The Hidden Cost of Large Jobs
Drain Time Analysis at Scale

Joseph 'Joshi' Fullop IV
National Center for Supercomputing Applications (NCSA)
University of Illinois at Urbana-Champaign
Urbana, IL, USA
Email: fullop@illinois.edu

***Abstract*—At supercomputing centers where many users submit jobs of various sizes, scheduling efficiency is the key to maximizing system utilization. With the capability of running jobs on massive numbers of nodes being the hallmark of large clusters, draining sufficient nodes in order to launch those jobs can severely impact the throughput of these systems. While these principles apply to any sized cluster, the idle node-hours due to drain on the scale of today's systems warrants attention. In this paper we provide methods of accounting for system-wide drain time as well as how to attribute drain time to a specific job. Having data like this allows for real evaluation of scheduling policies and their effect on node occupancy. This type of measurement is also necessary to allow for backfill recovery analytics and enables other types of assessments.**

***Keywords*-drain time; scheduling; supercomputer; backfill;**

I. Introduction

In this paper we discuss different ways of measuring and accounting for the time and resources attributed to scheduler sequestering for the purpose of allowing jobs or reservations to run. We look at accounting for this drain in terms of the system as a whole, as well as attributing drain to individual jobs. We further delve into some of the analytics that can be done with this information and how it can be used to drive policy.

II. Defining Drain Time

'Drain Time' in its simplest form for a single node is the amount of time that node is held in reserve and prohibited from running workload in order to allow enough nodes to become available to start another job. Since jobs run across multiple nodes, the metric, more loosely referred to as 'Drain Time', is actually the sum of time spent by a set of nodes and is measured in node-hours(nHrs), or node-seconds(nSecs). This is in line with how jobs are measured in their utilization of a system and the capital in which an allocation is granted.

One specific differentiation that must be made is between this definition and the terminology that schedulers and batch systems will sometimes use to signify the state where a node is not currently able to accept additional workload due to administrative action[1].

III. System-Wide Drain Time Accounting

A. Available Data

On the Blue Waters supercomputer, the scheduler of choice is Moab by Adaptive Computing. One log line is generated for each node on every scheduler iteration. Each line contains a timestamp, the node identifier, the current state, a reservation list and a job list. Below is a condensed sample log line for clarity.

```
2014-12-31T00:03:19.696-0600 21166 INFO
Node '27453' status: state='Idle'
rsvlist='1292428,1292433' joblist='none'
```

The sheer volume of just these records amounts to over 1.7TB covering a period of around 2 years.

B. Accounting Method

The method of accounting is an accrual of time in seconds for each node into an appropriate state accumulator for the assessed time period. Since each log line has only the one timestamp, we will require a second log line for that node in order to determine the number of seconds spent in that state. Therefore the node will not accrue time until the next record is encountered.

1) Per Node Accumulators: This consists of a multi-dimensional array, with the prime index being the node identification number. The secondary axis is the set of six possible states, which are {*Down, Idle, Busy, Running, Drained, and Draining*[1]}. The third axis is the number of combinations of the existence of jobs or reservations in the *rsvlist* and *joblist*. This is of course a constant of four possibilities.

To accomplish the task at hand, we will also need to store the last state information for each node so that the appropriate state accumulator can be incremented, once we determine how much time has passed between the scheduler iterations.

[1]Note - not to be confused with our definition of 'Drain'. See Adaptive Computing's documentation[1] for further differentiation.

2) Drain State Determination: Our data structure allows us to account for all possible combinations of states that we have identified. However, for the purposes of this paper, we are only concerned with the time spent draining. Other accounting tasks can easily be accomplished with this as well. Down time assessments or node-occupancy (*i.e. utilization*) accounting are a couple of examples.

The identification of a drain state is simply the combination of when the *state* is 'Idle' and there exists a reservation list. The reservations are presented in sorted order of upcoming occurrence. Therefore, the first in the list will be the next job or reservation to take place. This is important for drain accounting for jobs, discussed later.

3) Aggregation and Basis: Aggregation can be done across any vector of the multi-dimensional array. We can determine the drain time for the entire system by summing the appropriate accumulators for all nodes. This will give us the total node-seconds the system spent in drain as determined by the available data. However, we must also have some level of perspective. We will need to determine the basis, or how much time does the data cover, and how many nodes are relevant. Since a scheduling iteration outputs information for every node on each cycle, summing the total seconds accounted for any one of the nodes should represent the time basis. And furthermore, all nodes should have the same sum. The basis for the number of nodes could be static. Here you can use the number of nodes in the system, or at least the number of usable nodes. The node-hours basis would be

$$((basis\_seconds) * (basis\_nodes))/(3600 seconds/hour)$$

In the case of Blue Waters, it has 26,846 schedulable nodes[2]. This gives us a basis of

$$(86,400 * 26,846)/3600 = 644,304$$

node-hours per day potentially available.

