"OK, Just don’t spend too much time on it ..."

Henry B. Winsor, MedFocus
Brian L. Bournival, MedFocus

ABSTRACT:

The thought of building a program testing system from scratch may seem daunting at first glance, but the actual necessary components are few and the time needed is probably much less than you think. This paper covers the basic components and design of a small performance testing system currently in use at a large company. The major points presented are: 1) selecting the experimental design, 2) identifying and stating the test subject programs as simply as possible, 3) reviewing the available usage information and determining what is relevant, 4) automating the experimentation process for improved accuracy, 5) collecting and verifying the data in a non-manual fashion, 6) verifying the validity of any assumptions after the experiment, and 7) using both numerical and graphical methods for data analysis and presentation. Although the system was written for a UNIX platform, most of the ideas are applicable to all systems, and the few UNIX-specific items are clearly identified.

INTRODUCTION:

The purpose of this paper is to present the concepts needed to construct a performance testing system. When most people think of a system, they start thinking in terms of man-years of resource commitment. The ideas presented here represent a system that has been installed in a department at a large company, built entirely of small pieces over three years, with a total time expended of less than 200 hours. The test subjects are SAS® programs, containing at a minimum the code fragments to be tested, although some test subjects have exceeded 10,000 lines. The testing is iterative and non-interactive, normally run overnight on a networked Hewlett-Packard workstation using HP-UX.

FORMAL TEST DESIGN:

As you might assume, the design is the most important part of the entire system. Because we intend to test relatively simple, independent modules of SAS code, we decided to use a form of randomized-block design called paired comparison. What this means in non-statisticalese is that we’re going to execute first one then the other of a pair of test subjects, match up the results, and determine if there is a significant difference at any of the steps, or in the overall program. It’s the simplest possible comparison, which results in much of the rest of the system being very uncomplicated as well.

There are some theoretical drawbacks to using such a simple design. One major problem is that you cannot test for differences between more than two test subjects, which inhibits your ability to test for possible interactions appearing between the code fragments being tested. It was decided that by using reasonable care in constructing the test subjects, we could eliminate any concern for interactions. The central idea is to eliminate every possible source of undesirable variation between your test subjects, keeping only the differences that you wish to test.

Another problem is of some importance. The degrees of freedom associated with the test results are reduced, because you cannot consider each program execution an independent observation. Instead, each pair of program executions, consisting of one each from the two test subjects, can be considered as an independent observation. The importance of independence cannot be overemphasized; it’s a primary assumption of the statistical tests utilized for testing for significant differences, and if you violate this condition, your analysis process will be flawed, and can give you inaccurate results.

So, there is an intentional tradeoff present in the design, that we may substitute
increased numbers of program executions to compensate for the reduction in degrees of freedom. This is possible because of our assumption that cpu time is normally freely available, and we can usually run as many iterations as necessary. If you are working at a place with less excess capacity than ours, you may have problems using this design.

It’s also important to do your selection of which executions’ results to match up before you actually execute the test subjects. We do this in the simplest possible method, by running each test subject in a sequence, so that records in the output data file are adjacent. This also allows us not to have to decide on and live with a set number of iterations to execute. If you want to match up executions from the two test subjects more randomly, you’re still going to have to do your assignment beforehand, so you’re going to have to know how many there will be in each group.

**TEST SUBJECTS:**
The test subjects are just SAS programs, specially constructed for testing. For example, if the item of difference is to test whether a WHERE clause in a SET statement is more efficient than using a WHERE statement in the body of the DATA step to extract a value and place it in a macro variable, you would have programs similar to the following two examples:

```sas
LIBNAME TEST ‘whatever’;
DATA _NULL_; SET TEST.DATA;
WHERE ITEM=COUNT;
   CALL SYMPUT(’CNT’,COUNT);
RUN;

versus:

LIBNAME TEST ‘whatever’;
DATA _NULL_; SET TEST.DATA
   (WHERE=(ITEM=COUNT));
   CALL SYMPUT(’CNT’,COUNT);
RUN;
```

We recommend that you keep the test subject programs as similar as possible, even to the extent of keeping variable names and other seemingly inconsequential things identical, even when your knowledge of SAS indicates that it shouldn’t matter. It is important that the only difference between the test subjects be the item you wish to test, you don’t want to take the chance that something may inadvertently confuse the issue.

