**Programming Techniques for Optimizing SAS Throughput**
Steve Ruegsegger, IBM Microelectronics, Burlington, VT

**ABSTRACT**
What is the primary bottleneck for SAS when it analyzes large datasets? What can a programmer do to mitigate that bottleneck and optimize their SAS code for throughput? This paper will discuss several programming techniques which will increase the throughput of SAS analysis, particularly when analyzing large datasets. When these techniques were implemented in one example program, the real-time throughput improved 5x.

**INTRODUCTION**
In the IBM chip factories, we use SAS as one of our data analysis engines. We have 100’s of jobs running simultaneously on our large 32-CPU AIX servers. Our datasets typically run in the 100K’s to 10M’s of observations. We have had periods of time when the server performance has severely suffered due to an overloading of the system. Sometimes, the solution for an overloaded server is to buy more hardware. However, inefficient coding may be adding a significant and unnecessary load on the server. Optimized code may free up wasted resources and return the server to its proper performance levels.

This paper describes the programming techniques we employed for SAS analysis throughput optimization. It’s not a comprehensive list. The focus is to understand the primary bottleneck of a server running a ‘large number’ SAS jobs which are analyzing large datasets. When that bottleneck is understood, then several programming techniques can be used to mitigate that bottleneck. Examples of these techniques will be given. Additionally, the results of a before-and-after test case will be presented.

**SINGLE STRATEGY**
In order to optimize, one has to know in what direction to optimize – i.e., what are the bottlenecks that we are trying to avoid? It has become increasingly clear that running many ‘large dataset’ SAS jobs, the single, primary bottleneck to throughput is disk IO. This is actually quite intuitive:

1. All other HW components – CPUs, memory, backplane, even gigabit ethernet network connection – are incredibly fast compared to the disk drives.
2. SAS is incredibly disk-IO intensive. It accesses every dataset from disk, not memory.

The first point stems from the fact that the disk drives are mechanical. Physical read/write heads have to get to a physical location on a spinning disk. This movement is much slower than accessing memory or crunching numbers in silicon which are happening at the speed of switching transistors. The comparison of these data transactions is ms vs ns, i.e. multiple orders of magnitude.

The second point is an extremely important point of SAS’ architecture. SAS does not store datasets in memory. Everything goes to the ‘slower’ disk drives. This means that **every DATA and PROC command in SAS performs both disk IO reads and writes.** This architecture has the advantage of being robust and scalable (both of which are SAS’ competitive advantages), however, it also creates this disk IO bottleneck.

If disk IO is the slowest component of the server and also the most intensive part of SAS, then our primary programming strategy to optimize SAS throughput, especially for large datasets, is to:

**do everything we can to not read/write to the disks.**

Below are seven programming techniques which can be used to implement this single strategy. These techniques are

---

1 Our 32-CPU server runs 100’s of SAS job simultaneously. A “large number” of SAS jobs is probably best defined as 2 or 3 times the number of CPUs.
focusing the programmer to think in terms of data IO. As one writes the DATA and PROC blocks to get the desired analysis, these techniques are asking the programmer to also consider the number of times their code is forcing SAS to read or write entire (large) datasets to disk. For example, combining multiple DATA blocks into one DATA block is an obvious way to reduce unnecessary disk reads and writes and removes that load on the disk IO. This can contribute to significant server performance improvement.

These techniques do not increase the length SAS code, nor do they necessarily make the code any more complicated or ‘harder’ to get analysis. They are intended to get the programmer to realize that their jobs with large datasets will run on a busy and shared server which cannot afford wasted resources. The desire is to get the programmer into the mindset of not only answering “how do I get the analysis I need?”, but also answer “how can I get that in a manner which minimizes the number of disk IO read/writes?”

ANALYSIS
The problem and solution described here can be illustrated with a ‘real-life’ example. Figure 1 shows a real-time server monitor tool for AIX called nmon. The images below show a snapshot of the 32 CPU’s utilization. The image on the left shows the “normal” performance profile where there are higher amounts of CPU usage (green U) than Waiting (red W). The Waiting (red W) utilization means that the CPUs are waiting for data – from the disk drives in this case – before it can continue a calculation. The image on the right shows what happens when the server becomes IO bound. Every CPU is waiting for data from disk. This causes the server to “feel very sluggish” and very little analysis is being generated. Basically, SAS cannot get data from the datasets with which to do analysis. The CPUs are simply doing nothing – just waiting for data.

