Paper 012-2009

Maximum SAS®: Analyzing and Increasing Performance

Myron A. Chandler, SAS Institute Inc., Cary, NC

ABSTRACT

Application performance is a topic that concerns every developer at one time or another as it is arguably the second most important measurement of any software system's success directly behind correctness. When attempting to improve the performance of an application, the first and often most difficult task is to identify the elements that are most responsible for the delays in time.

This paper profiles how a real-world, poorly performing software testing system was analyzed using SAS® and the performance gains that were achieved by rewriting data queries, by restructuring code blocks and removing unnecessary code, and by altering SAS® configuration options.

INTRODUCTION

This paper describes a process that was used to analyze and improve the testing portion of a mission-critical application in an effort to make the production application more efficient and more reliable.

What should be taken away from this paper is an understanding that there are many areas to explore when you attempt to improve application performance and that nothing should be overlooked. Complex code is usually assumed to be the culprit. As detailed later, even the most uncomplicated code can be a large contributor to an application's less-than-desirable performance.

At the time of this writing, the profiled application existed as a SAS Release 8.2 software system. The discussion of its enhancements will apply to multiple versions. Although some portions of this paper are generally applicable to all software developers, the content is best consumed by SAS programmers who have a knowledge of SAS/AF® software, the SQL procedure, Base SAS® software, and general SAS system options. A working knowledge of the SAS/AF SCL Performance Analyzer is helpful but not required.

PROBLEM

The Testing Harness is a collection of Perl and SAS scripts that are used within the MIS department of SAS headquarters to automate the creation of setinit and SAS installation data files by mimicking existing SAS customer installation data and running the data through a license-creation facility. After each program executes, a set of hundreds or thousands of text files are created that can be manually inspected for correctness and programmatically compared to a pre-existing set of benchmark files to detect changes. Generally, after developing code in a playpen area, a developer measures the impact of the changes by running the Testing Harness and comparing the playpen code output with production environment code output.

The best case scenario is that for every change a developer makes, the Testing Harness is run for every SAS installation, for every current SAS customer. Because there are thousands of SAS installation sites, the amount of time required to use the Testing Harness with the complete set of data is formidable. That delay is extremely prohibitive when trying to react to customers in a fast-moving, production environment.

Currently, developers define a subset of data to use in each of a suite of test cases, and then they run the test case that covers the type of changes that were made. These smaller test cases can usually be run in less than one hour and provide a sufficient level of confidence before moving code into production. While not common, there have been issues that went undetected by the smaller test cases and have reached the production environment. To discover these types of issues before reaching production, I was assigned the task of speeding up the Testing Harness so that significantly larger amounts of data could be processed in the same amount of time.

STRATEGY

I immediately came up with three options to accomplish my goal: upgrade the hardware, upgrade the software, and rewrite the slow parts of the application. At that time I had the least amount of experience with the application of anyone on the development team, so the first two options were the most appealing.

The hardware upgrade took some time to get into place (make a request, get it approved, and so on). After getting that process started, I began looking at upgrading SAS. Unfortunately, upgrading the application to run on SAS 9 would require many more developer resources than were available due to other significant efforts already underway and the large number of application dependencies in the production environment.

My last option was to rewrite the slow parts of the application. To do this, I first needed to profile the application and identify the sections of the code that were the slowest, but also the least risky sections to change. I integrated the SAS/AF SCL Performance Analyzer into the application startup sequence to profile the entire application on a medium-sized program run. I then saved the performance data, analyzed the results, rewrote slow sections of code, and repeated the process.

METHOD

These are the specific steps that I followed:

1. Modify the startup sequence of the Testing Harness to include the SAS/AF SCL Performance Analyzer. Each test had its own autoexec.sas file that used the AF command to call the initial SAS program; since I wanted the entire application profiled, I modified this AF command to include the profiler option. Specifically, I changed the command from

   ```
   af c=library.catalog.startup.scl
   ```

   to:

