streams

Sfio defines a stream type called `Sfile_t` and a collection of functions to create, manipulate, and destroy streams. A stream can be created from a file descriptor or memory. It is common to call a memory stream a string stream. Streams can be created for reading, writing or both and its I/O mode can be set for byte-oriented or line-oriented.

The primary mechanism for creating a stream is `sfnew()`:

```
sfnew(Sfile_t* f, char* buf, int size, int fd, int type)
```

`f` is a stream pointer. If it is `NULL` or not closable, a new stream is created; otherwise, `f` is a stream to be closed and reused in making a new stream. `buf` and `size` refer to a buffer and its size when `size` is positive. If `size` is 0, the stream is unbuffered and if `size` is negative, the actual buffer will be allocated by the system. `fd`, refers to a file descriptor when the stream models a file stream and is ignored otherwise. `type` is a bit pattern defining the type of stream. It includes the bits:

`SF_STRING`: This bit indicates that the stream is an array of bytes in memory, not a file. If the stream is opened for reading, `buf` and `size` define the data and its extent.

`SF_READ` and `SF_WRITE`: These bits specify read and write mode.

`SF_LINE`: This bit turns on the line mode for a stream. The line mode of the standard input stream, `sfstdin`, is automatically turned on if the stream is unseekable. For write streams, `SF_LINE` means flushing the buffer whenever the new-line character is output. For read streams, `sfscanf()` matches literally the appearance of a new-line character in the format string. This provides a convenience for matching patterns such as `‘‘%s\n’’` on unseekable streams without unnecessary blocking.

`SF_SHARE`: This bit indicates that the stream corresponds to a file descriptor that may be shared elsewhere by a different stream or a different process. The file position will be reset to its logical location before a call to `read()` or `write()`.

`SF_APPEND`: This bit indicates that the stream is a file opened for append. On systems where this mode is not available, a seek to the end of the file will be performed before a call to `write()`.

Stream objects can also be created from higher level functions: `sfopen()`, and `sfpopen()`. `sfopen()` is similar to the `freopen()` function in *Stdio*. However, when the first argument is `NULL`, it creates a new stream. In addition, if the third argument is `"s"` or `"s+"`, the second argument, instead of being a file name, is a string to be opened for read or read and write. `sfpopen()` is similar to `popen()` in *Stdio*:

```
sfpopen(Sfile_t* f, const char* cmd, const char* type)
```

Here, `cmd` is a command connected to the application via pipes. `type` determines the type of connection. Unlike *Stdio*, an `sfpopen`-stream can be opened for both read and write.

When a stream is no longer needed, it can be closed with `sfclose()`.

3.3 Reading and Writing Streams

All legal I/O operations can be performed on a stream in arbitrary order. There are five ways to do I/O on streams:

- *Byte or number oriented*: The function `sfgetc(f)` returns the next byte from the stream `f`. If there are no more data, it returns `SF_EOF`. The function `sfungetc(f,c)` can be used to push back a byte into the stream. Unlike *Stdio*, there is no limit to the number of bytes that can be pushed back into a stream. The inverse of `sfgetc()` is `sfputc()` which writes a byte into the stream. Other functions, `sfgetl()`, `sfputl()`, `sfgetu()`, `sfputu()`, `sfgetd()`, and `sfputd()`, are available to read and write integral and floating point values in a portable format. The coding format is portable as long as the bit order within a byte is the same across the relevant machines and if two corresponding types have the same size. The coding for floating point values rely on the system provided functions `frexp()` and `ldexp()`. Most programs manipulate only small values; our coding method takes advantage of this and uses a minimal number of bytes. This saves disk space when storing large amounts of data. The cost of encoding and decoding data is usually well paid for by the time saved in accessing disk data.

- *String-oriented*: The function `sfgetr(f,rsc,string)` reads up to and including the record separator character `rsc`. If `string` is positive, the record separator character is turned into the nul character to make the input into a C string. There is no hard limit on the length of the input string. The library either uses the stream buffer space or allocates space as necessary to store the input. After a call to `sfgetr()`, the function `sfslen()` can be called to get the length of the input.

