Paper 11825-2016

Life in the Fast Lane: Optimisation for Interactive and Batch Jobs

Nikola Marković, Boemska T.S. Ltd.; Greg Nelson, ThotWave Technologies LLC.

ABSTRACT

We spend so much time talking about GRID environments, distributed jobs, and huge data volumes that we ignore the thousands of relatively tiny programs scheduled to run every night, which produce only a few small data sets, but make all the difference to the users who depend on them. Individually, these jobs might place a negligible load on the system, but by their sheer number they can often account for a substantial share of overall resources, sometimes even impacting the performance of the bigger, more important jobs.

SAS® programs, by their varied nature, use available resources in a varied way. Depending on whether a SAS® procedure is CPU-, disk- or memory-intensive, chunks of memory or CPU can sometimes remain unused for quite a while. Bigger jobs leave bigger chunks, and this is where being small and able to effectively exploit those leftovers can be a great advantage. We call our main optimization technique the Fast Lane, which is a queue configuration dedicated to jobs with consistently small SASWORK directories, that, when available, lets them use unused RAM in place of their SASWORK disk. The approach improves overall CPU saturation considerably while taking loads off the I/O subsystem, and without failure results in improved runtimes for big jobs and small jobs alike, without requiring any changes to deployed code. This paper explores the practical aspects of implementing the Fast Lane on your environment. The time-series profiling of overnight workload finds available resource gaps, identifies candidate jobs to fill those gaps, schedules and queues triggers, changes application server configuration, and monitors and controls techniques that keep everything running smoothly. And, of course, are the very real, tangible gains in performance that Fast Lane has delivered in real-world scenarios.

INTRODUCTION

This paper will focus on the optimisation of batch performance. Rather than advocate efficient coding techniques and focus on improving individual job performance, it will take a holistic approach to batch optimization, where possible avoiding low-level details. There will be no mentions of page faults. It will instead suggest techniques for collecting detailed resource utilisation data which isn't otherwise readily available, and show how that data can enable the programmatic real-time reassignment of memory resource to improve the performance of an environment as a whole. Most importantly the optimisations this paper suggests aim deliver a substantial performance improvement without requiring you to modify, re-test or re-deploy any program code that has previously been approved and scheduled to run in production.

THE ROLE OF STORAGE IN A SAS DEPLOYMENT

The notion that SAS® can become 'disk bound' or 'I/O bound' has been discussed in a number of previous papers. Understanding exactly what that means requires a closer look at how SAS works[1]:

Data processing with SAS is performed in steps. A program normally reads in data from an input data store, processes it using a combination of DATA steps and/or PROCs, and finally persists the output data by writing it out to another data store location. This flow of data for a typical SAS program is represented visually in Figure 1.

Each of the vertical arrows in the graphic represents a movement of data; most of these movements (blue, green, orange arrows) involve reading or writing data from or to a *block storage device*. Block storage devices, even modern solid-state based ones, are generally still two orders of magnitude slower

[1] Note: the discussion here is limited to native SAS processing and does not cover processing that may occur within an external database.

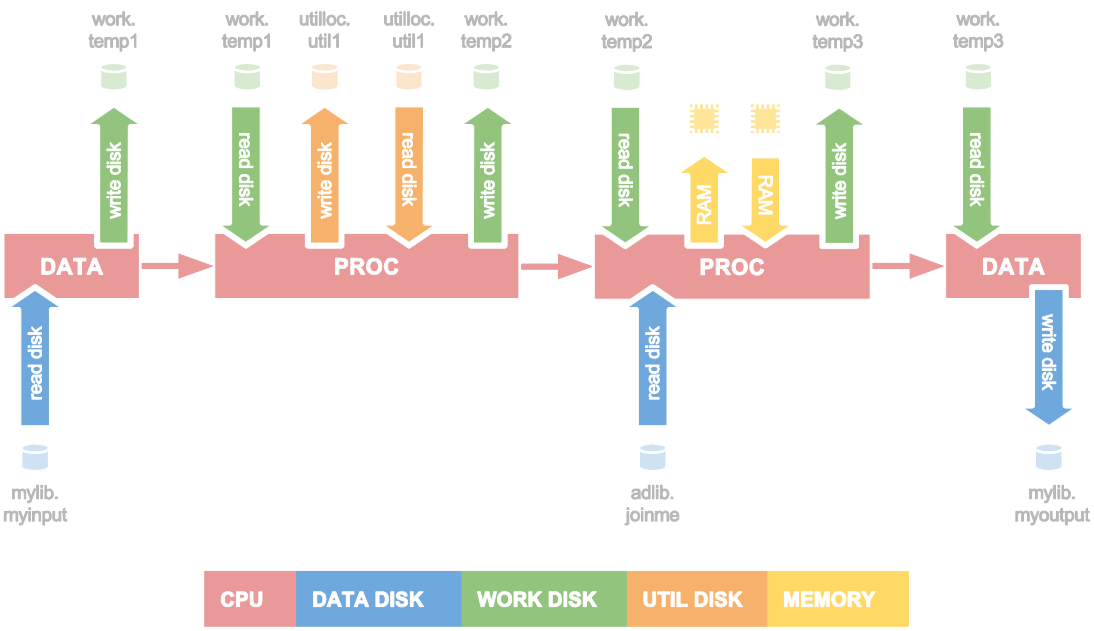

Figure 1: The flow of data in a typical SAS® Program

than *Memory (RAM)* at passing data to the CPU for processing, or being able to receive and store the results of a calculation. If the work a particular section of a SAS program is doing on the data is not very complex, the CPU will find the work easy and demand that it is supplied with data at a faster rate in order to stay busy. This is where the *throughput* of a disk device can so often become a limiting factor; if the storage devices cannot keep up with demand, the CPU begins to spend large portions of its time either waiting for processed data to be written to disk, or waiting for input data to arrive from disk. When this bottleneck condition occurs, the capacity of the entire system is no longer defined by the processing capacity it has, but by the rate at which its storage devices can supply its processors with data.

The behaviour of the typical SAS program in Figure 1 therefore highlights the points at which data is read from or written to disk. By calculating the total volume of data that a given program must read from and write to a disk device to complete its processing, it is possible to arrive at the *total I/O cost* for that program. Unlike the duration of execution for a program, this cost is generally predetermined; there is a fixed volume of input data that must be processed, and the logic of the program dictates where and how often that data must be read from and written to the various storage devices in the system. Each program has an I/O cost. The I/O cost for each different stages of a typical SAS program is visualised in Figure 2, separated into the different storage devices that make up a typical SAS system.

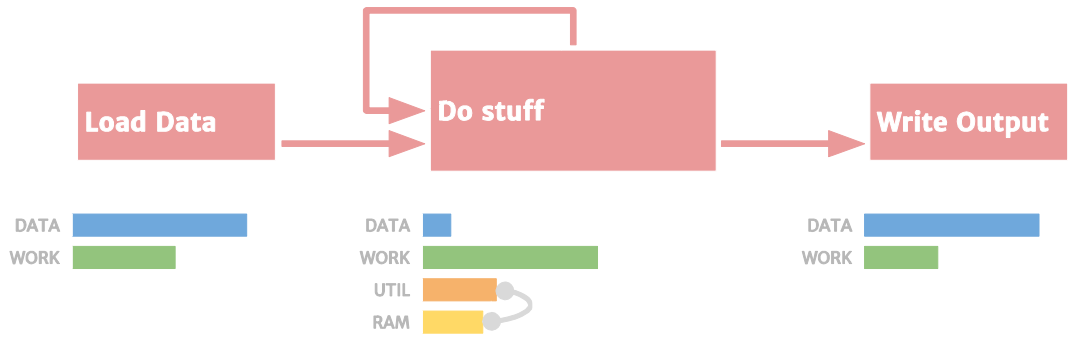

Figure 2: The I/O of a typical SAS® Program split by storage device

Different programs perform different operations, using resources in different ways[2]. Some, *Type A programs,* perform a single operation on one large dataset for a long period of time. Figure 3 describes the typical I/O cost of these programs.

