Monitoring and Analyzing Job Performance Using Resource Utilization Reporting (RUR) on A Cray XE6/XK6 System

Shi-Quan Su, Troy Baer, Gary Rogers, Stephen McNally, Robert Whitten, Lonnie Crosby
National Institute for Computational Sciences
Oak Ridge, TN 37831 Email: shiquansu@hotmail.com

***Abstract*—Resource utilization in high performance computing centers is fundamental information to gather on an ongoing basis. This information helps support staff to understand, and to predict more acurately, the usage patterns of users. This information is invaluable when deciding on day to day configurations of the system, and possibly long term upgrades to the hardware, or installations of future systems. Cray's Resource Utilization Reporing (RUR) software gathers node level statistics before and after users jobs are run. The base package includes data plugins to gather statistics, such as accounting data, and output plugins to save information in an orderly manner.**

This paper describes work completed at the National Institute for Computational Sciences (NICS) measuring the impact and scalability of enabling RUR on an XE6/XK6 system. To determine if enabling RUR would have any negative impact on system performance, metrics were analyzed from benchmarks generated using the NICS Test Harness, a framework which simulates the typical workload and is used during acceptance testing, when RUR was both enabled and disabled on the system. Also presented in this paper are the tools and methods used to extend the output from the basic data plugins to include other local system level data to generate a more detailed resource utilization snapshot. The integration between NICS and the XSEDE Metrics on Demand (XDMoD) project is discussed along with the utilization of XDMoD in the broader scientific community.

***Keywords*-RUR; resource utilization; job performance monitoring; Cray XE6/XK6; XDMoD; Test Harness;**

I. Introduction

Most high performance computing (HPC) centers run their systems as either time-shared or space-shared resources. There are growing needs that the service providers, usually the supercomputing center staff, want to keep track of the user activities on the machines. As these systems have become larger and more power-hungry, it has become more common for centers to collect more and more metrics on user activities on their systems[1]. These metrics can provide insight into system tuning and identify poor application performance. Such information provides the clues of how to tune the system for better throughput, and may also release a signal warning the potential system failures. The general users may also benefit from identifying the application performance weak spot from analyzing the resources utilization.

Cray system software has long provided solutions for measuring performance, and this software has evolved over time. The latest iteration of Cray's performance monitoring software is Resource Utilization Reporting (RUR), which was first released in 2013. RUR offers a standard framework for implementing job monitoring and reporting for various Cray systems.

This paper will describe a study to investigate using RUR to collect job performance metrics and integrate the output to various external software systems, including the TORQUE batch environment and the XSEDE Metrics on Demand (XDMoD) reporting system. This study was conducted on Mars, a Cray XE6/XK6 system at the National Institute for Computational Sciences (NICS) of the University of Tennessee, which is located at Oak Ridge National Laboratory. Mars is a single cabinet hybrid system with 20 XE6 and 16 XK6 compute nodes.

II. Quantifying RUR Overhead

The RUR module is designed to be low-noise and scalable on large scale systems [5]. The RUR module collects data from compute nodes on an individual aprun level. Each RUR output record is labeled by the combination of jobid and apid. Understanding the overhead and performance impact of enabling RUR on the job level is critical for support staff to deciding whether the service can be used in production operation. It is also imperative to quantitatively state the overhead under a realistic user environment. In a real life computational environment, multiple users perform disparate activities on the machine at the same time. These activities include compiling and linking applications, running jobs via a batch system, and transferring and archiving data.

It was prudent to simulate a realistic user workload on a smaller XE6/XK6 test system, Mars, instead of the larger XC30 production system, Darter, at NICS. This testing approach allows more tests to be completed in a shorter timeframe without having any direct impact on users' daily work.