- *Block-oriented*: The function `sfread(f,buf,n)` tries to read `n` bytes from `f` into the buffer `buf`. It returns the number of bytes actually read. Similarly, `sfwrite(f,buf,n)` writes `n` bytes to the stream `f`. The function `sfmove(fr,fw,n,rsc)` is used to move data from the stream `fr` to the stream `fw`. If `n` is non-negative, it indicates the number of units to move; otherwise all data in the stream is moved. The movement unit is either a byte or a record delineated by the record separator `rsc` when `rsc` is non-negative. The below example skips `10` lines from the standard input, then moves the next `10` lines to the standard output.

  ```
  sfmove(sfstdin,NULL,10,'\n');
  sfmove(sfstdin,sfstdout,10,'\n');
  ```

- *Formatted I/O*: The functions `sfscanf()` and `sfprintf()` provides ways to read and write data that are formatted. These functions are similar to the `fscanf()` and `fprintf()` functions of *Stdio* though we have provided a few extensions so an application can interpret unknown patterns or provide a different way to get or assign formatted arguments. See the appendix for details.

- *Direct stream buffer access*: The fastest method for performing I/O using *Sfio* is via the function `sfreserve(f,n)`. This reserves a data block of size at least equal to the absolute value of `n`. For a read stream, a data block is a segment of data and for a write stream, it is a buffer suitable for writing. If `n` is non-negative, the stream IO location is advanced by `n`. If `n` is negative, `sfreserve()` does not change the current stream location but it locks the stream so that further accesses to the stream will be prevented. Both unlocking and advancing stream IO position is done via a subsequent `sfread()` or `sfwrite()` call. Below is an example of using `sfreserve()` to get access to a write buffer, put some data into it, then advance the write pointer past the data.

  ```
  if((buf = sfreserve(f,-n)) != (char*)0)
  {    n = process(buf,n);
       sfwrite(f,buf,n);
  ```

```
    }
```

3.4 Stream Synchonization

When a stream corresponds to a file, *Sfio* either buffers data or use memory mapping to reduce the number of `read()` and `write()` system calls. This means that the logical seek location of a stream may not correspond to the seek location of the underlying file. The `sfsync(f)` function can be used to cause the two seek locations to synchronize. For a write stream, any buffered data will be output. For a read stream, if the stream is seekable, the file seek location is rolled back as necessary. The call `sfsync(NULL)` synchronizes all streams.

Streams can be grouped together into auto-synchronizable pools using the `sfpool(f,poolf,type)` function. If `poolf` is `NULL`, `f` is taken out of its current pool, if any. Otherwise, `f` is pooled with `poolf`. The `type` argument of `sfpool()` can be any non-empty combination of `SF_SHARE`, `SF_READ` and `SF_WRITE`. In each pool, the most recently accessed stream is at the head of the pool. A `SF_SHARE` pool consists only of write streams. It is useful to serialize buffered data for all streams in a pool as buffered data will be move from stream to stream as streams are switched. If the pool is not `SF_SHARE` and if the type of the current stream matches `type`, it will be synchronized before the new stream becomes current. Below is an example of pooling the standard input and output streams so that the standard output is always flushed before reading from the standard input:

```
sfpool(sfstdin,sfstdout,SF_WRITE);
```

A typical use of `sfpool()` is to preserve the order of I/O operations on streams sharing the same file. For example, the following example guarantees that the output to the standard output and standard error streams which typically go to the terminal will be in the right order. Further, if both streams are buffered, the number of `write()` calls will be minimized as if data is output to a single stream.

```
sfpool(sfstderr,sfstdout,SF_SHARE);
sfprintf(sfstdout,"This is a");
sfprintf(sfstderr," sentence.\n");
```

3.5 Stream Stacking

There are many applications in which it is desirable to insert a stream of data at some point of input. For example, the C preprocessor routinely needs to insert a new macro definition or an include file. The `sfstack(base,f)` function inserts a new stream `f` on top of the stack referred to by the `base` stream. All I/O operations are requested via the `base` stream but actually performed on the top stream. A top stream will be popped and closed on end-of-file. The stack can also be popped by specifying `NULL` for the second argument of `sfstack()`. Below is an example of using stream stacking. The function `process()` checks to see if a line of text defines an include file and returns its name.

```
while((s = sfgetr(sfstdin,’0,1)) != NULL)
{   char *include = process(s);
    if(include)
    {   Sfile_t *f = sfopen(NULL,include,"r");
        if(f)
            sfstack(sfstdin,f);
    }
}
```

In the next section, we give an example showing how a discipline can be used in conjunction with stacking to catch the end-of-file event of the inserted file and reset certain state information such as the current file name and line number.