   ```
   af c=library.catalog.startup.scl SCLPROF=YES
   ```

   (It is also possible to profile only specific sections of code if you are running the program interactively -- see the SAS documentation for more details.)
2. After the test finished running, the interface to the SAS/AF SCL Performance Analyzer displayed. I then clicked through the individual sections and saved the resulting data sets to a pre-defined SAS library.
3. I then did a Windows copy of the data from the library on the test machine to a library on my local machine. I created folders for each major set of code revisions. I ended up with eight folders, each with the data sets from that particular run. It's also good idea to save a copy of the SAS/AF catalog that goes with each run because you can click from the analyzer's interface directly into the code and see the line-by-line statistics.
4. I stepped through the analyzer results and kept track of the SCL programs and line numbers that took the most time to execute.
5. Based on the notes made in the previous step, I searched through SAS documentation, white papers, the SAS Companion for Windows, and anything else I could find on the SAS intranet about performance relating to the statements in question.
6. I made code modifications based on my research and reran the test.

RESULTS

The first test that I ran using unmodified code on a subset of the data completed in about 40 minutes. After I followed the process above and modified the application code, I reduced the execution time by almost 50 percent; the modified code completed the same test with the same data in just under 22 minutes.

These are the changes I made:

- **USE I/O EFFICIENTLY**
  In the original code, there was essentially a one-to-one correspondence between the number of FPUT and FWRITE functions used. The FPUT function was called 742,648 times and took a total of 2.19 seconds to execute. The FWRITE function was called 748,912 times and took a total of 139.50 seconds to execute. That is an expected outcome since FPUT is a memory-based function and the FWRITE function is a much slower I/O-based function.

To reduce the I/O, I increased size of the output buffer from the default 256 bytes to 64K (using the BLKSIZE and LRECL options for the FILENAME statement), increased the consecutive calls to the FPUT statement, and then decreased the number of calls to the FWRITE function (this required adding a newline character to the end of each FPUT statement). The actual program code is too extensive to list here. As an example, this arbitrary block of code could be rewritten:

```
f1 = ' ';
rc = filename(f1,'c:\temp\file1.txt');
fid = fopen(f1,'o');
do i = 1 to 100;
  rc = fput(fid, 'some text');
  rc = fwrite(fid);
end;
```

The following code block executes faster as long as the amount of text in the loop does not exceed the number of bytes defined in the FILENAME function. Notice that the newline character is appended to the end of the text in the FPUT function:

```
f1 = ' ';
rc = filename(f1,'c:\temp\file1.txt', 'DISK', 'BLKSIZE=65536 LRECL=65536');
fid = fopen(f1, 'o');
do i = 1 to 100;
  rc = fput(fid, 'some text'||byte(13));
end;
rc = fwrite(fid);
```

The result was that the FPUT function was called 1,134,090 times but still only took a total 2.52 seconds to execute. (See Figure 1.) The FWRITE function was called 354,386 times and took 67.30 seconds to execute. The total time to execute that set of output statements was reduced from about 141 seconds to about 70 seconds.