A NICS customized version of Test Harness platform was selected for the user's workload simulation. The Test Harness platform is originally developed at ORNL [9], and has been used in the acceptance test on several Cray systems such as Jaguar[11], Kraken, Darter, and Titan. The Test Harness includes two major components: the Test Harness Python library, which manages the test runs, and the set

of benchmarks from various applications. The Test Harness initially compiles and links the applications, then submits the benchmark runs to the batch system, collects the job outputs, and compares the results against the built in test. The Test Harness records the results locally and archives the job data into the High Performance Storage System (HPSS) at Oak Ridge National Laboratory (ORNL) campus. The Test Harness automatically restarts the process until a predetermined time passes, or a predetermined number of runs have completed. During the simulation, the administrators can query the current status of the test runs and the success rate of the test runs. NICS developed and maintains a customized version of Test Harness for its computational resources.

Three applications from the NICS Test Harness: NAMD, NWCHEM, and GROMACS were selected. All three applications have large user communities and are actively run on NICS resources. NAMD, is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems. NWCHEM is a scalable computational chemistry package, ackling molecular systems including biomolecules, nanostructures, actinide complexes, and materials. GROMACS is a versatile package to perform molecular dynamics, i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles.

Two benchmarks were chosen for each applications. The NAMD benchmarks were apoa1 and stmv. The NWCHEM benchmarks were aump2 and c60_pbe0. The GROMACS benchmarks were d.dppc and d.villin. All benchmark examples are available online. For each benchmark, two job sizes were chosen. Combined with the enabling/disabling of RUR, there are a total of $3 \times 2 \times 2 \times 2 = 24$ distinct sets of data.

The runs were completed in the following manner. RUR was enabled on the system. Twelve jobs were submitted to the batch system, distinguished by application, benchmark, and job size. Each benchmark job that was submitted to the batch system, in turn, created an auxilary job to run immediately after the completion of the benchmark job. The RUR output was recorded immediately before the benchmark job was placed on the compute nodes, and again immediately after the benchmark job completed. The auxiliary job would then record the outcome of the benchmark job (pass, failed, or inconclusive) in tabular form, archive the job results, and restart the rebuilding of the application for a new run. The auxiliary job only executes on the service nodes, thus there is no RUR output generated. At any given time, there are twelve sets of jobs running, eligible, or held on the system. Each set of jobs is either the benchmark job or the auxiliary job. This simulates a realistic workload on the system where multiple users are performing various activities all at the same time on the system.

Each set of tests were run continuosly for a 36 hour test period. Each benchmark was compiled and executed hundreds of times. After the completion of the initial 36 hour test, RUR was disabled and the same set of runs were started for the comparative 36 hour test.

Benchmark	Cores	RUR	Jobs	Ave. Run Time	Std. Dev.
stmv	128 128	on off	388 398	129.71 129.68	0.37 0.45
stmv	256 256	on off	227 240	89.73 88.49	1.47 1.29
apoa1	32 32	on off	1102 1261	81.67 81.86	0.42 0.48
apoa1	64 64	on off	1385 1519	46.21 46.25	0.36 0.78
c60_pbe0	32 32	on off	60 52	322.81 322.60	2.69 2.88
c60_pbe0	64 64	on off	98 80	167.45 167.64	1.21 1.38
aump2	32 32	on off	56 60	349.77 349.67	2.71 2.05
aump2	64 64	on off	89 75	184.92 185.50	1.41 2.05
d.dppc	64 64	on off	71 55	114.02 114.05	0.16 0.16
d.dppc	128 128	on off	96 77	64.68 64.77	0.16 0.18
d.villin	32 32	on off	109 86	74.22 74.26	0.15 0.13
d.villin	64 64	on off	134 104	44.85 44.92	0.37 0.35

Table I
A TABLE OF TEST HARNESS BENCHMARKS PERFORMANCE

After obtaining the full 24 sets of data, basic job statistics were computed for both cases. These statistics included the average runtime and standard deviation for each job. Table I illustrates the average run time with RUR on and off. These runtimes have no statistically significant difference. The runtime difference between RUR on and RUR off is well within the standard deviation of the average runtime. Thus we conclude that RUR does not have any negative impact on runtimes.