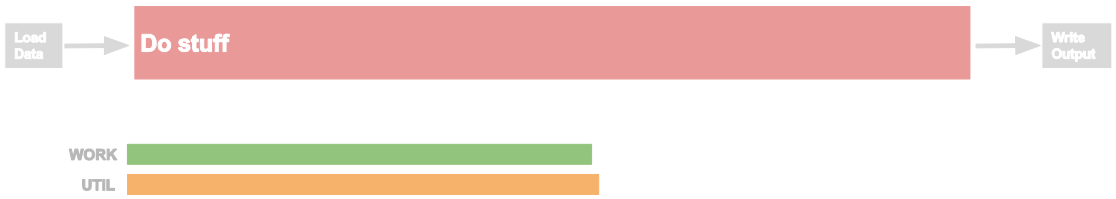

Figure 3: A short program with a big input dataset (*Type A*)

Other programs are longer in that they have more steps, make more passes or perform more operations on a smaller volume of data. Figure 4 describes these longer, *Type B* programs.

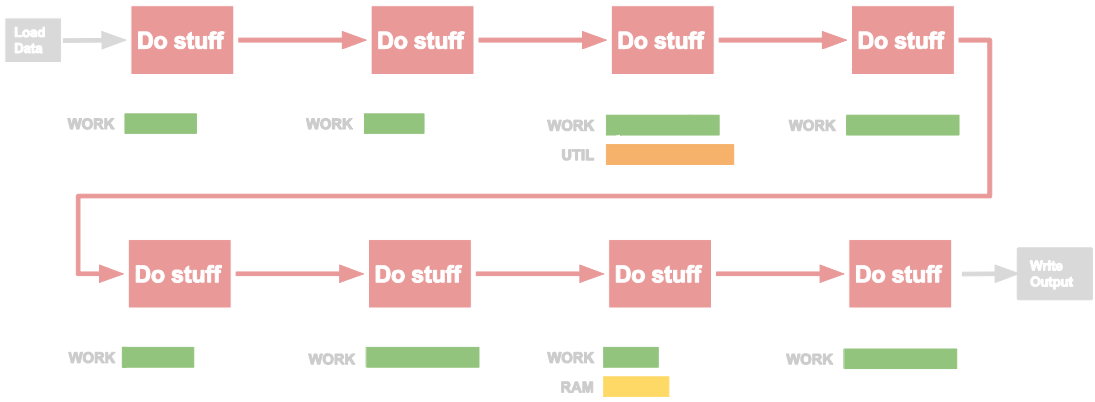

Figure 4: A long program processing a smaller volume of data (*Type B*)

The example *Type A* program performs one big operation on a lot of data, but only writes it and reads it two times: once to and from the UTIL disk, and once to (and eventually from) the WORK disk. The example *Type B* program reads and writes a much smaller amount of data for each of its steps, but there are many more of them; when combined, the amount of data these steps read and write, the throughput, is the same the *Type A* program. The amount of processing that the CPU needs to perform is also the same for both programs; in the latter program, it just happens to be split into smaller segments.

Often these two programs would be considered identical by most schedulers: the *CPU cost* and *I/O cost* of the programs is identical, and they therefore take a similar amount of time to execute. However, there is a key difference between them. Figure 5 shows the *pattern of disk space usage over time* for the *Type A* program.

---

[2] *At this point it is worth pointing out that, when it comes to processing data, the I/O cost of loading source data and writing result data is inherently compulsory and cannot be avoided. For this reason, this paper will ignore the I/O cost associated with these operations.*

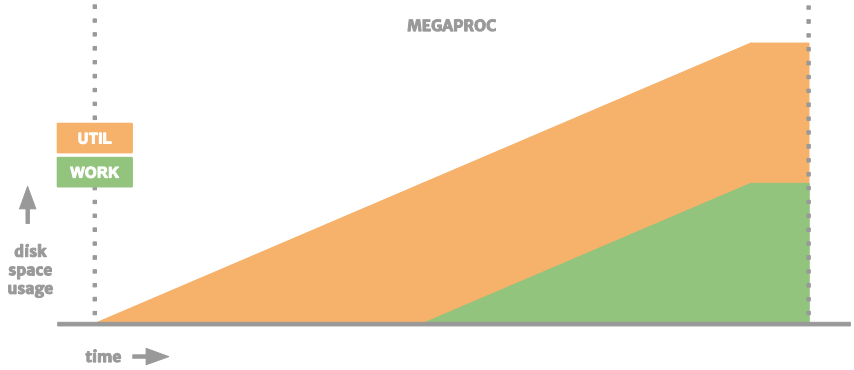

Figure 5: Pattern of Disk Space Usage for a *Type A* program

This *Type A* program creates one large Utility file, then with another pass reads that Utility file and creates a similarly sized WORK dataset, then reads the WORK dataset, writes its results out and finishes. Fairly typical of a job that performs a merge. In comparison, Figure 6 describes the pattern of disk usage for a Type B program.

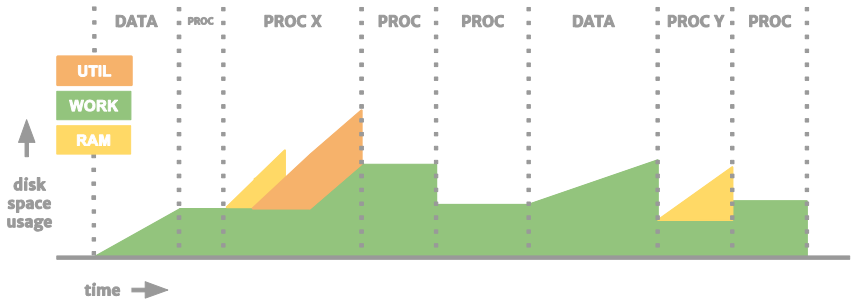

Figure 6: Pattern of Disk Space Usage for a Type B program

This program runs for a similar length of time but processes its data in much smaller chunks. Although it's *I/O cost* is the same as the Type A program, it is clear that the *total amount of space* it requires throughout its duration can be much smaller. It is also more likely to make multiple passes over the same dataset.

Utility files are used by some SAS procedures to store temporary working data. SAS may also sometimes try to store this temporary data in RAM (Memory) to avoid the I/O cost associated with using the disk-based Utility location. For procedures like the SORT Procedure, whether that procedure ends up using RAM or the Utility disk depends on the size of the input dataset: if the dataset can fit into memory, then it can be stored there and the I/O cost of that intermediate data movement avoided. In Figure 6, the temporary data for PROC Y fits into available memory. If it didn't, it would be written to the Utility location, and re-read on the next pass, like in the case of PROC X.

Philosophically, the way developers utilise the SASWORK library to store temporary datasets during the execution of their program is not too different to the way that those SAS Procedures utilise the disk-based Utility location as their temporary storage during their own execution. What enables SAS Procedures to sometimes use Memory instead of disk is the fact that they are atomic, and mostly aware of their temporary storage requirements. They also have the luxury of being able to *attempt* loading their data into memory and then failing gracefully if they hit their limit, knowing that they have a much larger Utility disk to fall back on.

SAS Programs don't have that luxury with SASWORK. They execute one step at a time and have no way of predicting how much temporary space they will require before they complete. There is no graceful fall-back for an 'out of disk space' condition, as the disk is the largest storage device available. This is why disks for intermediate SASWORK storage have to be generously provisioned, and also why we can even then occasionally run out of space on those disks.

SAS Programs also act entirely independently of each other, and regardless of the size of the data, every program will attempt to read from the disk at a rate dictated by the complexity of the step it is currently executing. Simply put, this means that the more programs are using a disk in parallel, the more demand is put on that disk. When disk controllers start to approach the point of saturation, their performance can often start to degrade in a non-linear way, and small improvements can often make a big difference.

