Producing Efficient SAS® Applications
Daniel J. Squillace
SAS Institute Inc. Cary, N.C.

OBJECTIVE

The objective of this paper is to acquaint you with DATA step coding techniques, PROC usage considerations, and SAS System options which generally result in more efficient SAS applications. Topics we will cover include:

- Motivation for efficient applications
- DATA step considerations
- Commonly used procedure considerations
- SAS/FSP®
- SAS/GRAPH®
- SAS Display Manager
- Some SAS system options
- Data set block sizes

MOTIVATION FOR EFFICIENT APPLICATIONS

When an application begins to cost more than the user is willing to pay, then we have a motivation to make it more efficient. The cost of an application may be expressed in terms of resource utilizations or data center charges. Applications which are resource intensive or which are run frequently are usually the first to require attention.

In general, the additional effort you put into making an application more efficient should be in proportion to the savings which can be gained. An effective way to produce more efficient applications is to incorporate good efficiency practices into your programming style. Once you do so, efficient practices will become second nature and will require extra effort on your part only in unusual situations. If you have to spend too much time worrying about efficiency or if you find yourself resorting frequently to clever and obscure code, you are probably using the wrong tool for the job!

DATA STEP EFFICIENCY CONSIDERATIONS

DATA Step Logic

The DATA step executes an implied loop. For example:

```sas
DATA B;
INFILE A;
INPUT X Y Z;
```

is equivalent to

```sas
DATA B;
INFILE A;
INPUT X Y Z;
OUTPUT;
RETURN;
```

Processing performed during each loop iteration includes resetting all variables which have not been RETAINED to missing. If you refresh each variable during each iteration of the DATA step yourself or are willing to forego having the variables reset to missing, you can save some time by explicitly controlling the looping yourself as shown below:

```sas
DATA B;
INFILE A END=EOF;
DO UNTIL(EOF);
INPUT X Y Z;
OUTPUT;
END;
```

The percentage CPU time reduction resulting from this technique will depend on the number of variables in the DATA step and the amount of processing done within the DATA step. There are other things to consider when using this technique. For one, `_N_` will always be one. For another, "INPUT @;" will have to be terminated with an explicit "INPUT;".

Consolidating Constants

The DATA step compiler does not recognize and consolidate duplicate constants. Therefore, you can reduce DATA step memory requirements by assigning constants to variables. For example instead of

```sas
X=0;
Y=0;
Z=0;
```

use

```sas
ZERO=0;
X=ZERO;
Y=ZERO;
Z=ZERO;
```

or better yet

```sas
RETAIN ZERO 0;
X=ZERO;
Y=ZERO;
Z=ZERO;
```
Variable Length Considerations

For numeric variables, the LENGTH statement affects only external storage requirements. Numeric variables of less than length 8 are kept in memory and used in computations as length 8 but are truncated when written to a SAS data set. Therefore, computation time is not affected by numeric variable length. However CPU time and external storage savings can be realized by specifying shorter lengths for numeric variables because more observations will be stored in each block.

Before specifying short lengths on your numeric variables, you need to be aware of precision implications when truncating variables. See (SAS Institute 1985, 157) and (Langston 1987) for further information.

DATA Step Initialization

Make sure initialization processing is done only once. Use _N_ =1, a missing value check, or your own first-time variable to control initialization logic execution. For example

```
DATA A;
RETAIN DATE;
DATE = TODAY();
```

is better coded as

```
DATA A;
RETAIN DATE;
IF DATE = . THEN DATE = TODAY();
```

Symbolic variables &SYSDATE and &SYSTIME are sometimes suitable in place of the TODAY() and DATETIME() functions. These variables contain the date and time of the SAS session initialization. For an example consider

```
DATA A;
RETAIN DATE "&SYSDATE"D
TIME "&SYSTIME"T;
```

DROP and KEEP Data Set Options

Use the DROP or KEEP data set options on input data sets instead of the DROP or KEEP statements. This technique will save CPU time by not bringing unnecessary variables into the DATA step. For example

```
DATA A;
SET B;
KEEP X Y Z;
```

is better coded as

```
DATA A;
SET B (KEEP = X Y Z);
```

The results of two tests comparing the relative CPU times of these techniques are shown below.