III. RUR CONFIGURATION EXPERIENCES AND LIMITATIONS

RUR includes a two different types of plugins by default: data plugins and output plugins. Data plugins record information on the node immediately before a job is placed on the compute node and immediately after the job is completed. Output plugins are used to record this data. NICS enabled all of the standard plugins described in the Cray system software management guide [5]. This included the taskstats, timestamp, memory, and energy plugins. The file output plugin was enabled, and all results were stored on the attached Lustre file system.

The standard RUR data plugins, and the file output plugin are enabled by editing the basic alps.conf and rur.conf files as shown in Apendix section. In order to enable RUR, beyond setting the correct configuration option, ALPS needs to be restarted on the login nodes. The file output plugin was

chosen, as opposed to the lightweight log manager, in order to have the RUR output sent to a centralized location that is shared between service and compute nodes. The Lustre filesystem was an ideal location not only because it was mounted on Mars on all the appropriate nodes, but it was also mounted on other systems, so post-processing of the RUR data could be done on other nodes if necessary.

IV. GENERATING SINGLE JOB RECORD

The information recorded using the timestamp, taskstats, and memory data plugins were used in generating a single job record. The timestamp plugin and the taskstats plugin output one record for each aprun command in a job. The record can be identified by the combination of jobid and apid. The memory plugin outputs one record for each node in each aprun command in a job. The record can be identified by the combination of jobid, apid and nid. Consider a job including M aprun commands, each aprun command runs n_i processes on N_i nodes, i from 1 to M. The RUR module will generate M timestamp plugin records, M taskstats plugin records, and $\sum_{i=1}^{M} N_i$ memory plugin records for a single job labeled by the unqiue jobid.

There is one major alternative setting that can be configured in the taskstats plugin where the arg is set to ”xpacct, per-process”. Under this setting, the taskstats will generate $\sum_{i=1}^{M} n_i$ records for a single job. The volume of data generated and stored on disk soon becomes an issue under this alternative, and as such is beyond the scope of this paper.

Practically, both the users and the support staff want to know the information on a single job level. This naturally leads to a question of how to generate a single job record. The answer of the above question pertains to the balance between how much information the user needs and how much information the system can offer without significantly affecting the job performance on the user's code. The satisfied solution is that the user has a single job record with all the information needed and no significant overhead added to the job execution.

There is another important issue of generating single job records. RUR is designed to collect information on the compute node level. For the majority of batch jobs running on the supercomputers, the execution of aprun commands is the major part of the job. Hence it is a good approximation to treat the information from RUR as the whole job information. But for some jobs, which spend a significant part of runtime on service node, such as performing I/O, executing system commands, the single job record need to include the information from job level (both service node(s) and compute node(s)) to loyally represent the job.

There is no doubt that there are a lot of approaches and implementations at different institutions to solve the above issue. At NICS, the recommended approach is to combine all the RUR records from the unique jobid first, then query the local pbsacct database with the jobid to suplement the single job record. Combining the job accounting information and the RUR records, it is possible to obtain single job records including most of the desirable information.

The administrators at NICS maintain a set of scripts to query the local pbsacct database. The pbsacct utility provides the following information: system name, job id, user name, group name, charge account, job name, number of processers requested, queue name, submit time, start time, end time, wall time requested, allocated host list, exit status, node where the job was submitted, and the user's batch job script.

After parsing the RUR output and querying the pbsacct database, a single job record can be generated in a json dictionary style. Each job record is stored in a file. These records can be injected into external analysis systems for further processing, such as XDMoD.

V. INTEGRATION WITH TORQUE

The TORQUE batch environment [7] is an open source fork of the Portable Batch System [12], maintained by Adaptive Computing and including patches from a wide variety of sources including academia, national laboratories, and industry. TORQUE is widely used in HPC on everything from small commodity clusters to leadership-class Cray systems such as Blue Waters [3] and Titan [6]. While TORQUE does not include system analysis or workload analysis tools, a number of third-party tools exist that can ingest the TORQUE accounting logs, including Gold [13], pbsacct [1], and XDMoDt [2].