THE ROLE OF MEMORY IN A SAS DEPLOYMENT

If a SAS Procedure knows how much temporary space it requires by looking at the size of its input datasets, and it can check that it has enough Memory available to cover that need, it should be able to reserve it for temporary storage and use it in place of the Utility disk, avoiding the unnecessary I/O cost of writing to and reading from disk. Of course, the data will still have to be written to and read from that Memory; in some cases (e.g. internal sort), the amount of data actually written to memory is slightly greater than would have been written to the Utility disk. Regardless, the speeds at which RAM is capable of reading and writing data compared to a block device is so much greater, that this relative I/O cost of writing data to memory is, for all intents and purposes, a non-issue.

The latest generation SAS® High Performance products take full advantage of these speeds to deliver a super-fast in-memory analytics capability. Being able to read data from memory at a faster rate than it can be processed means that SAS is able to fully engage all of the available CPUs in parallel, computing at an unprecedented rate and ensuring full use of the CPU resources that have been provisioned.

To make this possible, these new environments require a large amount of Memory to hold that unprocessed input data. As a result, the hardware that SAS is deployed on is often configured with as much RAM as it can take. Doing this makes sense: memory is getting cheaper, and the more of it is available, the greater the size of data that can be analysed using those in-memory techniques, meaning more data sources can be exploited.

Figure 7 describes the typical memory usage of a modern SAS deployment over a 24-hour period.

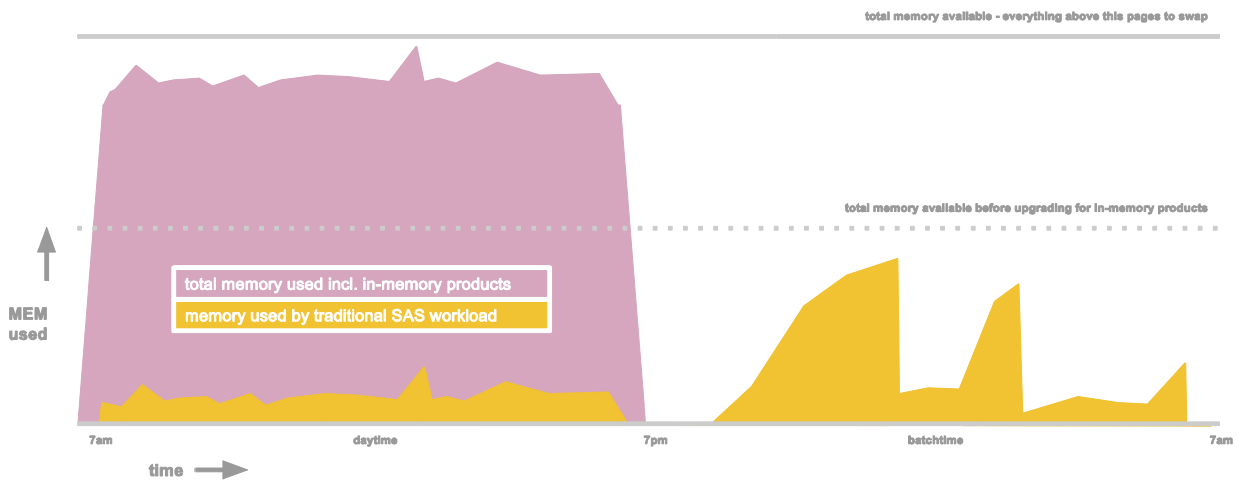

Figure 7: Typical Memory Usage of a modern SAS Environment over a 24 hour period

An increasing number of modern SAS environments are used for high performance in-memory analytics during the day, and traditional SAS Batch processing through the night. This works well, but a lot of that

super-fast RAM is often left underutilised during that batch window; in addition, there has been some discussion about whether the way Linux uses the free memory as filesystem cache can sometimes actually be detrimental to the SAS program performance.

This paper suggests the partial repurposing of that unused super-fast storage so that a certain proportion of the traditional SAS batch workload can utilise it for temporary storage in place of the SASWORK disks, in the hope of reducing the total I/O cost of the batch window as a whole. The rest of this paper discusses the ways of achieving that goal.

REPURPOSING MEMORY

Most modern operating systems provide a method of allocating a block of free memory and making it accessible via the filesystem. Linux offers a choice of memory-backed filesystems; any reasonably modern kernel providing the following options:

1. **RAMFS**: Bottomless. In some ways similar to SASFILE. Write too much data and the system will run out of memory and the Linux OOM process killer will start terminating programs.
2. **TMPFS**: Successor to RAMFS. Can be limited in size and the size changed on the fly. If system starts running low on memory it will page its contents out to swap.
3. **Kernel Module (brd)**: Space presented as a fixed size block device, meaning a filesystem has to be built on it, naturally hard limited on available space.

Of these choices, TMPFS is best suited for our purposes. It requires no formatting, can be sized on the fly, and has a safety net it in the form of a swap partition. A mounted TMPFS 'partition' will appear as just another directory on the disk, and any contents written to that directory will be stored directly in virtual memory. This makes it possible to use TMPFS as a means of letting SAS address that spare memory as if it was just another SASWORK location - albeit superfast one, with *zero effective I/O cost.* Figure 8 shows the portion of batch window memory that would be repurposed for TMPFS.

As touched on previously, SAS programs have no way of predicting how much SASWORK space they will need, nor how much space they will have available. Compared to disk-based temporary storage, the amount of RAM available is considerably smaller, and instructing all programs in a batch to use a smaller SASWORK location will almost certainly cause the batch to fail (very quickly). The challenge is therefore to operate the Batch in such a way that this unused memory is used safely. One way of doing this is letting only a select number of suitable SAS programs use it.

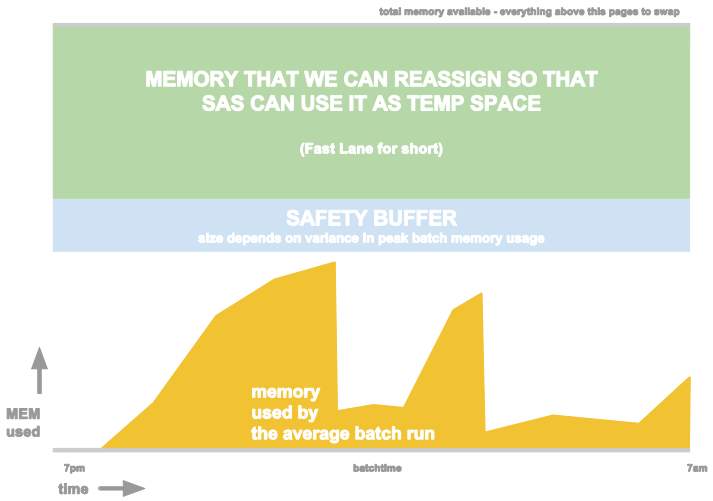

Figure 8: Repurposed Memory Capacity during the Batch Window

Although SAS programs themselves cannot predict how much temporary space they will use without actually using it, it should be possible to predict their temporary usage requirements given that program code deployed to run in a batch generally executes on a regular basis, unchanged. As long as sufficient data has been collected, it should be possible to make a reasonably accurate prediction as to how much temporary space each job in a batch is likely to use. Once this attribute is known, the most predictable jobs with small enough requirements can be reassigned to use the new 'Fast Lane' SASWORK, while the rest can continue to run as normal. Affected or not, all jobs should see a marked improvement in performance, as long as it is the I/O subsystem of the environment that is the performance bottleneck.

From here on in the paper will refer to this bit of memory that has been reassigned as TMPFS as the 'Fast Lane'. It's just easier than referring to it as the bit of memory that has been reassigned as TMPFS.

