This CUG paper is a preprint of the final paper published in the CCPE Special Online Issue of CUG 2017 at
http://onlinelibrary.wiley.com/doi/10.1002/cpe.v30.1/issuetoc

How to write a plugin to export job, power, energy, and system environmental data from your Cray® XC™ system

Steven J. Martin, David Rush, Matthew Kappel
Cray Inc.
{*stevem,rushd,mkappel*}*@cray.com*

Cary Whitney
LBNL/NERSC
clwhitney@lbl.gov

***Abstract*—In this paper we take a deep dive into writing a plugin to export power, energy, and other system environmental data from a Cray® XC™ system. With the release of the System Management Workstation (SMW) 8.0 software, Cray has enabled customers to create site-specific plugins to export all of the data that can flow into the Cray Power Management Database (PMDB) into site-specific infrastructure. In this paper we give practical information on what data is available using the plugin, and how to write, test, and deploy a plugin. We also share and explain example plugin code, detail design considerations when architecting a plugin, and look at some practical use cases supported by exporting telemetry data off a Cray® XC™ system. This paper is targeted at plugin developers, system administrators, data scientists, and site planners.**

The plugin feature was developed by Cray in response to discussions with and requirements from select members of the CUG XTreme SIG. This paper demonstrates lessons learned developing prototype plugins that export data off Cray® XC™ systems using Kafka, Redis Pub/Sub, and RabbitMQ. This plugin capability is in-use internally at Cray, used in production at NERSC, and is under consideration for deployment on systems at LANL and Sandia.

***Keywords*-Power monitoring; energy efficiency; power measurement; Cray XC40**

I. INTRODUCTION

The plugin capability described in this paper is supported on Cray® XC™ systems running the Cray SMW 8.0 UP01 release and newer. The “xtpmd” daemon runs on the Cray SMW and handles System Environment Data Collection (SEDC) [1] and high-speed power telemetry data. The xtpmd supports plugins with the help of xtpmd_plugd. Shared memory is used to communicate between xtpmd and xtpmd_plugd, and xtpmd_plugd interacts with the plugin using a defined C API.

The plugin capability is the result of requests from Cray customers, and driven by interactions with the Cray XTreme Systems SIG, starting with a special “System Monitoring Collaboration” meeting in San Jose, CA, on Jan 28, 2016. XTreme members at the January meeting requested access to SEDC and Cray high-speed power and energy data as a stream before it is injected into the Cray PMDB [2]. Cray followed up at CUG 2016 with slides [3] giving a preview of this feature. A prototype plugin was then developed using Redis Pub/Sub [4] transport and coded using Hiredis. [5] This proof of concept was demonstrated as part of the Trinity phase-2 factory trial. That demonstration highlighted the ability to stream data out of the Cray management plane and into site specific infrastructure. Code for the factory trial plugin was shared with ACES and NERSC. The NERSC team has now integrated xtpmd plugins as a new source of data for their “Center-wide Data Collect” [6].

In previous work [7] [8] [9] the Cray advanced power management development team members have outlined Cray® XC™ system power management capabilities and CAPMC workload manager interfaces. The latest publication on Cray® XC™ system power management is available online [10].

This paper is organized as follows: In Section II, we take a detailed walk through example code shipping with the SMW software release and cover the basics of a minimal plugin that writes data in csv format to local files. In Section III we cover the types of data available via the plugin infrastructure. Section IV covers the plugin system configuration file. Section V dives into methods and considerations for testing and deploying a customized site-specific plugin. In Section VI we look at a set of design considerations for plugin development. Section VII gives an overview of the plugin development, design, lessons learned, as well as a few screen shots from work at NERSC.

II. PLUGIN CODE BASICS

This section details basic concepts used in the example plugin code supplied on the SMW 8.0 software releases. In brief, the example plugin writes the data from each of the five sources to CSV formatted output files. The example code can be found in the **smw:/opt/cray/hss/default/pm/xtpmd_api** directory. The two files that make up the example are **xtpmd_plugin.h** and **xtpmd_plugin_csv.c**. A Makefile is not included in the example directory, so we share one in Listing 1.

Listing 1. Example Makefile

```
# Makefile to build:
#     /opt/cray/hss/default/pm/xtpmd_api/xtpmd_plugin_csv.c

OBJ = xtpmd_plugin_csv.o
LIB = xtpmd_plugin_csv.so
```

This paper has been submitted as an article in a special issue of Concurrency and Computation Practice and Experience on the Cray User Group 2017.

```
CFLAGS += -O2 -fPIC -I/opt/cray/hss/default/pm/xtpmd_api/
CFLAGS += $(shell pkg-config --cflags glib-2.0)
LDFLAGS += $(shell pkg-config --libs glib-2.0)

$(LIB) : $(OBJ)
	$(CC) -shared -o $(LIB) $(OBJ) $(LDFLAGS)
```

The header file **xtpmd_plugin.h** defines required data structures, helper functions, and prototypes for functions required when implementing a plugin. The first two function prototypes are defined as shown in Listing 2.

Listing 2. Instance Parameter Access

```
char *plugin_get_instance_var(char *name);
void plugin_free_instance_var(char *value);
```

These helper functions are provided by the plugin infrastructure. The first is intended for reading key=value instance parameters from the plugin's configuration file. The second is provided to free memory allocated by the first. The default configuration file for xtpmd plugins is: **smw:/opt/cray/hss/default/etc/xtpmd_plugins.ini** and it supports the example plugin code. We will discuss instance parameters (or instance variables) and the configuration file in more detail in Section IV.

Next, the header file defines the **pmd_data_opts_t** data structure shown in Listing 3, which is used by the plugin initialization functions to register callback functions for process streaming telemetry data from any of the four supported data sources. The **user_data** pointer will be passed to the functions when they are called, allowing callbacks to manage plugin-specific state. Comments in the header file should be read by plugin code developers looking for more details.

Listing 3. typedef struct pmd_data_opts_t

```
typedef struct {
  void(*recv_uint64)(void *user_data, uint64_t ts, int
       source, int id, uint64_t value);
  void(*recv_double)(void *user_data, uint64_t ts, int
       source, int id, double value);
  void(*flush)(void *user_data);
  void(*close)(void *user_data);
  void *user_data;
} pmd_data_opts_t;
```

The **enum** shown in Listing 4 defines application and job related event codes that can be expected for plugins that register to receive application and job information.

Listing 4. Event Enumeration

```
enum {
  APP_CMD_TYPE_START =     1,
  APP_CMD_TYPE_END =       2,
  APP_CMD_TYPE_SYNC =      3,
  APP_CMD_TYPE_SUSPEND =   4,
  APP_CMD_TYPE_RESUME =    5,
  JOB_CMD_TYPE_START =     6,
  JOB_CMD_TYPE_END =       7,
  JOB_CMD_TYPE_SUSPEND =   8,
  JOB_CMD_TYPE_RESUME =    9
};
```

The **pmd_job_opts_t** structure shown in Listing 5 is used to initialize handling of application and job related information. It is much the same as the **pmd_data_opts_t** structure previously defined other than the prototype for the **recv_apevent** callback function.

Listing 5. typedef struct pmd_job_opts_t

```
typedef struct {
  void(*recv_apevent)(void *user_data, uint64_t ts,
       uint32_t event, uint32_t userid, char *jobid,
       uint64_t apid, uint32_t *nids, uint32_t nids_len);
  void(*close)(void *user_data);
  void *user_data;
} pmd_job_opts_t;
```

The comments at the end of the **xtpmd_plugin.h** file are important. They show the interface for the initialization functions needed to implement a working plugin (see Listing 6). A valid plugin will implement one or more of these initialization functions, but need not define all five.

