Extended Data Formatting Using Sfio

Glenn S. Fowler, David G. Korn and Kiem-Phong Vo
AT&T Laboratories – Research
180 Park Avenue, Florham Park, NJ 07932, U.S.A.
gsf,dgk,kpv@research.att.com

Abstract

The ANSI-C Standard defines Stdio as the I/O library for C programs. Despite its ubiquitous use, Stdio has well-documented deficiencies in various areas including data formatting. The Sfio library provides an alternative to Stdio with improved functionality, robustness and performance. In particular, Sfio extends the data formatting functions so that applications can deal with arbitrary scalar objects, avoid unsafe operations and even define their own conversion patterns. This paper discusses these formatting enhancements.

1 Introduction

The Stdio `printf()/scanf()` family of functions [1, 5] are the de facto standard for formatting data in C programs. Many implementations of the C++ I/O operators [9] `>>` and `<<` are also based on the `printf()/scanf()` functions. Despite this popularity, the Stdio formatting functions have a number of shortcomings:

- *Inadequate handling of abstract scalars*: The formatting functions only deal with primitive scalar types. To format an abstractly defined scalar object, it is customary and even necessary to cast it to some presumed larger scalar type. For example, on most platforms, a file offset object declared with the Posix type `off_t` would need to be casted to a `long` for printing. This trick is not portable since, on a modern platform, `off_t` may be defined on top of a newer and larger type such as `long long`.

- *Unsafe data scanning*: String scanning with Stdio always runs the risk of overflowing the buffer because there is no way to tell the scanning function the buffer size. Buffer overflow bugs often corrupt memory leading to disastrous consquences. These bugs are also hard to detect.

- *Inextensible interface*: It is useful to be able to extend the defined set of conversion patterns or even to redefine some of them based on specific needs. For example, if an application defines a type `Coord_t` for spatial coordinates, it would be nice to be able to define a corresponding formatting pattern, say `%C`, to print or scan such a type. This cannot be done in the current formatting framework.

- *Inadequate reuse*: The POSIX Standard [8] defines commands such as **printf** to format data in the same style as the corresponding Stdio functions. Since applications cannot access the format parsing and argument processing code in the formatting functions, each tool must invent its own way to perform these tasks. This unnecessarily duplicates work already done in library functions and does not help to improve interface consistency across tools.

The Sfio library [3, 7] was introduced in 1991 as a better alternative to Stdio. In particular, the Sfio data formatting functions outperformed their Stdio counterparts due to faster integer and floating point value conversion algorithms. Although these early Sfio formatting functions addressed the mentioned portability and robustness issues in Stdio, they were still inflexible so that applications could not adapt them for specific needs.

Starting from the 1997 release of Sfio, we experimented with extending the formatting functions to allow both non-standard patterns and alternative argument processing. The early extensions were useful but we found through experience that the framework was incomplete and cumbersome to use. For example, formatting flags and values such as width and precision were not properly packaged and passed between library and application code when processing non-standard conversion patterns. It was also impossible to redefine existing conversion patterns. Since then, we have redesigned the extensions to enable much more natural cooperation between the formatting functions and applications.

The rest of this paper summarizes the new formatting features and gives examples of how to use them. A performance study comparing Sfio and various Stdio versions on the basic printing and scanning tasks shows that Sfio outperforms Stdio despite the additional features.

2 Extended data formatting

The formatting extensions include portable scalar formatting, safe data scanning, dealing with integers in general bases, and the ability to define new formatting patterns or redefine existing ones. To accomodate the new extensions, the general forms of Sfio printing and scanning patterns are respectively:

```
%[pos$][flag][(tstr)][width[.precis[.base]]]z
%[*][pos$][flag][(tstr)][width[.width.base]]z
```

Arguments such as `pos$`, `width`, etc. are the same as defined in the ANSI-C Standard. The `base` argument is introduced to accomodate generalized scalar and string processing.

The argument `(tstr)` is used to define a string that will be passed to an extension function if one is defined. Section 2.5 discusses how applications can use such data for non-standard conversion patterns and argument processing.

The below subsections discuss the new extensions. We mostly present printing examples, but scanning examples work in a similar way.

2.1 Portable scalar formatting

Certain platforms provide 64-bit integer and floating point values via types such as `long long` and `long double`. These types are handled differently in different Stdio implementations. For example, the Microsoft-C version provides an `I64` flag to specify a 64-bit integer while other Unix platforms use the more general flag `ll` for the same purpose.