3.6 Stream Discipline

The function `sfdisc(f,disc)` pushes a new I/O discipline onto the stream `f`. A discipline specifies functions to replace the system calls `read()`, `write()`, and `lseek()`, and to handle exceptions. The argument `disc` is either `NULL` to pop the discipline stack or a pointer to a `Sfdisc_t` structure. This structure contains four publically visible fields: `(*readf)()`, `(*writef)()`, `(*seekf)()`, and `(*exceptf)()`. The first three fields specify alternative I/O functions. If one of these functions is `NULL`, it is inherited from an earlier discipline on the discipline stack. For completeness, the bottom of the stack of a file stream always contains the discipline consisting of system calls. The fourth field, `(*exceptf)()`, defines an exception handler. The stream `f` is always synchronized before its discipline stack is manipulated. After successful manipulations, the seek location and the stream extent may change to reflect that as defined by the current discipline.

A discipline I/O function, `(*readf)()`, `(*writef)()`, or `(*seekf)()`, is called with 4 arguments. The first argument is the stream pointer. The second and third arguments correspond to the second and third arguments of the respected system call. The fourth argument is the discipline structure, `disc`, itself. Note that since a discipline function is invoked during a stream I/O operation, the stream remains locked during its execution. A discipline function should avoid using I/O system calls directly and use `sfrd()`, `sfwr()` and `sfsk()` instead. The latter functions invoke lower level discipline functions and ensure proper exception handling.

The exception function, `(*exceptf)()` is called when a read or write exception happens, when a stream is being closed, or when the discipline is being reset. A read or write exception occurs when the discipline I/O function returns a zero or negative value. During a call to `(*exceptf)()`, if it is at the top of the discipline stack, the stream will be opened for general operations. `(*exceptf)()` is called as `(*exceptf)(f,type,disc)`. Here, `type` is: `SF_DPOP` when the discipline is about to be popped of the discipline stack, `SF_DPUSH` when the discipline is about to be pushed down in the stack, `SF_NEW` when the stream is being renewed, `SF_CLOSE` when the stream is being closed, `SF_READ` when an exception happens during a read operation, and `SF_WRITE` when an exception happens during a write operation. The current *Sfio* function will examine the return value of `(*exceptf)()` for further actions: negative for immediate return, zero for executing default actions associated with the exception, and positive for resuming execution as if no exception happened.

Here is an example of using a discipline to copy data from the standard input to the standard output where all upper case characters are translated to lower case.

```
lower(Sfile_t* f, char* buf, int n, Sfdisc_t* disc)
{   int c;
    n = sfrd(f,buf,n,disc);
    for(c = 0; c < n; ++c)
            buf[c] = tolower(buf[c]);
    return n;
}
...
Sfdisc_t Disc = { lower, NULL, NULL, NULL, NULL };
sfdisc(sfstdin,&Disc);
sfmove(sfstdin,sfstdout,SF_UNBOUND,–1);
```

As usual, there are different solutions with trade-offs in ease of coding, ease of understanding, reusability, and efficiency. The above solution is attractive because it isolates the translation act into a single, easily recognizable and reusable function. The cost of an extra function call per buffer filling is minimal since it typically occurs only once per 8Kb of data.

In practice, most stream disciplines require state information. This can be done by defining application-specific discipline structures. For example, stacked streams frequently require the save and restore of line numbers and file names. A discipline structure for a stacked stream can be defined as:

```
typedef struct _stk_disc
{   Sfdisc_t   disc;
    int        line;
    char       *file;
} Stk_disc_t;
```

The discipline exception function will restore these values when the stream is about to be closed:

```
stk_except(Sfile_t* f, int type, Sfdisc_t* disc)
{   if(type == SF_CLOSE)
    {   Line = ((Stk_disc_t*)disc)->line;
        File = ((Stk_disc_t*)disc)->file;
    }
    return 0;
}
```

The application code needs only set this discipline before stacking the new stream:

```
Stk_disc_t *stk_disc = (Stk_disc_t*)calloc(1,sizeof(Stk_disc_t));
stk_disc->disc.exceptf = stk_except;
stk_disc->line = Line;
stk_disc->file = File;
sfdisc(f,(Sfdisc_t*)stk_disc);
sfstack(sfstdin,f);
```

3.7 Extensions to sfprintf() and sfscanf()

The formatted I/O routines `sfprintf()` and `sfscanf()` have a few extensions for added security and to allow an application to tailor their processing. The string scanning patterns `%C`, `%lc`, `%S`, `%ls`, `%{}` and and `%l[]` are analogues of `%c`, `%s`, and `%[]` but they require a buffer size.