Listing 6. Initialization Functions

```
int init_sedc_bc_ctx(int version, pmd_data_opts* opts);
int init_sedc_cc_ctx(int version, pmd_data_opts* opts);
int init_pmdb_bc_ctx(int version, pmd_data_opts* opts);
int init_pmdb_cc_ctx(int version, pmd_data_opts* opts);
int init_jobs_ctx(int version, pmd_job_opts* opts);
```

When the plugin is loaded, each of the functions that the implementation has defined will be called. This performs one-time initialization which registers the callback function with its associated data pointers.

The example **xtpmd_plugin_csv.c** begins by defining prototypes for the five callback functions that it will register. The prototypes match parameters defined in the **pmd_data_opts_t** and **pmd_job_opts_t** data structures discussed earlier in this section.

The implementation-specific **csv_ctx_t** structure is defined as shown in Listing 7. Member elements in this structure are used to track state for the output file specific to each data stream being collected by the plugin. Pointers to these data structures are returned in the **user_data** elements of the **pmd_data_opts_t** and **pmd_job_opts_t** structures by the plugins initialization functions. More complicated plugins are expected to replace **csv_ctx_t** with their own data structures optimized for whatever state is needed. For example, the NERSC plugins we describe in Section VII track RabbitMQ [11] connections.

Listing 7. Implementation Specific State

```
typedef struct {
  FILE   *lf;
  char   *lf_dir;
  char   *lf_name;
  char   *lf_prefix;
  int     tcheck;
  int     period;
  GTimer *timer;
} csv_ctx_t;
```

The function **static csv_ctx_t *init_ctx(int version, char *prefix)** is called for each plugin data source and performs all needed initialization work. That work includes:

- Determining if the data source (**prefix**) is enabled,
- Allocation of the **csv_ctx_t** tracking data structure,
- Reading in parameters from the configuration file,
- Opening the source-specific output file, and
- Adding the initialized **csv_ctx_t** structure to **ctx_lst** list.

III. Data Sources

The **xtpmd_plugin** infrastructure enables access to data streams falling into these five categories:

- **Blade-level SEDC data**
- **Cabinet-level SEDC data**
- **Blade-level power and energy (PMDB) data**
- **Cabinet-level power and energy (PMDB) data**
- **Application- and job-level information**

All data available directly via the plugin interface is also published into the Cray PMDB. Other data present in the Cray PMDB can be useful to plugin developers, consumers of the data provided by the plugin(s), or both. See section VI for more information.

A. Blade-Level SEDC Data (BC SEDC)

Blade level SEDC data available for different blade types varies. All Cray® XC™ blades have sensors that support temperature, voltage, current, and power, as well as a few miscellaneous status sensors. For example, a system in the Cray Chippewa Falls data center containing a mixture of supported blade types has over 562 unique blade-level sensor IDs. For brevity, we only show a subset of them in detail in this paper. Table I lists sensors in these categories with Nodes 1-3, and socket 1 edited out. The command **xtgetsedcvalues -l -t bc** can be used to query these data on an SMW running the latest Cray software.

B. Cabinet-Level SEDC Data (CC SEDC)

Cabinet level SEDC data is also available for a large numbers of sensors falling into the same basic categories as seen for the blade-level SEDC sensors in III-A, with 191 unique sensors on the same system sampled in III-A. The command **xtgetsedcvalues -l -t cc** can be used to query this type data on an SMW running the latest Cray software.

C. Cabinet- and Blade-Level Power and Energy Data (CC and BC PMDB)

The PMDB data is more normalized. Table II shows the default enabled sensors supported on Cray® XC™ systems.

As noted in [12] the CPU and Memory sensors listed in Table II are supported in the newest Cray® XC™ blades.

D. Application- and Job-Level Information

The application- and job-level data available via the plugin is a bit different than the previous four sources that support streaming sensor data. While the application- and job-level data stream out of this interface, the application and job data is event-driven. The supported events are enumerated in the **xtpmd_plugin.h** header file, as shown in Listing 4 of Section II. The nine defined events are all handled by the same user registered handler. Listing 8 shows output from the example plugin running on a system at the Cray Chippewa Falls data center. Some of the defined events may

Table I
BC SEDC Data

ID	Sensor Description	Unit
1257	BC_T_ARIES_TEMP	degC
1300	BC_T_NODE0_CPU0_TEMP	degC
1301	BC_T_NODE0_CPU1_TEMP	degC
1308	BC_T_NODE0_CPU0_CH0_DIMM0	degC
1312	BC_T_NODE0_CPU0_CH1_DIMM0	degC
1636	BC_T_NODE0_S0_VRM_CTEMP	degC
1637	BC_T_NODE0_S0_VRM_MTEMP	degC
1796	BC_T_NODE0_PCH_THERMAL	degC
1200	BC_V_VDD_0_9V	V
1201	BC_V_VDD_1_0V	V
1202	BC_V_VDD_1_0V_OR_1_3V	V
1203	BC_V_VDD_1_2V_GTP	V
1204	BC_V_VDD_1_2V_HSS	V
1206	BC_V_VDD_1_8V_HSS	V
1207	BC_V_VDD_2_5V_HSS	V
1208	BC_V_VDD_3_3V_PDC	V
1210	BC_V_VDD_3_3V_HSS	V
1211	BC_V_VDD_3_3V_MICROA	V
1213	BC_V_VDD_5_0V	V
1216	BC_V_VDD_13_0V_STDBY	V
1258	BC_V_ARIES_VDD_VCORE	V
1259	BC_V_ARIES_VDD_1V0	V
1260	BC_V_ARIES_VDD_1V8	V
1261	BC_V_ARIES_VDD_3V3	V
1262	BC_V_ARIES_VDD_13V0	V
1293	BC_V_BB_VERT_IVOC0_ECB_SOURCE_VOLT	V
1654	BC_V_NODE0_S0_VCC_OUT	mV
1708	BC_V_NODE0_S0_MVIN	mV
1710	BC_V_NODE0_S0_VDR01_OUT	mV
1712	BC_V_NODE0_S0_VDR23_OUT	mV
1263	BC_I_ARIES_VCORE_CURRENT	centiA
1264	BC_I_ARIES_1V0_CURRENT	centiA
1265	BC_I_ARIES_1V8_CURRENT	A
1287	BC_I_BB_VERT_IVOC0_ECB_VDD_CURRENT	A
1288	BC_I_BB_VERT_IVOC1_ECB_VDD_CURRENT	A
1676	BC_I_NODE0_S0_VCC_IN	mA
1680	BC_I_NODE0_S0_VCC_OUT	mA
1732	BC_I_NODE0_S0_VDR01_IN	mA
1734	BC_I_NODE0_S0_VDR23_IN	mA
1736	BC_I_NODE0_S0_VDR01_OUT	mA
1738	BC_I_NODE0_S0_VDR23_OUT	mA
1468	BC_P_NODE0_CPU0_CH0_DRAM_ACC	J
1469	BC_P_NODE0_CPU0_CH1_DRAM_ACC	J
1470	BC_P_NODE0_CPU0_CH2_DRAM_ACC	J
1471	BC_P_NODE0_CPU0_CH3_DRAM_ACC	J
1500	BC_P_NODE0_CPU0_VCC_ACC	J
1516	BC_P_NODE0_CPU0_PCKG_ACC	J
2776	BC_P_NODE0_GLOBAL_PROC_POWER	W
1266	BC_H_ARIES_VRM_FLT	status
1267	BC_H_ARIES_VRM_FLT_BITS	status
1268	BC_H_ARIES_VRM_HOT	status
1273	BC_H_NODE0_IVOC_ECB_FAULT	status
1295	BC_H_BB_VERT_IVOC0_ECB_FAULT	status
1296	BC_H_BB_VERT_IVOC1_ECB_FAULT	status
1844	BC_L_NODE0_CPU0_MEM_THROTTLE	%
1845	BC_L_NODE0_CPU1_MEM_THROTTLE	%
1872	BC_L_NODE0_CPU0_CPU_THROTTLE	N
1873	BC_L_NODE0_CPU1_CPU_THROTTLE	N