Sfio generalizes the `ll` flag to deal with the largest primitive types on a particular platform. In fact, to ensure portability, Sfio provides types such as `Sflong_t`, `Sfulong_t` or `Sfdouble_t` that are always mapped to the largest primitive types available. The following examples show how to use the `ll` flag in printing or scanning objects with large types:

```
Sflong_t   intval;
sfprintf(sfstdout,"%lld", intval);

Sfdouble_t fltval;
sfscanf(sfstdin,"%llf", &fltval);
```

The `ll` flag enables printing of abstract types that may be mapped to different primitive scalar types on different platforms. For example, the familiar ANSI-C `size_t` for memory size and the POSIX `off_t` for file offset are often mapped to `unsigned int` and `long` respectively. But `off_t` may also be mapped to the type `long long` on platforms that support very large files. To print a value defined by an abstract scalar type, one should cast it to the largest corresponding scalar type and use the `ll` flag with an appropriate conversion pattern. For example, an `off_t` value should be printed by casting to `Sflong_t` and using the pattern `%lld` as follows:

```
off_t   offset;
sfprintf(sfstdout,"%lld", (Sflong_t)offset);
```

Unfortunately, the above trick does not work with scanning since the scanned value must be stored in a location with a specific type. Printing performance is also suboptimal if arithmetic operations on such large types are more expensive than that on normal types. Various proposals are being debated by the C9X Standard Committee [4] to solve this problem. For Sfio, since we already needed to provide the Microsoft-C flag `I64` for portability, we simply took the opportunity and generalized this flag to allow specification of objects with arbitrary sizes. The below examples show how this works:

```
sfprintf(sfstdin,"%I4d",intval);
sfprintf(sfstdin,"%I*d",sizeof(intval),intval);
sfscanf(sfstdin,"%I*f",sizeof(fltval),&fltval);
sfprintf(sfstdout,"%I64d",big_long);
```

The first line indicates that the integer value `intval` is an object whose size is 4 bytes, i.e., a 32-bit integer. The second line is more general and supplies the size of `intval` via '`*`'. This will work with integers of any size. The third line is similar to the second line but for scanning a floating point value. The fourth line shows that, for compatibility with Microsoft-C, the value 64 can be used to identify a 64-bit integer.

The above use of 64 to indicate bit size instead of byte size is potentially ambiguous. However, it will be a long time before we need to worry about machines with 64-byte words. In the mean time, it solves a practical problem.

2.2 Safe data scanning

The string scanning patterns `%s`, `%c` and `%[]` are often unsafe to use due to buffer overflow problems. The aforementioned `I` flag can be used to define buffer sizes. Specifying a buffer size does not limit the amount of scanned data. Rather, scanned data exceeding the buffer limit are discarded. Below are two scanning examples where the second one is slightly more general than the first:

```
char buf[10];
sfscanf(sfstdin,"%I10s",buf);
sfscanf(sfstdin,"%I*s",sizeof(buf),buf);
```

In both cases, at most `9` bytes will be copied into the buffer. Further input data will be scanned but discarded. Sfio reserves one byte from the buffer for the final null character.

2.3 Integers in general bases

The patterns `%i`, `%u` and `%d` can format in bases from `2` to `64`. The syntax `[width[.precision[.base]]]` is used so that a base is defined if and only if exactly two dots have appeared. If a base is not validly defined, base `10` is used. Below are the 64 digits used to represent numbers:

```
0123456789
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ @_
```

Pattern	Value	Result
`%..2d`	123	1111011
`%#..2d`	123	2#1111011
`%#..16d`	12345	16#3039
`%#..34d`	-12345	-34#an3
`%#..63d`	123456789	63#7QKgA

Table 1: Integer values in general bases

Table 1 shows examples of printing numbers in general bases. The flag `#` outputs a number in the form `base#representation` where `base` is decimal and `representation` is in the digits for that base.

2.4 Character and string arrays

In addition to handling characters and strings, the string patterns `%c` and `%s` can also print null-terminated arrays of characters or null-terminated arrays of strings. To format an array, a separator must be supplied based on the syntax `[width[.precision[.separator]]]`. That is, a separator is defined if and only if exactly two dots have appeared. When the separator is given in the format string, it must be a non-alphanumeric character and appear immediately before the conversion pattern. Each formatted character or string always obeys the layout rules defined by `width` and `precision`. Below are three example formatting calls and results:

```
sfprintf(sfstdout,"%..:c", "abc");
    a:b:c

sfprintf(sfstdout,"|%6..*s|", '|', words);
    |  trez|  tres| three|

sfprintf(sfstdout,"%..s", words);
    treztresthree
```

The second and third examples assume that the null-terminated array `words` contain three words: `trez`, `tres` and `three`. In the second example, the field width for each word is 6 and the '*' means to get the separator from the argument list. In the third example, the separator is not defined so the strings are simply output one after another.