The formats `i`, `d`, and `u` are extended to print integers in non-standard bases. Here is the general form for the `d` format: `%[width].[precision].[base]d`. The forms for `i` and `u` are defined similarly. For example, `%..36d` prints a signed integer in base `36` and `%..64u` prints an unsigned integer in base `64`. Currently, `base` can range from `2` to `64`. If `base` is not defined or if it is not within the defined range, it is taken to be `10`. The modifier # will output the number in the form `base#number`. The digits to represent numbers are:

`01234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ@_`

Three additional customizing patterns are supported: `%&`, `%@` and `%:`. The `%&` pattern indicates that the next argument in the variable argument list is the address of a function that is used to interpret patterns not already defined by the respective utility. The `%@` pattern indicates that the next argument is the address of a function to get arguments in the case of `sfprintf()` and to assign values in the case of `sfscanf()`. This is useful, for example, to write an interpreter version of `printf()` where the arguments are unknown at compile time. Note that both extension functions are local to an invocation of `sfprintf()` or `sfscanf()`. This allows recursive calls of these functions (via the extension functions) to define different types of processing. Finally, the pattern `%:` indicates that the next two arguments are a pattern string and a corresponding variable argument list to be inserted. The processing continues with the new pattern string until it is exhausted, then resumes with the earlier string.

4. Stdio Compatibility

To ease the transition for programs currently based on *Stdio*, we wrote both source and binary compatibility packages. At the source level, the `FILE` type is simply redefined to be `Sfile_t`. Most *Stdio* functions and constants can be emulated by macro definitions to their *Sfio* counterparts. This is done in an emulation `stdio.h` header file. A few functions such as `printf()` and `scanf()` that accepts variable argument lists are reimplemented based on the *Sfio* versions.

Binary compatibility means that the library must work with code already compiled with the standard *Stdio* header file. In general, this is unsolvable because the the `FILE` structure may be defined in some non-standard way which would affect macro functions such as `getc()` or `putc()` that directly accesses certain internal fields. A working solution for most *Stdio* implementations is to write a program to generate from the header file `stdio.h` an emulation structure that contains the fields `_cnt`, `_ptr`, `_flag` and `_file` in the exact locations that they appear in the `FILE` structure. This keeps correct the embedded pointers in macro functions such as `getc()` or `putc()`. Then, functions are written to provide one-to-one mappings between the set of `FILE` emulation structures to the set of `Sfile_t` structures. Each *Stdio* function is reimplemented to map a given `FILE` structure to its

`Sfile_t` counterpart before calling the appropriate *Sfio* function. The reimplemented functions maintain the `_flag` field up to date with respect to I/O exceptions. Finally, by initializing the `_cnt` field to be 0, the macro functions `putc()` and `getc()` are trapped to call `_filbuf` and `_flsbuf`, internal *Stdio* functions to fill and flush buffers. Reimplementing these functions to call their *Sfio* counterparts and to set `_cnt` and `_ptr` to the appropriate values completes the job.

5. Performance

5.1 I/O Benchmark Results

To measure relative performances of *Sfio* and comparable packages, *Stdio* and *Fio*, we wrote a benchmark program that exercises the basic I/O functions. The program is based on the *Stdio* interface so that we can measure the performance of *Stdio*. To measure the performance of *Sfio*, we simply recompiled the program using the source *Stdio* emulation package. Benchmark data were obtained under a variety of environments. To reduce variance, each set of data was an average of five different runs at night time. In the data tables, the first column shows the functions being tested. The second column shows the amount of data processed in units of lines or kilobytes. or number of seeks in the case of *seek+rw*. The remaining columns partitions into set of three each. In each set of columns, the first shows the amount of *cpu* time, the second *system* time, and the third throughput rate which is computed by dividing the total amount of data in K-bytes by the total cpu and system time. The first three tests show results of block I/O where each block is of size 8Kb. The *revrd* test reads blocks in reverse order, i.e., starting from the bottom of the file. The next three tests show results of block I/O with blocks of size 757, a number that does not divide 8K. The tests *copy&rw* and *sfmove* copy a large file by either `fread()` and `fwrite()` or the `sfmove()` primitive of *Sfio*. The test *seek+rw* performs a sequence of seeking to a random location, reading a block of size 8K, then copying that block to location 0. The tests *fputc* and *fgetc* perform byte I/O. The *fputs*, *fgets* and `revgets` tests perform line I/O. *revgets* reads lines in reverse order. Both the `printf()` and `scanf()` tests exercise all common conversion modes: `%c`, `%d`, `%o`, `%x`, `%f`, `%e` and `%s`.