Test 1: 17,000 observations
- 106 variables, keep 7
  - KEEP statement: 1.4
  - KEEP option: 1.0

Test 2: 303 observations
- 190 variables, keep 25
  - KEEP statement: 1.9
  - KEEP option: 1.0

Use of the KEEP or DROP data set options can be particularly beneficial whenever you are interested in selecting only a few variables from a large observation.

Input Record Selection

When reading records from an external file, read the selection variables first. Then read the remaining variables only if the record is to be kept. For example

```
DATA A;
INFILE B;
INPUT JOBNAME CPUTIME IOCOUNT DASDSP PAGESEC;
IF JOBNAME = 'THEJOB';
```

is better coded as

```
DATA A;
INFILE B;
INPUT JOBNAME @;
IF JOBNAME = 'THEJOB';
INPUT CPUTIME IOCOUNT DASDSP PAGESEC;
```

In a test of reading 2,000 records with 10 numeric variables and one selection variable, the relative CPU time for the first technique compared to the second was 2 to 1.

Array Processing

Arrays as implemented in the SAS DATA step language are very convenient, but the convenience carries a heavy performance penalty. Array indices are floating point numbers and array elements can be of varying length. These characteristics add considerably to the processing overhead as can be seen from the following example.

```
ARRAY A(5) A1-A5;
DO I = 1 TO 5;
A(I) = ZERO;
END;
```

is eight times more expensive than

```
A1=ZERO;A2=ZERO;A3=ZERO;A4=ZERO;A5=ZERO;
```
IF (logical OR expression) versus IF-THEN-ELSE

Use IF-THEN-ELSE constructs or SELECT statements instead if an IF statements with logical OR expressions. Execute tests in order of decreasing probability; that is, check the most likely conditions first. For example,

```plaintext
IF JOB = 'A' OR
   JOB = 'B' OR
   JOB = 'C' OR
   JOB = 'D'
THEN TYPE = 'T1';
ELSE TYPE = 'T2';
```

is better coded as

```plaintext
IF JOB = 'A' THEN;
ELSE IF JOB = 'B' THEN;
ELSE IF JOB = 'C' THEN;
ELSE DO;
   TYPE = 'T2';
   GO TO NEXT;
END;
TYPE = 'T1';
NEXT:
```

If you don't like GO TO statements, the GO TO can be eliminated by putting TYPE='T1' before the first IF statement. The SELECT statement will generate the same code as the example given above but is more concise.

```plaintext
SELECT (JOB);
   WHEN('A');
   WHEN('B');
   WHEN('C');
   WHEN('D');
   OTHERWISE DO;
   TYPE='T2';
   GO TO NEXT;
END;
TYPE='T1';
NEXT:
```

A test case was constructed for each of the examples given above in which ten conditions were checked. The relative CPU time values are shown below.

<table>
<thead>
<tr>
<th></th>
<th>Match on first test</th>
<th>Match on last test</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical OR</td>
<td>3.8</td>
<td>1.5</td>
</tr>
<tr>
<td>IF/THEN-ELSE</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>SELECT</td>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>

The LINK Statement

The LINK statement has obvious advantages in terms of clarity and structure, but GO TO or in-line code are often faster. For example

```plaintext
SELECT (SMFREC);
   WHEN (6) LINK T6;
   WHEN (30) LINK T30;
   OTHERWISE;
END;
```

takes more CPU time than

```plaintext
SELECT (SMFREC);
   WHEN (6) GO TO T6;
   WHEN (30) GO TO T30;
   OTHERWISE;
END;
```

The extra overhead of LINK is due to subroutine linkage code which is avoided in the GO TO example. In a test of LINK versus GO TO with short subroutines (10 assignment statements), the version with LINK took 1.4 times more CPU time than the version with GO TO.

Avoiding Unnecessary Statement Execution

Use IF-THEN-ELSE or SELECT instead of sequential if statements. For example, instead of

```plaintext
IF X=1 THEN C=D;
IF X=2 THEN C=E;
IF X=3 THEN C=F;
```

or

```plaintext
SELECT (X);
   WHEN (1) C=D;
   WHEN (2) C=E;
   WHEN (3) C=F;
   OTHERWISE;
END;
```

Take statements which do not need repetitive execution out of DO loops.

