Resource Utilization Reporting, Two Year Update

Andrew Barry
Cray Inc.
Saint Paul, USA
e-mail: abarry@cray.com

Abstract—*In the two years since Resource Utilization Reporting was announced, half a dozen variants of the software have been released as part of CLE releases. Sequential releases have included additional plugins to record a growing list of data points, as well as more output plugins and controls for how the data is presented. This document reviews the purpose and design of RUR, and describes the addition of functionality through time.*

I. Introduction

Since the earliest days of time-sharing systems, administrators have wanted to track what users are doing on a system. Given limited system resources, carefully tracking usage of the resources can affect site policies for how users share resources, or billing. Usage statistics may also impact future upgrade plans, and requirements for successor systems.

Over the years, Cray has supported a number of accounting tools, including BSD process accounting, Comprehensive System Accounting (CSA), and Mazama Application Completion Reporting. Each of these tools reported a fixed set of tracked data elements, some quite a modest list, and some with more complete data. All of these tools supported scalability adequate for system sizes at the time of their initial release, but struggle to support systems with many thousands of compute nodes.

Starting with the CLE4.2UP02 release, Cray has supported a new feature: Resource Utilization Reporting (RUR). RUR allows for the collection of large, extensible, and arbitrary sets of data through a plugin mechanism, and allows for data to be stored and reported through another plugin mechanism, which is similarly versatile. RUR is also designed to support greater scalability than previous solutions.

II. How RUR Works

RUR comprises both a scalable framework for collecting utilization data from the nodes in a user's applications, and also plugins that capture the exact data elements of interest to the administrator. The framework works in five main phases: launch, staging, collection, post-processing, and output.

- Launch happens immediately prior to application launch, and allows each data plugin to launch any required data collection process, or to record the initial state of some resource. RUR refers to this as the “pre” component of data staging. This happens on every compute node used by the application.
- Staging happens immediately following application completion. In RUR terminology, this is the “post” component of data staging. Staging includes closing any data collecting processes, and recording the state of resources. Resources used after the application can be compared to the state prior to the application. This data is then stored in a staging file on the compute node. This happens on every compute node used by the application.
- Collection happens when the staged data on each compute node is collected to the login/mom node used to launch the application. This is done through the scalable fanout tree also used by Cray Node Health Checker.
- Post processing occurs on the login/mom node, and summarizes per-node data into a condensed representation.
- The Output phase happens on the login/mom node, and includes copying the summarized data to one or more logs, files, databases, or other form of permanent storage.

Resource Utilization Reporting is built around a plugin architecture, supporting both data plugins and output plugins. Data plugins collect data about a particular sort of system resource, such as general Unix process accounting, accelerator usage, energy usage, etc. Output plugins store summarized RUR data to a particular form of permanent storage such as a log, a text file, a database, or similar. Several data plugins and output plugins are included in the RUR package in CLE. If an administrator finds that these don’t collect a piece of desired data, he or she may create a custom plugin. The plugins within the CLE package are published with a BSD license, and may be used as a guide for creating custom plugins.

Data plugins include two components: staging, and post-processing. During the launch and staging phases of RUR, each data plugin staging component is called on each compute node, where it (typically) captures state prior to and after application run-time, and compares the two. During the post-processing phase of RUR, each data plugin post-processing component is called on the login node, and summarizes the data from the compute nodes, in a fashion appropriate to that data.

III. Example of RUR Usage

In this example, an administrator has configured ALPS to run RUR, by setting the ALPS config file to call the RUR prologue and epilogue scripts as part of the apsys prolog and epilog, respectively. The administrator has set the RUR config file to run the taskstats data plugin, and the LLM output plugin. To test RUR, the administrator runs three simple commands: sleep, hostname, and uptime.

```
crayadm@login1$ aprun –n 1 hostname
Nid00020
Application 2627651
crayadm@login1$ aprun –n 1 uptime
 14:42pm  up  3:35,  0 users,
 load average: 0.03, 0.11, 0.19
Application 2627652
crayadm@login1$ aprun –n 1 sleep 1
Application 2627653
```

