

DATA Step Efficiency and Performance

Jeffrey A. Polzin, SAS Institute Inc., Cary, NC

INTRODUCTION

Since the first inception of the SAS[®] DATA step language and its subsequent evolution over the years, the SAS DATA step language has appeared to be simple; yet, it can also be deceptively complex. Sometimes, the most obvious change in SAS DATA step code may make a difference in the behavior or in the performance of the program, or it may make no difference at all. Sometimes, an obscure change can have a dramatic effect on the program's efficiency and performance.

This paper will look at a number of areas that a SAS programmer can utilize to enhance the performance of DATA step execution. Both execution time and space with respect to program efficiency will be considered and concrete suggestions will be made.

When programming in the SAS DATA step language or in the SAS language in general, the programmer has to be aware of how an application is developed so as to obtain the greatest possible amount of execution efficiency. This paper highlights areas in which an application programmer can increase the efficiency of his applications and areas in which the SAS DATA step and the SAS System assist in the efficient execution of those applications.

EFFICIENCY MEASUREMENT

To gain an understanding of efficiency, we must first look at how it is measured. There are five primary measurements that describe efficiency in terms of program performance: CPU time, elapsed time, I/O, memory, and data storage. The first two measurements are baseline measurements, in that all the others impact these in one way or another. Generally, when one measurement is reduced or increased, it influences the others in varying degrees.

When analyzing program efficiency and its impact upon performance, two primary resource measurements are sought: CPU and memory. These measurements described the situation of how to minimize both the time and space requirements of an executing program. It is a difficult balance to manage either one so as not to overly influence the other or impact the other measurements of I/O and elapsed time.

The performance measurements of I/O and data storage are directly related to the volume of data and its pattern of access. The larger the amount of data to be processed or the degree to which the data are randomly accessed has an impact on both measurements. In most environments, I/O does have an impact upon the measurement of elapsed time, because for each I/O request, control is relinquished by the user's program to the operating system to perform requested work. There are other factors external to the executing program that do impact the total elapsed time required to execute a program, but they are not within the scope of this discussion.

USER CONTROLLED EFFICIENCIES

During application development, the developer must have a thorough knowledge of the requirements of the requested analysis and its associated data. In this section, techniques are discussed that assist in the improvement of overall application efficiency. Areas are highlighted and in many cases simple examples of concepts are presented. Keep in mind that each of these suggestions may not be the most appropriate in all situations, as

each suggestion is completely dependent upon the aspects of the application requirements.

Data Storage

SAS data sets are the most efficient storage mechanism for data managed by the SAS System in terms of CPU and I/O overhead. Whenever possible, processing and management of data should be accomplished using SAS data sets. When this is not possible, the SAS System does provide the capability to process and manage data stored in external data formats, such as an external database system or external files. SAS/ACCESS[®] software and DATA step views provide the ability to process and manage this external data.

Many applications require the use of summarization data to provide additional analysis information. Efficiencies can be gained by summarizing data in the minimal number of steps possible. The SUMMARY procedure is one of the most widely used tools in the SAS System to provide data summarization. A step can be defined so that PROC SUMMARY generates all the summarization data needed by subsequent steps.

Consider the following sequence of DATA and PROC steps with respect to using multiple invocations of PROC SUMMARY to provide summarization data.

```
PROC SUMMARY DATA=INPUT NWAY;  
CLASS X Y;  
VAR A;  
OUTPUT OUT=SUMMARY;  
RUN;
```

```
DATA REPORT;  
SET SUMMARY;  
...  
RUN;
```

```
PROC SUMMARY DATA=INPUT NWAY;  
CLASS X Y;  
VAR B;  
OUTPUT OUT=SUMMARY;  
RUN;
```

```
PROC CHART DATA=SUMMARY;  
VBAR B;  
RUN;
```

Rather than using two separate PROC SUMMARY steps to provide summarization information, both steps can be combined to generate the summarization in a single PROC SUMMARY step.

```
PROC SUMMARY DATA=INPUT NWAY;  
CLASS X Y;  
VAR A B;  
OUTPUT OUT=SUMMARY;  
RUN;
```

```
DATA REPORT;  
SET SUMMARY (DROP=B);  
...  
RUN;
```

```
PROC CHART DATA=SUMMARY;  
VBAR B;  
RUN;
```

Placing data into a SAS data set is most efficient when the minimal amount of information is saved. Analyze the use of variables and remove those that are not needed in further processing in subsequent job steps. Effective use of the DROP and KEEP statements can improve the performance of data access and data storage management.