TORQUE version 5.0 added a field in its accounting logs for recoding the energy used by a job, called $resources_used.energy_used$. The initial implementation of this can be found in $src/resmom/cray_energy.c$ in the source code [4]. This implementation targets RUR, specifically the energy plugin, as its underlying data gathering infrastructure. The TORQUE developers have acknowledged that this implementation is a prototype and will be refactored for greater generality in the long term [8].

To test the integration between TORQUE and RUR, NICS upgraded the TORQUE instance on Mars to version 5.0.1 and then enabled RUR with the energy plugin on the system. Additionally, the NICS pbsacct software was updated to handle the new field. However, RUR reported each job's energy consumption as zero, which was then propagated to the TORQUE accounting logs and on to the NICS pbsacct instance. Eventually it was discovered that the RUR energy plugin does not support energy measurement on the XE6/XK6 hardware platform, but that fact was not documented in the Cray system software manual describing RUR [5]. At that point, implementing the RUR power plugin on Mars was abandoned. NICS has requested that Cray update the documentation to reflect the fact that RUR does not support energy measurement on XE6 and XK6 systems.

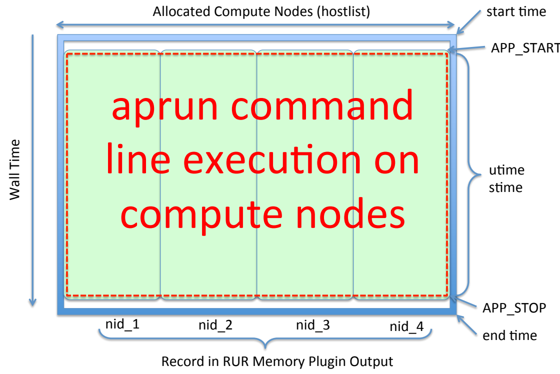

Figure 1. A schematic show of a simple job structure

VI. INTEGRATION WITH XDMOD

The XSEDE Technology Audit Service (TAS) is an NSF funded project at the Center for Computational Research (CCR) of the State University of New York (SUNY) at Buffalo to audit the XSEDE project's activities. One of this project's main products is XSEDE Metrics on Demand (XDMoD), a software system which provides web-based access to metrics and analytics for the computational resources of the XSEDE service providers (SPs). The XDMoD (https://xdmod.ccr.buffalo.edu) is designed to audit and facilitate the operation and utilization of XSEDE, the most advanced and robust collection of integrated advanced digital resources and services in the world. Similarly, Open XDMoD, the open source version of XDMoD, is designed to provide similar capabilities to academic and industrial HPC centers. The XDMoD tool includes both a web-based interface and a back end to collect and store job performance data. The level of detail available in this data varies by system. For example, the Stampede system at TACC uses the TACC_Stats system to provide very detailed information including hardware performance counters, interconnect statistics, and Lustre statistics. There is currently no real equivalent to TACC_Stats on Cray systems; however, the hope is that RUR might be able to provide similar functionality. In this section, we will discuss that possibility in detail, including an enumeration of the development work needed to make it a reality.

The XDMoD is a package under active development. The developers have identified a set of metrics for each single job run on the supercomputer. These metrics aim to represent the typical usage on the supercomputers. Analysis of these metrics offers various information which helps staff members and users to utilize the resources more efficiently.

The current set of metrics includes three subsets. The first subset includes: organization name, machine name, local job id, job name, project account, user name, job directory, executable, exit status, number of granted processing elements (PEs), queue name, number of requested nodes, array of allocated host names, actually used node count, shared mode, actually used core count, available core count, submit time, eligible time, start time, end time, wall time, requested wall time, wait time, cput time, node time, error message.