2.5 Extended format processing

Applications can both define new conversion patterns and redefine existing ones. In addition, it is also possible to use call-back functions to get the objects to be formatted instead of getting them from the function argument list. This ability is important for implementing certain Posix commands such as **printf** that mimics the Stdio function with the same name but whose arguments are given as strings on the command line.

In the below, we first describe the mechanisms to extend formatting, then give examples of how they may be used. Readers not yet acquainted with these extensions may prefer to reverse the order by reading the examples first before learning the details of the mechanisms.

2.5.1 Formatting environments and stacks

A typical formatting call has as input arguments a formatting string with conversion patterns and a corresponding argument list of the objects to be formatted based on the specified patterns. Such a formatting string and argument list is called a *formatting pair*.

To extend pattern processing, we define *formatting environments* in which both formatting pairs and associated call-back functions can be given. We further allow these multiple formatting environments to be stacked on top of one another on an *environment stack* for recursive pattern processing.

```
Sffmtext_f   extf;
Sffmtevent_f eventf;

char*        form;
va_list      args;

int          fmt;
ssize_t      size;
int          flags;
int          width;
int          precis;
int          base;

char*        t_str;
int          n_str;
```

Figure 1: The extended formatting environment

A formatting environment is of the type `Sffmt_t` with elements as shown in Figure 1:

- The first four members of an `Sffmt_t` object should be set by the application before passing to the formatting function. The event-handling function `eventf`, if not `NULL`, is called to process events such as popping the stack. `form` and `args` define a new formatting pair if `form` is not `NULL`. The extension function `extf`, if not `NULL`, is called to process conversion patterns.

- The next six members of `Sffmt_t` are used by the formatting function and `extf` to exchange data about the pattern being processed. For example, on the call to `extf`, the formatting function sets `fmt` to the pattern being processed. On return, `extf` may reset that field to redirect further processing.

- The last two members of `Sffmt_t` are used by a formatting function to pass to `extf` the `(tstr)` string that Sfio allows in specifying a conversion pattern. Section 2.5.5 shall discuss a use of such strings to unify formatting at the command level.

Each formatting call maintains a separate formatting stack whose bottom is a virtual formatting environment that consists only of the original formatting pair. A new conversion pattern `%!` is used to either push a new formatting environment onto the formatting stack or change the extension functions of the top environment. This works as follows:

- When the pattern `%!` is encountered during processing of a format string, the formatting function obtains the corresponding `Sffmt_t` object. Then, if the `form` field of this object is not `NULL`, the new environment is pushed onto the stack and processing will start with the new formatting pair and extension functions. On the other hand, if `form` is `NULL`, only the extension functions of the current top environment are changed to the new ones and processing continues with the current formatting pair.

- The stack top is popped whenever its format string is completely processed or if a call to an extension function returns a negative value. When this happens, the current `eventf` function will be called to allow the application to perform any finalization actions (e.g., freeing the formatting environment object).

- To process a conversion pattern, the formatting function first fills the relevant `Sffmt_t` fields with data such as the current states of the format string and the argument list, the formatting pattern, object size, flags, width, precision, etc. Then, it makes the call `(*extf)(f,v,fe)`. Here `f` is the stream, `v` is a pointer to an object suitable for storing a scalar or pointer value, and `fe` is the given `Sffmt_t` object.

- The return value of `extf` is handled as follows:

  - A negative value pops the stack. Processing will continue with the newly revealed top environment if there is one. If there is no more environment, the formatting function will return.

  - A positive value means that `extf` has finished formatting this pattern and also indicates the amount of stream data that `extf` reads or writes. The calling formatting function will record this amount, then continue processing of the format string.

  - A zero value indicates that the formatting function should take over processing this pattern. The extension function may redirect processing by modifying the `Sffmt_t` object to change the formatting pattern and other associated formatting attributes. In fact, if the original pattern was not one already defined by Sfio, `extf` should reset the field `fmt` to one already defined. Otherwise, this conversion pattern will be ignored.

```
 1. timeprint(Sfio_t* f,Void_t* v,Sffmt_t* env)
 2. {    if(env->fmt == 't')
 3.      {    time_t t = va_arg(env->args,time_t);
 4.           *((char**)v) = ctime(&t);
 5.           env->size = -1;
 6.           env->fmt = 's';
 7.           env->flags |= SFFMT_VALUE;
 8.      }
 9.      return 0;
10. }

11. error(char* form, ...)
12. {    Sffmt_t    fmt;
13.      va_list    args;
14.      static int count;

15.      va_start(args,form);
16.      fmt.form = form;
17.      va_copy(env.args,args);
18.      fmt.extf = timeprint;
19.      fmt.eventf = (Sffmtevent_f)0;

20.      sfprintf(sfstderr,"Error #%d, %!.\n",
                  ++count, &fmt);

21.      va_end(args);
22. }

23. error("%t:\n\tTrying to allocate %d bytes",
          time(0), 1024);

    Error #1, Tue Dec 1 00:39:46 EST 1999:
          Trying to allocate 1024 bytes.
```