OPTIMISING A BATCH WORKLOAD

CANDIDATE SELECTION

Earlier this paper contrasted two similar SAS programs, *Type A* and *Type B*, both requiring similar CPU and I/O cost: *Type A* ran a single SAS Procedure and created one big temporary file, and *Type B* took the same amount of time and resource to run but used less temporary space. Most jobs fall somewhere in between in these two examples on the resource usage spectrum; since a typical batch window can execute tens of thousands of SAS programs a night, most environments should see a good spread of short big-temp-file to long small-temp-file programs. That spread of programs and their temporary storage requirements is illustrated by Figure 9:

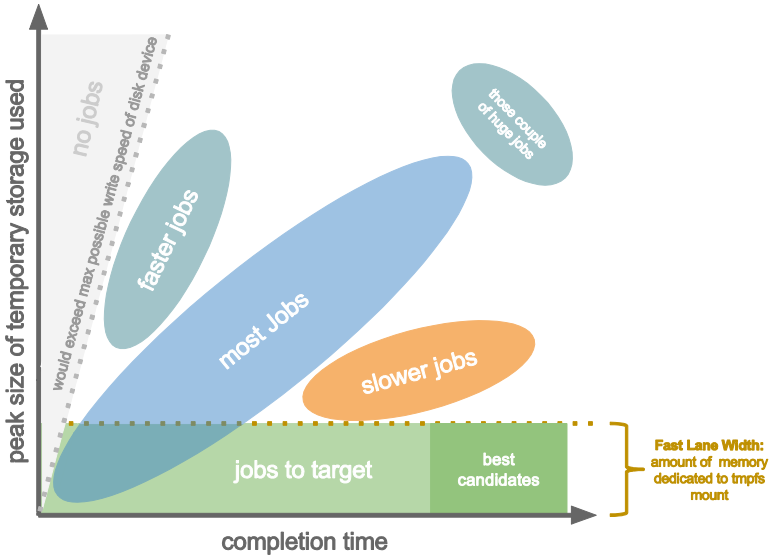

Figure 9: Spread of Temporary Storage Requirements and Program Completion Time

No jobs will land in the leftmost segment of the graph, as writing that much storage in that time would be beyond the peak throughput capacity of the disks. Close to that edge would be the performant jobs that get good disk throughput, use up a relatively large amount of space and complete quickly. The majority of jobs would land somewhere in the middle; then towards the X axis would be the slower jobs, the ones that seem to just take a long time. Then, there will be the jobs whose temporary storage requirements would be satisfied by the capacity of the Fast Lane.

In order to be able to make an accurate enough prediction of how much temporary space a SAS program is going to use the next time it runs, it is very important to measure its performance over a sufficient number of executions to deduce its pattern of disk usage. Batch runs can vary considerably, and the amount of temporary storage a job needed to complete its run yesterday is likely to be different today; whether it is only slightly different or very different compared to other jobs most often depends on the job itself. Figure 10 illustrates this range in variance, showing the target characteristics of a candidate job:

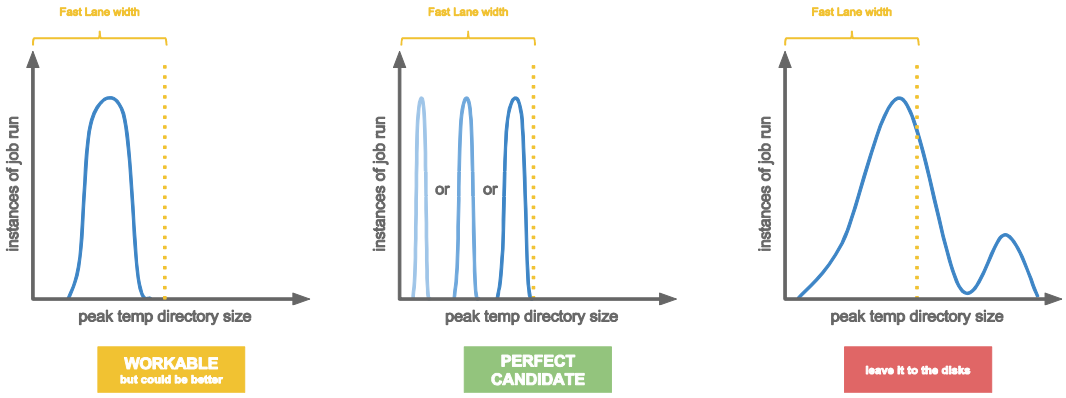

Figure 10: The Spread of Historical Temporary Space Requirements for Different Programs

Constant Job Profiling is paramount to the success of this technique. There will always be jobs that on some days complete in a few minutes and on some take over two hours, but with sufficient measurement, it should be possible to identify the jobs that process a similar amount of data and use a consistent amount of temporary space day-on-day. With enough data it should be easy to select those jobs that are most consistent in their temporary space usage, and use a sufficiently small amount of it that can be satisfied by the capacity of the Fast Lane.

WORKING WITH SCHEDULERS

Once the candidate jobs for the Fast Lane have been identified, they need to be instructed to do so without the need to make any changes in program code. We will assume that control of the *timing* at which the jobs in question are dispatched is outside the scope of this paper, and job triggering is a task that must be controlled by the existing scheduler. This was a major factor in the design of the solution presented in this paper.

The solution is represented in its simplest form in Figure 11:

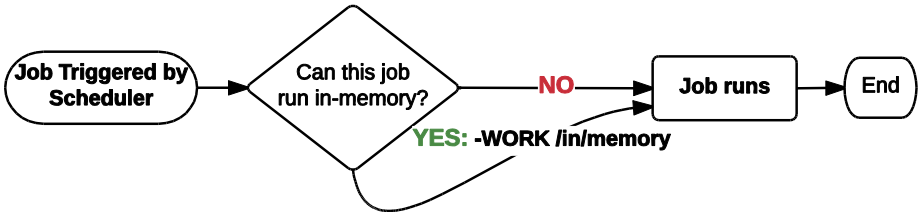

Figure 11 - High Level Solution

1. When a job is triggered, it is triggered via the **sasbatch.sh** script. This script is passed a number of parameters, one of which is the **name of the job** (input file), via the **-sysin** script parameter. That job name is extracted into a variable.
2. As the output of the statistical analysis of all batch jobs, an output '**cost file**' will be produced for each job that is considered to be a candidate for in-memory Fast Lane execution. Producing a separate cost file for each job makes it easier to source the cost config for the job being executed; instead of searching a lists, the script can simply source a file of the same name (or an md5 hash thereof) using the variable extracted in Step 1.
3. Once the cost value has been sourced from the cost file, it can be compared to the designated **limit** (the amount of available memory, the Fast Lane Width), and a decision can be made to either run the job in its standard configuration, or redirect it to the Fast Lane by means of a modified **-WORK** parameter such as **-WORK /in/memory** (pointing at the Fast Lane mount point)

The simpler technical solution to this problem involves a programmatic reorganisation of the batch queues performed as part of the (regularly scheduled) analysis which considers the size and variance of disk space usage by jobs on previous runs to identify suitable candidate jobs. Jobs are reorganised to run in their own Queues, against a separate Batch Server context configured to only run using the Fast Lane as temporary storage. Jobs are run sequentially, ensuring that the limit set for the Fast Lane is exceeded as infrequently as possible. This configuration is described in Figure 12.