The second subset includes: cpu idle percentage, cpu system percentage, cpu user percentage, flops average per core, clock ticks per instruction on average per core, L1D cache load drop off percentage, clock ticks per l1d load on average per core, total data transferred over the memory bus, standard deviation of cpu user for all used cores, (max - min) / max cpu user over all used cores, memory usage including system service per node, memory used by the OS including the page and buffer caches per node.

The third subset includes a number of interconnection and accelerator hardware counters, which relies on additional plugins. It is beyond the scope of focus here.

The first subset can be extracted from the job accounting data base. The second subset needs to combine both the job accounting data and the RUR output.

Consider a simple job as shown in Figure 1. Each mint green color box represnts one compute node. The job requests 4 nodes, and there is only one aprun command inside the job script, which uses all 4 nodes. The time spent on the service node is ignorable.

In this case, the hostlist information from the accounting data base is identical to enumerating the nid key in the RUR memory plugin output. The walltime calculated from the difference between start time and end time is about the same as the difference between the APP_START and APP_STOP from the RUR timestamp plugin. The sum of utime and stime in RUR taskstats plugin output, the area of the red dashline rectangle, can be approximated by the area of the blue solid line rectangle, which is the Wall Time $\times$ Allocated Compute Nodes.

Next, we study a complex job as in Figure 2. Still the job requests 4 compute nodes, but only 3 of them are actually used. There are 4 aprun commands in the job, labeled by $apid_i$, i from 1 to 4. $apid_1$ command runs on 2 nodes. After $apid_1$ finishes, $apid_2$ and $apid_3$ commands start at the same time, $apid_2$ runs on 1 node, using only a portion of cores on the node (such as having a -N option after aprun), $apid_3$ runs on 2 nodes and finishes earlier than $apid_2$. The $apid_4$ command runs after both $apid_2$ and $apid_3$ finishes, and runs on 3 nodes. After all the aprun commands finishes, there is a significant span of the job executing system commands on the service node.

We generate the metrics from the single job record of above complex job. The metrics set is stored in a JSON-dict format as the single job record. We customize the original definitions of the metric to handle the job with multiple aprun commands. We demonstrate how to obtain several metrics below.

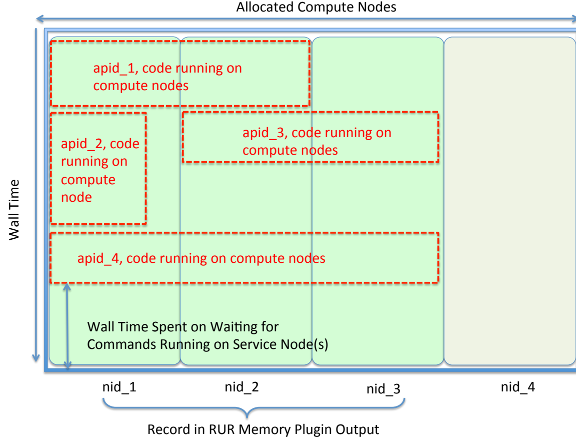

Figure 2. A schematic show of a complex job structure

We first generate two extra key-value-pair items. The first item has the key "used nodes list", and has the value as a list of nids. The nid(s) in the list appear(s) at least once as the value of the RUR memory plugin output "nid" key. In Figure 2 example, it is ("used nodes list": [nid_1, nid_2, nid_3]). The second item has the key "aprun nodes dict", and has the value as a dict with apid key and nodes list value. The nodes list is the list of nid appear in each aprun memory plugin output. In Figure2 example, it is ("aprun nodes dict": {$apid_1$: [nid_1, nid_2], $apid_2$: [nid_1], $apid_3$: [nid_2, nid_3], $apid_4$: [nid_1, nid_2, nid_3]}). Then we also need a mapping function ppn_{nid} between nid and the number of cores on the node. Since it is common that the Cray system has heterogeneous structure, which includes more than one type of node. For example, the Mars has both 32-core XE6 nodes and 16-core XK6 nodes. If the user does not specify the feature in the PBS option, it is highly possible that the scheduler schedules a mixed node pool to the job.