Figure 2: An error processing function

2.5.2 Defining a new pattern

Figure 2 shows how to implement a function `error()` that prints all normal conversion patterns and also supports a new pattern `%t` to convert a clock value to a date string. This example also shows how the formatting stack is used.

Lines `1-10` define an extension function `timeprint()` to interpret the new conversion pattern if it is specified. Other patterns are simply deferred to the calling formatting function.

Lines `3-4` obtain the time value and convert it to a date string. The use of `time_t` to get a value off of an argument list is possible here because `timeprint()` is an application routine. Both `time_t` and `ctime()` are defined in the ANSI-C Standard.

Lines `5-7` reset the formatting pattern `env->fmt` to 's' and `env->size` to `-1` and also add the bit flag `SFFMT_VALUE` to `env->flags`. These actions tell `sfprintf()` that `timeprint()` is returning a null-terminated string to be printed. Although not necessary in this example, the extension function should always make sure that associated formatting attributes such as width, precision and base are reset properly along with resetting a conversion pattern.

Lines `11-22` define a function `error()` to print error messages with embedded conversion patterns including `%t`.

Lines `15-19` construct a formatting environment `fmt` from the function arguments and the extension function `timeprint()`.

Line 17 shows the use of the macro function `va_copy` to copy argument lists. This macro function is provided by Sfio for portability.

Line `20` calls `sfprintf()` to do the actual formatting. This call first outputs an error count. Then when it encounters the pattern `%!`, it stacks `fmt` to start processing the arguments of `error()`. When that is finished, the stack is popped and processing returns to the original formatting string to output the final period.

Line `23` gives an example of how `error()` may be called to print an error message concerning an allocation error. The `%t` pattern is treated by `timeprint()` in the described manner. However, `timeprint()` simply returns `0` for the `%d` pattern so that `sfprintf()` will continue with normal processing. An example output is shown after the `error()` call.

2.5.3 Redefining a pattern

Figure 3 shows an example that redefines the system-defined pattern `%c` and also defines a new pattern `%C` to print a pair of real numbers in two different ways, as a complex number or as a two-dimensional coordinate. The former is presented as a pair of numbers in parentheses while the latter is presented in angle brackets.

Lines `1-4` define the object type `Obj_t`, a `struct` with two floating point value members.

Lines `5-17` defines the extension function `objprint()` to print an `Obj_t` object based on the specified formatting patterns. The `default` clause of the `switch` statement shows that `objprint()` returns `0` on all conversion patterns other than `%c` and `%C`. This means that `sfprintf()` will continue processing them normally.

Lines `8-13` show how recursive calls to `sfprintf()` are used to process the patterns `%c` and `%C`. In each case, data is output directly to the stream. The output amount is returned to indicate to the the original `sfprintf()` call that the pattern has been completely processed and also so that `sfprintf()` can correctly update its output count.

```
 1. typedef struct obj_s
 2. { double   x;
 3.   double   y;
 4. } Obj_t;

 5. objprint(Sfio_t* f,Void_t* v,Sffmt_t* env)
 6. { Obj_t* o;
 7.   switch(env->fmt)
 8.   { case 'c': /* print a complex number */
 9.      o = va_arg(env->args,Obj_t*);
10.      return sfprintf(f,"(%g,%g)",o->x,o->y);
11.     case 'C': /* print a coordinate pair */
12.      o = va_arg(env->args,Obj_t*);
13.      return sfprintf(f,"<%g,%g>",o->x,o->y);
14.     default :
15.      return 0;
16.   }
17. }

18. Sffmt_t fmt;
19. fmt.form = (char*)0;
20. fmt.extf = objprint;

21. Obj_t obj = {1.11, 2.22};

22. sfprintf(sfstdout,"%!%c\n",&fmt,&obj);
    (1.11,2.22)

23. sfprintf(sfstdout,"%!%C\n",&fmt,&obj);
    <1.11,2.22>
```

Figure 3: Printing user-defined data

Lines 18-20 construct a formatting environment. The field `fmt.form` is set to `NULL` so that only the extension function of the current top environment would be changed to `objprint()`.