![Figure 1: Snapshots of CPU utilization in “normal” and “diskIO bound” states](image)

Figure 2 shows why we reached the conclusion that the Waiting is for Disk IO. These images are from the topas utility in AIX which monitors disk IO. The disks are sorted by “% busy.” The image on the left is showing that a “normal” server state has the busiest disks at 60-90% busy. However, during a “diskIO bound” server state, the top 5 disks are “100% busy.” This can be characterized as “thrashing” disks, where the requests for data (from SAS DATA and PROC blocks) are coming...
at the disks much faster than they can be delivered. So, the requests pile up, the CPUs go into Wait states and the entire server loses performance.

Figure 2: Snapshots of disk utilization in “normal” and “diskIO bound” states

One can also get performance analysis at much smaller resolution with the SAS option FULLSTIMER, which according to SAS is provided: “to collect performance statistics on each SAS step, and for the job as a whole and place them in the SAS log.” This provides the real time and CPU time for each block. Additionally, depending on the OS, this option also provides details on page swaps and page faults which can be used to diagnose memory or disk IO bottlenecks, respectively. By using FULLSTIMER, this bottleneck/performance analysis can be done at the individual code block level.

EXAMPLE

Below is an example of the magnitude of increased throughput realized when these techniques were utilized. The SAS script used in this example reads in a log file, performs some summarizations, then makes plots. It was originally written without the techniques below. When server performance began to suffer, this program was re-written with the optimization techniques described below. The original code was stored in CVS, and was used as a control. The old and new versions were each run 3 times, one immediately after the other. No new features were added or deleted to the new code – it was modified to incorporate these optimization techniques. In fact, there is only a 4-line delta in their lengths. Yet, there was a significant improvement in speed. Below is a table showing the differences.

The unix time command was used to record the real and user times. The real time is a good metric of system performance since it is directly affected by the Wait states.

```
$ time /usr/sas/sas82 -batch -work $SASTMP $sasconfig metrics.timeslide.sas
```

2 http://support.sas.com/rnd/scalability/tools/fullstim/index.html
You can see that there was 3x, 6x and 8x improvement in the new code. Notice that the original script took over an hour on Server1 on the first day – compared to 9 min for the optimized code. It is interesting to note that Server3 is an older box than Server1 (fewer and slower CPUs and hard drives), but since it was not as overloaded, it out-performed the newer, more expensive and “faster” Server 1. The optimized code still increased the throughput on Server3, but not as dramatically as Server1 since Server3 was not as ‘diskIO bound’ as Server1.

<table>
<thead>
<tr>
<th>Date time</th>
<th>server</th>
<th>Original code</th>
<th>Optimized code</th>
<th>Throughput increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>3/21 ~3pm</td>
<td>Server1</td>
<td>real 1h6m20.76s</td>
<td>real 9m5.72s</td>
<td>6x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>user 0m32.24s</td>
<td>user 0m8.78s</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sys 1m14.47s</td>
<td>sys 0m11.16s</td>
<td></td>
</tr>
<tr>
<td>3/22 ~9am</td>
<td>Server1</td>
<td>real 54m48.77s</td>
<td>real 7m30.77s</td>
<td>8x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>user 0m29.20s</td>
<td>user 0m8.42s</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sys 0m59.51s</td>
<td>sys 0m8.79s</td>
<td></td>
</tr>
<tr>
<td>3/22 ~10am</td>
<td>Server3</td>
<td>real 3m18.73s</td>
<td>real 1m24.08s</td>
<td>3x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>user 0m55.52s</td>
<td>user 0m16.89s</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sys 0m45.96s</td>
<td>sys 0m8.14s</td>
<td></td>
</tr>
</tbody>
</table>

**TECHNIQUE #1 -- SASFILE**

There is a wonderful, undocumented command that was introduced in SAS v8.2. According to V9 documentation, the sasfile command “opens a SAS data set and allocates enough buffers to hold the entire file in memory.” It puts a dataset into memory for all future reads. Therefore, this removes future, multiple disk reads when accessing a dataset from any DATA or PROC request! By now it’s clear that this exactly is our primary strategy!