Data Set Management

Use PROC COPY instead of the DATA step SET statement to copy SAS data sets. That is, instead of

```plaintext
DATA DDA.B;
SET A;
```
it is better to code

PROC COPY IN=WORK OUT=DDA;
SELECT B;

As an example, the relative CPU time to copy a 100,000 observation data with 26 variables using SET compared to using PROC COPY is 2 to 1.

Use PROC APPEND instead of DATA step SET statements to add new observations to a data set. For instance, instead of

DATA A;
SET A B;

use

PROC APPEND BASE=A DATA=B;

COMMONLY USED PROCEDURE CONSIDERATIONS

Univariate Statistics Procedures

Procedures UNIVARIATE, MEANS, and SUMMARY can all be used to calculate mean, standard deviation, min, max, etc. However PROC UNIVARIATE is somewhat more expensive than the other two procedures and should be used only when percentiles, quartiles, and/or distribution graphs are required. The real question then is when to use PROC MEANS and when to use PROC SUMMARY. PROC SUMMARY is as efficient as PROC MEANS when calculating statistics for an entire population. Differences begin to appear when you specify BY or CLASS variables. In general, PROC MEANS by itself is more efficient in this situation; however, if the data must be sorted before being used by PROC MEANS, you will often find that PROC SUMMARY with CLASS is more efficient than PROC SORT followed by PROC MEANS. Also, if you plan to run analyses at several levels, it is more efficient to use PROC SUMMARY and then subset the output dataset by _TYPE_ than to make multiple PROC MEANS runs. Tests were run against two 10,000 observation data sets with PROCs MEANS, SUMMARY and UNIVARIATE. The first data set had two BY/CLASS variables with ten values each; the second had three BY/CLASS variables with ten values each. Statistics were gathered for four variables. The chart below shows the relative CPU times for the test runs.

Univariate Test Results

<table>
<thead>
<tr>
<th></th>
<th>10 by 10 by 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEANS</td>
<td>0.6</td>
</tr>
<tr>
<td>SORT=MEANS</td>
<td>1.0</td>
</tr>
<tr>
<td>SUMMARY</td>
<td>1.0</td>
</tr>
<tr>
<td>UNIVARIATE</td>
<td>1.9</td>
</tr>
</tbody>
</table>

PROC TABULATE

PROC TABULATE calculates descriptive statistics and cell percentages and then generates tabular reports from these statistics. It is often more efficient to generate the statistics with PROC MEANS or PROC SUMMARY with the NWAY option first and then invoke PROC TABULATE. This is a case of two being less than one! For example inserting

PROC SUMMARY NWAY;
CLASS DIV DEPT YRMON;
VAR TSOCPU;
OUTPUT OUT=SUMWAY SUM;

before

PROC TABULATE;
CLASS DIV DEPT YRMON;
VAR TSOCPU;
TABLE (DIV*DEPT ALL), YRMON*TSOCPU(SUM PCTSUM=DIV*DEPT ALL);

produces the same results as the PROC TABULATE alone, but requires less cpu time. A test case run with the above code took 20% less cpu time.

Print Procedures

Supply formats for numeric variables. Use the UNIFORM option to reduce CPU time. PROC QPRINT can be substantially less expensive than PROC PRINT, but does not have all the features of PROC PRINT. Conversely, PROC QPRINT has some capabilities not found in PROC PRINT including multi-column formatting. See Technical Report P-146 for PROC QPRINT documentation. The following table shows the relative CPU times required to print 1,000 observations with seven numeric variables.

PRINT/QPRINT tests

<table>
<thead>
<tr>
<th></th>
<th>** No Formats Specified **</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINT</td>
<td>2.00</td>
</tr>
<tr>
<td>PRINT UNIFORM</td>
<td>1.65</td>
</tr>
<tr>
<td>QPRINT</td>
<td>1.10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>** Formats Specified **</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINT</td>
<td>1.84</td>
</tr>
<tr>
<td>PRINT UNIFORM</td>
<td>1.45</td>
</tr>
<tr>
<td>QPRINT</td>
<td>1.00</td>
</tr>
</tbody>
</table>
SORT Considerations

FILSZ parameter If your system sort utility supports the FILSZ parameter, set FILSZ in your SAS system options. This option can improve sorting efficiency because it will cause the sort to be given more accurate information about the number of records to be sorted.