Following this, the administrator checks the messages log on the SMW.

```
c0-0c0s1n1 RUR 11086 p0-20150326t103954
[RUR@34] uid: 12795, apid: 2627651, jobid:
0, cmdname: /bin/hostname, plugin: taskstats
{"core": 0, "exitcode:signal": ["0:0"],
"max_rss": 704, "stime": 8000, "wchar": 101,
"rchar": 4524, "utime": 0}

c0-0c0s1n1 RUR 11537 p0-20150326t10395
4 [RUR@34] uid: 12795, apid: 2627652, jobid:
0, cmdname: /usr/bin/uptime, plugin:
taskstats {"core": 0, "exitcode:signal":
["0:0"],"max_rss":868, "stime":12000,
"wchar":155, "rchar":7875, "utime":0}

c0-0c0s1n1 RUR 11727 p0-20150326t10395
4 [RUR@34] uid: 12795, apid: 2627653, jobid:
0, cmdname: /bin/sleep, plugin: taskstats
{"core": 0, "exitcode:signal": ["0:0"],
"max_rss": 732, "stime": 8000, "wchar": 92,
"rchar": 4524, "utime": 0}
```

IV. RUR Evolution

From its initial release in CLE4.2UP02 RUR has slowly added functionality in the form of additional data plugins, more output plugins, greater configurability, and more modes of operation.

A. CLE4.2UP02 and CLE5.1UP00

The initial release of RUR included only the taskstats, energy, and gpustats data plugins and the LLM and File output plugins. The taskstats plugin provides basic process accounting, including cpu, memory, and filesystem utilization metrics, and process exit codes. The gpustat plugin provides GPU and memory utilization metrics for processes using nvidia accelerators. The energy plugin reports the amount of energy used, in joules, by all of the compute nodes in the application. The LLM plugin writes RUR data to the LLM log on the SMW. By default this ends up in the messages-date file. The 'file' plugin writes the RUR output to a flat text file in a filesystem accessible from the login/mom node; If the filesystem is not a shared filesystem, each login/mom node will write its own file. While this initial release of RUR has been superseded by others with more features, these plugins remain the most used.

B. CLE5.1UP01

The second release of RUR added plugins and expanded existing ones. The taskstats plugin was modified to report application exit-codes as an exit-code:signal pair. This is similar to what is reported to a user's shell. Previously the reported value was an integer of the two values concatenated. In this release the timestamp plugin was added, which prints the application starting time and ending time. The 'user' output plugin was added, which writes RUR data to a file in the user's home directory, for all applications he or she runs.

C. CLE5.2UP00

This release added the Kncstats plugin, and further improved the Taskstats plugin. The Kncstats plugin provides the same process accounting statistics as the initial taskstats implementation, but for processes run on Intel Knights Corner accelerators, when running autonomous mode applications. The memory plugin was also added in this release. It reports a wide array of Linux kernel memory allocation statistics for each compute node used by the application. Because the memory plugin output is not summarized, it can create a very large volume of data, and is most applicable for diagnosing a known problem, rather than running all of the time.

The taskstats plugin, was given two optional arguments: extended-process-accounting and per-process accounting. Extended-process-accounting (xpacct) aims to collect all of the process statistics formerly collected by the CSA tool. The per-process accounting option instructs taskstats to not summarize the accounting data, instead reporting on every process on every compute node in the application. Obviously this can create a great deal of data, and is generally not appropriate during regular system operation, but may be suitable to debugging periods.

D. CLE5.2UP01

In this release, no new plugins were added. Both the taskstats and energy plugins were given a new configuration option: json-dict. This option instructs the post-processing component of the plugin to output data in a JSON dictionary format. The dictionary format is generally simpler to parse with complex data-types as elements. Both plugins also gain a json-list option, which provides the previous format.

When the energy plugin is configured to use the json-dict format, it also reports a greatly expanded list of energy statistics including information on accelerator energy usage, and thermal and power throttling.

In this release, the user output plugin has also been extended with the opt-in argument. When this is selected, RUR data will only be written to a user’s home directory if they indicate their interest in the data by creating one of several files in their ~/.rur/ directory.

E. CLE5.2UP03

This release introduces job scope RUR, extended config file semantics, service node RUR, and RUR scalability enhancements.

Job scope RUR does not run from the apsys prolog/epilog, rather out of the WLM prologue and epilogue. In this mode, RUR captures data for the entire span of the job, rather than for the individual applications that constitute the job. This can provide subtly different metrics, compared to looking at the RUR data from the constituent applications. One example of this is energy usage in a job where some nodes are idle for some part of the job’s duration.

Imagine a job that reserves four nodes, but the first application is a single-threaded pre-processing task. For that time one node is busily working, while three sit idle. Then the compute job runs for several minutes. In this scenario, three nodes sit idle for a couple minutes, and only a single node is busy for the entire five minutes. Application scope RUR only tracks utilization for nodes that are part of a running application. Job scope RUR would capture energy data from all nodes, for the duration of the job, including the time spent idle. Idle nodes use a lot less power than a busy node, but the total can add up across many nodes.