Lines 21-23 initialize an object `obj` with the shown values, then print it both as a complex number and as a two-dimensional coordinate. The resulting outputs are shown along with the respective calls.

2.5.4 Application-defined arguments

Figure 4 shows how to extend `sfprintf()` so that the values to be formatted can be obtained either from the argument list or via a call-back function that gets them from the process environment.

Lines 1-18 define the function `envprint()` to process environment variables. The special processing is done only when an environment variable name is given via the use of the `(tstr)` syntax.

Lines 4-5 construct the name of the environment variable. This explicit construction is necessary because the `(tstr)` string `env->t_str` is not necessarily null-terminated.

```
 1. envprint(Sfio_t* f,Void_t* arg,Sffmt_t* env)
 2. {  char name[1024], *v;

 3.    if(env->n_str > 0)
 4.    {  memcpy(name,env->t_str,env->n_str);
 5.       name[env->n_str] = 0;
 6.       if((v = getenv(name)) && *v)
 7.       {  *((char**)arg) = v;
 8.          env->size = -1;
 9.          env->fmt = 's';
10.       }
11.       else
12.       {  *((char*)arg) = '?';
13.          env->fmt = 'c';
14.       }
15.       env->flags |= SFFMT_VALUE;
16.    }

17.    return 0;
18. }

19. Sffmt_t ft;
20. ft.extf = envprint;
21. ft.form = (char*)0;

22. sfprintf(sfstdout,"%!%s=%(*)d\n",
              &ft, "LINES", "LINES");
    LINES=24

23. sfprintf(sfstdout,"%!%s=%(*)s\n",
              &ft, "SHELL", "SHELL");
    SHELL=/bin/ksh

24. sfprintf(sfstdout,"%!%s=%(*)s\n",
              &ft, "UNKNOWN", "UNKNOWN");
    UNKNOWN=?
```

Figure 4: Application-defined arguments

Lines 6-15 attempts to obtain the value of the specified environment variable. If this value exists, it is returned in the given argument `arg`. The conversion pattern is changed to '`s`' since this is a string. If the value does not exist, the character '`?`' is returned and the conversion pattern is accordingly changed to '`c`'. In either case, the flag `SFFMT_VALUE` is set to indicate that further processing of the returned value is needed by the orginal `sfprintf()` call.

Lines 19-21 set up a new formatting environment. Since the field `form` is set to `NULL`, only the extension function of the current top formatting environment on the formatting stack will be changed.

Lines 22-24 give examples of printing the names and values of three environment variables: `LINES`, `SHELL` and `UNKNOWN`. In each case, the conversion pattern `%!` is used to change the extension function to `envprint()`. After that, processing continues with the current formatting string and argument list. This would cause the name of the variable and the character '`=`' to be output. Then, the `*` directive in the "`(tstr)`" construct obtains the second instance of the variable name from the argument list to pass on to `envprint()`. In turn, the `envprint()` call computes and returns the value of the specified environment variable in the manner described above.

2.5.5 Command-level formatting

Commands like **ls**, **ps** and **find** can produce data in tabular formats. Classic implementations provide a variable format controlled by option flags, each flag enabling another column in the formatted output. These commands have been independently extended by various groups to allow `printf`-style specifications, but because of the earlier lack of a programmable `printf` interface, such extensions are often incompatible.

The Sfio "`(tstr)`" construct allows a common syntax for extending formatting at command level. For example, our **ls** command provides a **-f** *format* option that accepts format parameters of the form:

%[-+][*width*[.*precis*[.*base*]]](*id*[:*subformat*])*char*

Here, *id* is `path` or any member of the `<sys/stat.h> stat` structure (with the leading `st_` omitted.) If *char* is **s** then the string representation of the item is formatted; otherwise, the integer form is formatted. Consider the below example option:

```
-f '%(mode)s %(mtime:time=%H:%M:%S)s %(path)s'
```

This would print:

- The file mode in the style of **ls -l**,
- The file modify time using the `strftime`(3) format `%H:%M:%S` (hours, minutes, seconds), and
- The file path name.

Within the **ls** implementation, such an option is simply passed to the formatting function `sfprintf()` after a formatting environment has been set up with an appropriate extension function that knows how to interpret the mentioned `(tstr)` strings. Then, `sfprintf()` parses the format string and calls the extension function for actual formatting.