There is no inherent limit in the size of test subjects that can be tested by this system. We not only examine the differences in the actual part of the program where the code is situated, but also the rest of the program to see if there is any additional effect on other steps. For example, we’ve seen the resources used in a PROC SORT step quintuple after a sloppy use of macro variables. You can assume that the part of the program before any change is going to perform identically (especially if you test for it occasionally) in the two test subjects, but you should never assume that the same is true after any change.

Testing simple fragments is easy, but issues can get more complicated in longer programs, especially when we’re testing whether several DATA steps can replace one. At times we’ve inserted dummy DATA steps into one of the test subjects to keep the number of steps comparable in a set of test programs, and correctly aligned. Eventually we will be taking the record from each pair of a test set and differing the comparable values, so we have to make sure that we are comparing equivalent steps from one of the pair to the other.

**AVAILABLE DATA:**
We run our test programs using OPTIONS FULLSTIMER. On our UNIX system, we get four statistics per DATA step and overall program usage. These four are real time, user cpu time, system cpu time, and memory usage. You should get similar statistics using the same option on any computer system.

Of these, memory usage and real time are normally only used to check for problems in the experiment run. Memory usage is not very useful in that memory is allocated in large blocks, so there is not sufficient detail present to indicate more than a very gross difference. Any
such difference is reported when available, but this has been a rare occurrence.

Real time is slightly more useful, but our network is shared, so we have little control over who may be either using the cpu or generating large amounts of network traffic. We only use real time to check for interference in the test run, by looking for any unusual pattern present in the data. There will usually be some extreme differences in run time between executions of the same program, but a continual pattern of deviance can indicate problems with either the setup or program execution process.

So, we use the various cpu times for the purposes of comparison. These are values specific to each program execution, and assuming that the system is properly set up, will be accurate and consistent. A simple test for this is to execute the same program fragment several hundred times or so, then compare the reported values for each statistic. If your system is reliable, real time values may show a lot of variance, but the cpu values should test as being all from the same population, normally distributed, with a minimal value for variance.

UNIX differentiates between user and system cpu times, CMS differentiates between virtual and total cpu usage, and we imagine that other systems will report something similar, appropriate for each. Please make sure that you have a clear idea of what each statistic is measuring. There's nothing quite so embarrassing as reporting a value to be used as a rationale for change and have someone point out that you're using the wrong one, such as using user cpu time to evaluate a system-intensive process.

There are two issues concerning the FULLSTIMER option. First is the option utilizes more system resources, so your programs will take longer to complete than they would without the option. Also, some operating systems' versions of SAS recognize it as an option that can be placed within a test program, others require it to be set at initialization time. For both of these concerns, we use a separate "config" file for testing, so that the option is always set at initialization time.

THE DATA COLLECTION PROCESS:

The next issue is how to extract the run data from the log files. You can do this by hand, but that could get a little tedious, especially if you've set up for 500 iterations, which means 1000 log files to open and process.

We've written two SAS programs to process each log file. They both do roughly similar things; they each read in each line of the log file in sequence, looking for the appropriate strings that indicate a reporting value. When the right string is found, then the numeric value is input into a character dataset variable. As an example, the string "real time" should uniquely identify the line where the real time statistic is reported. Of course, this will differ for each system, so you should check a sample of your output to see what strings you need to identify.

It may seem unusual that we are reading in data inherently numeric into a character variable. On the UNIX operating system, the SAS system switches it's output format for time usage statistics, e.g. 59 seconds will be reported as 59 seconds, but 61 seconds will appear as 1:01.00. To get around this, we have written a small macro that checks the character value for the presence of a colon. If found, we don't try to scan in the the next word, which might take us to the next line, instead we automatically report the unit as 'colon'. There is an apparently new UNIX-specific option in SAS 6.10 called STIMEFMT, that allows you to force the format to be consistent, making our macro unnecessary.

Where the two extraction programs differ is what they do with the read-in values. The first accumulates the test values over the individual steps, reserving a different variable for the overall total reported at the end of the program. This has an additional benefit; it collects data for every individual step in the test program, allowing us to identify the maximum number of steps needed to set the file sizes width for the second extraction program's output file.
The second extraction program does not start reading in values at the beginning of the log file. Instead, it looks for three specific strings that tell it where to start collecting; strings containing in sequence the program name, execution date, and execution time. We use an autocall macro to insert the three strings into the test subjects in a consistent fashion. When the extraction program finds these strings, it writes the reported values to the output file, then starts reading in the numeric values. When finished, each record in the flat file will contain all the desired data collected from one log file.