Data Processing

When processing SAS data sets, it is important to reduce the number of variables that have to be processed. By reducing the set of variables read from a SAS data set, reductions in the amount of CPU and memory resources utilization can be gained.

When reading data from external files using the SAS DATA step INFILE statement, effective use of the INPUT statement can provide reductions in CPU utilization. Minimize the number of fields processed by the INPUT statement; this is most especially true with fixed column input records. It is not necessary to read all INPUT fields if only one or two are required for processing. This technique may not be possible with list style input where fields are separated by delimiting characters, such as the space character.

Assume the following sample data are contained in the external file reference DATA :

```
12APPLE
15FIG
29PEAR
```

```
/* Read the data */
DATA;
INFILE DATA;
INPUT A 2. B $CHAR5. C 12.;
KEEP A;
RUN;
```

In the example above, notice that only variable A is saved in the output data set. The program can be modified so as to eliminate the INPUT operation for the variable B.

```
DATA;
INFILE DATA;
INPUT A 2.;
RUN;
```

A similar technique can also be applied to applications that process data in which the data records are made up of multiple record types by reading only the minimal amount of data to determine the type of record that is to be processed.

```
DATA;
INFILE DATA;
INPUT TYPE 2.;
IF TYPE = 1 THEN
  INPUT NAME $CHAR20. SCORE 5.;
ELSE
IF TYPE = 2 THEN
  INPUT TEAM $CHAR20. RANK 5.;
...
RUN;
```

Data Subsetting: IF and WHERE Statements

Reducing the number of observations that a step uses for processing can be a very effective means of increasing the performance of a job step. Depending upon the style of data subsetting incorporated in a SAS job, the combining of multiple data subsetting steps can be very beneficial. The use of the DATA step subsetting IF and WHERE statements can reduce the

amount of data processed within a single job step.

There is a very important distinction between the subsetting IF and the WHERE statement that needs to be noted. The subsetting IF selects data based upon the evaluation of the IF expression, after an observation has been completely read into the DATA step program, whereas the WHERE statement only surfaces observations to the DATA step program that meet the conditions of the WHERE expression. Selecting one technique over another, with respect to performance considerations, is very difficult due to the unknown nature of the data and its organization. Generally, the DATA step subsetting IF is just as efficient as the WHERE statement; however, the WHERE statement efficiency impact can be better realized by the use of indexes on the input data sets.

Due to the levels in the DATA step environment at which the subsetting IF and WHERE statements operate, differences in the results of BY processing should be noted. All data are provided to a DATA step when the subsetting IF is used, and the setting of the BY variables FIRST. and LAST. values is based upon all observations contained in the data set. With the WHERE statement, the setting of the BY variables FIRST. and LAST. values is based upon the observations surfaced by the WHERE statement evaluation process.

Indexes

As suggested in the discussion of the WHERE statement, the definition and use of indexes on SAS data sets can provide significant performance improvements in the overall execution of a SAS data processing job. Indexes are used to optimize the retrieval of data from SAS data sets when a step uses the BY statement to group data together or when WHERE statements or WHERE expressions are used to subset the amount of data processed from a SAS data set. The DATA step subsetting IF currently cannot use the information contained in indexes to assist its analysis. SQL views use indexes to optimize the processing of data table joins or interactions with other views.

Arithmetic and Logic

The evaluation of expressions is the most widely used construct provided by the SAS DATA step language. When performing expression evaluation it is important that evaluation of common expressions be minimized as much as possible. Common expression evaluation can waste valuable amounts of CPU time.

Similarly, the use of constants rather than constant expressions in subsequent expression evaluation can also be beneficial to the performance of a DATA step program. Use of the MACRO language and MACRO variables to propagate constants in place of constant expressions is recommended. Condense constant expression as much as possible, since the DATA step does perform limited constant expression evaluation between constants of an arithmetic operation, though this capability is currently limited until more specialized constant expression evaluation is supported.