However, a restriction is that “a SAS data set opened by the sasfile statement can be used for subsequent input (read) or update processing but not for output or utility processing.” That is, a file in memory from sasfile cannot be edited or changed -- only read. Therefore, this programming technique uses sasfile only on the main datasets in the code -- datasets that you can setup once and not change anymore, but read from them over and over. For example, I use this technique when I have defined a dataset with my raw, large data to analyze. Then, once setup, I put it into memory with the sasfile command and read from it to do all my analysis. For example, I might do Shewhart, EWMA, T-test analyses as well as to make wafer, lot, daily, weekly, and monthly charts, as required. All these subsequent DATA and PROC steps are now reading from memory and not from disk. This will significantly reduce the requests to the disk drives. If the dataset is large, with 10’s of cols and millions of rows, this removes millions of read-requests from the disk drives.

To use sasfile, simply use the command

```
sasfile <dataset> open;
```

to put the data into memory, and

```
sasfile <dataset> close;
```

to “unlock” it from memory so you can edit it again. You will get an error if you try to edit a dataset that’s been put up into memory with the sasfile command.

Like all good things, there are tradeoffs to consider. It is not a good idea to start using sasfile commands everywhere. If the memory on the system fills with data, then the OS will begin page swapping… which will cause disk IO! That’s the very thing we are trying to avoid, and we’ve just gone back to square one. In general, for our large servers, we have ~ 200Gb of RAM, so sasfile provides an excellent way to use that ‘real’ memory and reduce disk IO requests. Usually, one dataset per script is in memory – the main, raw dataset being analyzed.

---

3 It’s now documented in v9, but since we are a manufacturing site, we are still on v8.2 and have not yet migrated to v9.
TECHNIQUE #2 -- INDICES

When *sasfile* is used, an error will occur if that dataset is used in a PROC SORT. This is because PROC SORT is an *edit* – it requests the dataset read from disk, performs the sort algorithm, then writes out the new dataset to disk in the desired sorted order. But this is exactly what we are trying to avoid. To make matters worse, anytime you use a BY statement, the dataset *must* be pre-sorted by the same keys. This can cause many PROC SORTs on the same dataset throughout one’s code. For small datasets on a single SAS box, this is not a problem. However, for large datasets on a very busy SAS server, this will be a detrimental load on the server.

These problems can be avoided by using indices. An index is a separate file with keys that tells SAS how to order the main dataset, yet the main dataset is not re-written in that new order. **By using indices, all the sorting is done “up-front,” as it were,** and we can eliminate many PROC SORT’s (and disk IO) used simply for BY statements.

Using indices is a key additional component when implementing the *sasfile* technique described above. By “pre-defining” all your sorts with indices, you do not need to use PROC SORTs throughout the code to satisfy BY statements. Therefore, the optimized programming strategy for large datasets is to define appropriate indices and then put the dataset into memory. All the subsequent PROC and DATA blocks can still utilize the appropriate BY statements to get the desired analysis – but without a re-read of the large dataset from disk!

Indices are defined by the PROC DATASETS command. You can create multiple indices in one PROC block. Below is an example where four indices are created for the dataset called *zscore*. The first three are compound indices and the fourth is a simple, single variable index. These indices are used for BY statements in various PROC MEANS, UNIVARATE, GPLOT, etc.

```sas
%put * making indices *;
proc datasets library=work nolist;
  modify zscore;
  index create i1=(sasbox enddttm);
  index create i2=(sasbox date hour);
  index create i3=(date hour);
  index create zby;
run;
quit;

sasfile zscore open;  * put this main dataset into memory;
```