The data will be presented in four tables. Table 1 shows benchmark results on a two-processor Solbourne running SUN OS4.0, comparing *Sfio*, *binary* (the *Stdio* binary emulation package), and *Stdio*. In this version, *Sfio* uses memory mapping for all read streams. This is why `sfmove()` is twice as fast as copying data by block reads and block writes. The functions `sfprintf()` and `sfscanf()` are much faster than their counterparts in *Stdio* due to new data conversion algorithms which, among other improvements, avoid division and multiplication which are expensive on a SPARC machine.

Table 2 compares *Sfio* and *Stdio* on a VAX 8650 running 4.3BSD. Many parts of the native *Stdio* package are implemented in assembly code, especially `fputs()`, `fgets()` and `_doprnt()`, the guts of `fprintf()`. In the VAX version of *Sfio*, `sfprintf()` is written entirely in C but `sfgetr()` and `sfputr()` contain a few `asm()`s to access hardware instructions for block search and block copy.

Table 3 compares *Sfio* and *Stdio* on an Intel 386 machine running System V Release 3.2. Since this machine has limited disk space, we usesd smaller data sets.

Table 4 compares *Sfio*, *Stdio* and Hume's *Fio*. This is done on a MIPS machine running UMIPS. Again because of limited disk space, we used smaller data sets. As *Fio* does not have the same interface as *Stdio*, a few emulation macros were written to map the needed functions. The poor performance of *Fio* analogues of `fputc()` and `fgetc()` is due to their implementation as subroutines.

5.2 Reimplemented System Utilities

A goal of *Sfio* is to be sufficiently efficient so that applications can be based on it without fear of performance loss. To test this, we rewrote several standard UNIX utilities including *ksh*, *cat*, *wc*, *cut*, *pack*, and *unpack*. *Ksh* is a complex program that exercises virtually all aspects of *Sfio*. *Cat* is an I/O bound program and, to a lesser extent, so is *wc*. *Cut* is a non-trivial example where I/O is still likely to matter. *Pack* and *unpack* show how the Huffman coding can be implemented as a discipline. For example, *unpack* entails opening a packed file, setting the unpacking discipline, then transferring data as if it is in plain text.

		Sfio			*binary*			*Stdio*		
test	size	user	sys	Kb/s	user	sys	Kb/s	user	sys	Kb/s
fwrite	10000K	0.03	2.04	4813	0.09	2.13	4504	0.05	2.03	4796
fread	10000K	1.00	0.62	6163	1.07	0.66	5755	0.74	1.43	4608
revrd	10000K	0.31	1.62	5188	0.35	1.54	5298	0.79	1.39	4597
fw757	10000K	1.12	1.97	3236	1.25	2.06	3016	0.96	1.91	3481
fr757	10000K	1.20	0.58	5610	1.36	0.66	4944	0.96	1.38	4278
rev757	10000K	1.25	2.66	2560	1.71	2.58	2325	1.21	15.28	606
copy&rw	10000K	1.13	2.88	2496	1.25	2.94	2389	0.88	3.87	2105
sfmove	10000K	0.01	1.90	5228						
seek+rw	2000S	0.83	6.35	2268	0.90	6.07	2315	1.52	5.87	2185
fputc	5000K	3.92	1.14	987	4.29	1.08	929	4.18	1.14	940
fgetc	5000K	3.78	0.44	1184	4.02	0.23	1175	3.93	0.79	1059
fputs	50000L	2.15	1.03	1538	2.63	1.18	1285	2.07	1.08	1549
fgets	50000L	2.21	0.25	1981	2.58	0.33	1681	2.05	0.70	1783
revgets	50000L	2.56	1.25	1309	3.49	1.28	1028	4.15	36.59	122
fprintf	50000L	6.06	1.00	506	6.81	0.97	458	16.71	1.31	196
fscanf	50000L	6.56	0.29	518	7.47	0.41	454	17.90	1.00	187