For many years, implicit conversion operations have not been as efficient during expression evaluation as one would expect. Over time, many programmers have found that the use of the PUT/INPUT function was much more efficient at data type conversion than that provided in normal expression evaluation. Recent changes to the DATA step environment have made the PUT/INPUT technique less efficient than the implicit conversion due to code generation optimization techniques. Current programs still function as they do today, but as new applications are developed, keep this information in mind.

```
/* Implied Type Conversions */
```

```
DATA;
LENGTH CKEY $ 10;
RETAIN NKEY 0;
CKEY = '1234567890';
NKEY = CKEY;
CKEY = NKEY;
RUN;
```

```
/* Explicit Type Conversion using */
/* INPUT and PUT functions */
```

```
DATA;
LENGTH CKEY $ 10;
RETAIN NKEY 0;
CKEY = '1234567890';
NKEY = INPUT(CKEY,10.);
CKEY = PUT(NKEY,10.);
RUN;
```

The evaluation of expressions containing missing values is generally very expensive due to the requirements of detecting and reporting missing value exceptions. Some host platforms must check each arithmetic operand for missing values in the generated code, while other host platforms are assisted by the floating point hardware to detect and signal the SAS System, thereby eliminating explicit missing value checks in the generated code. In either case, the propagation of missing values during expression evaluation is very expensive over all. As a programmer, the order in which you specify the evaluation of an expression can have a significant impact upon the effects of missing value propagation. By analyzing the variables involved in expressions that have a high probability of incurring missing value propagation, one can reorder the expression so as to move those variables to the end of the problematic expression. By doing so, the effects of missing value propagation are limited to a smaller set of expression evaluation operations.

Assume that the data set DATA contains the variables A, B, and C and that missing values are present only in variable A.

```
DATA;
SET DATA;
SUM = A + B + C;
RUN;
```

In the example above, the addition of variables A and B generates a missing value whenever variable A is missing. This missing value is propagated into the addition of C to complete the expression evaluation. The evaluation causes two missing value exceptions to occur, requiring the DATA step to maintain information about the location of each exception.

By moving variable A later in the evaluation of the expression, the number of missing value exceptions is reduced by one occurrence. Given the overhead required to process a missing value exception, substantial savings in CPU utilization can be realized.

```
DATA;
SET DATA;
SUM = B + C + A;
RUN;
```

As with missing value propagation, expression evaluation ordering can have a significant impact upon the performance of evaluation within conditional statement expressions. Recent changes to the DATA step compiler have improved the efficiency of evaluation of conditional statements like the IF and DO/WHEN/SELECT

statements. These improvements are due to the utilization of a technique, Boolean short circuiting, that reduces the number of AND and OR expressions that need to be evaluated to determine the TRUE or FALSE conditions present in these two statements. Ordering the conditions in descending probability of occurrence eliminates the need for further evaluation of expressions within conditional statements by detecting that the complete conditional expression is TRUE or FALSE at the earliest point of evaluation.

Assume the following frequencies at which the variables or expressions A, B, and C are non zero values.

- Freq(A) = 10
- Freq(B) = 513
- Freq(C) = 3004

```
DATA;
IF A | B | C THEN ...
...
RUN;
```

With the example above, more time is spent evaluating the conditions for A and B when the data have a higher probability that the C variable or expression has a higher number of occurrences present in the data. By changing the order of the IF condition so that the order of evaluation is based upon the expected occurrence of the data, you can improve overall performance of an executing DATA step program.

```
DATA;
IF C | B | A THEN ...
...
RUN;
```

A similar programming construct can be used for conditional evaluation when mutually exclusive conditions exist in a SAS DATA step program. When only one condition can be true for a given observation, use a series of IF-THEN/ELSE statements or a DO SELECT group rather than a series of IF-THEN statements without ELSE statements.

```
DATA;
SET DATA;
IF X = 1 THEN ...
IF X = 2 THEN ...
IF X = 3 THEN ...
...
RUN;
```

```
DATA;
SET DATA;
IF X = 1 THEN ...
ELSE
IF X = 2 THEN ...
ELSE
IF X = 3 THEN ...
...
RUN;
```