Notice, that *after* creating the indices, the *sasfile* command is used. Now, the main dataset, *zscore*, is in memory and has four indices already pre-defined. Now as this main, large dataset is used over and over again for analysis, even with different BY statements, SAS only goes to memory -- no disk IO! For example,

```sas
proc means data=zscore noprint;
   by sasbox date hour;
   output out=started n=n;
run;
```

and then this command later on,

```sas
proc univariate data=zscore noprint;
   where date ge today() - 2 and sasbox eq "sassrv";
   by date;
   var z;
   output out=ptile median=p50 p95=p95 p99=p99 max=max n=n;
run;
```

Each PROC has different BY statements, yet a PROC SORT is *not* required before each one. Each BY statement simply uses
the appropriate index. Therefore, by using sasfile and indices together, I have removed many disk IO read/writes as I use this main dataset over and over again.

The difference between BY and CLASS statements is subtle, as both provide independent analysis for the variable discrete values. Since CLASS variables are not required to be pre-sorted, it might seem an optimal way to remove a sort or index from a large dataset. However, SAS uses that physical sorting to its advantage. By knowing the data is sorted with the BY statement, SAS does not have to search the large dataset to make sure it has all obs for a class. So, while not studied in detail, experience has shown that BY is faster than CLASS, presumably due to less diskIO due to a priori knowledge.

Obviously, both sasfile and indices have a price. They each have some overhead. The programmer clearly needs to evaluate when using sasfile and creating indices provides the desired benefit. In general, use these two techniques for “main, large datasets” which can be setup before the analysis, remain unchanged, and then followed by many analysis commands.

**TECHNIQUE #3 -- VIEWS**

One way to think of a view is like a “formula.” That is, a view is a formula to create a dataset -- it’s like a “virtual” dataset. Specifically, SAS does not actually create a new dataset that is written out to disk. It simply “remembers the formula” for how to recreate the dataset in memory when needed. Using a view is useful for removing the read/write of creating a new dataset. Rather than using a DATA block to make a small change to a dataset only to re-read the data for a subsequent PROC, a view can be used to simply record that change and then make that change when the view is used.

To create a view, you can to use PROC SQL or a DATA block. When using SQL, simply use “create view” rather than “create table.”

As an example, the table below shows the original code on the left where a whole new dataset was created simply to add a calculated value. That required a disk read and write of the large dataset zdata. (This is the bottleneck we want to avoid.)

The optimized code on the right defines a view (there is not a dataset on disk called zview). This removes the read and write of the large dataset. The view creates the new variable z “on the fly” every time it’s accessed, but the CPUs have plenty of resources to do that, and this is a much better tradeoff than to reread and rewrite this large dataset.

<table>
<thead>
<tr>
<th>Original code</th>
<th>Optimized code</th>
</tr>
</thead>
<tbody>
<tr>
<td>proc sql;</td>
<td>proc sql;</td>
</tr>
<tr>
<td>create table zscore as</td>
<td>create view zview as</td>
</tr>
<tr>
<td>select a.*,</td>
<td>select a.*,</td>
</tr>
<tr>
<td>(a.var - b.mean)/b.std as z</td>
<td>(a.var - b.mean)/b.std as z</td>
</tr>
<tr>
<td>from zdata a</td>
<td>from zdata a</td>
</tr>
<tr>
<td>inner join tmp b on a.zby = b.zby</td>
<td>inner join tmp b on a.zby = b.zby</td>
</tr>
<tr>
<td>quit;</td>
<td>quit;</td>
</tr>
<tr>
<td>/* get stats of new variable z */</td>
<td>/* Now the VIEW calculates z from zdata (which is already in memory with sasdata) */</td>
</tr>
<tr>
<td>proc means data=zsore noprint;</td>
<td>proc means data=zview noprint;</td>
</tr>
<tr>
<td>where abs(z) &lt; 3.0;</td>
<td>where abs(z) &lt; 3.0;</td>
</tr>
<tr>
<td>by zby; var var;</td>
<td>by zby; var var;</td>
</tr>
<tr>
<td>output out=tmp mean=mean std=std n=num median=median;</td>
<td>output out=tmp mean=mean std=std n=num median=median;</td>
</tr>
<tr>
<td>run;</td>
<td>run;</td>
</tr>
</tbody>
</table>
At the subsequent PROC MEANS, SAS reads large dataset zdata and actually runs the PROC SQL 'formula' creating the new variable z - only in memory. Then, the PROC MEANS uses the view results from that PROC SQL. The new code, by using the view, eliminated a disk IO read and a write to the intermediate zscore dataset.