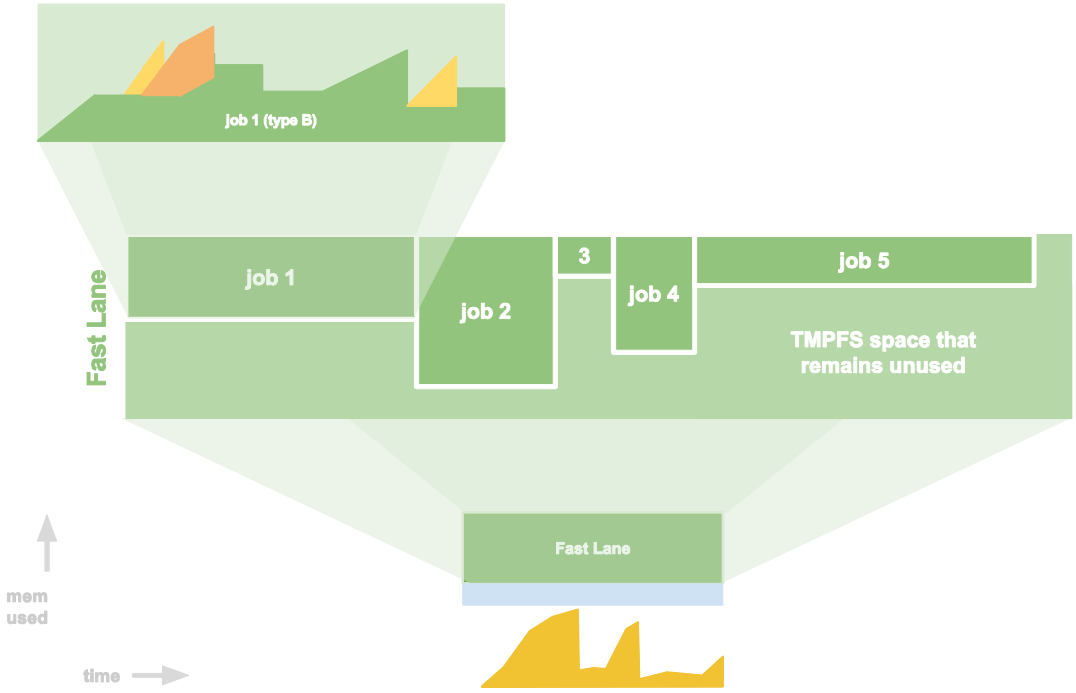

Figure 12: A 'Single-File' Fast Lane

One drawback of this approach is that the candidate jobs have to ideally be either *completely independent* of each other. Alternatively, the dependencies can be honoured by scheduling each of the jobs to run in the Fast Lane queue in sequence; however, including jobs for their relationship to other jobs rather than their own performance pattern is likely to result in some underutilisation of the Fast Lane. This dependency element can complicate candidate selection, but is a non-issue in situations where the majority of jobs are independent and scheduled by individual users with a scheduler such as *cron*. Depending on available server capacity and the volume of candidate jobs available, it may even make sense to increase the number of queues dedicated to the Fast Lane to enable a higher number of jobs to be run in parallel.

With larger, more complicated flows however, the lack of proper dependency handling could disqualify a high proportion of potentially excellent candidates from using the Fast Lane. In these cases, another way of approaching this problem is the 'Fast Lane Load Balancer'.

THE 'FAST LANE LOAD BALANCER'

Rather than picking out candidate jobs and altering queue configurations on a regular basis, it is possible to make the decision around whether a job should use the Fast Lane on the fly as the job is triggered. This approach sounds more complex, but in reality requires far less modification of an existing SAS batch configuration; everything here can be achieved with some relatively simple modifications to the sasbatch.sh and sasbatch_usermods.sh initialisation scripts.

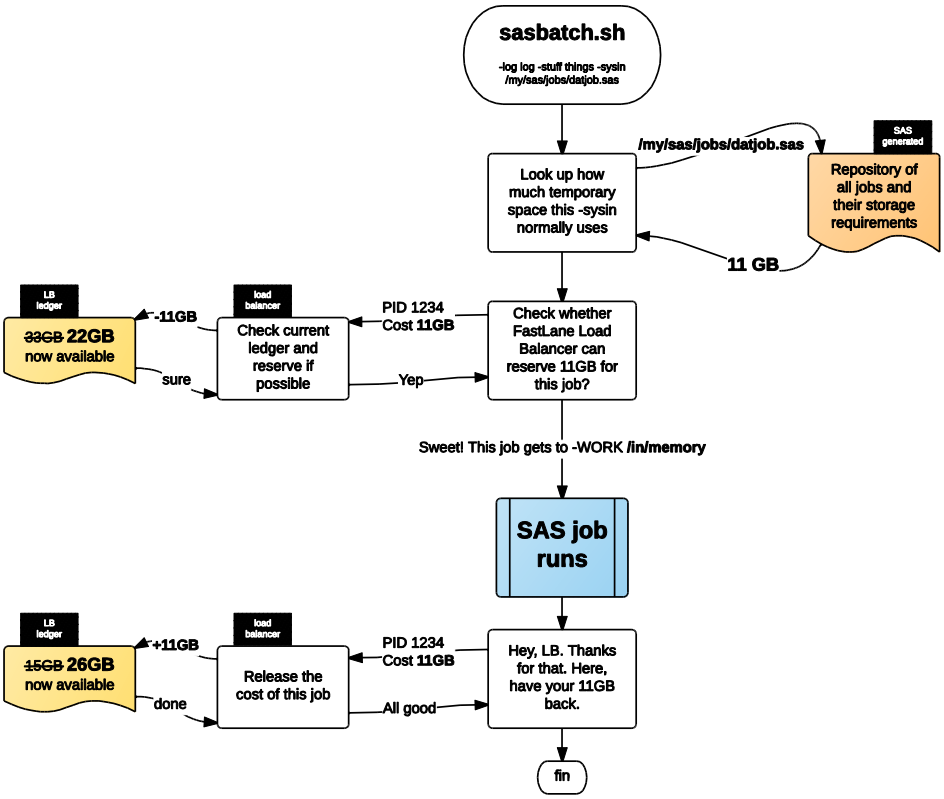

Figure 13: Execution Flow when using the Fast Lane Load Balancer configuration

This is how the Load Balancer idea works:

1. **sasbatch.sh** begins execution, the **name of the job** is sourced from the **-sysin** parameter, the **cost** of that job is loaded and it is compared to the current **limit** (the Fast Lane *width*)
2. sasbatch.sh issues a ***reservation request*** to the LB for the amount of temporary space the candidate job is expected to consume (that job's *cost*). It does this by writing a **request file** to a **landing area** to be processed by the LB for that node. sasbatch.sh then **sleeps** (~2s).
3. The Load Balancer runs an update routine at a regular interval (default 500ms) as follows:
    a. First the LB processes the 'release request' trigger files, updates the **ledger** and clears down the release trigger landing area (more in step 6 below)
    b. Then, one at a time it processes all reservation requests that have been made since the last interval. For each request, it does the following:
        i. Looks at the request **cost**, looks at **ledger** and looks at currently set **limit**.
        ii. If there is enough availability to satisfy the request, increases ledger by request amount and writes a result file instructing the use of the Fast Lane -WORK location
        iii. If the ledger cannot accommodate the request it is **ignored**
        iv. The request trigger file is **deleted**
4. After 2 seconds, sasbatch.sh wakes up and checks whether a result file has been produced by the LB. If so, -WORK /in/memory/work/${pid} and -UTILLOC /in/memory/util/${pid} are appended to the **USERMODS_OPTIONS** array. If no result file is found, no changes are made to the job parameters.
5. **The job runs**.
6. After the job has completed, sasbatch.sh makes sure that the /in/memory/${pid} directories have been cleaned up. It then issues a *release request* to the LB, letting it release the temporary space that it had reserved for the execution of the job that has just finished executing, before exiting.

7. As part of the first half of the ledger update routine outlined above, on each iteration the LB collects each of the release request trigger files created as and subtracts their reservation value from the ledger so that the capacity can be reserved by future candidate jobs.

The resulting Fast Lane utilisation pattern is shown in Figure 14. This movement of the LB Reservation Ledger is also detailed throughout the duration, explaining how the arrangement of jobs occurs.