Of course, the nodes basis could be dynamic. If you are interested in framing the percentage in terms of available node-hours instead of theoretically possible, you could sum all non-down states instead of multiplying basis nodes by basis seconds. However for the following analyses, we use a static basis.

The seconds basis calculation should also be used to determine if you have a quality set of data. Blindly using a percentage of drain time over basis time could be dangerous as you may not know if you only have data for a portion of the day. The seconds accounted for should be approximately 86,400 for a day.

One of the practicalities of storing log data is breaking it into files by date. This does have an implication, especially in terms of determining basis. Since two scheduler iteration times are necessary to determine accumulation seconds, a day log file will provide all but an average of one cycle per day. One cycle will be lost, since the last, partial cycle will not accumulate because the next cycle will be in the next daily log file. Additionally, iterations are not in sync with the day cycle as it might be if it were a cron job. So the seconds from midnight to the first iteration of the day will also be unaccounted for. The shortage is the length of one scheduler iteration on average if data is stored and assessed in this manner. Given that scheduler iterations are usually on the order of a couple of minutes, the basis shortage is generally negligible in the scope of a day.

Where sanity checking the basis seconds really comes in handy is when you have a maintenance period that spans the midnight barrier. If the scheduler was stopped the day before and not resumed until sometime the next day, there would be a notable skew in the seconds basis. If percentages are calculated on a purely static number of node-hours for the day, the percentage would error on the low side. Of course, we could just use and compare total node-hours, but normalizing to a daily percentage is far more approachable and understandable for a broad audience.

C. Benefits of Information Availability

Having a daily number of node-hours or percentage of time spent draining is a great first step and will allow for a number of capabilities. First, the ability to track and visualize leads to having a better feel for what is a normal level of drain time for a particular system. The variability will also lend to determining how sensitive that system is to the submitted workload. Submitted workload can significantly impact the amount of drain time experienced by a system. A mix of sporadic, high-priority, large jobs amid small, long-running jobs would likely have a greater drain time than a mix with regular stacks of same-sized jobs.

This method of accounting is good for determining the impact of systemic changes on the overall system utilization and scheduling efficiency. Changing a scheduling policy for a period of time can now be compared to previous policies in terms of the drain overhead.

D. Shortcomings of the System-wide Method

While the system-wide drain time accounting method is useful for seeing things at a high level, it is insufficient for determining how policy or other changes may be affecting the user experience. Drain time directly relates to the time a job spends in the queue, and therefore how long a user has to wait for their job to launch. It is further inadequate for determining how policies affect different job classes, whether that is by size, queue or requested wall time.

IV. Per-Job Drain Accounting

A. Accounting Method

The accounting method for the per-job method is built upon the per-node method. Once the number of seconds to accumulate has been determined for a node, if the state is determined to be draining, those second are added to the running total for the appropriate job. Recall that the first job or reservation in the *rsvlist* is the next one scheduled to run on that node and is therefore the job that is responsible for the drain.

1) Per Job Accumulators: The implementation of the per-job accumulators is an associative array where the jobid or reservation maps to a node-seconds accumulator.

2) Drain Across Day Barriers: Another of the issues with storing log data in day-partitioned files is that jobs can accumulate node-seconds of drain in multiple periods. To get the full accounting for a job, the fragments need to be grouped and summed on the job or reservation id.

3) Unallocated Idle Time: One of the special cases that shows up in the data is when the *rsvlist='none'*. The code will recognize the literal 'none' as the job or reservation id when the state is 'Idle'. When the state is Idle, and there are no reservations for that node, then there is no work for that node. This can be an indicator of inadequate submitted workload.

On a technical level, the jobid 'none' should be removed from drain time summaries since it is not drain time.

V. Analytics of Drain Time

In this section we will examine ways to use the drain time accounting and present some data from Blue Waters.

A. Refining Scheduler Policy

1) Identifying Oversight: Once a scheduling policy is decided upon and implemented, it is always a good practice to evaluate and determine whether those policies are having the desired effect. The time jobs spend in the queue or the node occupancy rate are often considered. Drain times could also be evaluated since they are directly related to node occupancy rates. More specifically, jobs that exhibit inordinately high drain time after policy changes should be scrutinized. These outliers are generally manifestations of edge cases that may have been overlooked in the definition of the scheduling policy.