The first metric we show is "cpu idle percentage" (ρ), the original definition is:

$$\rho = \frac{\sum_{i=1}^{n} x_i}{n} \tag{1}$$

where n is number of cores the job ran on. x_i is cpu idle percentage of each core. We calculate ρ as:

$$\rho = 1 - \frac{\sum_{i=1}^{k}(utime_i + stime_i)}{T \times N} \tag{2}$$

where k is the number of aprun, $utime_i$ and $stime_i$ are from the RUR taskstats plugin record with $apid_i$, T is the wall time elapsed for the whole job, N is the number of cores actually assigned to the job, sum up the values of ppn_{nid} with the nid in array of allocated host names from job accounting information. The idea is that the result needs to include data from multiple aprun executions.

The second metric is "memory usage including system service per node" (μ), the original definition is:

$$\mu = \frac{\sum_{i=1}^{H} \frac{x_i}{C_i}}{H} \tag{3}$$

where x_i is the mean memory used on node i, C_i is the number of cores on node i and H is the number of nodes on which the job ran. We calculate μ as:

$$\mu = \sum_{j=1}^{k} w_j \times \frac{\sum_{i=1}^{n_j} m_i}{n_j} \tag{4}$$

where k is the number of aprun, n_j is the number of nodes in the jth aprun, m_i is the "Active(anon)" value in the RUR memory plugin output with $apid_j$, and nid_i. The w_i is the weight calculated as below:

$$w_j = \frac{(APP_STOP_j - APP_START_j) * N_j)}{T \times N} \tag{5}$$

where APP_STOP_j and APP_START_j are values from RUR timestamp plugin with $apid_j$. T and N are defined the same as above. The idea is that we weight the average memory usage per node among different aprun commands by node hour weight.

Finally we describe the designed workflow for injecting RUR output to XDMoD. On Mars, RUR is configured to write job performance data to a file on a Lustre scratch file system. The data in that file would be validated and filtered to remove potentially sensitive information. Then the python script described in the section "GENERATING SINGLE JOB RECORD" will be run. The script combines the RUR record labeled by (jobid, apid) to a single job record (jobid), then executes system command "jobinfo" to query the pbs accounting data base with the jobid, and fills the relevant result into the correct fields of the single job record. Then the single job record will be passed into a Javascript run by application "node" to generate the desired metrics. The metrics will be stored in the JSON-dict format file for each jobid. The final "single job record and metrics" files would then be transferred to an instance of the XDMoD system at CCR, where an automated pipeline would ingest the data. CCR developers would then re-validate the data before storing it in the central data warehouse. The front-end XDMoD GUI would then pull data from the central data warehouse per users online requests.

VII. Conclusions

The RUR provides a low noise, scalable approach to collect performance data from the compute nodes. The RUR framework allows users develop customized plugins for various purposes. It appears that RUR is a stable utility going forward serving the Cray community. There are substantial development works from multiple aspects related to RUR.

The data generated by the default plugins show that RUR, in its current state, is at best a supplement for the accounting

data recorded by whatever resource management system the site implements. The existing plugins can give additional data on CPU time, memory, energy, and GPU usage on the compute nodes that is not typically available in the resource manager accounting logs on Cray systems. However, most of the deeper performance information desired by third-party analytics systems such as XDMoD and provided by other monitoring systems such as TACC Stats [10] are not available through RUR. Specifically, plugins for processor performance counters, interconnect performance counters, and file system performance counters (such as for Lustre) are not currently available for RUR on XE6/XK6 platforms. In principle, Cray customer sites could implement plugins for processor and Lustre file system performance counters, but it would be difficult for anyone other than Cray to implement a plugin for interconnect performance counters.

VIII. Acknowledgment

This material is based upon work performed using computational resources supported by the University of Tennessee and Oak Ridge National Laboratory's Joint Institute for Computational Sciences (http://www.jics.utk.edu). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the University of Tennessee, Oak Ridge National Laboratory, or the Joint Institute for Computational Sciences.