Job scope RUR may be configured in the place of application scope RUR, or both may be enabled at the same time. The apid field, of all job scope RUR output, will always be zero.

```
crayadm@login1$ qsub –I –l mppwidth=1
qsub: job 2001549.sdb ready
crayadm@login1$ aprun –n 1 hostname
Nid00020
Application 2627661
crayadm@login1$ aprun –n 1 uptime
 14:42pm  up  3:35,  0 users,
 load average: 0.03, 0.11, 0.19
Application 2627662
crayadm@login1$ exit
qsub: job 2001549.sdb completed
```

In the RUR output data, the constituent applications and the full job report all have the same jobid, but the apid and cmdname differ.

```
c0-0c0s1n1 RUR 11086 p0-20150326t104954
[RUR@34] uid: 12795, apid: 2627661, jobid:
2001549.sdb, cmdname: /bin/hostname, plugin:
taskstats {"core": 0, "exitcode:signal":
["0:0"], "max_rss": 704, "stime": 8000,
"wchar": 101, "rchar": 4524, "utime": 0}

c0-0c0s1n1 RUR 11537 p0-20150326t104954
[RUR@34] uid: 12795, apid: 2627662, jobid:
2001549.sdb, cmdname: /usr/bin/uptime,
plugin: taskstats {"core": 0,
"exitcode:signal": ["0:0"],"max_rss":868,
"stime":12000, "wchar":155, "rchar":7875,
"utime":0}

c0-0c0s1n1 RUR 11727 p0-20150326t104954
[RUR@34] uid: 12795, apid: 0, jobid:
2001549.sdb, cmdname: N/A, plugin: taskstats
{"core": 0, "exitcode:signal": ["0:0"],
"max_rss": 868, "stime": 20000, "wchar":
247, "rchar": 12399, "utime": 0}
```

In the case of the taskstats plugin, job scope does not provide any additional data. The information gleaned from the job scope RUR report could be created from a simple arithmetic combination of the reports from the constituent applications. As mentioned above, this is not true for the energy plugin.

```
crayadm@login1$ qsub –I –l nodes=4
qsub: job 2001550.sdb ready
crayadm@login1$ aprun –n 1 setup.sh
Application 2627671
crayadm@login1$ aprun –n 4 compute.sh
Application 2627672
crayadm@login1$ exit
qsub: job 2001550.sdb complete

c0-0c0s1n1 RUR 11086 p0-20150326t114893
[RUR@34] uid: 12795, apid: 2627671, jobid:
2001550.sdb, cmdname: setup.sh, plugin:
energy {“energy”: 4165}
c0-0c0s1n1 RUR 11086 p0-20150326t115037
[RUR@34] uid: 12795, apid: 2627672, jobid:
2001550.sdb, cmdname: compute.sh, plugin:
energy {“energy”: 43007}
c0-0c0s1n1 RUR 11086 p0-20150326t115040
[RUR@34] uid: 12795, apid: 0, jobid:
2001550.sdb, cmdname: N/A, plugin: energy
{“energy”: 50292}
```

	Job 50292j	
Sum 47172j	Setup 4165j	Compute 43007j
Node1	Setup 4165j	Compute 10752j
Node2	Idle 1040j	Compute 10752j
Node3	Idle 1040j	Compute 10752j
Node4	Idle 1040j	Compute 10752j

Figure 1.

Figure 1 summarizes the RUR data graphically. Note that the dark grey field and red field do not match. In this example, that the three nodes idle during the setup application use a significant fraction of the energy used by the one node that is setting up the data. This is over a short period, and may not be representive the steady-state power usage of idle nodes.

In order to simultaneously configure job scope RUR and application scope RUR, two options are available. One option is to have a config file for each. Alternately, if the config files are very similar, it may be simpler to use configsets. Each plugin definition in the RUR config file can be tagged with a ‘config_sets’ descriptor, which can take one or more config set names. When the RUR scripts are called with the command line argument "-s" option, and a config set name, only those plugins tagged with a matching config set name will be run. Thus different plugin definitions with different options and arguments, can be defined for the job scope case, and for the application scope case.

Another feature added in the 5.2UP03 release, is support for running RUR plugins on service nodes. Like job-scope RUR, this is will require either a separate configuration file, or config-sets to control RUR behavior. A list of targeted service nodes must be supplied to the RUR scripts. Cray does not currently provide any plugins for collecting useful data from service nodes, though that capability is available for custom plugins.