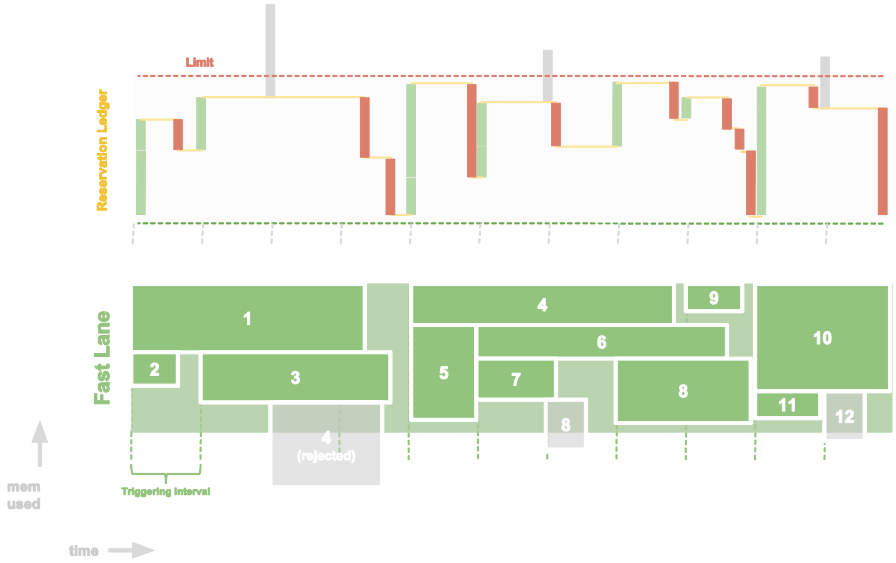

Figure 14: Fast Lane Utilisation with Load Balancer and Ledger

PREREQUISITES AND CONSIDERATIONS

For an environment to benefit from the optimisation techniques discussed in this paper, the following must apply:

1. The environment in question must be 'disk-bound'. Low CPU utilisation and high iowait can be symptomatic of a disk-bound system, but this should be confirmed with detailed monitoring.
2. The environment must have enough available Memory (RAM) throughout its Batch window to support moving some temporary I/O activity to RAM. or there must be scope for reducing the environment's use of RAM
3. The batch must have a sufficient number of programs scheduled which meet the criteria of consistently creating temporary directories small enough to fit into the available Memory.

The usage pattern that can be expected when the Fast Lane Load Balancer is operating correctly is shown in Figure 15.

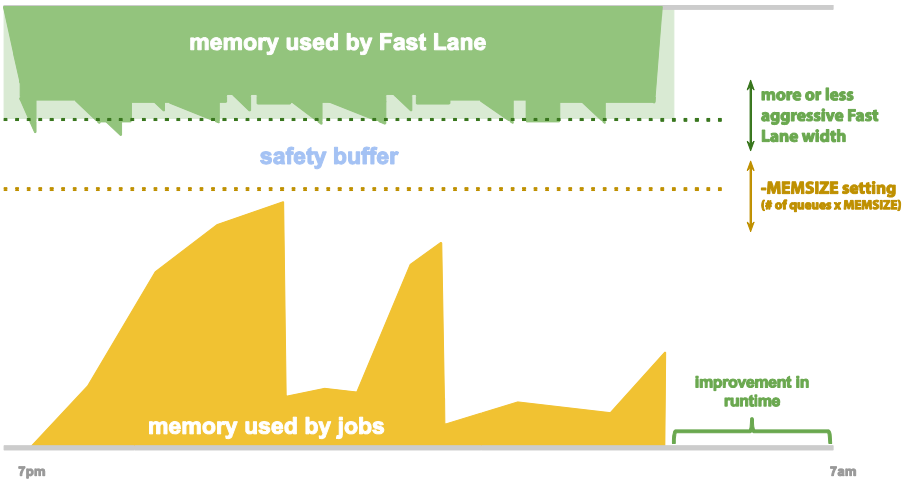

Figure 15: Typical Memory Usage Pattern after implementation of Fast Lane Load Balancer

Figure 15 also shows the parameters that can be altered to 'tune' the approach for best results. Specifically, *reducing* the amount of memory available to SAS can sometimes have a positive effect, for these reasons:

- For as long as the Scheduler restricts how many jobs are run in parallel (i.e. using lanes), limiting MEMSIZE caps the overall usage of memory by SAS and allows for the Fast Lane width to be increased safely with a reduced safety buffer. This allows memory to be used for SASWORK as well as in place of Utility file storage. Depending on where the current Fast Lane width sits in relation to the average temporary directory size for all jobs, increasing the selection threshold can increase the number of eligible candidate jobs considerably.
- Some SAS procedures can be inefficient with how they use the memory allocated to them. For example, PROC SORT will always *attempt* to use Memory made available to it by loading as much data as will fit up until the SORTSIZE limit is hit. When that limit is hit, it will use the disk no differently to how it would if the memory made available to it was much lower. This can be detrimental to performance both because the CPU time spent trying to load that data into memory is wasted, and because the memory that it reserves for the duration of the step could be better used elsewhere, i.e. by other programs in the Fast Lane.

GENERAL SUGGESTIONS

The following are some things to keep in mind when trying to implement this technique.

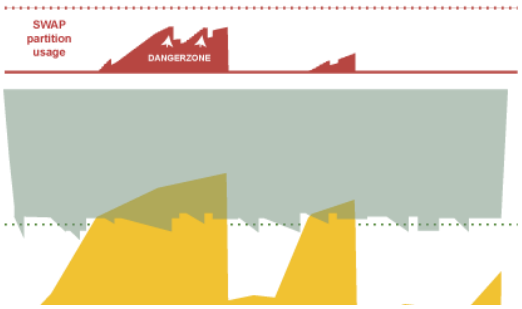

Figure 16: reduced performance upon collision of Fast Lane and conventionally used memory

Constant Monitoring is Important

The periodic sizing of SASWORK and Utility directories is fundamental to this method. However, constantly measuring the size of a large number of directories with full of active files can take its toll on performance, particularly on very busy block devices. If the impact is severe, consider reducing the interval at which the directories are sized. The TMPFS directories can and should be sized more frequently than disk-based ones as the performance hit on them is considerably smaller.

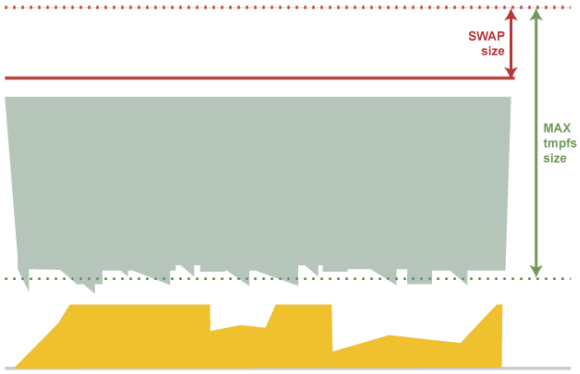

Figure 17: Reducing MEMSIZE to accommodate for a wider Fast Lane

It is OK to Stop Optimising

The aim of this exercise is to maximise CPU utilisation, not to always use all available memory. If your SAS environment was sized correctly, it should not take too much work to bring the CPU saturation back up. Once you have achieved the performance improvement you desire, it's ok to stop.

Maximise Fast Lane concurrency

SAS programs don't consider other programs running alongside them, and every program that runs will demand its own I/O throughput. If the number of concurrently running programs is managed by the scheduler by means of a fixed number of lanes, then redirecting as many concurrent programs as possible to run in the Fast Lane is likely to have the greatest effect in reducing the I/O demand from the disks.