Table II
CC AND BC PMDB DATA

ID	Sensor Description	Unit
0	Cabinet Power	W
1	Cabinet Energy	J
2	Cabinet Voltage	mV
3	Cabinet Current	A
8	Cabinet Blower Power	W
16	HSS Power	W
17	HSS Energy	J
32	Node 0 Power	W
33	Node 0 Energy	J
36	Node 0 CPU Power	W
37	Node 0 CPU Energy	J
40	Node 1 Power	W
41	Node 1 Energy	J
44	Node 1 CPU Power	W
45	Node 1 CPU Energy	J
48	Node 2 Power	W
49	Node 2 Energy	J
52	Node 2 CPU Power	W
53	Node 2 CPU Energy	J
56	Node 3 Power	W
57	Node 3 Energy	J
60	Node 3 CPU Power	W
61	Node 3 CPU Energy	J
68	Node 0 Memory Power	W
69	Node 0 Memory Energy	J
76	Node 1 Memory Power	W
77	Node 1 Memory Energy	J
84	Node 2 Memory Power	W
85	Node 2 Memory Energy	J
92	Node 3 Memory Power	W
93	Node 3 Memory Energy	J

not be seen by a plugin depending on the workload manager used at the customer site.

The example in Listing 8 shows two sequences of job_start, app_start, app_end, job_end. Both jobs are run by the same user, reserve and use only one node (nid 2), and run a single application. Note that lines are wrapped to fit the formatting of this paper.

Listing 8. Example Application Output Data

```
ts=1490734709583873,event=6,userid=1205,jobid=3880.sdb,
    apid=0,nids=',2'
ts=1490734713023993,event=1,userid=1205,jobid=3880.sdb,
    apid=706691,nids=',2'
ts=1490734718511193,event=2,userid=1205,jobid=3880.sdb,
    apid=706691
ts=1490734719788394,event=7,userid=1205,jobid=3880.sdb,
    apid=0
ts=1490734786322464,event=6,userid=1205,jobid=3881.sdb,
    apid=0,nids=',2'
ts=1490734789715572,event=1,userid=1205,jobid=3881.sdb,
    apid=706717,nids=',2'
ts=1490734795285564,event=2,userid=1205,jobid=3881.sdb,
    apid=706717
ts=1490734796573906,event=7,userid=1205,jobid=3881.sdb,
    apid=0
```

IV. CONFIGURATION FILE OVERVIEW

The default configuration file for xtpmd plugin support is located on the Cray SMW or Cray external PMDB node at **/opt/cray/hss/default/etc/xtpmd_plugins.ini**. This file supports the example plugin code.

The default file that ships with the SMW release is filled with useful comments. For a more in depth description of the file format refer to [13].

Listing 9. Configuration File

```
# Copyright 2015 Cray Inc. All Rights Reserved.
#
# This is the configuration group that tells xtpmd where
# to find plugins and more specifically what plugins it
# should load.
[plugins]
## xtpmd communicates data to its plugins via a shared
## memory region. It can be tuned anywhere between 4-16MB
## in size. The default size is 4MB.
# shmsize=4194304
##
## Xtpmd can, if filesystem permissions allow for it,
## write its shared memory ID to a file when it starts
## up. This is provided for future use cases. As of now
## it is unused.
# shmid_path=/var/run/xtpmd/xtpmd.shmid
##
## Xtpmd loads plugins with the help of a sandbox process.
## The binary by default is called "xtpmd_plugd". If the
## binary is installed in a non-standard location xtpmd
## may be notified by editing this parameter
# plugd_path=/opt/cray/hss/default/bin/xtpmd_plugd
##
## Plugin instances are enabled by adding them to this
## string list. Elements are delimited with a semicolon.
## To disable a plugin, remove it from the list. To run no
## plugins, comment out the list.
##
## Instance list examples include:
##
# instances=csv;other
# instances=csv
# Plugin instances are configured using a dedicated ini
# file section named "plugin_<instance>" where instance
# is the name listed in the list above. It is permissible
# to have multiple instances of a single plugin as well.
[plugin_csv]
# The only required configuration parameter is the path
# to the shared object. This parameter is used by the
# plugin instance loader.
object=/opt/cray/hss/default/lib64/xtpmd_plugin_csv.so
# All other parameters are arbitrary. They are specific
# to this plugin instance. The CSV writer plugin has a
# few. They are:
# The base directory for CSV files. This parameter should
# probably be changed to a location with lots of storage
# space.
log_dir=/tmp
# Should the CSV writer log CC level PMDB data? If so a
# new CSV file will be created every <period> seconds.
# In this case a new one will be created every hour
pmdb_cc_enabled=yes
pmdb_cc_period=3600
# Blade & Node level PMDB data.
# Rotate a new file every 10 minutes
pmdb_bc_enabled=yes
pmdb_bc_period=600
# Cabinet level SEDC data.
# Rotate a new file once an hour
sedc_cc_enabled=yes
sedc_cc_period=3600
# Blade & Node level SEDC data.
# Rotate a new file every 10 minutes.
sedc_bc_enabled=yes
sedc_bc_period=600
# Alps application / batch data.
# Rotate a new file every day.
application_enabled=yes
application_period=86400
# Where timestamps are involved, the CSV plugin writes
# them out in number of microseconds since the UNIX Epoch.
```

Listing 9 shows a version of the default configuration file where text was modified to remove blank lines, and wrap lines to fit the required format of this paper.

One line that will likely need to be edited when testing or deploying a plugin in supervised mode is the **instances=csv** (key-value pair). It is commented out in the file, and to run the **plugin_csv** plugin, that line would need to be uncommented. As the comment line **instances=csv;other** suggests multiple plugins can be started and run concurrently.

The **object=*path_to_valid_lib.so*** must point at a plugin file runnable by the crayadm user.

The **xtpmd_plugd** support code reads in all of the **key-value pairs** for the plugin at load time, and makes them available to the plugin via the **char *plugin_get_instance_var()** helper function. With the exception of the **object=*path_to_valid_lib.so***, it is the responsibility of the plugin to define and implement handling of plugin-specific **key-value pairs**.

V. Plugin Testing

This section describes how to test an xtpmd plugin. Compiling and running the provided "csv" example plugin helps verify the environment is working.

To test a plugin, one can run it by hand or in what we call *unsupervised* mode. As shown in Figure 1, the command **ipcs** can be run to find **shmid** and **shmsize** parameters needed to call **xtpmd_plugd** from the command line. The third parameter is the **plugin_*name*** which in this case is **plugin_csv**. The fourth parameter is the configuration file to use.

```
my-smw~:> /tmp/xtpmd_plugin> ipcs | grep -v postgres
------ Shared Memory Segments --------
key        shmid   owner   perms bytes   nattch status
0x00000000 1441793 crayadm 600   4194304 1      dest
0x00000000 2031623 crayadm 600   524288  2      dest
0x7a060813 1900553 crayadm 600   4194304 2

my-smw~:> xtpmd_plugd 0x7a060813 4194304 plugin_csv ./
    xtpmd_plugins.ini
```

Figure 1: Unsupervised Mode Testing

There are two easy ways to kill an unsupervised plugin. The first is to enter **Ctrl-D** in the terminal window where it is running. The second is to issue a **kill -9** ***pid_of_plugin*** from another window.