Rather than testing the variable X in each IF statement, it is more efficient to change the flow of control of the IF statement evaluation to take advantage of the mutually exclusive conditions. By simply using the ELSE statement with the IF/THEN statement, additional reductions in CPU utilization can be realized.

```
DATA;
SET DATA;
IF X = 1 THEN ...
ELSE
IF X = 2 THEN ...
...
RUN;
```

The effective use of the DO loop for many programming operations is extremely important. Too much evaluation within a loop can be inefficient due to the waste of valuable system resources. Review the contents of a looping construct to move constant expressions or expressions not based upon the loop control variables, whether implicitly or explicitly specified, to outside of the loop control. Equally important is to order the evaluation of IF statements in terms of descending probability of use within a DO loop.

```
DATA;
SET DATA;
DO I = 1 TO N;
  X = N * N;
  C = X * SORT(I);
  ...
END;
RUN;
```

In the example above, the calculation of the variable X is not influenced by any variable changes during the DO loop execution. By moving this calculation outside of the DO loop, additional CPU savings can be realized.

```
DATA;
SET DATA;
X = N * N;
DO I = 1 TO N;
  C = X * SORT(I);
  ...
END;
RUN;
```

Data Compression

If disk storage space is a major concern, the use of data compression techniques can provide a solution. Shortening of the length of data stored can have a great impact upon the incurred storage costs. The use of the LENGTH statement can be the easiest technique of all. Using the LENGTH statement with character variables is simpler than using the LENGTH statement to specify short lengths for numeric variables because of the host platform floating point representation and the desired precision of the result. The restrictions of the various host platforms is discussed in the "LENGTH Statement" in Chapter 9, "SAS Language Statements," in *SAS Language: Reference, Version 6, First Edition*.

Using PROC FORMAT formats and informats to encode shorter values for data can be a very effective technique for data compression. A similar, but simpler technique, is to store numerics as characters. Keep in mind that there is a break-even point where the size of the formatted numeric value will be longer than that of a numeric variable with a short length in terms of physical storage requirements.

With the introduction of Version 6 of the SAS System, the BASE I/O engine provides the capability for data compression. The data compression technique employed uses run-length compression to reduce consecutive runs of bytes into a smaller number of data bytes within an observation. This data compression technique, as do most commonly used methods, requires additional CPU and

memory resources to accomplish the management of the compressed data.

Views

The use of SAS views can improve the efficiency of a SAS step but, more importantly, views can increase the available programming resources by allowing the views to be reused in multiple applications. Views can be coded once and then used in many steps within a single SAS job or by many other SAS jobs. There are three primary types of views provided by the SAS System: SQL, DATA step, and SAS/ACCESS views. For this discussion, only the SQL and DATA step views are considered.

A SQL view is generally characterized by its ease of use and expressive power of the underlying SQL language upon many data organization constructions. Many SQL queries can be accomplished using the SAS DATA step, but these queries are likely to be more complex or contain a large set of distinct operations that are more efficiently processed by the SQL environment than by using multiple steps in the DATA step language.

DATA step views fit well into the SAS System by allowing programmers to encapsulate existing DATA step code to function like SAS data sets. In most circumstances, the use of DATA step views reduces the creation of redundant data by providing the data upon demand. The importance of DATA step views is that the view inherits the full power of the SAS data step language to retrieve data from other SAS data sets, SAS views, external files, or to dynamically generate it. Whether the performance characteristics of DATA step views is the same or greater than that of other SAS data views is highly dependent upon the exact nature of the step reading the view, the options specified for the step and the methods in which the referenced view obtains or generates its data. All WHERE expression evaluation is handled in the DATA step view rather than in the SAS I/O Engine Supervisor, thereby reducing the overhead of data movement between the DATA step view and the SAS I/O Engine Supervisor. Improvements to the management of the views execution environment have also been made to reduce the impact upon system resources.

Stored Programs

The use of stored programs can also provide benefits to application developers by eliminating compile time overhead of the DATA step program, because the stored program does not have to be recompiled each time it is invoked in a SAS job. Stored programs can be compiled outside of the environment where it is used, and a single stored program can be used by many SAS data processing applications. As a side note, DATA step stored programs are used to implement the support for DATA step views.