SORT and SORTT You can use either your own system sort utility or one supplied with the SAS System to sort SAS data sets. Your system sort is used by setting SORTPGM=name and invoking PROC SORT. The SAS System sort may be used by setting SORTPGM=SAS or by invoking PROC SORTT. PROC SORTT is efficient when sorting small data sets. However whether it is more efficient than your system sort depends on which utility it is and how it is installed. Comparisons run on the SAS Institute production CMS system against SyncSort™ release 6.0c installed in a shared segment show the crossover point comes at about 300 observations (observation size was 232 bytes). The same tests run against the current MVS SyncSort release reached a crossover point at about 2000 observations.

SAS/FSP CONSIDERATIONS

The FSEDIT procedure provides the AUTOSAVE command to allow you to control how often checkpointing is done by closing and reopening the data set. The system default is to do this every 25 observations; hence, you will lose no more than 25 observations if the system crashes during an editing session. There is a definite tradeoff between overhead and data loss (or re-entry) potential in setting this option. Setting AUTOSAVE to 1 costs about five I/Os per observation with a single variable data set in a small directory.

SAS/GRAPH CONSIDERATIONS

Use hardware fill capability (POLYGONFILL) on devices for which it is applicable. POLYGONFILL results in much less CPU time, faster drawing time, and fewer output lines to SPPOOLed devices.

Use empty fonts for previewing. For example, use XSWISSE instead of XSWISS and XSWISSBE instead of XSWISSB.

Generate graphs in batch jobs and store them in GREPLAY catalogs for later viewing. This is usually effective because in many installations batch CPU usage rates are lower for batch than for interactive work and even lower for off-peak than for prime time work.

Use native device drivers instead of vendor access methods when available.

IBM 3179G vector graphics devices are much faster and use considerably less CPU time and communications line time than program symbol set devices. These devices are more expensive than PS devices; however, if you are a moderate to heavy graphics user, you are likely to save money in the long run.

SAS DISPLAY MANAGER

AUTOROLL LOG ON/OFF command The AUTOROLL LOG ON command causes a separate I/O to be sent to the terminal for each line written to the log window. The most noticeable effect of this was heavy data traffic and slow response on remote lines. Local users usually do not notice a response problem due to the many terminal I/Os. However, the CPU time cost is heavy in both cases. AUTOROLL LOG OFF causes the log window to be updated only when you are prompted for input. At that time, of course, you may scroll the log backwards to see what transpired since your last input. The system default is AUTOROLL LOG OFF. To convince yourself this should be your installation default, run PROC OPTIONS both ways and compare the results. The CPU time ratio is about 10:1.

SPOOL/NOSPOOL option The SPOOL option is intended to be used when running the SAS System interactively without the display manager. When SPOOL is in effect, SAS input statements are stored in the WORK library for later retrieval by the %INCLUDE command. You should set NOSPOOL when running under the display manager because the RECALL command provides similar function.

SAS SYSTEM OPTIONS

Number of history generations The GEN=n option specifies how many generations of history data are to be kept for SAS data sets. When GEN is nonzero, the SAS source statements used to create the data set are stored in the directory. This storage of history information can take up considerable library space and add significantly to the number of I/O operations required to run a SAS program. If you specify GEN=0, you can expect to reduce I/O by 10% or more. The amount of space saved by GEN=0 will vary over a wide range. I have seen extreme cases in which the data occupied less than 2% of the space required for a data set; the rest of the space contained history information.

There are times, of course, when keeping history information is entirely appropriate, perhaps in studies where it is necessary to keep an audit trail of how the data was processed. Ideally, it would be best to set GEN=0 as your normal default and set it higher only when you have a particular reason to keep
history information. At the very least, you should have large production jobs for which
the source code is well controlled and documented explicitly set GEN=0.

MEMFILL option This is a very expensive debugging option. Please do not turn it on
unless SAS Institute specifically asks you to do so for problem analysis.

MACRO processing options

IMPLMAC and MAUTOSOURCE Options One
of the significant enhancements included in
Version 5 SAS software is an autocall facility
for macros. It is no longer necessary to explicitly include each macro you want to use.
Macros may instead be pulled in and compiled
on demand from a macro library. This facility
can save resources in that macros will not be
retrieved and compiled unless required. The
macro autocall facility is turned on by
specifying the MAUTOSOURCE option..