Note if you are having problems starting your plugin, that you need to run it as **crayadm** and all the files must have correct permissions for crayadm. A good source if debug information is the **power_management-YYYYMMDD** log file found in the **smw:/var/opt/cray/log** directory. This is the file that captures **stderr** and **stdout** when a plugin is running in supervised mode.

Listing 10 shows the important parts extracted from a configuration file on a testing and benchmarking system in the Cray Chippewa Falls data center where we are running a Kafka [14] plugin. The **key-value pairs** are used to pass configuration information into the plugin. This offers a great deal of flexibility without need to recompile the plugin or implement an ad hoc configuration file. The output was modified to obscure some IP address and internal system name information.

To test a new version of this plugin without shutting down the running copy:

- Copy the configuration file into a sandbox directory where crayadm has read permissions so that it can be edited without a chance of modifying the running running plugin if or when it gets restarted
- Edit the line "object=" to point at the new plugin.so
- Optionally edit:
  - The "kafka_broker_ip=" line to use a test broker
  - The "*_topic= lines"
  - The "*_enabled= lines"
- Start the modified plugin as shown in Figure 1.

Listing 10. Kafka Example xtpmd_plugins.ini

```
[plugins]
instances=kafka
[plugin_kafka]
object=/home/crayadm/kafka/kafka-plugin.so
kafka_broker_ip=XXX.XXX.XX.XX:31092,XXX.XXX.XX.XX:31092
pmdb_cc_enabled=yes
pmdb_cc_topic=xtpmd-XX-pmdb-cc
pmdb_bc_enabled=yes
pmdb_bc_topic=xtpmd-XX-pmdb-bc
sedc_cc_enabled=yes
sedc_cc_topic=xtpmd-XX-sedc-cc
sedc_bc_enabled=yes
sedc_bc_topic=xtpmd-XX-sedc-bc
application_enabled=yes
application_topic=xtpmd-XX-application
pmdb_nodes_enabled=yes
pmdb_nodes_topic=xtpmd-XX-pmdb-nodes-key-value
sedc_sensor_info_enabled=yes
sedc_sensor_info_topic=xtpmd-XX-sedc-sensor-info-key-value
pmdb_sensor_info_enabled=yes
pmdb_sensor_info_topic=xtpmd-XX-pmdb-sensor-info-key-value
sm_expand_enabled=yes
sm_expand_topic=xtpmd-XX-source-key-value
```

Testing Checklist:

- **Talk to the system administrator before starting!**
  - Testing new processes on the SMW could impact system stability, or the ability of the SMW to respond to RAS events.
  - Appropriate planning and agreement with system stakeholders is required.
- Test for memory leaks.
- Test whether your plugin causes a high load on the SMW.
  - /usr/bin/top is your friend
- Make sure required libraries are in the path for crayadm at boot time, or your plugin may run in unsupervised mode but fail in production.
- If your plugin writes to local files, make sure something prevents them from filling up the file system.

VI. Plugin Design Considerations

Design considerations covered in this section fall into the following categories:

- Limiting The Plugin's Impact
- Library Usage
- Time Stamp Formatting
- Translating Binary Fields
- Getting Data Off Node
- Other Formatting Considerations

A. Limiting The Plugin's Impact

Now that we have a plugin environment to stream the data from the SMW, we have to be conscious of what resource this streaming capability will use. Since the SMW is the main management system for the Cray® XC™ system, additional processes and/or functions should be evaluated and reviewed by the administration team. Considerations would include but are not limited to: memory usage of the plugins, network usages in both number of connections and bandwidth, processes activity such as forking or threading and disk usage. If the desired plugin may use or consume too many of these resources, using an external PMDB server is an option to consider. The two primary designs detailed in this paper both stream the data off of the SMW using minor amounts of memory and network connection. In these designs, if a bandwidth issue were to arise on the SMW, the use text-based messages would likely be the main concern.

B. Library Usage

There are two primary ways to deal with any additional libraries that the plugin may require to run. The least intrusive method for the SMW may be to compile the plugin with static libraries. This may require additional care in the plugin creation. The other primary method would be to load or compile the different libraries the plugin may require on the SMW itself. Building and storing shared libraries on the SMW may be accomplished by placing them in **/usr/local/lib** which will be loaded from **/etc/ld.so.conf** file. This is the method that NERSC has used, and this configuration has survived an SMW software upgrade.

C. Timestamp Formatting

The callbacks for all five plugin data sources return an unsigned 64-bit 'ts' parameter that represents the number of microseconds since January 1, 1970 UTC. There are routines available to convert this data into several different string formats including time zones that match a site's location. It could be argued that not translating into a string is the correct choice for plugins where a computer is the primary consumer, as conversion can be done at any time if needed.

The code in Listing 11 is used in several internal plugins to convert 64-bit 'ts' into a standards based string. The choice to use Coordinated Universal Time (UTC) was made to remain site-independent, and avoid disruptive events such as Daylight Saving Time.

Listing 11. Generate RFC 3339 encoded Time String

```
/**
 * gen_time_string - Generate RFC 3339 encoded string,
 *                   relative to the Coordinated Universal
 *                   Time (UTC) from given ts microseconds.
 *
 * @param  ts[in]  Microseconds since January 1, 1970 UTC
 * @return RFC 3339 encoded string
 */
static char *gen_time_string(uint64_t ts)
{
  GTimeVal tv;

  if (ts == 0)
    ts = g_get_real_time();
  tv.tv_sec  = ts / 1000000;
  tv.tv_usec = ts % 1000000;
  return g_time_val_to_iso8601(&tv);
}
```

D. Translating Binary Fields

There are multiple fields provided to the five callback functions that at some point need to be translated from their native binary representation into something another computer program or a human can understand. Timestamps were covered in VI-C. This section describes nid-to-cname, source-to-cname, and scan ID translations.

The code sequence in Listing 12 loads a hash table with 'nid' to 'cname' translations. These translations can be useful for plugins implementing **recv_apevent** handlers. For clarification, a 'nid' is Cray shorthand for a Cray Node ID. Cray uses the term 'cname' for strings that represent physical locations in the system. All compute nodes in Cray® XC™ systems have both a 'nid' and a 'cname'. Translating back and forth can be necessary for tasks such as calculating total power consumption of a set of nodes assigned to a job or application.

Listing 12. Load nid2cname Hash Table From PMDB

```
str = g_strdup("SELECT comp_id, nid_num FROM pmdb.nodes");
res = PQexec(conn, str);
if (PQresultStatus(res) == PGRES_TUPLES_OK) {
  int row, count = PQntuples(res);
  for (row = 0; row < count; row++) {
    g_hash_table_insert(pmdb_hash->nid2cname,
        GUINT_TO_POINTER(strtoull(PQgetvalue(res, row, 1)
        , NULL, 0)), g_strdup(PQgetvalue(res, row, 0)));
  }
} else {
  printf("Failed: >>> %s <<< !\n", str);
  PQfinish(conn);
  return -1;
}
g_free(str);
```

Cray 'cnames' are used to name physical locations of cabinets, blades, nodes, and other components. The next code block (Listing 13), shows code that loads cabinet-level source to 'cname' translations into the **source2cname** hash table. These translations are needed for cabinet-level SEDC and PMDB sensor data.