Arrays

Effective use of DATA step arrays can provide a small improvement in overall efficiency when they are used in a DATA step program. A DATA step array is not a physical array of data but a logical mapping over a set of variables defined within the step. To assist the DATA step compiler in providing efficient support of array references, it is important that no two arrays share the same variables. The reason for this is the technique employed to gain access for reading and updating the values referenced by the array. When one or more variables is shared between arrays, more time is required to navigate to the actual storage locations of the data.

```
DATA;
ARRAY A A1-A5;
ARRAY AB A1-A5 B1-B5; /* Array contains elements */
                        /* from the array A */
...
RUN;
```

As with array element sharing, the same effects are encountered if the array contains both retained and unretained variables, or in the case of character variables if elements of the arrays are of a different length. Access costs to elements are increased, because all the elements cannot be contiguously allocated and must be spread out in memory, thereby eliminating contiguous element access for reading and updating.

```
DATA;
LENGTH A1-A5 $ 10 B1-B5 $20;
RETAIN B1-B5;
ARRAY AB A1-A5 B1-B5; /* Array contains elements */
                        /* with different sizes and */
                        /* mixed retained and */
                        /* unretained variables */
...
RUN;
```

If the purpose of arrays is to calculate information, and the result of the calculation eliminates the need for the values of the array elements to be saved, the use of `_TEMPORARY_` style arrays is recommended. `_TEMPORARY_` arrays do not require variables to be defined or managed but provide the data storage to contain the array elements during array accessing and evaluation.

It is highly recommended to reduce or eliminate the use of the *array of arrays* construct that many older SAS programs may contain. An example of the *array of arrays* construct is as follows:

```
DATA;
  ARRAY A(J) A1-A10;
  ARRAY B(J) B1-B10;
  ARRAY AB(I) A B;
  DO I = 1 TO 2;
    DO J = 1 TO 10;
      AB = I * J;
    END;
  END;
```

This old form of array processing is very expensive to use due to the requirements of the implementation to support it. Access to the array elements cannot be generated as in-line code but must use external function calls to navigate through complex data structures to gain access to the underlying data.

The array of arrays example can be converted to use explicitly subscripted arrays to provide better performance characteristics.

```
DATA;
ARRAY AB(2,10) AB1-AB20; or ARRAY AB(2,10);
DO I = 1 TO 2;
  DO J = 1 TO 10;
    AB(2,10) = I * J;
  END;
END;
```

SAS CONTROLLED EFFICIENCIES

In this section, information is presented describing the involvement of the SAS System components that assist in the efficiency of SAS program execution. The SAS System has been modified heavily since its first introduction to be as efficient as

possible, while at the same time being able to provide a very robust set of features and functionalities.

SAS Data Sets

The I/O Engine Supervisor and BASE engines have been modified to provide better management of the data storage structures, thereby reducing the overhead necessary to maintain user data. In pre-Version 6 systems, the overhead to manage a single observation was 4 bytes. When the initial Version 6 of the SAS System was introduced, this overhead increased by a factor of 3 to 12 bytes. With this substantial overhead increase, significant performance problems in terms of CPU, I/O, and disk storage were encountered and reported by many SAS sites. With Release 6.07 of the SAS System, the new lean file format for SAS data sets was introduced, significantly reducing the management overhead of each observation to a single bit.

Additional algorithmic changes have also made significant strides in the reduction of CPU and I/O resource utilization when accessing data sets sequentially. The technique of fast pathing reduces the CPU overhead to gain access to an observation within a SAS data set. With fast pathing, access to the observation is handled by reading a page from the data set and returning the first observation out of the data page, with subsequent observation accesses just incrementing pointers into the data page until the number of observations contained within the page is exhausted. Additional observations are retrieved by reading the next data page and continuing the pointer manipulation process.

Changes to the compression algorithms and to the number of buffers and their corresponding buffer sizes also contribute to greater efficiency of SAS data set I/O. Better choices for these values were determined and installed as part of the BASE I/O engine with the introduction of Release 6.07 of the SAS System. The algorithm used for observation compression was optimized to reduce the CPU resources utilized during both the compression and decompression operations.