Figure 5 shows parts of an extension function `lsprint()` to interpret the above example `(tstr)` strings for printing the path name and modification time of a file. Although this code is not the same as

```
 1. typedef struct _lsfmt_s
 2. {   Sffmt_t      fmt;
 3.     struct stat* sb;
 4.     char*        path;
 5. } Lsfmt_t;

 6. lsprint(Sfio_t* f,Void_t* arg,Sffmt_t* env)
 7. {
 8.     Lsfmt_t* ls = (Lsfmt_t*)env;

 9.     if(...path name...)
10.     {   *((char**)arg) = ls->path;
11.         env->size = -1;
12.         env->flags |= SFFMT_VALUE;
13.         return 0;
14.     }
15.     else if(...modification time...)
16.     {   char buf[1024], pattern[1024];

17.         ...extract strftime() pattern...
18.         strftime(buf, sizeof(buf), pattern,
19.                  localtime(ls->sb->st_mtime));

20.         return sfwrite(f,buf,strlen(buf));
21.     }
22.     ...
23. }
```

Figure 5: Printing file modification time

in our implementation of the **ls** command, it shows how the formatting extensions may be used.

Lines `1-5` define a type `Lsfmt_t` that combines the `Sffmt_t` type, a `struct stat*` for a file status object, and a `char*` for the file name. In this way, the **ls** application can pass along the file status data and file name to the formatting function. C casting rule allows a pointer to a `Lsfmt_t` object to be treated as a pointer to an `Sffmt_t` object. This way of extending a data structure to be passed back and forth between the library and the application code is commonly used in our libraries based on the discipline and method library architecture[10].

Line `9` identifies a formatting request for a path name via examining the string `env->t_str` to see if it defines the id `path`.

Lines `10-14` simply return the path name as a string to be further processed by the calling formatting function.

Line `16` identifies the print modification time request by examining the string `env->t_str` to see if it defines the id `mtime`.

Line `18-20` extracts from the `form` field the conversion string `%H:%M:%S` to pass to the ANSI-C function `strftime()`. This conversion string is assumed to be stored in the buffer `pattern`. The function `localtime()` is called first to convert the `time_t` value `env->sb->st_mtime` to an object of the type `struct tm` as required by `strftime()`. Both `strftime()` and `localtime()` are defined in the ANSI-C Standard.

Line `21` writes the result out to the given stream and returns the number of bytes written. Subsequently, the formatting function continues with processing the format string.

3 Performance

The new features do add complexity to the formatting functions. Since many applications, especially those based on Stdio, only use the basic formatting tasks, we need to assure that their performance are not adversely affected by the new features when they are not used. Toward this end, we perform a study comparing Sfio against various Stdio versions on basic data printing and scanning tasks.

	Hardware	**MHZ**	**OS**
O	Pentium II	200	SCO UNIX 3.2
K	Pentium II	333	UWIN/WIN32
F	Pentium II	333	Linux 2.2.12-20
W	Pentium II	450	BSDI 4.0.1
D	HP9000/889	400	HP-UX B.10.20
G	UltraSparc2	2x300	SUNOS 5.6
R	SGI Origin 200	4x270	IRIX64 6.5
T	DEC-Alpha	500	UNIX V4.0D

Table 2: Tested platforms

Table 2 shows the platforms used in the performance study. The first four systems are PCs running various Unix operating systems. UWIN/WIN32 is David Korn's UWIN system [6] that provides a Posix layer on top of the WIN32 environment. The last four are large servers from Hewlett-Packard, Silicon Graphics, Sun Microsystems and Digital Equipment running some respective Unix operating systems available from the vendors.

The conditions of the experiments were as follows:

- A test program prints 25,000 lines out to a file, then scanning the same lines back. Each line contains an instance of each of the patterns: `c,d,o,x,f,e,s`. The amount of data generated per run is about 1.7Mbs.

- To ensure uniformity, we wrote a single test program based on the Stdio interface. To test Stdio on a particular platform, we simply compiled the program using the native `stdio.h` header and library. To test Sfio, we compiled the program using the source compatibility header `stdio.h` provided by Sfio. This header mapped Stdio calls to Sfio calls mostly via macros. Such mappings did add some more work to the Sfio tests but we deemed that to be insignificant compared to the work done by the formatting tasks themselves.

- For UWIN/WIN32, the native WIN32 Stdio was used instead of the UWIN Stdio since the latter is just the same source compatibility interface provided by Sfio.

- The test programs always performed I/O to a same file in `/tmp`. In our environment, this ensured that the file would be on a disk local to the processing computer.

- All test runs were performed at night on lightly loaded machines. In fact, most machines were single user during the tests except for platform R, a large compute server.