Listing 13. Cabinet ‘cnames’ into source2cname Hash Table
```
str = g_strdup("SELECT cname2source(name) as source, name
    FROM sm.expand('rt_l1', 's0')");
res = PQexec(conn, str);
if (PQresultStatus(res) == PGRES_TUPLES_OK) {
  int row, count = PQntuples(res);
  for (row = 0; row < count; row++) {
    g_hash_table_insert(pmdb_hash->source2cname,
        GUINT_TO_POINTER(strtoull(PQgetvalue(res, row, 0)
        , NULL, 0)), g_strdup(PQgetvalue(res, row, 1)));
  }
} else {
  printf("Failed: >>> %s <<< !\n", str);
  PQfinish(conn);
  return -1;
}
g_free(str);
PQclear(res);
```

Listings 14 and 15 are near clones of the above, with only the “SELECT” string that reads data from PMDB changing to read out blade- and node-level translations, respectively.

Listing 14. Blade ‘cnames’ into source2cname Hash Table
```
str = g_strdup("SELECT cname2source(name) as source, name
    FROM sm.expand('rt_l0', 's0')");
res = PQexec(conn, str);
if (PQresultStatus(res) == PGRES_TUPLES_OK) {
  int row, count = PQntuples(res);
  for (row = 0; row < count; row++) {
    g_hash_table_insert(pmdb_hash->source2cname,
        GUINT_TO_POINTER(strtoull(PQgetvalue(res, row, 0)
        , NULL, 0)), g_strdup(PQgetvalue(res, row, 1)));
  }
} else {
  printf("Failed: >>> %s <<< !\n", str);
  PQfinish(conn);
  return -1;
}
g_free(str);
PQclear(res);
```

Listing 15. Node ‘cnames’ into source2cname Hash Table
```
str = g_strdup("SELECT cname2source(name) as source, name
    FROM sm.expand('rt_node', 's0')");
res = PQexec(conn, str);
if (PQresultStatus(res) == PGRES_TUPLES_OK) {
  int row, count = PQntuples(res);
  for (row = 0; row < count; row++) {
    g_hash_table_insert(pmdb_hash->source2cname,
        GUINT_TO_POINTER(strtoull(PQgetvalue(res, row, 0)
        , NULL, 0)), g_strdup(PQgetvalue(res, row, 1)));
  }
} else {
  printf("Failed: >>> %s <<< !\n", str);
  PQfinish(conn);
  return -1;
}
g_free(str);
PQclear(res);
```

In Listing 16, we load SEDC scan ID descriptions into the **sedc_ids** hash table. The descriptive strings describe the sensor with a human-readable name, as well as its unit of measure. The data shown in III-A and III-B were generated from the same PMDB table that this code is accessing.

Listing 16. SEDC Scan ID into sedc_ids Hash Table
```
str = g_strdup("SELECT sensor_id, sensor_name, trim(both
    from sensor_units) FROM pmdb.sedc_scanid_info");
res = PQexec(conn, str);
if (PQresultStatus(res) == PGRES_TUPLES_OK) {
  int row, count = PQntuples(res);
  for (row = 0; row < count; row++) {
    g_hash_table_insert(pmdb_hash->sedc_ids,
        GUINT_TO_POINTER(strtoull(PQgetvalue(res, row, 0)
        , NULL, 0)), g_strdup_printf("%s,%s", PQgetvalue(
        res, row, 1), PQgetvalue(res, row, 2)));
  }
} else {
  printf("Failed: >>> %s <<< !\n", str);
  PQfinish(conn);
  return -1;
}
g_free(str);
PQclear(res);
```

In listing 17 we load PMDB scan ID descriptions into the **pmdb_ids** hash table. The descriptive strings describe the sensor with a human-readable name, as well as its unit of measure. The data shown in III-C was generated from the same PMDB table that this code is accessing.

Listing 17. SEDC Scan ID into pmdb_ids Hash Table
```
str = g_strdup("SELECT sensor_id, sensor_name, trim(both
    from sensor_units) FROM pmdb.sensor_info");
res = PQexec(conn, str);
if (PQresultStatus(res) == PGRES_TUPLES_OK) {
  int row, count = PQntuples(res);
  for (row = 0; row < count; row++) {
    g_hash_table_insert(pmdb_hash->pmdb_ids,
        GUINT_TO_POINTER(strtoull(PQgetvalue(res, row, 0)
        , NULL, 0)), g_strdup_printf("%s,%s", PQgetvalue(
        res, row, 1), PQgetvalue(res, row, 2)));
  }
} else {
  printf("Failed: >>> %s <<< !\n", str);
  PQfinish(conn);
  return -1;
}
g_free(str);
PQclear(res);
```

Slight variations of code shown in Listings 12 through 17, have been used in several Cray internal plugins as well as the plugins running in production on the NERSC Cori system. Discussion of Cori in more detail is provided in Section VII.

E. Getting Data Off-Node

In addition to the Kafka method listed above, one could use RabbitMQ to send data from the SMW. Listing 18 shows a concentrated listing of the code to create a connection to RabbitMQ. The connection is built by first creating a new amqp connection and then setting the SSL options. The secure path is then used to connect to the RabbitMQ server. A defined account and password are needed to authenticate the connection. The connection handle is placed in the rcon structure of the plugin to be used when sending the data. Most of the error checking has been removed to conserve space.

Listing 18. NERSC RabbitMQ Connection Code Example
```
...
rcon->conn = amqp_new_connection();
socket = amqp_ssl_socket_new(rcon->conn);
amqp_ssl_socket_set_verify_peer(socket, 0);
amqp_ssl_socket_set_verify_hostname(socket, 0);
status = amqp_socket_open(socket, rcon->hostname, rcon->
    port);
r = amqp_login(rcon->conn, "/", 0, 131072, 0,
    AMQP_SASL_METHOD_PLAIN, rcon->login, rcon->password)
    ;
```

```
if (r.reply_type != AMQP_RESPONSE_NORMAL) {
 fprintf(stderr, "rabbitMQ login failure\n");
 sleep(3600);
 exit(1);
}
amqp_channel_open(rcon->conn, 1);
r = amqp_get_rpc_reply(rcon->conn);
...
```

Code shown in Listing 19 publishes the data to the RabbitMQ server.

Listing 19. NERSC RabbitMQ Publish Code Example

```
...
props._flags = AMQP_BASIC_CONTENT_TYPE_FLAG |
    AMQP_BASIC_DELIVERY_MODE_FLAG;
props.content_type = amqp_cstring_bytes("text/plain");
props.delivery_mode = 2; /* persistent delivery mode */
status = amqp_basic_publish(ctx->conn, 1,
     amqp_cstring_bytes(ctx->exchange),
     amqp_cstring_bytes(ctx->routingkey), 0, 0, &props,
     amqp_cstring_bytes(message));
...
```

F. Other Formatting Considerations

Formatting the data before transmission will obviously slow down the processing speed of the plugin. However, one reason for processing in the plugin is to speed up the ingest on the receiver side. We formatted the data into JSON syntax and added a bit of enrichment, which can only be gathered on the Cray SMW (or Cray external PMDB node). This provided a nice trade off. No official timing was done, but in general when JSON is fed to Logstash to feed Elastic [15], Logstash was able to process almost 2 times more data.