External File I/O

As with the optimization of the SAS data sets, the fast pathing algorithm has been applied to the access of data to and from external files. In fact, the fast pathing technique was first introduced in the external file I/O support and later incorporated into the SAS I/O engine architecture. This support is most noticeable when using the DATA step FILE and INFILE statements to access data from external files.

The INPUT and PUT statement support for column I/O operations is now generated in-line as much as possible to reduce CPU resource utilization.

```
DATA;
INPUT @10 X 2. @X Y 5. +Y Z 3.;
...
RUN;
```

Informats, Formats, Functions, and Subroutines

A great deal of time has been spent in analyzing the performance of these system level components in an effort to increase their efficiency. Special attention was given to the conversion type formats and informats, due to their use by other formats and informats within the system. Over 100 formats and informats were improved through the incorporation of changes identified during the performance analysis process. On average, between 300% to 400% in performance improvement gains have been measured.

The use of functions by SAS application developers is the mainstay of many applications. Special attention has been given to the most commonly used functions provided by the SAS System. Modifications to the identified functions and subroutines realized a net gain of improvement to over 95 functions when compared to Release 6.06 of the SAS System and a net gain to over 50 functions when compared to Release 5.18 of the SAS System.

Missing Value Detection

As previously discussed in the user controlled efficiency section, the ordering of missing value propagation within a SAS expression can have a significant impact upon the utilization of CPU resources. Additional system level changes have assisted in improving the detection and management of missing value detection, propagation, and reporting. The DATA step algorithm for missing value tracking has been greatly improved by generating different in-line code information to provide immediate tracking information rather than using search algorithms to determine the location of missing value exceptions. These changes have improved the processing of missing value exceptions by 50%. The use of the ERRORBLOCK data step option can also provide performance gains by letting the DATA step track missing value exceptions on a complete expression boundary rather than at the subexpression boundary within an expression.

IF Statements - Boolean Expression

The DATA step has also seen significant improvements in the area of Boolean expression handling by incorporating the optimization technique of Boolean short circuiting. This change impacts applications when multiple conditional expressions are found together in an IF or DO/WHEN/SELECT statement where OR or AND operators are used. An example of such a construct is shown in the following statement:

```
IF VAR1 OR VAR2 OR VAR3 OR VAR4 THEN
  PUT 'One of these is true';
```

The idea behind Boolean short circuiting is to eliminate redundant expression evaluation as early as possible during the evaluation of the IF condition. The programmer can improve the performance of an application that contains this style of IF conditional logic by ensuring the ordering of the AND and OR expressions proceeds from a high to low probability of occurrence.

Subexpressions contained within the IF conditional logic that exhibit side effects prevent the Boolean circuiting of conditional logic. A side effect, with respect to SAS expressions, is characterized by expression evaluations that must be evaluated completely from one iteration to the next and the evaluation cannot be short-circuited. In the SAS DATA step it is expected that specific functions within an expression are always executed regardless of the potential for optimization. Specifically, the random number functions and system level functions are designated to be functions that produce side effects.

```
IF x > 5 AND RANUNI(0) < .5 THEN ...
```

The RANUNI(0) function creates a side effect that requires it to be executed every time to ensure that the random number stream is generated in the same order as expected.

Code Generation Optimizations

Code generation optimizations of the DATA step compiler are continually identified and placed into service. These changes provide for smaller generated machine code programs or

compiled programs and better utilization of machine resources.

Explicitly subscripted arrays within the DATA step have also been optimized. This optimization is especially apparent when the subscript is a constant value. In these cases, the array references are pre-evaluated when the DATA step program is compiled, yielding CPU use that is roughly one-third what was used in Release 5.18 of the SAS System. Also, new with Version 6 of the SAS System is DATA step support for multidimensional explicitly subscripted arrays.

Use of implicitly subscripted arrays is not recommended as their performance characteristics were not improved and they tend to use much more CPU resources than explicitly subscripted arrays.

Even simple DO loops have been optimized. All three forms of loops (iterative, WHILE, UNTIL) exhibit improvements in CPU utilization. Although it would require an enormous number of loop iterations to achieve a noticeable change in an application, improvements to this area exemplify the effort given to decreasing the CPU requirements within the DATA step.