TABLE 1. Solbourne and SUN OS4.0

		Sfio			*Stdio*		
test	size	user	sys	Kb/s	user	sys	Kb/s
fwrite	10000K	0.11	3.90	2496	1.11	4.09	1924
fread	10000K	0.11	1.78	5284	0.79	1.71	3996
revrd	10000K	0.18	1.88	4854	1.00	2.00	3336
fw757	10000K	0.97	4.32	1893	0.83	4.42	1904
fr757	10000K	1.07	1.78	3514	0.80	1.72	3976
rev757	10000K	1.47	4.98	1550	2.15	19.02	472
copy&rw	10000K	0.26	4.97	1911	1.87	5.19	1416
sfmove	10000K	0.04	4.87	2035			
seek+rw	2000S	1.23	7.41	1851	3.02	7.87	1469
fputc	5000K	8.84	1.93	464	10.18	1.92	413
fgetc	5000K	6.74	1.07	640	8.00	1.04	553
fputs	50000L	1.33	2.15	1405	1.15	2.21	1456
fgets	50000L	1.21	0.93	2290	0.93	0.90	2659
revgets	50000L	2.29	2.30	1063	5.12	68.17	66
fprintf	50000L	18.86	1.49	174	22.28	1.67	147
fscanf	50000L	25.34	0.93	134	42.11	1.18	81

TABLE 2. VAX 8650 and 4.3BSD

Table 5 compares different versions of *cat*, *wc*, *cut*, *pack* and *unpack*. The data were obtained on a two-processor Solbourne running SUN OS4.0. Each program is run with two different datasets, one of size 50000 bytes and the other 5 million bytes. The data for each utility will be presented in two successive rows, the smaller dataset first, then the larger one. Where it is applicable, we show data of three different versions of the tool, SUN OS, System V, and our implementation. Note that *cut*, *pack* and *unpack* are the same on SUN OS and System V.

The data shows that our versions of the standard utilities perform at the same level or better than the other versions. The *cat* test shows that, even with the direct use of `read()` and `write()` for data transfer, the System V *cat* still lost in system time to its SUN-OS and *Sfio* counterparts because the latter use memory mapping. The data for *wc* shows another aspect of efficiency at the library level. In

		Sfio			*Stdio*		
test	size	user	sys	Kb/s	user	sys	Kb/s
fwrite	1000K	0.02	2.91	340	0.02	2.87	346
fread	1000K	0.02	3.26	305	0.24	3.87	242
revrd	1000K	0.03	2.91	340	0.24	3.26	285
fw757	1000K	0.31	2.84	317	0.41	3.16	279
fr757	1000K	0.39	3.40	263	0.43	3.67	243
rev757	1000K	0.47	5.58	165	0.42	5.44	170
copy&rw	1000K	0.03	6.37	156	0.25	6.65	144
sfmove	1000K	0.01	6.60	151			
seek+rw	500S	0.46	7.66	491	1.32	11.58	309
fputc	500K	3.81	1.50	94	3.73	1.75	91
fgetc	500K	3.44	1.70	97	3.16	1.86	99
fputs	10000L	2.72	2.60	185	2.66	3.09	169
fgets	10000L	2.58	3.28	167	2.58	3.83	152
revgets	10000L	2.32	4.18	149	4.20	32.74	26
fprintf	10000L	32.96	2.01	19	32.66	2.38	19
fscanf	10000L	30.60	2.56	20	38.60	2.63	16

