Writing Efficient C Programs Using the SAS/C® Compiler

Oliver Bradley, SAS Institute Inc., Cary, NC

ABSTRACT

If your programs are written in C and you use the SAS/C® compiler, there are a number of ways to make your programs run faster. This paper addresses ways to generate more efficient code and then discusses how to use C I/O efficiently.

IMPROVING CODE EFFICIENCY

Use the Global Optimizer

One of the easiest things you can do to improve the efficiency of your C code with Release 4.00 requires no code changes at all. Release 4.00 of the SAS/C compiler includes a Global Optimizer that performs optimizations such as merging common subexpressions, eliminating dead code, constant propagation, and strength reduction. The Global Optimizer also allocates registers, placing the most highly used variables for each section of code in registers. This eliminates any need for you to select and specify register variables by hand.

Take Advantage of Leaf Functions

A *leaf* function is a function that calls no other functions. This property means that the function is always at the end of a calling sequence. The compiler can tell that a *leaf* function calls no other functions and takes advantage of this. Instead of needing a fresh allocation of stack space, a *leaf* function uses a fixed area of storage in the CRAB for its automatic variables (as long as they do not exceed 128 bytes). This leads to a much more efficient entry and return sequence for these functions. You can take advantage of this by making heavily used functions *leaf* functions where possible.

Select the Prologue You Need

You can select a prologue for function calls based on your needs for speed, interlanguage communication and diagnostic information. You can make this choice at execution time through a runtime option or at compile time via the _linkage external variable. Four prologues are available, ranging from a *debugging* prologue that maintains maximum diagnostic information to a *minimal* prologue that does not even check for stack overflow. Provided that your program does not use interlanguage communication or call C from assembler, you can maximize its speed by using the minimal prologue once it is in production, if it does not use recursive algorithms whose stack space requirements cannot be predicted in advance.

Take Full Advantage of New Switch Optimizations

Release 4.00 uses some new techniques to generate code for switch statements. These techniques use indexed tables with one- or two-byte entries, so they execute quickly while minimizing data space used. The possible algorithms are

1. a sequence of CRs
2. a sequence of CLIs
3. a single-level direct lookup table
4. a two-level direct lookup table.

Algorithms 3 and 4 are the new indexed algorithms. For each switch encountered in the source file, the code generator analyzes the size and execution time that would result from each method and chooses the best method. For switches with a small number of cases, one of the first two methods will generally be used since the overhead of table lookup is not justified. For a larger number of cases, one of the indexed algorithms is used for all but highly sparse switch statements. However, these algorithms do require one table entry for each value in the switch range (the difference between the lowest and highest values in `case` statements). Thus, it is advantageous for you to reduce the range of your switch statements if possible since you can save a lot of table space this way. If you have one or two `case` values very different from the others, you might want to test for them separately or handle them at the `default` label (which is not part of the range).

Below is an example of an inefficient case statement and a corresponding efficient one. The difference is the range.

Inefficient:

```
switch c {
  case 5: ...;
  case 11: ...;
  case 12: ...;
  case 901: ...; /* This case increases the range */
  case 23: ...;
  case 33: ...;
  case 27: ...;
  case 28: ...;
  case  4: ...;
  case -1: ...;
  default: ...;
}
```

Efficient:

```
switch c {
  case 5: ...;
  case 11: ...;
  case 12: ...;
  case 23: ...;
  case 33: ...;
  case 27: ...;
  case 28: ...;
  case  4: ...;
  case -1: ...;
  default:
     if (c==901) ...;
     else ...;
}
```

The range of the switch can be reduced either by renumbering the cases (changing 901 to 29 in the example above) or in some cases by processing other values in the default clause. When case values are based on `#defines` or `enums`, it is relatively easy to control the case values to decrease the range of the switch. `Enums` are particularly useful for symbolic switching values since by default they are numbered consecutively. This results in high switch densities when they are used.

The two-table lookup technique and the encoding of the case destination address in a two-byte format makes table-based switching practical at lower densities than would otherwise be allowed. This same technique also allows more efficient handling

of multiple labels per set of statements. So the following increases the likelihood of getting a table-based switch, compared to separate code for cases 1, 5 and -1:

```
case 1:
case 5:
case -1:
  stmts...;
```

Table-based switches (methods 3 and 4 above) both handle multiple code regions (that is, code spanned by the switch is not limited to 4K). Region information is efficiently encoded in the table and requires only a few instructions to extract and use.

Use New ANSI Keyword const

ANSI added the `const` keyword to the C language. This keyword is used to specify variables that do not change during execution of the program, such as fixed tables.

Release 3.01 supported `const`, but Release 4.00 does more with it. Most external and static items that are declared `const` are placed in CSECTs and initialized at compile time rather than execution time. This leads to a saving in data space and run-time memory requirements, especially if your program is executed by multiple users simultaneously. The compiler can place `const` data in a CSECT because the keyword guarantees that it cannot be changed during execution of your program. Thus, placing it in a CSECT does not affect reentrancy. (Contrast this with the use of the NORENT compiler option, which places all external and static data in a CSECT whether it is modified or not.)