This technique of using views is useful when there are (large) temporary datasets that are used for intermediary results before an analysis PROC. This is especially powerful when combined with the two techniques described above. If the 'base' dataset (zdata) is already in memory from a sasfile command with appropriate indices, the entire view is memory driven and not disk IO driven at all! Now, that’s a fast way to modify the dataset for analysis!

TECHNIQUE #4 -- WHERE

Many, many times a SAS programmer needs a subset from a larger dataset. Often a new, smaller copy of the data is created by using an if... then... delete; statement. This, of course, requires disk IO reads and writes which we are trying to avoid.

Rather than making a new dataset, the SAS programmer should use the WHERE qualifier. The WHERE qualifier can be thought of as an “input filter” (see Figure 3). That is, upon reading the large dataset -- before passing the data to the requesting PROC or DATA command -- only the observations which meet the WHERE qualification are ‘allowed through.’ So, the requesting PROC or DATA command acts as if it saw a ‘virtual data subset.’ This is a simple, fast way of selecting only a subset from a larger dataset without creating a brand-new dataset requiring unnecessary and wasteful disk reads and writes.

Figure 3: Sketch of how a WHERE filter is optimal over the 3 “if” row filters.

It is important to see the difference between the WHERE and the “subsetting IF” statements. The WHERE will “pre-read” only the required cols and then skip to the next obs if the WHERE condition is not met. The subsetting IF reads in the entire observation before checking the condition. Sometimes this is required. But if the subset condition is based on col values only, then the WHERE is much faster by removing the reading of all unnecessary cols. For large datasets, this is a significant optimization.

As we have seen before, if the large dataset is already in memory via sasfile, then using WHERE only uses memory and eliminates all disk IO reads. Additionally, if the WHERE clause uses an index, then the filter is even faster.

There are two primary methods to use the WHERE qualifier. The first is simply as a new line in any DATA or PROC block. An example from the test case program is shown below. Originally, a new dataset zscore2, was created by reading every row and every column (yuk) from the large dataset zscore and then ‘throwing away’ the rows not wanted from the

---

5 There is a lot of material on if/then/delete, subsetting if, and WHERE. Here is a place to start: support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a000201978.htm or www2.sas.com/proceedings/sugi31/238-31.pdf
if/then/delete. The new code *completely eliminated* the dataset `zscore2` and all the disk IO that went with it! That additional `WHERE` on the `PROC GPLOT` reads in one col from each obs and promptly skips the rest of the obs if not meeting the subset condition. In fact, by using a `WHERE` qualifier on the `zscore` dataset, which is already in memory from `sasfile`, there were *zero* disk IO reads and writes to get at this subset of data!

One technique used is to define a macro variable called `&where` which can be used in multiple places throughout the code in multiple `WHERE` clauses, but yet changed in one place at the head of the script (or input from an OS argument).