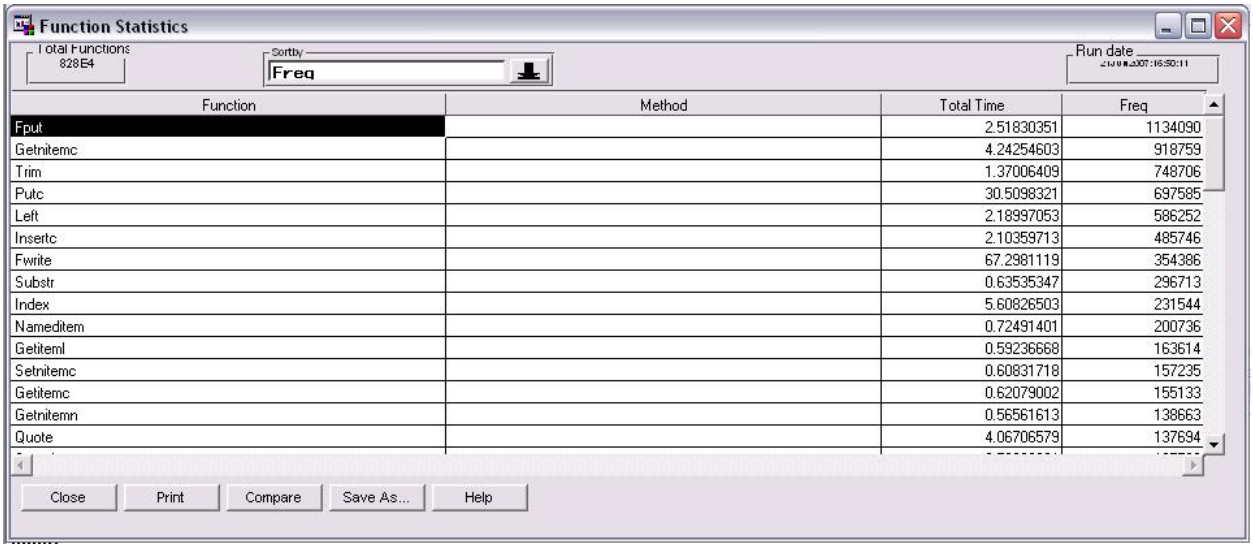

Figure 1. Sample Output from the SAS Dynamic Performance Analyzer – Examining FPUT Usage

- **REWRITE SQL INTO SIMPLE BLOCKS WHEN POSSIBLE**
  The lesson here is that it is good to know more than one way to write the same query. I confess to not knowing all the details of PROC SQL execution algorithms, but I do know SQL and can usually find several ways to write the same query. Sometimes longer, simpler blocks of SQL execute faster than shorter, indirect SQL blocks. For example, this correlated query took 23.32 seconds to execute:

```
delete * from work.release wr
   where prodid||release in (select prodid||release from product.release pr
                     where wr.prodid = pr.prodid and pr.relstat = 'X');
```

  I rewrote the query, and the new version executed in 2.28 seconds:

```
create table work.toRemove as
  select pr.prodid||pr.release as prodrel
   from product.release pr, work.release wr
   where wr.prodid = pr.prodid and pr.relstat = 'X';

  delete from work.release
   where prodid||release in (select prodrel from work.toRemove);
```

- **MINIMIZE FUNCTION USAGE INSIDE LOOPS**
  Whenever function or method calls inside a loop can be replaced by a constant or eliminated altogether, the resulting code will be faster. This code executed 1900 times for a total of 50 seconds:

```
   submit continue;
   data _null_;
   infile "&tmp_mediajob" length=linelen;
   file "&mediajob";
   input @1 string $varying80. linelen;
   if index(upcase(string),'="";') > 0 then do;
     i = index(upcase(string),'="";');
     string = substr(string,1,i-1)||'=" ";';
   end;
     put string $char80.;
   run;
   endsubmit;
```

  The rewritten code runs the same number of times but is just over ten seconds faster. Instead of using the `INDEX` function multiple times to locate a substring and the `SUBSTR` function to replace it with another substring, the new code uses a single call to the `TRANWRD` function to achieve the same results.

```
   submit continue;
   data _null_;
   length outString $80;
   infile "&tmp_mediajob" length=linelen;
   file "&mediajob";
   input @1 string $varying80. linelen;
   outString = tranwrd(string, '="";', '=" ";');
   put outString $char80.;
   run;
   endsubmit;
```

- **ANALYZE FILENAME STATEMENTS**
  It was surprising to me to see the amount of time the FILENAME statement takes to execute, both to initially assign a file reference and then to subsequently deassign it. This statement to assign a file reference ran 1900 times and took over 11 seconds to execute while other memory-based functions in the same sequence took less than one half of one second (the BLKSIZE and LRECL options increase the default output buffer size from 256 bytes to 64K):

```
rc=filename('FS','c:\temp\file1.txt', 'DISK', 'BLKSIZE=65536 LRECL=65536');
```

Even more surprising is that de-assigning the same file reference 1900 times took 41 seconds:

```
rc=filename('FS', '');
```

There is no way around assigning the file reference (which is subsequently used in the FDELETE statement or other statements), but I commented out the code that deassigns the file reference. Why? In my case, this is a program that is used for testing on a non-production server, and I was willing to take the risk of not clearing the file reference to save the 41 seconds. Obviously, I would not recommend doing this in a production application.

- **REDUCE METHOD/FUNCTION CALLS**

This code enhancement was basically recognizing one set of statements as being equivalent to a hopefully faster set of statements. This block of statements executed 3000 times, and each line took 47 seconds for a total of 282 seconds:

```
_self_.delete_File(infile1);
_self_.delete_File(infile2);
_self_.delete_File(outfile1);
_self_.delete_File(outfile2);
_self_.delete_File(aliasin);
_self_.delete_File(aliasout);
```

I wrote a new DELETE_FILESET method for this object that uses the SYSTASK statement to execute a DOS-based deletion of multiple files instead of using the FDELETE statement to delete one file at a time. The line below called the new method 3000 times and took just under 47 seconds to delete the same number of files.