2) Perpetually Sliding Jobs: Sometimes jobs can continually be bumped to make room for higher priority jobs. This is normal. Schedulers generally do a decent job of refactoring and getting a bumped job launched in due course. However, in reality, sometimes things do not work as intended. To identify perpetually sliding jobs, one can consider the set of jobs that are currently queued and sort them based on their ratio of accumulated drain time to job size. The top of the list should show those jobs that have accumulated the most drain time (normalized by job size). The threshold of concern will have to be established by what is considered 'normal' for any given system and workload.

B. Evaluating Scheduler Changes

Most operators of a supercomputer tend to strive for scheduling efficiency. Drain time is generally considered wasted time, or, at least, a source of inefficiency. Therefore, measuring this inefficiency can be used to determine relative efficiency. Simply considering a daily drain time value or percentage over an affected period can lead to quick conclusions about the impact of a scheduler modification.

Example Case: Topology Aware Scheduling

Figure 1 shows the percentage of drain time for four periods. From left to right, it shows an initial period of running a scheduler before the introduction of Topology Aware Scheduling (TAS), then the broad section of running under TAS, followed by an experimental period where TAS was turned off, then finally the re-introduction of TAS. Topology Aware Scheduling requires jobs to be placed on sets of nodes contiguous on the torus network and be of a certain shape. When scheduling is required to be done in these blocks, there is a greater propensity for an increase in drain time, resulting in a decreased node occupancy, given an unchanged workload. Performance improvements and an improved consistency are expected on individual jobs since they should be more isolated from interference by other workload. The extent of that speed-up can, and should, be measured and compared to the increased overhead of drain time costs to determine if such technologies present any real gain. At least this provides one factor that must be overcome to justify adoption. Measuring application performance improvements is a topic for another time.

The table below summarizes the drain time values for the stated periods.

Period	Average Drain Time (%)
Pre-TAS	9.1%
TAS	22.3%
Non-TAS Experiment	5.8%
Post Experiment TAS	21.3%

The Topology Aware Scheduling mode of operation, examined here, impacts the system in terms of increasing the time spent draining by 14.4% or more. This also directly detracts from node-occupancy (utilization).

C. The Real Value of Debugging Code

Drain losses are a practical part of getting work done. However, when the drain cost is sunk and no work materializes, then that is a particularly bad case that should be avoided, when possible. This occurs when jobs fail to launch or crash shortly after launch. These cases can be identified by job execution times being very short.

Using the node-hours ratio of $[DrainTime] : [RunTime]$, jobs that are responsible for a large drain and little work can easily be identified.

On Blue Waters, we examined a period of 1.4 years saw 13,500 jobs that accumulated drain time and then ran less that 30 seconds. These jobs generated 1.77 Million node-hours of drain. This is equivalent to 1.5 years of constant