RUR scalability enhancements is a feature created to improve RUR scalability to large node-count applications, and to lessen RUR's dependence on high scalability of the DVS filesystems. The problem is that the RUR config file, infrastructure scripts, the python base package, and RUR plugins are all hosted on the /dsl filesystem of the compute nodes. This is exported from the boot node, by way of DVS servers. When RUR is run on a compute node, all of these files need to be paged into the compute node via DVS. If RUR is simultaneously running from a large number of compute nodes, this can put a large load on the DVS servers, particularly if they are busy serving other data, or if the ratio of compute nodes to DVS servers is high. This contributes to RUR taking a long time to complete.

To reduce this problem, two steps were made in the 5.2UP03 release. The first enhancement is always used, which sends the config file from the launching login/mom node to compute nodes, as part of the launch message.

The second enhancement is optional, and requires using the compute_local_python configuration option in the RUR config file. If this is selected, RUR will not run compute node scripts out of the /dsl filesystem. Instead, the compute node components of RUR are compiled with pyInstaller, and installed into the ramdisk filesystem of the compute node. This packages the python scripts up with a python interpreter, into a stand-alone binary. Thus the infrastructure and plugins are loaded from the local filesystem, and do not depend on the /dsl filesystem and associated DVS servers. Using the compute_local_python configuration option reduces load on the DVS system of the Cray, and improves scalability. Running in this manner is still slower for large node-count applications, than it is for a single node, but the scalability is improved.

V. Sample RUR statistics from Cray internal Systems and a large customer system

Here are presented some sample RUR output from a XC30 system in Cray's datacenter. The system has one hundred and forty compute nodes, and is used for a mix of system software development, and application testing. This proves somewhat different from RUR output gathered from a very large XE6 belonging to a Cray customer.

A very average job on the Cray XC30 system, followed by a very average application on the customer XE6 system:

```
apid: 2787476, jobid: 169739.sdb, cmdname:
/cray/css/ostest/binaries/MPI_Test_p_F, plugin:
taskstats ['btime', 1425363224, 'etime', 321698,
'utime', 20000, 'stime', 68000, 'coremem', 164904,
'max_rss', 6968, 'max_vm', 95888, 'pgswapcnt', 0,
'minfault', 7341, 'majfault', 0, 'rchar', 338712,
'wchar', 704, 'rcalls', 312, 'wcalls', 30,
'bkiowait', 48332261, 'exitcode:signal', ['0:0'],
'core', 0, 'abortinfo', ['0']]
```

```
apid: 6821745, jobid: 1335865.nid11293, cmdname:
/u/team/user/NAMD_build.latest/NAMD_2.10_CRAY-XE-
MPI-system/namd2, plugin: taskstats ['utime',
189040172800, 'stime', 57707188000, 'max_rss',
191024, 'rchar', 195746806, 'wchar', 3844321678,
'exitcode:signal', ['0:0'], 'core', 0]
```

The Cray XC30 is much smaller than the XE6 system. As such, it runs smaller jobs. The maximum cpu usage of an application on the XC30 is 8,862,578 core*seconds, compared to 1,730,219,193 core*seconds on the XE6. The average is much closer: 6,706 core*seconds compared to 12,356 core*seconds. Similarly the maximum filesystem write usage of the XC30 was smaller: 5277 GB, compared to 3460 TB; again the average values are close at 1.5 GB and 3.0 GB. The not surprising analysis of the collected RUR data is that larger machines can and do run larger jobs, though modest sized jobs still dominate even very large machines. Another unsurprising statistic is that development machines run a lot of applications that complete with a failure. The success to failure ratio on the XC30 was 1.92, compared to 15.01 on the XE6.

The RUR statistics do not show two areas in which the systems differ quite severely. The XC30 in Cray's datacenter spends a lot of time idle, waiting for developer or administrator action, whereas the XE6 is almost never idle. The XC30 also averages 3.3 reboots per day, a very high level, even in a development environment.

VI. Conclusion

Resource Utilization Reporting has now been deployed on a large number of Cray customer systems, including some of the largest supercomputers in the world. RUR replaces several previous accounting tools on Cray systems, expanding the data variety that is collected, and allowing greater scalability. RUR is included in all currently supported releases of CLE and will be supported on future releases of CLE, currently in development. Cray continues to solicit input on useful features or plugins that might benefit users of this feature.