Instead of making six individual method calls with one filename each, the new code made one call with a string of concatenated filenames. Internally the new DELETE_FILESET method passed those filenames directly to the operating system to perform one delete operation:

```
_self_.delete_fileset('"'||infile1||'" "'||
                      infile2||'" "'||
                      outfile1||'" "'||
                      outfile2||'" "'||
                      aliasin||'" "'||
                      aliasout||'"');
```

Now look at the bulk of the code for the original DELETE_FILE method that appears below. It was originally called 24,578 times and took 271 seconds to execute:

```
if fileexist(delfile) then do;
   fname = ' ';
   rc=filename(fname,delfile);
   if (rc=0) then
      rc=fdelete(fname);
   rc=filename(fname,'');
end;
```

After rewriting the code, it executed 6432 times in 25 seconds. It was called significantly fewer times because many of the original method calls were redirected to use the new DELETE_FILESET method. Even if that output is multiplied by a factor of four to make it comparable to the original method usage, it would have conceivably executed over 25,000 times in 100 seconds, which is over 60 percent faster.

Notice that I removed the FILEEXIST function. It takes a long time to execute and the original function does no significant processing based on the function return code, the return code was always true and the code

block always executed. Although good programming practice might say check first and clean up afterwards, practically speaking, there is no reason not to remove those statements in this case:

```
fname = ' ';
rc=filename(fname,delfile);
if (rc=0) then rc=fdelete(fname);
```

- **EXPERIMENT WITH SAS CONFIGURATION OPTIONS**
  I reviewed the system options in the SAS Companion for Windows to get a list of potentially performance enhancing options. These are the ones that I updated or added:

```
SORTSIZE (limits memory available to SORT procedure; default is 2M)
MVARSIZE (maximum size for macro variables; default is 4K)
MSYMTABMAX (maximum size for macro symbol table; default is 4M)
BUFNO (number of buffers used for SAS I/O files; default is 1)
BUFSIZE (permanent buffer size for a SAS file; default is system)
```

  There are also a few Extended Server Memory Architecture options that were unavailable on the Windows server being used at the time because it did not have enough physical memory. These options could have had a dramatic effect on performance by caching entire data sets in memory:

```
MEMLIB (process work library in ESMA memory)
MEMCACHE 4 (use ESMA as file cache; default is 0-do not use)
```

CONCLUSION

This paper has presented a process that can be used with multiple versions of SAS to profile and improve application performance by using the SAS/AF SCL Performance Analyzer. The specific code modifications are representative of a small set of changes that could be applied to similar SAS programs.

In general, no area of an application should be overlooked during the analysis process. In this particular application, execution time was reduced by the following methods:

- using I/O statements more efficiently (using memory-based functions, larger memory buffers, and so on)
- rewriting SQL into simple (sometimes longer) blocks of code
- reducing function usage inside iterative loops
- analyze the FILENAME function usage
- reducing repetitive method calls (by increasing method parameters)
- experimenting with SAS configuration options

It is also worth noting that with every new release of SAS, one should always inspect the documentation for performance improvements; some are transparent (for example, underlying changes to existing procedures), and some require code modification.

RECOMMENDED READING

Please see the online SAS product documentation for all general questions:
http://support.sas.com/documentation/index.html

For the specific topics covered in this paper, see these topics for your version of SAS:

- SAS/AF SCL Performance Analyzer—see the SAS/AF Reference Development Tools in Help
  http://support.sas.com/documentation/onlinedoc/af/index.html
- SAS System Options, Operating Environment Specific Information
  http://support.sas.com/documentation/onlinedoc/base/index.html#companion

- SAS Procedures, SAS/SQL
  http://support.sas.com/documentation/cdl/en/allprodsproc/61869/HTML/default/a003229772.htm

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name: Myron A. Chandler
Enterprise: SAS Institute, Inc.
Address: SAS Campus Drive, Rm E378
City, State ZIP: Cary, NC 27513
Work Phone: 919-531-5482
E-mail: myron.chandler@sas.com