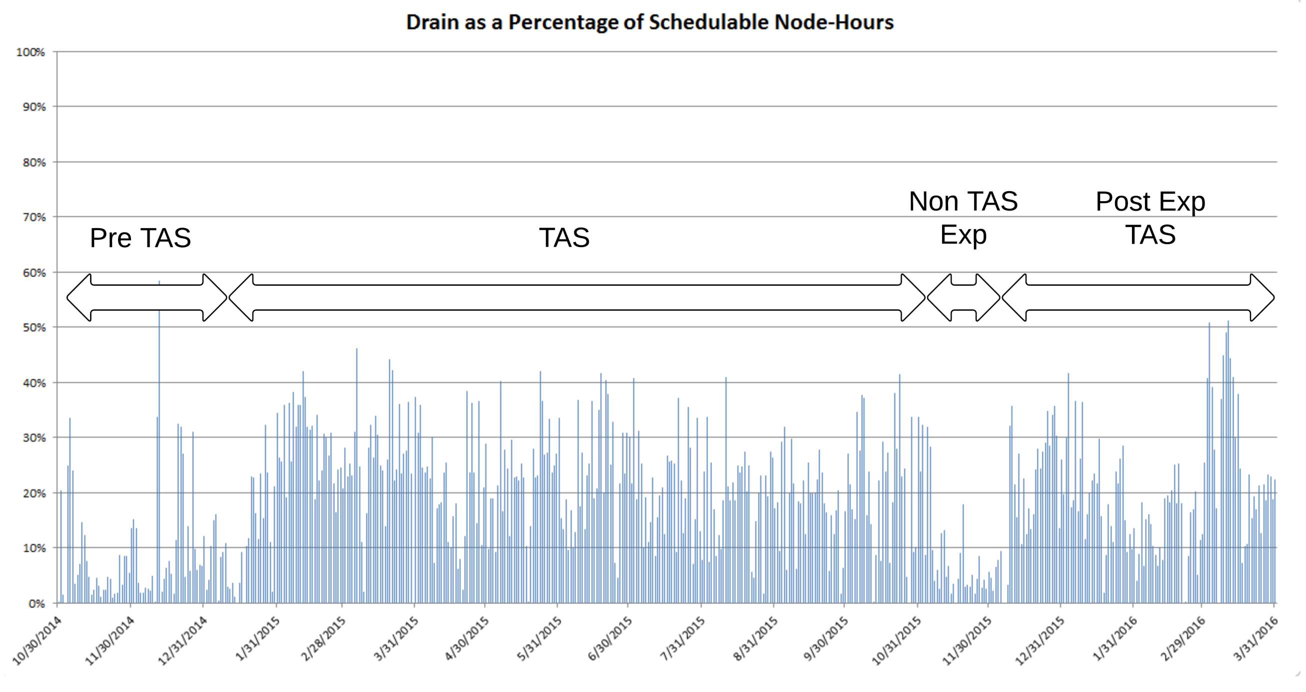


Figure 1. Daily Percentage of drain time across Topology Aware Scheduling periods.

run time on a 128 node machine of similar performance. These jobs can be categorized as 'failure to launch' types of jobs. Fortunately, this works out to average only about 0.5% of the daily machine capacity. NCSA's user services group actively engages users when these types of situations arise. But if ignored, it could easily grow as users continue to submit buggy or faulty workload.

D. Drain Time as a Function of Job Size

There exists a certain expectation that the bigger a job is in size, the more drain time it may require to launch. This growth should also be expected to be mitigated by the scheduler's ability to stack large jobs of similar size in succession such that they minimize the total drain time, much like race cars drafting to share the wind resistance.

In order to search for a trend, we take a look at the average drain time in node-seconds and how it relates to the size of a job. We split the job sizes up as follows and summarize for each size grouping. We also examine the four different periods separately to see if TAS has a discernible impact.

Group Name	Job Size (nodes)
Tiny	1-128
Sub1k	129-999
1k+	1,000-1,999
2k+	2,000-3,999
4k+	4,000-7,999
8k+	8,000-15,999
16k+	16,000+

Figure 2 shows a generally upward trend. However, it is by no means linear. The variability would also tend to suggest that there are other factors at play. Workload back-log is a usual suspect.

VI. Enabled Future Work Areas

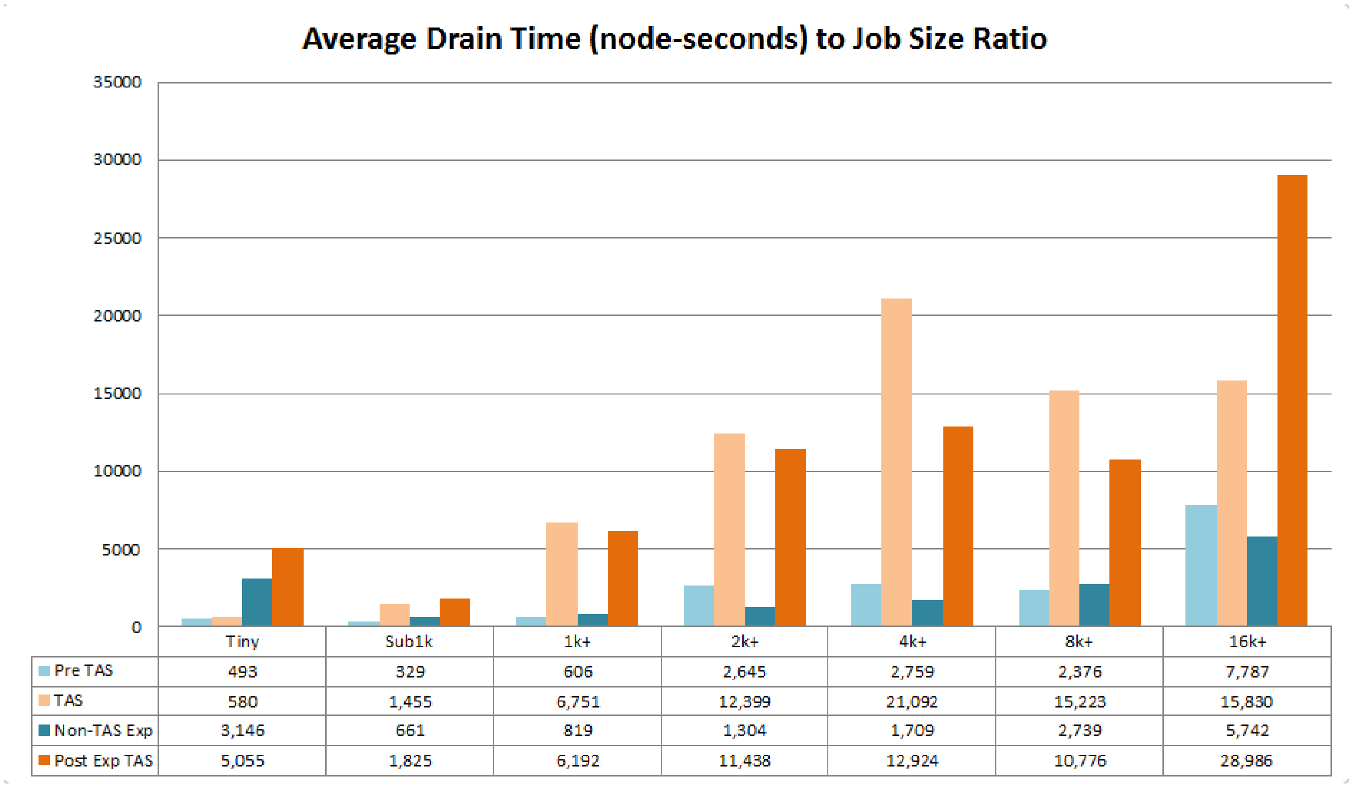
Having drain time numbers allows for further comparative analysis, as well as provides measurable evidence to warrant the pursuit of efficiency-improving efforts.