The example in Listing 20 shows a handler for PMDB streaming data that generates JSON formatted strings that are then passed to a plugin specific **xjson_output()** function. The **xjson_uint64** function calls **gen_time_string()** and uses the **source2cname** and **pmdb_ids** hash tables shown in Listings 13, 14, 15 and 17 above. The **xjson_uint64()** function can handle both ‘BC_PMDB’ data and ‘CC_PMDB’ data. A nearly identical **xjson_double()** (not shown) can handle ‘BC_SEDC’ data and ‘CC_SEDC’ data, it uses the **sedc_ids** hash table.

Listing 20. Example PMDB Data Callback

```
static void xjson_uint64(void *user_data, uint64_t ts, int
     source, int id, uint64_t value)
{
   xj_ctx_t *ctx = (xj_ctx_t *)user_data;
   gchar* source_str = g_hash_table_lookup(ctx->pmdb_hash
       ->source2cname, GUINT_TO_POINTER(source));
   gchar* id_str = g_hash_table_lookup(ctx->pmdb_hash->
       pmdb_ids, GUINT_TO_POINTER(id));
   char *tvs = gen_time_string(ts);

   xjson_check_for_rotation(ctx);
   snprintf(ctx->out_buf, (SD_MAX_LEN), "{\"ts\":\"%s\",\"
       cname\":\"%s\",\"id\":%d,\"id_string\":\"%s\",\"
       value\":%llu}\n", tvs, source_str, id, id_str, (
       long long unsigned int)value);
   g_free(tvs);
   xjson_output(ctx,ctx->out_buf);
}
```

Listing 21 shows formatted JSON generated by the **xjson_uint64()** function described above. The actual string generated is all on one line without extra spaces.

Listing 21. Example Job/Application JSON Output

```
{
  "ts": "2017-03-22T02:33:27.165380Z",
  "cname": "c5-0c0s2n3",
  "id": 56,
  "id_string": "Node 3 Power,W",
  "value": 51
}
```

The code examples in the section are from a plugin that writes its output to local files in plain text, or using **Lib Z** compression. The **json_output** function shown in Listing 22 checks if compression is requested and calls ether the **gzputs()** or the standard lib **fputs()** function. Unfortunately, error handling is slightly different for the two functions.

Listing 22. Example File Output Function

```
void xjson_output(xj_ctx_t *ctx, const char *output_string
    )
{
  int len = -1, rc = 0;

  if(ctx->use_gzip == TRUE) {
    len = gzputs(ctx->use.lfgz, output_string);
    if (len <= 0) {
      gzerror(ctx->use.lfgz, &rc);
      fprintf(stderr, "%s: Failed gzputs() to file %s,
           errnum: %d.\n", __func__, ctx->lf_name, rc);
      return;
    }
  } else { // Uncompressed output
    if (fputs(output_string, ctx->use.lf) < 0) {
      rc = errno;
      fprintf(stderr, "%s: Failed fputs() to file %s,
           errno: %d.\n", __func__, ctx->lf_name, rc);
      return;
    }
    len = strlen(output_string);
  }
  ctx->tfsz += len;
  return;
}
```

The long example in Listing 23 shows an implementation of an application and job information callback that formats its output as a string with JSON semantics. This example also uses hash tables to do the binary-to-string conversions that were built up in the code examples shown in VI-D.

Listing 23. Example Job / Application Callback

```
xjson_apevent(void *user_data, uint64_t ts, uint32_t event
     , uint32_t userid, char *jobid, uint64_t apid,
     uint32_t *nids, uint32_t nids_len)
{
  static char ap_out_buf[SD_MAX_LEN+AP_MAX_LEN];
  static char nid_list[AP_MAX_LEN];
  xj_ctx_t *ctx = (xj_ctx_t *)user_data;
  char *tvs = gen_time_string(ts);
  xjson_check_for_rotation(ctx);

  // Handle event we do not have a translation for
  if (event > sizeof(cte_strings)) {
    fprintf(stderr, "%s: invalid event %d\n", __func__,
         event);
    event = 0;
  }

  if (nids == NULL) {
    snprintf(ap_out_buf, (AP_MAX_LEN), "{\"ts\":\"%s\",\"
         event\":\"%s\",\"userid\":%u,\"job_id\":\"%s\",\"
         apid\":%llu}\n", tvs, cte_strings[event], userid,
          jobid, (long long unsigned int)apid);
  } else {
    int i, len=0;
```

```
    len += snprintf(nid_list, (AP_MAX_LEN), "{\"nid\":%u
        ,\"cname\":\"%s\"}", nids[0], g_hash_table_lookup
        (ctx->pmdb_hash->nid2cname, GUINT_TO_POINTER((
        guint64)nids[0])));
    for (i = 1; i < nids_len; i++) {
      len += snprintf(&nid_list[len], (AP_MAX_LEN - len),
          ",{\"nid\":%u,\"cname\":\"%s\"}", nids[i],
          g_hash_table_lookup(ctx->pmdb_hash->nid2cname,
          GUINT_TO_POINTER((guint64)nids[i])));
    }

    snprintf(ap_out_buf, (SD_MAX_LEN+AP_MAX_LEN), "{\"ts
        \":\"%s\",\"event\":\"%s\",\"userid\":%u,\"job_id
        \":\"%s\",\"apid\":%llu,\"nid_count\":%d,\"
        nid_cname_array\":[%s]}\n", tvs, cte_strings[
        event], userid, jobid, (long long unsigned int)
        apid, nids_len, nid_list);
  }
  g_free(tvs);\textbf{
  xjson_output(ctx,ap_out_buf);
}
```

Listing 24 shows formatted JSON generated by the **xjson_apevent()** function shown above. The actual string generated is all on one line without extra spaces.

Listing 24. Example JSON Formatted Output


```
{
 "ts": "2017-03-23T16:04:28.366738Z",
 "event": "APP_START",     "userid": 27216,
 "job_id": "1723832.sdb", "apid": 3801382,
 "nid_count": 2,           "nid_cname_array": [
  {"nid": 56, "cname": "c0-0c0s14n0"},
  {"nid": 57, "cname": "c0-0c0s14n1"}
 ]
}
```


VII. NERSC Plugin Deployment

At NERSC we have implemented the xtpmd plugin architecture based strongly on the work provided by Cray. Our extension added the ability to stream the data directly to RabbitMQ thus allowing it to flow directly into our NERSC data collect.

The NERSC design created a plugin for each of the five data sources described in Section III. This was a simple first step in parallelizing the data collection. This provided five different streams quickly. The next step was for data enrichment. Since the power and environmental data all contain the data's cname, while the job data contains the nid number, we had to enrich the job stream with some extra information, the corresponding cnames. This information is loaded into memory from the PostgreSQL database running on the SMW VI-D. We also loaded in the text strings of the different sensor IDs and conversions for the location IDs. Loading from the SMW database at startup guarantees that we have the most updated and correct data when inserting into the database. Also with Elastic [15], we do not have to worry if sensors change names or new ones get added when we do software upgrades, the new data is indexed as it arrives.

One other key task the job plugin does is break the node list of a running job into its individual nids and cnames creating two arrays. These arrays and all other information is placed on a single text JSON line and fed into RabbitMQ, at which point it moves into Elastic with minimal additional processing. All plugins for Cori basically feed the "NERSC Center-wide Data Collect" [6] at about 19K inserts a second or about 1.5 billion documents in about 66GB of storage a day.