Use New Bit Field Types

This technique saves you data space rather than execution time. Release 4.00 adds support for `char` and `short` bit field types. This means that bit fields that previously required a full `int` (four bytes) may now require only one or two bytes. If you have large numbers of bit fields, the saving is significant.

There are two ways to allocate `char` and `short` bit fields. You can specify `char` and `short` directly on the declaration. `signed` and `unsigned` are also supported, with `unsigned` being the default. Of course, such declarations are not portable. You can also use the compiler BITFIELD option to change the size of `int` bit fields from the default four. This leaves your program source code unchanged. Bit fields longer than 8 or 16 bits continue to be supported, too. They simply use as many `shorts` or `chars` as they need, up to the maximum of 32 bits. Since alignment requirements are reduced too, your structures may shrink even if they contain only bit fields larger than 16 bits.

Use Auto Variables for Highly Used Items

Consider the following code:

```
p->count += q->r->n;
x = func();
if (x)
   p->count += q->r->n;
```

If you look at an OMD of code like this, you'll notice that the compiler accesses *count* and *r* from memory after the function call, rather than keeping them in registers. Why is this? Because *func* might have changed them. The compiler generally doesn't know where p and q point and has to assume that *func*, or something it calls, has access to where p and q point and might change *count* and *r*.

Often, you know that *func* doesn't make changes of this sort. Look what happens if you recode this as

```
/* Earlier in current function */
auto int count, n;
count = p->count;
n = q->r->n;
/* ... */
count += n;
x = func();
if (x)
   count += n;
/* If needed, before return from function */
p->count = count;
```

count and *n* are kept in registers over the function call. By recoding it this way, you've told the compiler that these values aren't changed in ways it doesn't know about. The Global Optimizer is able to optimize the use of these values fully because it has maximum information about them. If your code is portable, compilers other than the SAS/C compiler can take advantage of the extra information you've provided.

Of course, I'm not suggesting that you assign every indirect access to a local variable. The savings make it worth doing for heavily used values where the speed of the code is a concern.

Use Memory Pooling

If your program allocates a large number of memory blocks of the same size, there are more efficient ways than `malloc` to do it. The SAS/C library provides the `pool`, `palloc`, `pfree` and `pdel` functions to manage storage pools. Allocating or freeing an element from a storage pool is extremely fast. Normally not even a function call is needed. If your program uses several sizes of memory blocks, you can define one pool for each size.

Use Built-in Functions

Many library functions are built-in; that is, the compiler generates in-line code for them. These functions include

- `memcpy` (copy a block of memory)
- `memset` (initialize a block of memory)
- `memxlt` (translate a block of memory)
- `memcmp` (compare two blocks of memory)
- `strcpy` (copy a string)
- `strlen` (find the length of a string)
- `strcmp` (compare two strings)
- `abs` (absolute value of an integer)
- `fabs` (absolute value of a double).

The built-in code is generally much faster than the function call. As well as avoiding function call overhead, the in-line code can take advantage of information the compiler has, such as knowing that a length is a constant. Many built-in functions generate faster code for a constant length, so you should specify the length as a constant whenever you can.

An enhancement to built-in functions for Release 4.00 allows you to get even better code for many of them when you specify a length that is variable but less than 256 or 32768 bytes. When you tell the compiler that the length is less than 256 by casting it to an `unsigned char` or that it is less than 32768 by casting it to a `short`, the compiler takes advantage of this information to generate a more efficient code sequence. For example:

```
memcpy(p, q, (unsigned char)len);
```

Sometimes you have a choice between a mem-prefix function and a str-prefix function. If you already know the length to be processed, then the mem-prefix function will always be more efficient, even though both are built-in. This is especially true if you also take advantage of the previous optimization. (If your code is portable, though, be aware that mem-prefix functions are not necessarily faster than str-prefix ones for other hardware.)

To get the built-in code, just include the appropriate header file, for example `<string.h>`. It's always a good idea to include the header file. That way, your code automatically takes advantage of new built-in functions in new releases.

IMPROVING I/O EFFICIENCY

Many of the I/O recommendations that follow are contrary to what a UNIX® programmer would expect. When porting programs from UNIX, be aware of the differences between UNIX and the IBM® mainframe operating systems.

Use True Relative Files Wherever Possible

The library does the most efficient I/O to *true relative* files. These are files that are directly byte-addressable. For Release 4.00, files with fixed-length, standard-sized records (on MVS, RECFM=FBS) are directly byte-addressable. These files can be readily shared with most non-C programs.

The most efficient way to access *true relative* files is through the `fopen/fseek/fread/fwrite` family of functions. These are the "low-level" functions on the IBM mainframe. `open/lseek/read/write` are implemented on top of the `fopen` family (the reverse of the UNIX situation).

If a file is NOT *true relative*, then you need to use the `open/lseek` family of functions to obtain byte addressability since `fseek` does not support complete byte addressability to these files in a UNIX-compatible way. The `open/lseek` functions copy files that are not *true relative* to a temporary file. The `open/lseek` functions do not copy a *true relative* file, but access is still slightly slower than via `fopen/fseek` because of the extra layer of functions.