We chose SAS as the language for these programs mostly for adaptability, we can easily port the program to another platform when needed. If you've ever tried to convert an EXEC to a UNIX script, you can understand why we are willing to write a SAS program to do something this complicated.

THE EXECUTION PROGRAM:
The next needed piece is a program that executes each pair of test programs sequentially, then calls the extraction program to get the data values from each program log. Ours is a UNIX script, but it has some features that will be applicable to any system. We used an operating system-specific script for this because all we're going to do is invoke SAS as a non-interactive process a few times, and manipulate files, two relatively simple concepts that can be easily coded in any scripting language.

First, it is set up to either loop for a specified number of iterations, or to discontinue after a certain set time. As mentioned before, we run our test processes at night, when considerably fewer people are using the network. We borrow machines from other people, who tend to be much more kindly disposed towards us if we have stopped running processes on their machines before they come in the next morning. So, our iteration program can be set to finish up and exit after a certain time in the morning, 7:00a is usually a safe time.

The execution program is set up to run either one or a pair of test subjects. When running a pair of programs, it calls a random number, then determines which one of the pair it should run first. If you always run program one then program two, you are introducing bias into the experimental process, because your experiment sequencing is no longer random. Also, after each execution, the process "sleeps" for five seconds before proceeding to the next test execution. You can introduce bias into the testing process, depending on your hardware, by not resting for at least a few seconds.

An additional feature of the execution program is that it will make the necessary arrangements to store the log files from every iteration. Sometimes unusual values will be collected, and having the log file accessible means the difference between guessing whether the outlier is the result of a transient network condition, and being able to make sure. If the value is a legitimate extreme value (often called an influential point) it should be retained, but if it can be shown that the value is extreme due to an uncontrollable external influence, it should be discarded.

In our case, we retain the log file by renaming the log file; we concatenate the iteration number to the file extension, so that the log file does not disappear each time the test subject is executed. This is easy to do on a UNIX system, but you may have to use another method, due to the file-naming conventions on your operating system.

One final thing, the execution program is intentionally designed not to shift any part of the testing process to another machine on the network; it executes everything on one host. While developing the performance testing system, we've found large performance differences between machines supposedly configured identically. If you make sure that every aspect of an experiment run executes on only one host, you have eliminated another potential source of bias.

EVALUATION TIME:
When the testing process has finished, it's time to start evaluating the outcome. For this
purpose we use three SAS programs. One uses the output from the first extraction program to simultaneously check for both testing and overhead use problems. The second takes the data from the second extraction program, and matches up the paired observations, then differences the pair. The process converts two input observations into one observation that contains all the original data and the difference between the two paired observations at every desired run/step boundary, as well for the overall usage. This program does input the time values as character, and depending on the units(seconds,colon) either converts the value to seconds, or just uses INPUT to convert the value to numeric, so that when we’re done, every value is expressed in terms of seconds. This observation is written to a permanent SAS data set, which is used by the third program to produce the plots necessary for evaluation.

For evaluation, we prefer to use SAS/GRAPH procedures whenever possible, rather than numerical procedures, such as UNIVARIATE. We’ve found that it’s easier to report many related statistics on one page with PROC GLOT output, making the evaluation and presentation of the results much easier, especially to a non-statistically trained audience. Of the time spent in building this system, roughly 30% of the time was spent in developing the plots that we use for evaluation, and we consider it time well spent.

The only use we currently have for numerical procedures is for internally verifying assumptions and process tracking, such as verifying that a data variable is distributed normally, and for calculating the range values for the data, so that we can keep the plot scales as consistent as possible.

CHECKING FOR TESTING PROBLEMS:

The first thing we do after running a test process is plot the overall real time, user cpu time, system cpu time, and memory usage of each test subject against the iteration number. This is for the purpose of looking for any patterns in the data that indicate problems with the test run. At this time, it may become evident from the plotted data that some observations need to be discarded, so we’ll simply edit a copy of the original data and get rid of the offending records.

One thing to remember, this is a paired comparison experimental design, so you will need to determine which are the problem records, which other records were paired with them, and delete all of these observations. The choice of a paired comparison design makes it necessary at times to remove otherwise valid and reliable data from the output data set.