- Times shown are totals of CPU and System time measured in seconds. Each data point was obtained as follows. Each test was run nine times. Then the highest two and lowest two values are discarded to remove certain outliers due to file caching effects on some platforms. The remaining five values are then averaged to produce the final result.

	Printing		Scanning	
	Sfio	Stdio	Sfio	Stdio
O	.82	1.01	.86	1.00
K	.42	1.96	.61	.73
F	.52	1.29	.66	1.22
W	.21	.43	.26	.30
D	.85	.90	2.06	2.07
G	.75	.85	.78	1.09
R	.40	.40	.49	.70
T	.25	.26	.33	1.39

Table 3: Timing results

Table 3 presents timing results on the mentioned platforms. Figure 6 shows the same data in bar charts. Below are a few comments on the data:

- Sfio outperforms Stdio on all platforms. Most of the improvement is due to new data conversion algorithms. For example, the decimal printing algorithm uses table look-up and an inline binary search to compute digits instead of

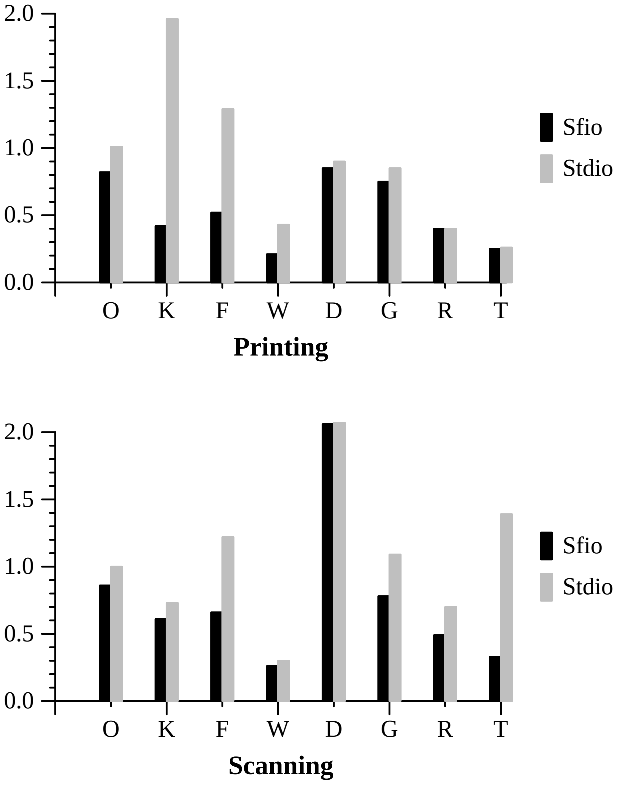

Figure 6: Formatting performance

the usual method of division and modulo by 10. This works particularly well on hardwares such as SUN SPARCs that use function calls for division and modulo. A full description of the Sfio conversion algorithms is not appropriate for this paper whose focus is on the new formatting features. Interested readers can peruse the freely available Sfio source code for details.

- It should be noted that Sfio is built from a single source base but configured differently on different platforms. For optimal performance, it is important that certain basic functions, e.g., string or memory copy, are matched to the best available methods on a particular platform. We use the tool Iffe[2] to automatically detect and compare different functions made available by a platform for the same purposes and generate appropriate configuration parameters.

- The native Stdio on platform K, Windows NT, has the worst printing performance relative to Sfio. Some of this poor performance is due the buffering strategy of the I/O package (i.e., small stream buffer) but a larger part is due to antiquated conversion algorithms.

- Platforms K and F are based on the same processor but use different operating systems, UWIN for K and Linux2.2.12 for F. The Sfio performance is slightly poorer on F than on K. To see if this difference is due to compilation environments, we reran the Sfio tests on platform K after recompiling with **gcc** version *egcs-2.91.66*, the same compiler on Linux2.2.12. The Sfio printing and scanning times on K are then `0.80s` and `0.77s` respectively. This shows that either `gcc` generates worse code than the native Microsoft-C compiler or its supporting libraries are not as optimized as the Microsoft-C ones, or both. Since the `gcc`-based timing results on K are also worse than that on F, it is likely that the support libraries for `gcc` on F are more optimized than on K.

- The printing performance of Stdio on platforms G, R and T is close to that of Sfio but its scanning performance is relatively much worse. This is particularly bad on T where Stdio scans data at a rate 4 times slower than Sfio. Since printing is more popular than scanning, perhaps these platforms recently improved their printing facilities though not the scanning ones.

- Platform D clearly has the worst performance in both printing and scanning. This is especially disappointing given the advertised processor speed. Since the timing results are similar between Sfio and Stdio, the poor performance must be due to the platform itself, i.e., the compiler or the support standard libraries.