A good alternative, if you want to do random access to a file but don't need complete UNIX-style byte addressability, is to use `fread` and `fwrite` in conjunction with the new ANSI `fgetpos` and `fsetpos` functions. These functions operate on almost all file types, are portable to other ANSI implementations, and are likely to operate on all new file types supported by the library in the future. These functions are also considerably more efficient than `lseek`.

Choose Binary I/O to Text I/O Wherever Possible

Under some operating systems, for example UNIX, there is little difference between binary and text I/O. On the IBM mainframe, there is a major performance difference. In a text file, certain characters are significant. For example, when writing a text file on the IBM mainframe, a new-line character becomes a record boundary. Under UNIX, the new-line is written out to the file like any other character. This means that the SAS/C run-time library must scan all output to a text file for special characters that have significance. This scanning process slows down text I/O compared with binary I/O, where no such scanning is needed. So use binary I/O whenever you have a choice.

If you do need to use text I/O, note that `fread` and `fwrite` have no performance advantage over `fgets`, `getc`, and `putc` for text files. This is because `fread` and `fwrite` need to do the same processing of special characters that `fgets`, `getc`, and `putc` do.

Use fread, fwrite, afread, or afwrite for Record-oriented I/O

The C file model does not contain the concept of a record, except perhaps as a string of bytes delimited by a new-line. If your C programs need to process IBM mainframe files on a record-oriented basis, then these functions provide the most efficient way to do it.

For files with fixed-length records, use `fread` and `fwrite`. Read and write an exact number of records at a time, that is, an integral multiple of the record length. It is not necessary to know the block size of the file. Use of `fread` and `fwrite` in this manner is portable.

For files with variable-length records, use `afread` and `afwrite`. If you don't know the record length in advance for `afread`, supply a size of one and a count equal to the maximum record length. This will read any record size up to the maximum successfully. It is not necessary to know the block size of the file. Of course, use of these functions renders your program non-portable.

Use DIV Files

IBM recently added *linear data sets* to VSAM, which are processed using a very efficient technique called *data in virtual* (DIV). Release 4.00 of the SAS/C compiler supports linear data sets as *true relative* files and accesses them via DIV. Because DIV processes data entirely in virtual memory, it provides an extremely efficient way of accessing large amounts of data, particularly if access is random. If your application permits, consider making its permanent files DIV files.

Use Low-level OS I/O or CMS I/O Functions

Low-level functions are available that provide more direct access to the OS and CMS file systems. (The OS functions were added in Release 4.00.) While they sacrifice portability, these functions provide the maximum performance, especially for specialized applications such as accessing multiple members in an OS PDS or searching for CMS files whose names match a pattern. Release 4.00 of the SAS/C CLINK utility for MVS uses the OS PDS I/O functions. You'll see by comparing its performance with Release 3.01 that these functions can improve the performance of a suitable application dramatically.

Avoid fgetc, fputc, and gets

The `fgetc` and `fputc` functions are the most inefficient functions in the C library because they do a function call for every character read or written. (The ANSI Standard requires them to be true functions, not macros.) Instead, use the `getc` and `putc` macros, which do a function call only when a buffer is exhausted. `getc` and `putc` are also more likely to take advantage of future compiler enhancements; `fgetc` and `fputc` are limited by the requirement for a function call. You can also allocate your own buffer and use any of the other I/O functions. Any other I/O technique outperforms `fgetc` and `fputc`.

The reason for avoiding `gets` is different. With `gets`, there is no limit on the amount of data read; data are read until a new-line is encountered. If the file contains an unexpectedly long record, memory following the `gets` buffer could be overwritten, causing unpredictable problems. Unless you are very sure that your program will never read such files, avoid `gets`. Use `fgets`, where the maximum length to read is a parameter.

Use Appropriate File Characteristics

Earlier I discussed using *true relative* files wherever possible. Another way to get better I/O performance on MVS is to choose a large file block size. Choose the largest block size that is compatible with your disks and with any other uses of the files. A large block size is an advantage whether or not the file is *true relative*.

Having chosen a large block size, take maximum advantage of it by reading or writing large amounts of data at a time, for example, by specifying a large number of items or item size or both to `fread`.

A very useful optimization for temporary files under MVS is to allocate them to virtual I/O (VIO) files. VIO files are written to and read from virtual memory. A moderate-sized file might remain in real memory for its entire usage. Even if parts of a VIO file need to be paged out, MVS handles paging very efficiently.

Put the Transient Library in LPA or a CMS Segment

Most of the SAS/C I/O functions are in the transient library and are loaded when needed. This means that there is a big performance advantage in placing the transient library in LPA or a CMS-shared segment. Loading is much faster, especially from the CMS-shared segment, and there is a further performance gain if the module has already been paged in for another user or job using C I/O.

See What the Library Has First

Before writing your own function, see what is available in the run-time library. The run-time library contains many functions. Areas such as string handling, memory allocation, I/O and specialized system interfaces have a wide variety of functions available.

If you do find a library function that does what you need, not only are you saved the trouble of writing and debugging your own function, but the library function is also likely to run very efficiently. All library functions have been written with efficiency in mind. As discussed earlier, many even generate in-line code.