TABLE 3. Intel 386 and System V Release 3.2

		Sfio			*Stdio*			*Fio*		
test	size	user	sys	Kb/s	user	sys	Kb/s	user	sys	Kb/s
fwrite	1000K	0.01	1.91	521	14.26	1.75	62	0.19	2.01	456
fread	1000K	0.01	0.91	1086	14.12	0.91	66	0.01	1.01	978
revrd	1000K	0.02	0.84	1152	13.84	0.92	67	0.04	0.79	1187
fw757	1000K	0.45	1.75	453	14.26	1.70	62	0.45	1.83	437
fr757	1000K	0.43	0.84	787	14.05	1.18	65	0.19	2.54	365
rev757	1000K	0.57	2.17	364	14.04	2.39	60	0.28	2.84	320
copy&rw	1000K	0.05	2.62	374	28.02	2.61	32	0.18	2.71	346
sfmove	1000K	0.01	2.65	376						
seek+rw	500S	0.74	5.79	612	111.92	6.53	33	0.92	6.23	559
fputc	500K	5.07	1.02	82	4.88	0.93	86	20.18	0.98	23
fgetc	500K	4.30	0.48	104	4.44	0.46	101	18.52	0.45	26
fputs	10000L	5.15	1.74	141	9.74	1.79	84	5.49	1.82	133
fgets	10000L	5.04	0.95	162	11.17	0.94	80	4.96	1.11	160
revgets	10000L	6.16	2.06	118	12.09	5.26	55	6.69	38.17	21
fprintf	10000L	20.74	1.18	31	60.08	1.40	11	52.33	1.29	13
fscanf	10000L	27.06	0.75	24	43.85	0.74	15			

TABLE 4. MIPS and UMIPS

this case, the SUN-OS version of *wc* consumes much more cpu time than System V version because it uses `getc()` for input while the System V version uses `fread()`. The *Sfio* version improves on the System V version via the use of `sfreserve()` and a new algorithm for detecting word and line boundaries. The big performance improvement in *unpack* is due in large part to a new unpacking algorithm which makes use of `sfreserve()` and discipline. The benefit of coding *unpack* as a discipline is that the discipline can be reused directly in other contexts.

6. Conclusions

We presented *Sfio*, a new library for stream input/output that can be used as a replacement for *Stdio*, the standard I/O library for C. A number of deficiencies in *Stdio* were described. We discussed how *Sfio*

	SUN-OS		*System V*		*Sfio*	
command	user	sys	user	sys	user	sys
cat	0.01 0.01	0.08 0.09	0.02 0.05	0.09 1.08	0.01 0.03	0.11 0.17
wc	0.10 5.72	0.09 0.75	0.04 2.55	0.08 0.69	0.04 1.57	0.10 0.68
cut	0.11 9.69	0.11 1.21	0.11 9.69	0.11 1.21	0.05 3.79	0.10 1.30
pack	0.11 9.92	0.16 3.76	0.11 9.92	0.16 3.76	0.09 6.87	0.11 1.40
unpack	0.24 19.65	0.13 2.96	0.24 19.65	0.13 2.96	0.10 6.90	0.10 1.59

TABLE 5. Reimplemented System Commands

avoids such deficiencies. Performance results from our own benchmark program and a few reimplemented utilities were shown The data shows that *Sfio* performs as well as or better than *Stdio* on popular hardware platforms running different UNIX versions. This is nice especially since many versions of *Stdio* that we compare *Sfio* against have been hand-tuned in assembly code. Standard utilities gain substantial efficiency when reimplemented based on *Sfio*. Despite many new features, the total size of the library is under 50K on a SPARC1+ workstation.

Beyond a being a better replacement for *Stdio*, *Sfio* introduces a number of new abstractions. Streams have been generalized to represent both files and in-core memory areas. Fast access to internal stream buffers is provided so that applications should never have to revert to bare system calls for efficiency. New methods are provided to handle synchronization of collections of streams, build new virtual streams by stacking and change stream I/O disciplines. Stream disciplines allow applications to tune the handling of exceptions and to change or augment the processing power of the underlying system calls. Many routine data processing tasks can be modeled and implemented by appropriate disciplines resulting in better code organization and code reuse. In our experiences, the combination of performance and new features makes *Sfio* a powerful tool that enhances our ability to write programs.

Acknowledgement

Glenn Fowler helped firm up the design and implementation of *Sfio*. Griff Smith contributed the main idea and code behind the new decimal conversion algorithm for `sfprintf()`. Finally, thanks are due to Andrew Hume and Griff Smith who helped to improve the presentation of the paper.

References

[AN] ANSI x3.159-1989, *American National Standard for Information Systems - Programming Language - C*, Amer. Nat. Sta. Ins., 1990.

[Hu] A. Hume, *A Tale of Two Greps*, Soft. Prac. & Exp., v.18, pp.1063-1072, 1988.

[KR] B. Kernighan & D. Ritchie, *The C Programming Language*, Prentice Hall, 1978.

[St] B. Strousup, *The C++ Programming Language*, Addison-Wesley, 1986.