<table>
<thead>
<tr>
<th>Original code</th>
<th>Optimized code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data zscore2;</code></td>
<td><code>%let where = date ge today() - &amp;ndays;</code></td>
</tr>
<tr>
<td><code>set zscore;</code></td>
<td><code>proc gplot data=zscore nocache;</code></td>
</tr>
<tr>
<td><code>if date le today() - &amp;ndays</code></td>
<td><code>title1 &quot;normalized performance&quot;;</code></td>
</tr>
<tr>
<td><code>then delete;</code></td>
<td><code>title2 &quot;z-score = number of sigma from mean&quot;;</code></td>
</tr>
<tr>
<td><code>run;</code></td>
<td><code>title3 &quot;data qualifiers: &amp;where&quot;;</code></td>
</tr>
<tr>
<td><code>proc gplot data=zscore2 nocache;</code></td>
<td><code>where &amp;where;</code></td>
</tr>
<tr>
<td><code>title1 &quot;normalized performance&quot;;</code></td>
<td><code>plot z * dttm = sasbox/</code></td>
</tr>
<tr>
<td><code>title2 &quot;z-score = number of sigma from mean&quot;;</code></td>
<td><code>frame grid legend = legend1</code></td>
</tr>
<tr>
<td><code>title3 &quot;data qualifiers: in last &amp;ndays days&quot;;</code></td>
<td><code>haxis = axis5 vaxis = axis3</code></td>
</tr>
<tr>
<td><code>plot z * dttm = sasbox/</code></td>
<td><code>vref = (0 3 6 20);</code></td>
</tr>
<tr>
<td><code>frame grid legend = legend1</code></td>
<td><code>run; quit;</code></td>
</tr>
<tr>
<td><code>haxis = axis5 vaxis = axis3</code></td>
<td></td>
</tr>
<tr>
<td><code>vref = (0 3 6 20);</code></td>
<td></td>
</tr>
</tbody>
</table>

The second way to use the `WHERE` qualifier is as a ‘dataset option.’ The context for this is to put a `WHERE=` statement in parenthesis next to any dataset name. This allows the `WHERE` qualifications to act as an input filter for only that one dataset. For example:

```
data active_users;
  merge zscore(where=(date > today() - 3)) users;
  by user;
  run;
```

**TECHNIQUE #5 -- RENAMING A VARIABLE**

In order to merge datasets together in a `DATA` block, the `BY` variables have to be named similarly in all datasets. Often, when datasets to be merged have different names for the key variables, the tendency can be for the more novice SAS programmer to make a temp dataset with newly named variables. This requires every single data point in the large dataset to be read and then written to disk. There are two very simple programming techniques to avoid this very unnecessary disk IO! The code on the left is from an actual script.

The `RENAME=` dataset option acts as an output filter to the dataset so that the command requesting the dataset sees only the renamed column. For the example above, the column called `lot_id` in the `inline` dataset now looks like it is named `lotid`. This “rename” is only in effect for the `MERGE`. It did not actually change the name of the column permanently.
TECHNIQUE #6 – COMBINE BLOCKS AS MUSH AS POSSIBLE

When developing SAS code, most programmers tend to break the complex analysis into lots of little tasks so that the intermediate results can be seen. This is good development technique but result in very suboptimal throughput. A programmer needs to go back and condense the code into as few DATA steps as possible.

A classic example is copying a dataset from slower archive network server into the current WORK directory (which is usually much faster). Often, a sort needs to be done after that copy (in order to do a merge or create a plot, for example). However, rather than performing a disk read and write for both the copy and then the sort, the sort could also perform the copy, thus eliminating one disk read and write.

<table>
<thead>
<tr>
<th>Original code</th>
<th>Optimized code</th>
</tr>
</thead>
</table>
| proc datasets;  
  copy in=archive out=work;  
  select bigds;  
  run;  
  quit;  
proc sort data=bigds;  
  by test_dt;  
  run; | proc sort data=archive.bigds  
  out=work.bigds;  
  by test_dt;  
  run; |

TECHNIQUE #7 – PROC APPEND

This technique is an example of knowing the PROCs available. Often, a large dataset needs to be appended with a smaller dataset. A common example is a “log” dataset which is storing chronological data events. When there are new events to be
appended to the larger historical dataset, often the DATA step will be used. However, this re-reads and re-writes the entire large dataset without making any changes (yuk). The programmer who is familiar with SAS PROC’s will use PROC APPEND, which will simply open the large dataset, move the file pointer to the end of the file, and start writing (appending) the new data.