A. *Some key points learned during this project.*

- Although restarting xtpmd should be non-intrusive, if HA is enabled on the SMW, this will cause the HSN to pause. HA seems to be configured to restart the whole subsystem if xtpmd has an issue. For this reason, it is recommended that SMW HA be set into maintenance mode for a xtpmd daemon restart.
- In the xtpmd_plugin.ini file, the default method to call the plugin prepends the value: 'plugin_' to the plugin name, see below in the example.
- Rebooting the SMW may cause a custom xtpmd_plugin.ini file to be overwritten with the default file. **(This is fixed in SMW 8.0 UP03.)**
- To restart a plugin without restarting xtpmd is easy. Just kill the plugin process. The xtpmd process will notice that the plugin has gone away and will restart it. This is also a good way to change configuration parameter.
- Make sure the libraries are in a location that the 'crayadm' user can read.

NERSC plugin to-do list:

- RabbitMQ is not a threaded library; the plugin must open multiple connections. Develop a way to round-robin plugin data flow to multiple RabbitMQ sockets.
- Devise graceful fallback and recovery if RabbitMQ goes away.
- Add monitoring for plugin communication failures.
- Develop a more graceful method to restart a plugin.

B. *About the code and configuration:*

The code itself is based on Cray's example from CUG2016 [3], extended to add the RabbitMQ library and some string processing to create a JSON output line which gets written to RabbitMQ. In Listing 25 you can see the configuration information that is in the Cori production **xtpmd_plugin.ini** file.

First, the plugin definitions (Listing 25 lines 1 and 2). These are the five defined NERSC plugins expected.

Listing 25. Plugin Definitions

```
[plugins]                                              1
instances=bp;bc;cp;cc;job                              2
[plugin_bp]                                            3
object=/opt/data_collect/pmd/blade_power_bp_plugin.so  4
power_bp_enabled=yes                                   5
hostname=rabbit.nersc.gov                              6
port=5671                                              7
exchange=ha-metric                                     8
routingkey=cori.corismw.sedc.power_bp                  9
system=cori                                           10
type=sedc                                             11
login=<account>                                       12
password=<passwd>                                     13
```

The ‘instances’ are the plugin names. That name is used in the next part of the configuration file. This block starts with the plugin name, which get the string ‘plugin_’ prepended to it.

Listing 25 line 3 starts the plugin-specific parameters for the **plugin_bp** plugin. Line 4 defines the full path to the shared library **bp_plugin.so** for this plugin. Next, on line 5 is an enable flag in case we would like to disable this plugin the next time xtpmd is restarted. The hostname and port (lines 6 and 7) are where the RabbitMQ service is located. Line 8 specifies the RabbitMQ exchange to be used; from that exchange telemetry gets routed based on the routingkey (in line 9) to Logstash which uses parts of the routingkey to place the data into the correct Elastic index. The system and type fields (lines 10 and 11) are used by Logstash to help manage the data flow. Finally, the last two lines are the account login and password information to initially connect to the RabbitMQ. At this point, we can make changes to the NERSC Center-wide Data Collect without having to change the plugin binaries.

We should have a word about data loss. Although the connections are all TCP and we have have not noticed any substantial data losses, the xtpmd infrastructure (by design) will drop data instead of backing up the stream in memory. If the plugin cannot keep up, xtpmd will drop what the plugin can not process. This would almost be like a UDP stream. We have not devised any method to save data either in memory or to disk if we have a RabbitMQ issue. If we have an Elastic issue, the RabbitMQ servers will do the spooling. Lastly, the ‘kill -9’ restart will also lose data both in the shutdown and restart at which time no data would be flowing to the NERSC infrastructure. With these reasons, we have tried to keep the plugins simple and fast; most enhancements and monitoring happening after it leaves the SMW and arrives at the NERSC infrastructure.

C. NERSC Cori Telemetry Visualization

This section show a few views created using the data being collected on Cori. The page layout is in Cori cabinet and row order, thus also giving a physical view to the data.

Figure 2 shows a detailed view of the power in each cabinet. Each graph is displaying the four nodes in that slot.

Figure 3 shows cabinet air temperatures. Note that blower cabinets are associated with the cabinet cname of the cabinet to its right. The exception is the end blower which has the cname of the cabinet to its left.

Figure 4 shows the detailed power of a slot. Each column lists the data for one of the four nodes in that slot. The top two graphs show the summary for the slot.

Figure 5 is the CPU temperature view. This is showing the maximum temperature of any of the CPUs during that time period. We do this to summarize all 192 CPUs in each cabinet. Drill-down graphs are implemented to view more detailed information.

Figure 2: Cori Cabinet Power

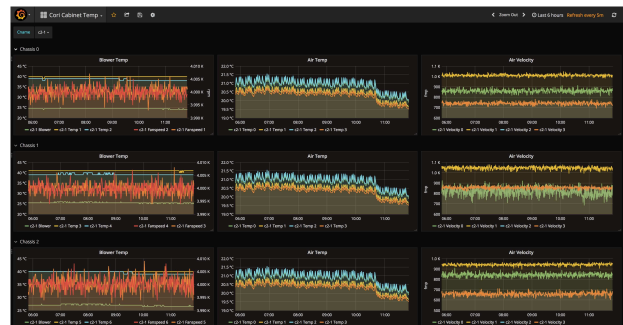

Figure 3: Cori Cabinet Temperature

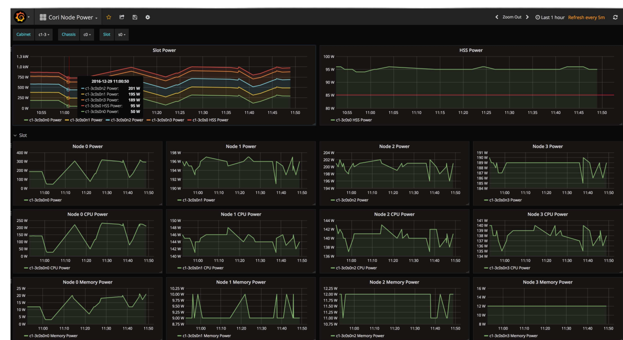

Figure 4: Cori Node Power

Figure 5: Cori CPU Temperature

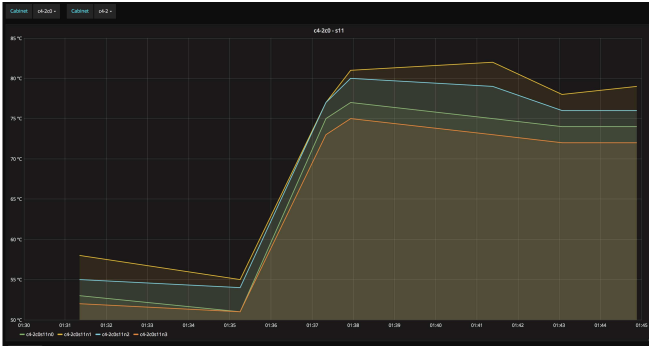

Figure 6: c4-2c0s11n1 Node Temp

Figure 7: c4-2c0s11 Power Profile

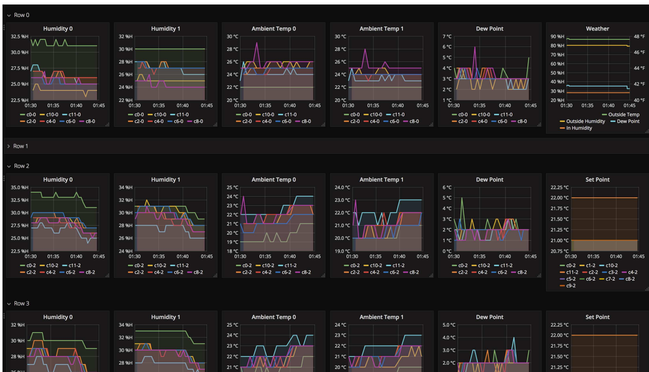

Figure 8: c4-2 Blowers

D. *NERSC Cori Problem Gathering*

NERSC recently had a temperature throttling event and the NERSC Center-wide Data Collect was used to examine the data visually. The summary that follows is for one node.