Check for rejected candidates

If you find a large number of your candidates jobs are being rejected by the LB because the ledger is full, consider either increasing the amount of memory you dedicate to the Fast Lane, or slightly lowering your candidate selection threshold. Theoretically, rejecting a smaller job in favour of a larger one is bad because it lowers the possible concurrency (see 'Maximise Concurrency' above, also Figure 18)

Consider reducing *swappiness*

In modern Linux kernels, *swappiness* is a virtual memory configuration parameter that dictates the tendency of the kernel to page VM out to the swap partition. Having a swappiness of 10 means that the kernel will begin paging to swap once the amount of free memory falls to 10%. A swappiness of 3 lowers this threshold to 3%. Lowering swappiness will allow for a more aggressive safety buffer.

Increase Swap space

And put your Swap partition on a fast, direct attached Solid State disk. If you already have such disks for SASWORK, consider cannibalising some of it and repartitioning as SWAP space. Your disk-based SASWORK requirements will be reduced by the Fast Lane, and having a fast swap partition will minimise the performance hit when an overlap occurs (see Figure 16).

Install more RAM

If your machines have the spare capacity it makes sense, as this technique can take advantage of it.

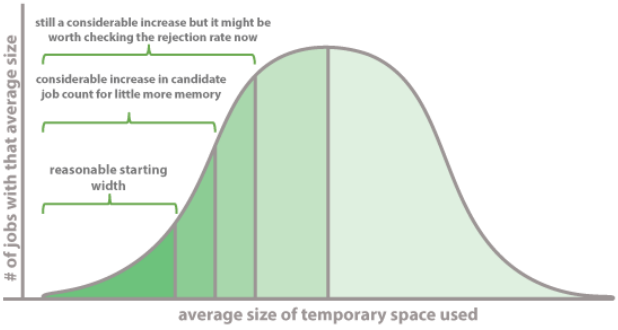

Figure 18: When to stop increasing Lane Width / Candidate Threshold

TEST RESULTS

With any optimisation technique, the effectiveness of the results depends on the state of the jobs being optimised. All environments differ and no two SAS workloads are ever the same. Every simulation will be flawed and any published optimisation results should therefore always be considered subjective. With that in mind, here is an oversimplified demonstration of our optimisation principle, just to show it working:

Sample Workload:

One 'Big Job' sorts a 60GB dataset twice, while 10 'Little Jobs' load a 6GB dataset into their SASWORK and sort it in opposite directions 10 times over. This test is simply trying to simulate the iterative behaviour of many small user-written programs – typically a series of data steps reading and writing mostly the same data – and contrast the use of the same free memory as OS Cache versus the Fast Lane.

The following two scenarios are shown:

1. All jobs run in parallel allowing free memory to be used as OS cache for SASWORK and UTIL directories (standard approach)
2. All jobs run in parallel, but the Load Balancer directs the smaller jobs to use the Fast Lane for SASWORK and UTIL directories, allocating the free memory to this instead of allowing the OS to use it as cache (optimised)

Test hardware: 24 Xeon X7542 CPUs (no HT), 256GB RAM, Fast Lane TMPFS partition size set to 300GB, 1TB Direct Attached SSD array capable of 1800MB/s throughput, split 50/50 as SASWORK / SWAP. The cache was cleared before each run to ensure a fair test.

Elapsed Time Comparisons:

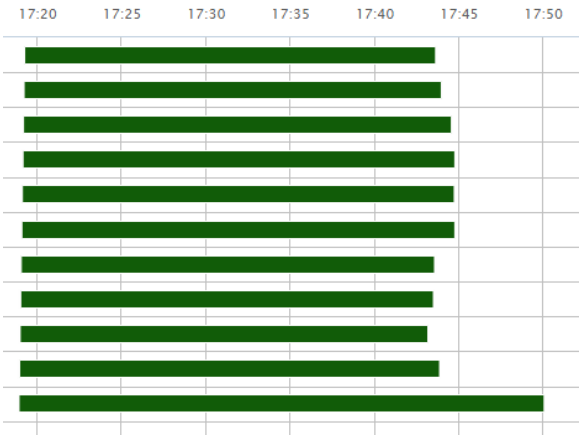

Figure 20: Runtime Pattern for Scenario 1 (normal)

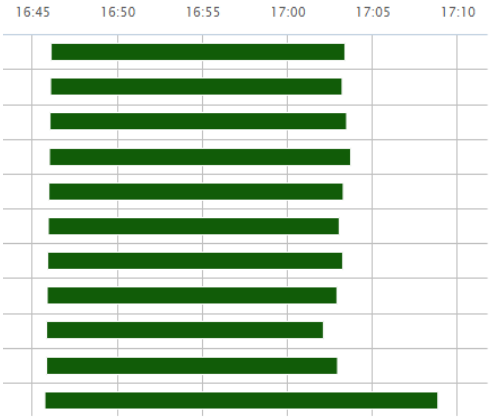

Figure 19: Runtime Pattern for Scenario 2 (optimised)

In the first run, the Big Job took 31m8s to complete. When the smaller jobs were redirected to the Fast Lane, the Big Job completed in 23m6s. The average time for a smaller job went from 25 to 17 minutes.

Resource Usage Comparisons:

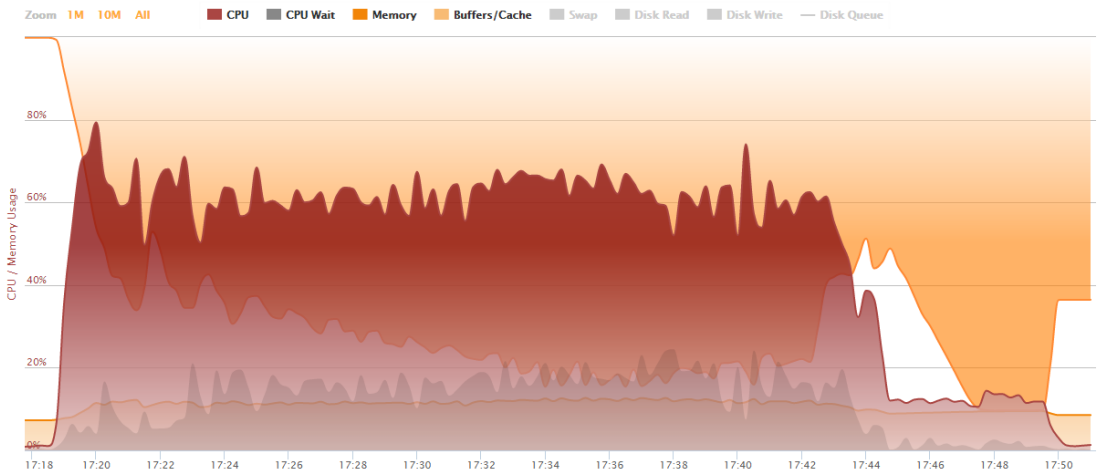

Figure 21: Server Resource Profile when using the Operating System cache (Scenario A)

The graphs on this page can be interpreted as follows:

- The Dark Red area is the overall CPU usage
- The Grey portion of that area is the portion of used time that the CPU spends waiting for IO.
- The Orange area adjacent to the bottom X axis is the percentage of memory in use by programs (SAS)
- The Orange area adjacent to the top X axis is the percentage of memory allocated by the kernel to be used as filesystem cache and buffers (this includes our Fast Lane)