The additional formatting features to define new patterns or redefine existing patterns do incur cost due to extra function calls. To see how much this cost might be, we wrote test programs to print a sequence of complex numbers whose real and imaginary parts are equal and range from `1` to `n`. For any given `n`, all programs produced identical output. Below are brief descriptions of the programs:

- **sfio%c**: This prints numbers using the method shown in Section 2.5.3.

- **sfio**: This prints numbers using the format string "`(%g,%g)`" in direct `sfprintf()` calls. For fair comparison with **sfio%c**, the program constructs `Obj_t` objects before using their parts in the printing calls.

- **complex**: This uses the `complex<double>` type in C++ and the output operator `>>` to print numbers to the standard output.

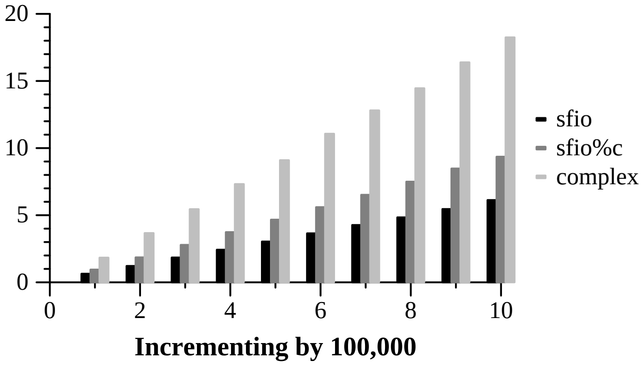

Figure 7: Times to print complex numbers

All programs were compiled on platform R using the compiler **g++** version 2.95. Figure 7 shows CPU+System times from test runs with `n` from 100,000 to 1,000,000 in increments of 100,000. Not surprisingly, the program **sfio** using direct `sfprint()` calls was fastest. The mapping of the new conversion pattern in **sfio%c** increased computation cost by about 50% relative to **sfio**. The program **complex** was slowest, about twice slower than **sfio%c** and three times slower than **sfio**. This shows that there is a significant performance cost to use the new extensions. However, this cost is not unreasonable in light of the cost incurred by a commonly used I/O facility in C++.

4 Conclusion

We discussed a number of extensions made to the printing and scanning functions in the Sfio library. These extensions enable safe and flexible manipulation of strings and scalar objects. Data formatting is fully generalized so that applications can provide their own interpretation of the conversion patterns and also define new ones. Examples were given to show how the new features enable applications that would be hard to build using Stdio.

A performance study was presented to show that, despite the additional formatting features, Sfio still performed as well or better than currently popular Stdio implementations when only standard formatting tasks are done. A separate experiment showed that the extended formatting features to define new patterns and/or redefine old ones could incur significant cost due to extra function calls. This cost should be balanced against the extra programming flexibility.

Although the Sfio's API is distinct from Stdio's, Sfio does provide source and binary compatibility packages for programs written on top of Stdio. The extensions discussed here are orthogonal to the features defined in the ANSI-C Standard[1]. Therefore, they can be transparently used by Stdio applications that are compiled or linked with the compatibility packages provided by Sfio.

Code availability

The source code for Sfio is freely available at:

`http://www.research.att.com/sw`

In addition, related commands and libraries are available at:

`http://www.research.att.com/sw/download`

References

[1] ANSI. *American National Standard for Information Systems - Programming Language - C.* American National Standards Institute, 1990.

[2] Glenn S. Fowler, David G. Korn, J.J. Snyder, and Kiem-Phong Vo. Feature-Based Portability. In *Proc. of the Usenix VHLL Conference*, pages 197–207. USENIX, 1994.

[3] Glenn S. Fowler, David G. Korn, and Kiem-Phong Vo. Sfio: A Buffered I/O Library. *Software—Practice and Experience*, Accepted for publication, 1999.

[4] ISO/IEC. *ISO/IEC International Standard 9899:1999(E) Programming Language - C.* IEEE, 1999.

[5] Brian Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice Hall, 1978.

[6] David G. Korn. Porting UNIX to Windows NT. In *Proc. of the 1997 Usenix Conference.* USENIX, 1997.

[7] David G. Korn and Kiem-Phong Vo. SFIO: Safe/Fast String/File IO. In *Proc. of the Summer '91 Usenix Conference*, pages 235–256. USENIX, 1991.

[8] Posix - part 2: Shell and utitilities, 1993.

[9] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1991.

[10] Kiem-Phong Vo. The discipline and method architecture for reusable libraries. *Software—Practice and Experience*, 30:107–128, 2000.