Appendix: Source *stdio* Emulation Header File

Below is the full text of the source emulation header file for *Stdio*. Note that the macro `_SFSIZEOF()` provides an added security check for a frequent use of `gets()` where the buffer is a declared fixed size array.

```
#ifndef _SFSTDIO_H        /* protect against multiple #includes */
#define _SFSTDIO_H

#include                  <sfio.h>

#define _IOFBF            0
#define _IONBF            1
#define _IOLBF            2
#define L_ctermid         9
#define L_cuserid         9
#define P_tmpdir          "/usr/tmp/"
#define L_tmpnam          (sizeof(P_tmpdir)+15)

extern char               *ctermid(char*);
extern char               *cuserid(char*);
extern char               *tmpnam(char*);
extern char               *tempnam(char*);
extern char               *_stdgets(char*, int n);
extern int                _stdprintf(const char*, ...);
extern int                _stdsprintf(char*, const char*, ...);
extern int                _stdscanf(const char*, ...);
extern int                _stdsetvbuf(Sfile_t*, char*, int, int);

#define stdin             sfstdin
#define stdout            sfstdout
#define stderr            sfstderr
#define FILE              Sfile_t
#define BUFSIZ            SF_BUFSIZE

#define fopen(f,m)        sfopen((Sfile_t*)0,f,m)
#define freopen(f,m,p)    sfopen(p,f,m)
#define popen(cmd,m)      sfpopen((Sfile_t*)0,cmd,m)
#define fdopen            _stdopen
#define tmpfile()         sftmp(SF_BUFSIZE)
#define fclose(f)         sfclose(f)
#define pclose(f)         sfclose(f)

#define fwrite(p,s,n,f)   ((_Sfi = sfwrite(f,p,(s)*(n))) <= 0 ? _Sfi : _Sfi/(s))
#define fputc(c,f)        sfputc(f,c)
#define putc(c,f)         sfputc(f,c)
#define putw(w,f)         (_Sfi = (int)w, sfwrite(f,&_Sfi,sizeof(int)) <= 0 ? 1 : 0)
#define putchar(c)        sfputc(sfstdout,c)
#define fputs(s,f)        sfputr(f,s,-1)
#define puts(s)           sfputr(sfstdout,s,'0)
#define fprintf           sfprintf
#define vfprintf          sfvprintf
#define vprintf(f,a)      sfvprintf(sfstdout,f,a)
#define _doprnt(fm,a,f)   sfvprintf(f,fm,a)
#define vsprintf          _stdvsprintf
#define printf            _stdprintf
```

```
#define sprintf           _stdsprintf

#define fread(p,s,n,f)    ((_Sfi = sfread(f,p,(s)*(n))) <= 0 ? _Sfi : _Sfi/(s))
#define fgetc(f)          sfgetc(f)
#define getc(f)           sfgetc(f)
#define getw(f)           (sfread(f,&_Sfi,sizeof(int)) == sizeof(int) ? _Sfi : –1)
#define getchar()         sfgetc(sfstdin)
#define ungetc(c,f)       sfungetc(f,c)
#define fgets(s,n,f)      _stdgets(f,s,n,0)
#define _SFSIZEOF(s)      (sizeof(s) != sizeof(char*) ? sizeof(s) : BUFSIZ)
#define gets(s)           _stdgets(s,_SFSIZEOF(s),1)
#define fscanf            sfscanf
#define vfscanf           sfvscanf
#define _doscan           sfvscanf
#define sscanf            sfsscanf
#define vscanf(f,a)       sfvscanf(sfstdin,f,a)
#define scanf             _stdscanf
#define vsscanf           _stdvssanf

#define fflush(f)         sfsync(f)
#define fseek(f,o,t)      (sfseek(f,o,t) < 0L ? –1 : 0)
#define rewind(f)         sfseek((f),0L,0)
#define ftell(f)          sftell(f)
#define setbuf(f,b)       sfsetbuf(f,b,(b) ? BUFSIZ : 0)
#define setbuffer(f,b,n)  sfsetbuf(f,b,n)
#define setlinebuf(f)     sfset(f,SF_LINE,1)
#define setvbuf           _stdsetvbuf

#define fileno(f)         sffileno(f)
#define feof(f)           sfeof(f)
#define ferror(f)         sferror(f)
#define clearerr(f)       (sfclrerr(f),sfclrlock(f))

#endif /* _SFSTDIO_H */
```