<table>
<thead>
<tr>
<th>Original code</th>
<th>Optimized code</th>
</tr>
</thead>
<tbody>
<tr>
<td>data bigds;</td>
<td>proc append base=bigds data=append;</td>
</tr>
<tr>
<td>set bigds update;</td>
<td>run;</td>
</tr>
<tr>
<td>run;</td>
<td></td>
</tr>
</tbody>
</table>

There are some consequences to this optimized approach of PROC APPEND. Since the ‘base’ dataset is not being re-written, the col structure cannot change. If the ‘new’ dataset does not perfectly match the column structure of the base dataset, then SAS will give an error. However, there is a FORCE option, which will allow the PROC to continue in spite of column differences. For example, additional cols from the new dataset will not be appended since they cannot be added to the base dataset. In general, I do not use the FORCE option, but prefer to write the code to ensure the two datasets have matching col structures.

ADDITIONAL THOUGHTS

The techniques above have all been ways for the SAS programmer to reduce disk IO, and thus improving SAS throughput for a busy server which runs analysis of large datasets and is prone to a disk IO bottleneck. Below are some additional ideas for batch submission management and WORK space setup which can also help with server performance.

Not running unnecessary jobs

We love to run batch jobs which make charts and reports, preferably overnight, which are ready for employee consumption in the morning. This is a good thing. However, these automated jobs often run unnecessarily and therefore consume resources without a gain. There are 2 situations which make a job ‘unnecessary:’

1. No new data – the result is exactly the same as the previous run.
2. No consumption of the analysis -- no one is watching or cares.

The first issue can be addressed with some simple checks at the beginning of a SAS script. In a dynamic manufacturing setting, if there is no new data for a particular analysis, then re-running past analysis on the same data is wasteful.

The second issue is harder to implement since there needs to be a mechanism to monitor the consumption of the analysis. If there is a ‘chart viewer’ or web content server, then the SAS programmer could check content server for any activity on a particular result set, and then call the analysis ‘inactive’ if some activity thresholds are not met.

WORK space setup

There are 3 primary strategies we considered for WORK space configuration.

1) Independent temp space from the OS. In Unix, /tmp is the temp space for the OS. Do not have that also be the WORK space for SAS. It will severely overload the system. Add additional harddisks dedicated for the SAS WORK space.

2) Stripe the WORK disks. In addition to independent hard drive for SAS WORK space, put them in a RAID configuration and stripe them. This will allow multiple disks to read a large dataset in parallel.

3) Furthermore, if there are many simultaneous users/jobs, setup multiple WORK spaces. For example, we have /SAS/A through /SAS/G on one large server. These are striped sets of independent disk drives. The user is assigned one of these WORK spaces when SAS is started, using the –work option. This could be done at random and allow the uniform distribution to spread out the load. However, we’ve put some intelligence into the startup script so that the “least busy” temp space is selected for that user at that time.
We have also created a RAM drive for the \texttt{WORK} space of a limited amount of SAS jobs. As mentioned previously, the size has to be managed so that page swapping doesn’t begin to occur. But if kept in check, running a SAS job with its \texttt{WORK} library defined in a RAM drive can remove most all diskIO and significantly improve throughput for analysis with large datasets.

**Dataset compression**

We have dataset compression turned “on” globally on our servers (via \texttt{SASROOT/sasv8.cfg} config file). Since our CPU’s are under-utilized, giving them the burden of compression before and after diskIO is a great tradeoff in order to have less bytes to read/write to the disks.

It’s beyond the scope of this paper to go into more details for these configuration ideas – but they are key components to running 100’s of SAS jobs simultaneously with large datasets on a powerful unix server.

**CONCLUSION**

Several programming techniques for optimizing SAS throughput have been described. The single strategy is to eliminate non-essential disk IO which is the slowest component of a server, and also the component that SAS most utilizes. However, by re-thinking programming methods and implementing these techniques, a demonstrated throughput improvement of 3x - 8x can be obtained.

**OTHER RESOURCES**

There are many other fine documents for programming techniques for throughput optimization. Here are some with which to start:

- [https://support.sas.com/edu/schedules.html?ctry=us&id=279](https://support.sas.com/edu/schedules.html?ctry=us&id=279)

**CONTACT INFORMATION**

Steve Ruegsegger  
IBM Microelectronics  
1000 River Road  
Essex Junction, VT 05465  
Email: ruegseggs@us.ibm.com

**ACKNOWLEDGMENTS**

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are registered trademarks or trademarks of their respective companies.