More than one macro may be stored in a single
library member provided that one macro in the
group matches the library member name. This
can be a handy way to bring in a group of
related macros with one directory search.

There is an interaction of MAUTOSOURCE with
another option, IMPLMAC, that you should be aware of. Specification of IMPLMAC allows you
to use statement-style macros in your SAS
programs. With IMPLMAC in effect, each SAS
statement is potentially a macro and the first
word (token) in each statement must be checked
to see if it is a macro call. When IMPLMAC is
in effect without MAUTOSOURCE, no special
checking takes place until the first
statement-style macro is compiled. When
MAUTOSOURCE is on, however, this checking
must be done unconditionally. The initial
occurrence of a word as the first token of a SAS
statement results in a search of the autocall
library. There can be a significant number of
directory searches, especially during the
compilation of a large DATA step, in addition to
the CPU time necessary to maintain and search
the symbol table. The combination of
MAUTOSOURCE and IMPLMAC can add 10% or
more to the CPU and I/O resources required for
a job. As an example, the compilation of a SAS
program containing about 420 statements
required 95 autocall library directory searches
and an additional 25% CPU time. For best
performance, you should use NOIMPLMAC as
your default.

MSIZE and MLEAVE options When macros are
compiled, they are stored in the work data
library in 2048-byte execution frames. These
frames are brought into memory during macro
execution. If insufficient storage has been set
aside to hold the execution frames, extra work
data set I/O is incurred during macro execution
to support frame paging. (MSIZE-MLEAVE/2k)
gives the number of memory frames available for
macro processing. The default is three. The
cost of having too few macro execution frames
can be significant while the cost of
overallocation is trivial. A good set of defaults
would be MSIZE=128k and MLEAVE=64k.

Data Set Block Sizes

The benefits of large block sizes have been well
documented for MVS. The SAS system under
MVS is no exception as can be seen from the
following table of relative elapsed times to
create a SAS dataset over a range of blocking
factors per IBM 3380 track.

<table>
<thead>
<tr>
<th>Blocks per track</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative elapsed time</td>
<td>1.0</td>
<td>1.4</td>
<td>1.8</td>
<td>2.3</td>
<td>2.8</td>
<td>3.3</td>
</tr>
</tbody>
</table>

SAS System performance under CMS can also be
greatly affected by logical block size. The
following table shows the relative elapsed time
and total CPU time for writing and reading a
SAS dataset containing 30,000 200-byte
observations. For more information, see
(Squillace 1987).

<table>
<thead>
<tr>
<th>Logical Block Size Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>(relative times)</td>
</tr>
<tr>
<td>block size</td>
</tr>
<tr>
<td>2k</td>
</tr>
<tr>
<td>4k</td>
</tr>
<tr>
<td>8k</td>
</tr>
<tr>
<td>16k</td>
</tr>
<tr>
<td>32k</td>
</tr>
</tbody>
</table>

A Note for CMS Users... If you are still
running 82.3 format libraries, convert them! SAS data library I/O under CMS is markedly
faster in Version 5 than it was in release 82.3
and requires substantially less resources.
Furthermore, the new format requires less
DASD space, especially for small files. The
minimum space required for a file under the old
format is two times logical block size.
Therefore, there was a definite disadvantage to
using a large block size as the default under
that format. Version 5 of the SAS System will
create all new libraries in the new format;
however, libraries created under release 82.3
will continue to be maintained in the old format.
You will clearly benefit by converting libraries
created under release 82.3 to the new format.
As an example, creation of a data set at 32k
blocksize in 82.3 library format takes 4.9 times

87
more elapsed time than creating the same data set using Version 5 library format. For further information see (Squillace 1987).

SUMMARY

We have covered a range of SAS programming practices which can make your SAS applications more efficient. Incorporating these practices into your programming style is an effective way to improve the efficiency of your applications.

References:


Thanks to John McCann of Morino Associates for contributing ideas to this paper.

SAS, SAS/GRAPH, and SAS/FSP are registered trademarks of SAS Institute Inc. Cary, NC. USA.

SyncSort is a trademark of SYNCSORT INC. Englewood Cliffs, NJ. USA.