Appendix

alps.conf

```
apsys
  prologPath /opt/cray/rur/default/bin/rur_prologue.py
  epilogPath /opt/cray/rur/default/bin/rur_epilogue.py
  prologTimeout 60
  epilogTimeout 60
/apsys
```

rur.conf

```
[global]
rur: False

[rur_stage]
stage_timeout: 10
stage_dir: /tmp/rur/

[rur_gather]
gather_timeout: 10
gather_dir: /tmp/rur/

[rur_post]
post_timeout: 10
post_dir: /tmp/rur/

[plugins]
gpustat: false
taskstats: true
timestamp: true
energy: false
memory: true

[outputplugins]
llm: false
file: true
user: false

[gpustat]
stage:  /opt/cray/rur/default/bin/gpustat_stage.py
post:   /opt/cray/rur/default/bin/gpustat_post.py

[taskstats]
stage:  /opt/cray/rur/default/bin/taskstats_stage.py
post:   /opt/cray/rur/default/bin/taskstats_post.py

[energy]
stage:  /opt/cray/rur/default/bin/energy_stage.py
post:   /opt/cray/rur/default/bin/energy_post.py

[timestamp]
stage:  /opt/cray/rur/default/bin/timestamp_stage.py
post:   /opt/cray/rur/default/bin/timestamp_post.py

[memory]
stage:  /opt/cray/rur/default/bin/memory_stage.py
post:   /opt/cray/rur/default/bin/memory_post.py


[llm]
output: /opt/cray/rur/default/bin/llm_output.py

[file]
output: /opt/cray/rur/default/bin/file_output.py
arg:    /lustre/medusa/grogers/RUR/output/rur.output

[user]
output: /opt/cray/rur/default/bin/user_output.py
arg:    single
```

References

[1] Troy Baer and Doug Johnson. pbsacct: A workload analysis system for pbs-based hpc systems. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*. ACM, 2014.

[2] Thomas R. Furlani, et al. Using XDMoD to facilitate XSEDE operations, planning and analysis In *Proceedings of the 2013 Annual Conference on Extreme Science and Engineering Discovery Environment*. ACM, 2013. doi 10.1145/2484762.2484763

[3] Blue waters user portal: Getting started. https://bluewaters.ncsa.illinois.edu/documentation.

[4] GitHub: adaptivecomputing/torque. https://github.com/adaptivecomputing/torque.

[5] Managing system software for the Cray Linux Environment. http://docs.cray.com/books/S-2393-51/.

[6] Titan user guide. https://www.olcf.ornl.gov/support/system-user-guides/titan-user-guide/.

[7] TORQUE resource manager. http://www.adaptivecomputing.com/products/open-source/torque/.

[8] David Beer. Energy measurement code in TORQUE 5.0. http://www.supercluster.org/pipermail/torquedev/2015-January/004811.html.

[9] Arnold Tharrington. An Overview of NCCS XT3/4 Acceptance Testing. https://cug.org/5-publications/proceedings_attendee_lists/2007CD/S07_Proceedings/pages/Authors/Tharrington/Tharrington_paper.pdf.

[10] Todd Evans, William L Barth, James C Browne, Robert L DeLeon, Thomas R Furlani, Steven M Gallo, Matthew D Jones, and Abani K Patra. Comprehensive resource use monitoring for hpc systems with tacc stats. In *Proceedings of the First International Workshop on HPC User Support Tools*, pages 13–21. IEEE Press, 2014.

[11] Wayne Joubert, and Shi-Quan Su. An analysis of computational workloads for the ORNL Jaguar system. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 247256. ACM, 2012. doi 10.1145/2304576.2304611

[12] Robert L Henderson. Job scheduling under the Portable Batch System. In *Job scheduling strategies for parallel processing*, pages 279–294. Springer, 1995.

[13] Scott Jackson. The Gold accounting and allocation manager, 2004. http://www.emsl.pnl.gov/docs/mscf/gold.