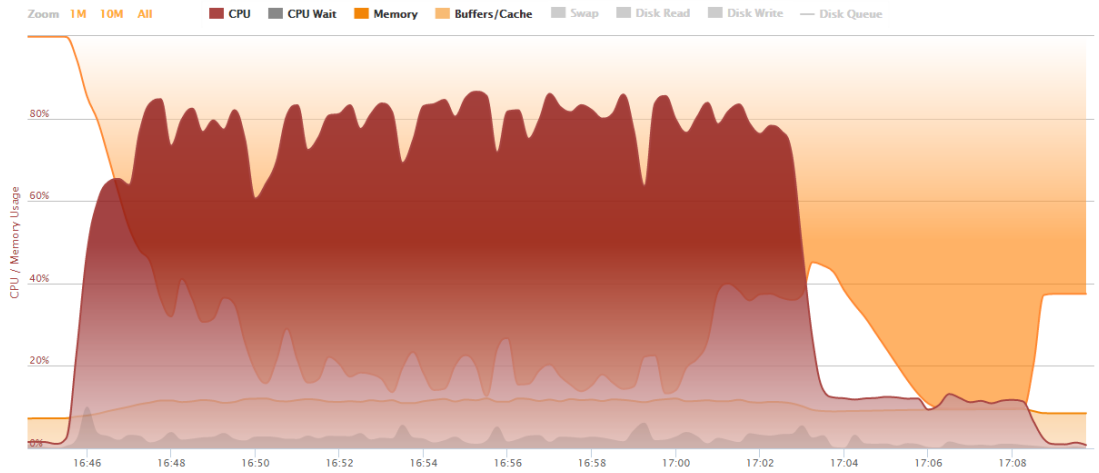

Figure 22: Server Resource Profile when using Fast Lane technique (Scenario B)

It is interesting to see that roughly the same amount of total memory, around 70-80%, is used for caching the temporary files that are being created or used by these SAS jobs. However, the OS cache requires constant management, iowait time is higher, disks are still busier and CPU saturation lower, leading to overall less efficient processing.

Disk activity comparisons:

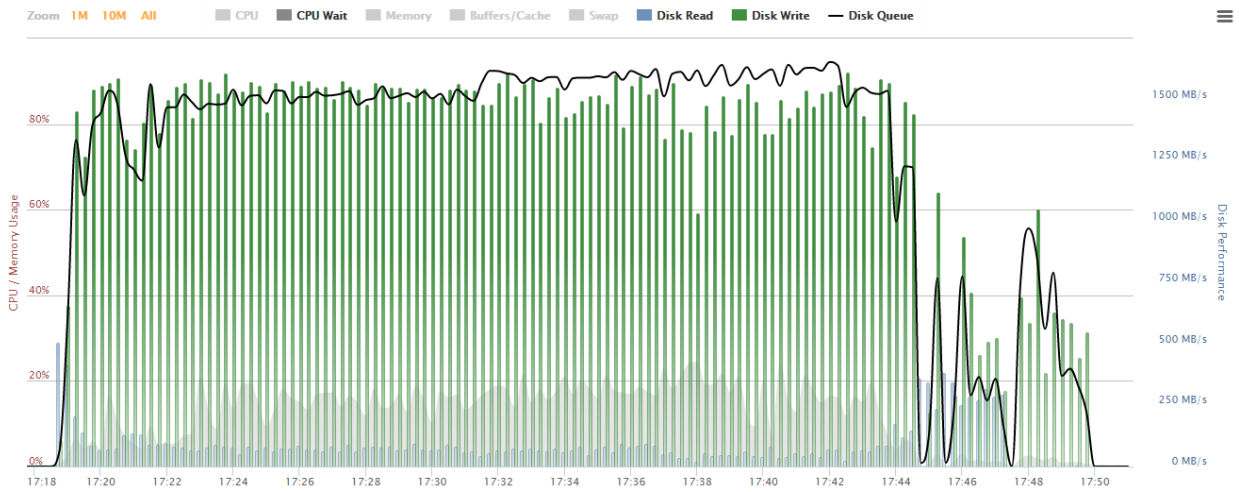

Figure 23: Disk activity profile when using the Operating System cache (Scenario A)

Interpreting the graphs:

- Green bars are Disk Write speed; Blue bars are Disk Read speed
- The black line shows the Disk Queue length, an indicator of congestion
- The grey area is CPU iowait

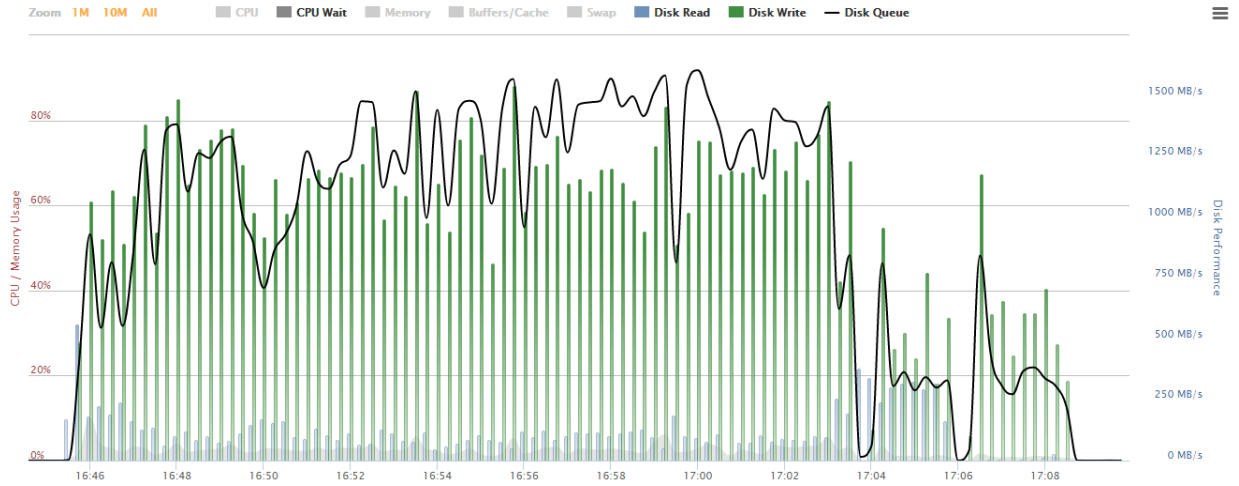

Figure 24: Disk activity profile when using the Fast Lane optimisation (Scenario B)

Figure 23 and Figure 24 compare the different Disk usage patterns between the two scenarios. It is clear that the disks are at their limit when running the jobs in parallel as normal, even with the memory in use by the OS cache. During the optimised run, the iowait is lower and the throughput less flat-out, suggesting that there is some throughput capacity still available from the disk.

CONCLUSION

Our findings suggest that the behaviour of many scheduled batch jobs can be very predictable, enabling an analytics-driven approach to job orchestration and optimisation through some form of predictive modelling. As this paper also shows, an often overlooked feature of SAS is the ability to parametrically control an array of system parameters for each individual program or session as it is launched; the parameters that result in the best performance may vary depending on the current state of the node they are executing on, and doing what is best for an individual job may not drive the best performance for the batch window as a whole. This suggests that a deeper level of integration between the retrospective analysis, a real-time monitoring component and the job triggering mechanism should yield a substantial improvement in overall performance.

At Boemska, our real-time monitoring product Enterprise Session Monitor™ for SAS® already collects detailed performance data for each individual job, including the amount of temporary storage used by a job at every point throughout its execution. It was the availability of this detailed historical data, otherwise only used for chargeback calculations, that led us to look at using past performance and behaviour as a basis for optimisation. This not only led to the technique presented in this paper, but also eventually led us to start working on a new job orchestrator product - an intelligent scheduler that relies on past performance to make better decisions about how jobs should be run, with the ability to reconfigure individual jobs on the fly depending on the current measured state of the system. Some form of the Fast Lane technique described in this paper is very likely to play a large part in the function of that product.

In the meantime, all of the scripts, components and best practice guidelines required for the implementation of the Fast Lane technique discussed here will be hosted as a community project on github.com/boemska/fastlane

CONTACT INFORMATION

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

Nikola Marković
Boemska
22 Upper Ground
London, SE1 9PD
United Kingdom
+44 (0) 20 3642 4643
nik@boemskats.com
http://boemskats.com

Greg Nelson
ThotWave Technologies, LLC
1289 Fordham Boulevard #241
Chapel Hill, NC 27514
United States of America
(800) 584 2819
greg@thotwave.com
http://www.thotwave.com