As an example of a record necessitating removal, consider the following situation that we have had to live with until recently. We currently work at a site with over 40 networked UNIX workstations available as hosts. The internal clocks on each workstation will get out of synchronization with each other, and the initial fix was to reset each machine’s internal clock once a day, at 5:05a. This reset would take place whether or not other processes were running on the machine, and occasionally it would fall within a DATA step. The needed adjustment would normally be small, but if the machine happened to be fast, the machine time would be set back. If this reset happened to occur in the middle of a DATA step, it could cause the DATA step to appear to finish before it started. The SAS compiler treats this small negative number as a very large positive number, so that we would get run times like 1193:02:30.31, as in HR:MIN:SEC/SEC/100. The end result would be a very large DATA step run time value, and a reduced overall run time value, and there was no record of the adjustment, so we couldn’t go back and recalculate the correct values.

So, what we’d have to do is check the values that appeared for any program that started before and finished after 5:05a, looking for any extreme results. When found, we’d edit out that record and it’s partner, although the partner’s execution may have been unaffected. Our system administrators have since found a better synchronization method, so this problem no longer appears.
ASSUMPTION VERIFICATION:

After we've corrected any evident runtime problems, it's time to verify the assumptions that we're going to be using in our performance checks. We look at the distribution of the reported time statistics, making sure that each is normally distributed. We usually don't need the assumption of normality because of the large numbers of test iterations, but it never hurts to make sure.

We've also made the assumption that the run order of the two test subjects does not matter. So we divide the data into two groups, by whether or not it was the first of the pair to execute, and make sure that the two samples are consistent with each other.

Assumption verification is a necessary process that we cannot stress the importance of too much. When you don't verify that your assumptions are defendable, you run the risk of reporting results from spurious tests as facts, which can only cause you problems in the future. We've seen cases where the results from bad testing have been reported as Gospel, and the reeducation process can be a painful one in which to participate.

CHECKING FOR OVERHEAD PROBLEMS:

The first performance check we perform is to look for problems in overhead. As you may be aware, the total run time listed at the end of a SAS program execution is not the sum of each DATA step's run time; instead, it represents the total time between the start and finish of the execution process. As such, it includes the time spent in each DATA step, plus the time spent processing external commands. In DATA step programming, this includes any time spent in the SAS macro processor, as well as any other SAS or external commands that do not need to execute within a run step boundary, such as any X command.

For those of you that don't think that the processing of external commands is of any consequence, let us share an example of some bad macro variable usage, or why white space can matter. We have four test programs that are identical in terms of all the non-blank characters. They each have the code:

```%let labvar = %cmpres (
... scatter 116 test names over 37 lines, no more than five to a line ...);
DATA dummy;
   dummy = 1;
RUN;
```

The purpose for the DATA step is to allow for one set of DATA step statistics to be printed, we'll come to why in a minute.

Two of the programs are in a variable record format, the other two are in a fixed format of length 80, or V versus F for variable versus fixed. Also, two of the programs have the lab tests below the first line left justified to the margin, the two others have the start of the line inset 20 characters, so that they align below the left parenthesis, or N versus I for non-indented versus indented.

The original programmer's thinking was that white space did not matter; that the %CMPRES function would work as designed and squeeze out all extra blanks. In one sense he was right; the %CMPRES function does work as designed; when you print out the resulting macro variable, there is one space between the test names.

These four test programs were executed once using SAS 6.08 on an IBM 9021 series mainframe, with CMS as the operating system. As you may expect, the overall execution times differ slightly, if you consider a difference of more than 30,000 percent slight.

<table>
<thead>
<tr>
<th>test subject</th>
<th>Total CPU Time, in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>V &amp; N</td>
<td>0.52 0.06</td>
</tr>
<tr>
<td>V &amp; I</td>
<td>17.45 0.06</td>
</tr>
<tr>
<td>F &amp; N</td>
<td>151.56 0.06</td>
</tr>
<tr>
<td>F &amp; I</td>
<td>153.59 0.06</td>
</tr>
</tbody>
</table>

When we execute the same programs
using SAS 6.09 on a networked UNIX machine, we get a similar result. Here, there are only two test subjects because UNIX does not have a fixed file format.