In many cases, bit string testing is now generated by the DATA step compiler rather than using a function call to do the test. This change applies to single character tests as well as numeric value tests. Unfortunately, the use of bit string testing on character strings with a length greater than one still exhibits a degradation in performance.

In the mainframe environment, a technique that was used to enhance the performance of the DATA step was the use of routines that do not require dynamic save areas (DSAs). These routines are more efficient because they do not require save areas to be acquired when they start or freed when they terminate. In order to make some routines not require a DSA, they often had to be modified such that called routines were relocated into higher level routines and, in some cases, use of automatic storage had to be reduced. Source code modifications were also made in some cases to bring small functions in-line rather than calling them at all. Also, in the mainframe environment, many of the specialized functions were optimized. These improvements came in the form of reduced path length for the most common uses, better use of the available instruction set, and removal of duplicate validation checks.

There have been additional optimizations to the DATA step compiler when actual machine instructions are generated:

- Changes in code generation produces smaller programs.
- Less CPU overhead is incurred during code generation.
- Compile time memory was significantly reduced when internal label descriptors were used.
- SUM, LAG, and DIF functions are now generated in-line.
- Numeric-to-character and character-to-numeric conversions are now generated in-line. A performance penalty is incurred when using PUT/INPUT functions to perform type conversions.

In future releases, more and more DATA step operations will be performed in the generated machine code.

ADDITIONAL PERFORMANCE INFORMATION

There are also numerous tuning options (both SAS System options and DATA step options) that can be used to adjust the performance characteristics of the DATA step. The use of the DATA step options should only be specified in completely tested production applications so as to eliminate the possibility that a coding or logic problem in an application program does not cause internal data corruption within unprotected areas of the program execution environment. A brief list of some of these options follows:

System Options

BUFNO=	Changes the number of buffers allocated for a given data set. Primarily used for sequential I/O to accommodate look-ahead I/O operations.
BUFOBS=	Determines the SAS data set buffer size based upon the number of observations requested to be placed within a single buffer.
BUFSIZE=	Set the buffer size for a given SAS data set.
COMPRESS=	Specifies whether observations written to the data set are compressed. <i>Understand the costs before attempting to use this option.</i>
REUSE=	Reuse space from deleted observations in a SAS data set.

DATA step options

ERRORBLOCK	Track expression errors across the complete expression rather than at the subexpression level.
NOMISS	Do not generate code to check for missing values. On machine architectures where missing value detection is handled by the hardware, any missing value exception will cause anabend of the DATA step program executing.
NOSTMTID	Do not generate code or support memory to keep track of error locations.
NOSUBCHK	Do not generate code to detect array subscripting errors. Programmers need to be very careful when using this option in that undetected subscripting operations could overwrite memory outside of the array.

DATA step options can be specified by following the DATA statement with a / (forward slash) followed by the option names, as for example:

```
DATA TESTOPT/NOSTMTID NOSUBCHK;
```

CONCLUSION

This paper has briefly described some techniques for the SAS programmer to use or that the SAS System uses to assist in the development of efficient DATA step programs. Information was presented that a programmer can use to develop DATA step programs to obtain the most optimal performance characteristics possible. Information was also presented on how the SAS System provides additional assistance to ensure that those programs can be executed in the most efficient manner. In the quest to obtain

the highest programming efficiency and its desired effect — performance — both the programmer and the SAS System must work together to achieve this ultimate goal.

A final note: SAS Institute is dedicated to producing software products that are of the highest quality and efficiency. If you are aware of specific problems with performance of the SAS System, please do not hesitate to bring this to our attention so that appropriate action can be taken for future product releases.

REFERENCES

SAS Institute Inc.(1991), *SAS® Language: Reference, Version 6, First Edition.*, Cary NC: SAS Institute Inc.

SAS Institute Inc.(1990), *SAS® Programming Tips: A Guide to Efficient SAS® Processing.*, Cary NC: SAS Institute Inc.

Herbert, Pat M., Polzin, Jeffrey A.(1992). "DATA Step Performance", *Proceedings from the Seventeenth Annual SAS® Users Group International Conference*, 956 - 959.

SAS and SAS/ACCESS software are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries.
* Indicates USA registration.