A. Back-fill Recovery Analysis

If it were possible to accurately determine that a job was run under a back-fill scenario, the total node-hours of back-fill jobs could be compared to the total node-hours of back-fill plus the total node hours of drain to determine back-fill efficiency.

B. Job Back-log Analysis

Using the unallocated idle time provided by the per-job drain for job 'none', one could begin to assess the available



	Tiny	Sub1k	1k+	2k+	4k+	8k+	16k+
🟦 Pre TAS	493	329	606	2,645	2,759	2,376	7,787
🟧 TAS	580	1,455	6,751	12,399	21,092	15,223	15,830
🟦 Non-TAS Exp	3,146	661	819	1,304	1,709	2,739	5,742
🟧 Post Exp TAS	5,055	1,825	6,192	11,438	12,924	10,776	28,986

Figure 2. Analysis of drain time as a function of size under different scheduling strategies

backlog of scheduled jobs. This can be used to determine if the lack of node-occupancy is due to scheduler inefficiencies or due to the lack of suitable workload. More classification of scheduled workload will be necessary as the unallocated idle time is only a component of what is needed for this task.

C. *Back-fill Dynamic Sub-scheduler*

An interesting idea to salvage drain time would be to dynamically schedule workload onto the nodes reserved for upcoming jobs. It is possible to determine what the currently largest back-fill job size is. Combine that with the time until the pending job launches and one could schedule a job to run in the shadow of upcoming workload. This would increase overall node-occupancy (utilization) for the system as a whole.

If the user community had workload that could be scheduled with a range of size and could run to termination, or at least make measured progress, the sub-scheduler could dynamically submit jobs on their behalf to take advantage of drained nodes. They could submit these parameters under which their code could execute and fund it from their existing allocation of node hours. Heavy discounts could also be used to entice usage of such a system.

An alternative use of this sub-scheduler could be to run benchmarking jobs with a fixed input deck. The run times of these benchmark jobs could be used to measure relative throughput of the system as a matter of practice. Doing this could provide vital data for performance analytics without displacing user workload to do so.

VII. Conclusion

The size of supercomputers today are large enough to warrant efforts to recover or take advantage of available cycles in the wake of large pending jobs. Innovative strategies in scheduler technologies also have the promise to improve system throughput. Before something like these efforts can be managed or properly evaluated, they must first be measured. We have provided two methods of measurement and some examples of their use.

Acknowledgment

The author would like to thank Mark Dalton of Cray and Rick McKay of Adaptive Computing for their verification of the interpretation of the log file information.

References

[1] 10.5 Managing Node State. *Moab Workload Manager*, Adaptive Computing., Web. 7 Apr. 2016.

[2] ”System Summary” *Blue Waters*, National Center for Supercomputing Applications, Web (https://bluewaters.ncsa.illinois.edu/hardware-summary) Web. 7 Apr. 2016.