First we searched for any throttling events in the log data, and found: **2017-03-23T01:39:08.965711-07:00 c4-2c0s11n1 xtconsole 2173 p0-20170322t150819 [console@34] CPU22: Package temperature above threshold,**

Figure 9: c4-2 Water Profile

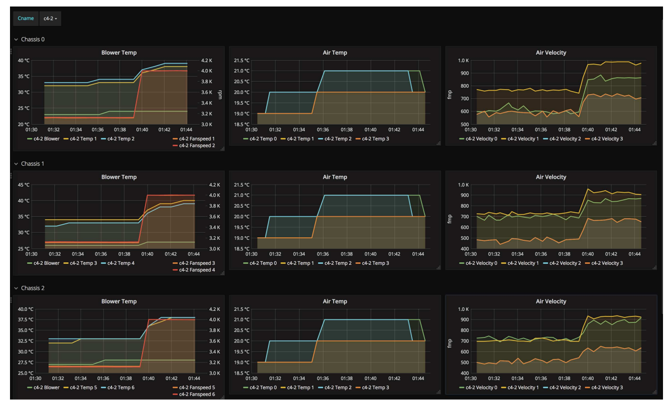

Figure 10: c4-2 Air Profile

cpu clock throttled

With the event time, **3/23 at 01:39** and location **c4-2c0s11n1**, we used the Grafana [16] dashboards to gather insight into what had happened on the system. Figure 6 shows the temperature of all four nodes in Slot 11, with Node 1 reaching 80C at about 1:39 and that it is the only processor to do so. In Figure 7 we show a graph of the affected slots power data at the time of the throttling event. The power profiles of all the nodes look very much alike, none showing any significant differences.

After viewing the CPU temperature, we now turned our gaze to the cabinet water data. Figure 9 is the water ingress and egress temperatures, the water pressure, and the value position. The steep part of the valve graph was taking place at the same time as the reported thermal throttling event.

Next we viewed the cabinet blower screens. In Figure 10, first column shows rising temperatures in each chassis and a large step in fan speeds. By Cray design, all blower cabinet fans in a row must run at the same speed. The fan speed is changed in response to the hottest socket in the row of cabinets. The valve position (controlling water flowing through each cabinet) is modulated as needed to control the

cabinet outlet air temp to a configurable set-point [17], [18].

In Figure 10, the third column shows air velocity and as expected with an increase in fan speed we see an increase in airflow. Figure 10, the second column shows the cabinet outlet air temperature. These air temps sensors are positioned in the airflow after the water coil has removed most (if not all) heat generated by the blades in the cabinet. There was a 1C rise about 3 seconds before the event, that would have triggered the change in water valve position noted above and seen in Figure 9. We should point out that the rise in outlet air temperature above the system set-point was temporary, and shows the short lag in thermal response to the change the inlet water valve position.

Lastly, Figure 8 shows the general environment of the blower cabinets. Note the outside weather as shown in the upper right corner of the figure was a steady 47.5F.

VIII. Conclusion

With CUG 2016 providing the groundwork for Cray® XC™ system xtpmd plugin development, this work dived more deeply into details needed to develop a plugin. This paper shows the process, starting with the base plugin calls, followed by plugin enhancements and data enrichment. Finally, the NERSC dashboard's data visualization demonstrates how the data was used to solve a problem facing the center. There is more work to do – this is only the beginning of the uses for the data, be they monitoring or machine learning.

This has been a collaborative effort between Cray, the user community, and customers to design and test the streaming telemetry plugin capability. This new monitoring capability allows customers to collect power, energy, thermal, and application data and to make that data available to system administrators, application developers, and the HPC research community as appropriate given site-level infrastructure and policy.

References

[1] "Cray System Environment Data Collection (SEDC)," (Accessed 27.March.17). [Online]. Available: https://pubs.cray.com/#/00446663-DC/FA00237186

[2] "Cray Power Management Database (PMDB)," (Accessed 27.March.17). [Online]. Available: https://pubs.cray.com/#/00446663-DC/FA00223922

[3] S. Martin, "CUG 2016 Cray XC Power Monitoring and Management Tutorial," *Proceedings of the Cray User Group (CUG)*, 2016, (Accessed 22.March.17). [Online]. Available: https://cug.org/proceedings/cug2016_proceedings/includes/files/tut103.pdf

[4] "Redis," (Accessed 27.March.17). [Online]. Available: https://redis.io/

[5] "Hiredis," (Accessed 27.March.17). [Online]. Available: https://github.com/redis/hiredis

[6] C. Whitney, T. Davis, and E. Bautista, "CUG 2016 NERSC Center-wide Data Collect," *Proceedings of the Cray User Group (CUG)*, 2016, (Accessed 22.March.17). [Online]. Available: https://cug.org/proceedings/cug2016_proceedings/includes/files/pap101.pdf

[7] S. Martin and M. Kappel, "Cray XC30 Power Monitoring and Management," *Proceedings of the Cray User Group (CUG)*, 2014, (Accessed 28.March.16). [Online]. Available: https://cug.org/proceedings/cug2014_proceedings/includes/files/pap130.pdf

[8] S. Martin, D. Rush, and M. Kappel, "Cray Advanced Platform Monitoring and Control (CAPMC)," *Proceedings of the Cray User Group (CUG)*, 2015, (Accessed 28.March.16). [Online]. Available: https://cug.org/proceedings/cug2015_proceedings/includes/files/pap132.pdf

[9] "CAPMC API Documentation," (Accessed 1.April.16). [Online]. Available: http://docs.cray.com/PDF/CAPMC_API_Documentation_1.2.pdf

[10] "XC Series Power Management Administration Guide (CLE 6.0.UP03) S-0043," (Accessed 27.March.17). [Online]. Available: http://docs.cray.com/PDF/XC_Series_Power_Management_Administration_Guide_CLE60UP03_S-0043.pdf

[11] "Rabbitmq," (Accessed 25.March.17). [Online]. Available: https://www.rabbitmq.com/

[12] S. Martin, D. Rush, M. Kappel, M. Sandstedt, and J. Williams, "Cray XC40 Power Monitoring and Control for Knights Landing," *Proceedings of the Cray User Group (CUG)*, 2016, (Accessed 22.March.17). [Online]. Available: https://cug.org/proceedings/cug2016_proceedings/includes/files/pap112.pdf

[13] "Desktop entry specification," (Accessed 31.March.17). [Online]. Available: https://freedesktop.org/wiki/Specifications/desktop-entry-spec/

[14] "Kafka," (Accessed 25.March.17). [Online]. Available: https://kafka.apache.org/

[15] "Elastic," (Accessed 27.March.17). [Online]. Available: https://www.elastic.co/

[16] "Grafana," (Accessed 27.March.17). [Online]. Available: https://grafana.com/

[17] G. Pautsch, D. Roweth, and S. Schroeder, "The Cray® XC© Supercomputer Series: Energy-Efficient Computing," 2013, (Accessed 22.March.17). [Online]. Available: http://www.cray.com/sites/default/files/WP-XC30-EnergyEfficiency.pdf

[18] B. Draney, J. Broughton, T. Declerck, and J. Hutchings, "Saving Energy with Free Cooling and the Cray XC30," *Proceedings of the Cray User Group (CUG)*, 2013, (Accessed 22.March.17). [Online]. Available: https://cug.org/