<table>
<thead>
<tr>
<th>test subject</th>
<th>User CPU Time, in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Overall</td>
</tr>
<tr>
<td>N</td>
<td>0.74</td>
</tr>
<tr>
<td>I</td>
<td>17.19</td>
</tr>
</tbody>
</table>

From a quick review of these two tables, we think you can see that white space does matter at times. The most reasonable explanation we’ve heard is that when the SAS compiler sets aside space for a macro variable, it allocates sufficient virtual memory to hold the entire contents, and this memory allocation can only increase, not decrease. In this case, the space needed is determined before the function %CMPRES is executed, so the compiler is unaware that it’s setting aside valuable memory blocks to hold a lot of blanks, and these blanks won’t even exist shortly.

In our contrived test programs (and the original was only slightly nicer than these), you can’t notice the ripple effect that results from creating such a large macro variable. Other steps and procedures that use dynamic memory will take longer to execute, simply because memory allocation has become a more complicated and time-consuming process. The end result is a program that takes much longer to execute in real terms, and the CPU usage also goes up dramatically, which is a big issue if you are charged for each second of usage. However, because the major difference in usage does not appear in any of the individual DATA step statistics, you can look for the reason for the abnormal usage for quite a while.

To identify problems such as this, we simply total the relevant statistic value for each data step, then compare this value to the overall total reported at the end of the program. We compare by taking the ratio of the overall value when divided by the calculated total. The value can never be less than one when your process is working correctly, and the larger the value, the greater percentage of your overall resources is being devoted to external processing. We use as a rule of thumb that any value larger than 3(CMS), or 2(UNIX), invites an examination of the program for design problems.

This is why the test subject programs had the DUMMY DATA step, so that there was something to calculate and compare with the overall time. For our four test subjects executed on CMS, the ratio ranges from a low of 7.667 to a high of 2558.833, all indicating that there is some concern with program overhead. In our case, we plot the ratio values for each pair against the test iteration number, so that we can look for any patterns in usage, and whether there is separation between the test subjects’ results.

![Graph showing ratio between cumulative and total usage.](image)

**CHECKING FOR SIGNIFICANT DIFFERENCE:**

As stated before, we use one program to prepare the evaluation data set, and one to create the plots. We use three plots, one showing the calculated difference at every run/step boundary, the second shows the data at each run/step boundary, and the third shows both overall difference and data. We do wish to emphasize that while the title says significant difference, this is a statistically significant difference, not necessarily important enough make someone modify their code.
Also, we are aware that we aren't really documenting that we are correctly following the procedures for formal hypothesis testing, so we really shouldn't be implying that we are, which is why we speak in terms of means and standard error bars. The end result is that the innumerate reader easily can tell from our plots whether there is a statistically significant difference at a certain point between the two test subjects.

The first plot, when no difference is indicated, should show a line of points that are horizontal and close to zero. We indicate the minimum, maximum, and mean, with standard error bars about the mean, indicating a confidence interval of slightly over 95%. When there is not a significant difference from zero, the horizontal reference line at zero should lie within the error bars for every run/step boundary number.

The third plot combines the presentation of the first two for the overall data, here using the maximum, minimum, mean with error bars. This is combinable because you will always only have three horizontal points to plot, whereas in the previous two, we've had as many as 60 horizontal points.

The second plot is of the run/step data from each test subject, plotted with just the mean and same standard error bars. We omitted the maximum and minimum points due to crowding on the plot. Here, the data from one test subject is overlaid on the other, with the horizontal axis shifted from one side to the other. When there is no difference, the plots should look virtually identical, with one shifted above the other. This plot is mostly just for reference, a difference of one second (from the first plot) can be more meaningful if it's in a step that uses few resources, as opposed to it being the difference between 159 and 160 seconds.
You may be wondering why we didn’t include the overall with the individual step data. Initially, it was done this way, but the scale difference between the individual steps and the overall value meant the individual values were almost featureless in an 81/2x11 plot. Also, remember that the overall time is the only place where usage from external resources (which includes the macro processor) is measured, so it can indicate a significant difference when one is not evident in the individual steps. We feel the data presented is less confusing if presented in three separate plots.

CONCLUSION:
The actual things you need to construct a system for performance testing SAS code is much less than you might think, and with some appropriate design in the right places, can be much less expensive than your employer imagines.

ACKNOWLEDGEMENTS:
The authors wish to thank Douglas R. Parrish and Mario Widal for contributing their ideas to the process that became this system, Daniel Stephens for being able to supply us with a coherent explanation of the difference between user and system cpu usage on a UNIX system, and especially our former boss, Robert W. (Tad) Braden, for making it possible for us to be able to do the work at all.

SAS and SAS/GRAPH is a registered trademark of SAS Institute Inc. in the US and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.