Use of the ERD for administrative monitoring of Theta

Alex Kristiansen
Argonne Leadership Computing Facility
Argonne National Lab
9700 Cass Ave, Lemont, IL
alexk@anl.gov

Abstract— **Monitoring the state of an HPC cluster in a timely and accurate fashion is critical to most system administration functions. For many Cray users, the first step in monitoring is ingestion of log files. Unfortunately, log parsing is an inherently inefficient process, requiring multiple software components to read and write from files on disk. Cray's own utilities use a message bus, the Event Router Daemon (ERD), for a wide variety of purposes. At the Argonne Leadership Computing Facility (ALCF), we have begun to use this message bus for monitoring via a client library written in Go, allowing us to read in structured data directly from Cray's services, and in many instances, bypass log files entirely. In this paper we will examine the implementation and utilization of this approach on our 4392 node XC40, Theta, as well as the overall benefits and drawbacks to using the ERD for real-time monitoring.**

I. INTRODUCTION

In 2016, the Argonne Leadership Computing Facility (ALCF) acquired Theta [4], which is now a 4392-node, 24-rack Cray XC40 system accompanied by a 10PB Lustre file system. Each Theta node is equipped with a 64-core Intel Xeon Phi 7230 and 192GB RAM. [1] For the first few months of Theta's operation, administration and system diagnosis was mostly preformed by searching various log files and using all vendor-provided tools. By comparison, the ALCF's last machine was an IBM BG/Q, and the BG/Q monitoring stack relied on highly structured and centrally databased data. [2] Although using a log ingestion stack, such as ELK, was possible, we ran into a problem: the hardware error logs, which provide error information for various components, such as PCIe chips, Cray Aries NICs and processors, was in a binary format. Cray provides a tool, `xthwerrlog`, to parse this binary data, but the human-readable, multi-line text was sub-optimal for machine parsing. There was an obvious use-case for placing this data in a database; we could be notified automatically of failures, and modern database tools would allow us to track long-term trends of correctable and uncorrectable errors, potentially allowing us to find and replace suspect hardware before it fails. In addition, such data, when in an easily parsed format, would be of research interest.

It was soon discovered that this hardware error data reached the log file by way of the ERD, where it was transmitted in (mostly) the same binary format as the log files. It was soon decided that reading this error data by way of the ERD was more efficient than tailing a binary log file. As the ERD is reachable from outside the System Management Workstation (SMW), this also meant that we could avoid running any analytics software on the SMW. An initial client library was written, and soon after we decided to start monitoring other data sources via the ERD, most notably the console logs and System Environmental Data Collection (SEDCv2) data. This data, once parsed, is shipped to Elasticsearch and InfluxDB in a structured form. Although SEDC data already exists in a Cray-provided database, our use of the ERD allowed us to effortlessly export the data in real-time to our own analytics stack using the same interface used by Cray's own tools, without running any additional software on the SMW. This design choice meant that analytics and management software stacks were kept isolated.

As a result, we have created a number of client libraries, all written in the programming language Go, that are responsible for reading from the ERD, as well as parsing data and inserting it into various databases. This set of libraries has been dubbed Deluge. While development is ongoing, Deluge has proven to be production-ready, stable, and easy to maintain. To aid in the maintenance of these libraries, we developed a number of harnesses that serve to automatically re-implement Cray's own libraries for parsing SEDC and hardware error data. These libraries allow the Deluge parsing backend to be regenerated with a single script, making it effortless to update in response to changes made in Cray's parsing logic. Deluge was developed without the benefit of detailed user-facing documentation for the ERD API. The presence of development libraries for the Hardware Supervisory System (HSS) is not well-advertised, so initial development was done using only tcpdump logs of ERD messages. With the help of Cray's development libraries for the Gemini Hardware Abstraction Layer (GHAL) and HSS, the rest of the development process proved to be trivial, as the provided header files contained the struct definitions needed to interface with various Cray components.

As far as we are aware, this approach to monitoring an XC40 appears to be unique, both in philosophy and design. The design and deployment of Deluge has proven to be much simpler than expected. This success can be attributed to a number of factors: the relative stability of the ERD's API, our use of the Go language, the choice to re-implement Cray's hardware error parsing logic in an automated fashion, and extensive unit testing to verify program correctness. As a result of the above factors, Deluge has proven to be useful in day-to-day operations, allowing us to store data in a machine-friendly and DevOps friendly format that is easy to visualize, monitor and store using modern, industry-standard tools.

The rest of this paper is structured as follows: In section II we will describe the structure of Deluge and its various helper libraries, and the justification for this design. In section III we will discuss the performance characteristics of the Deluge libraries, how that performance has evolved, and compare it to the corresponding Cray libraries. In section IV we will discuss how Deluge is being used in production at ALCF, and how it has assisted in the administration of Theta.

II. Design

We established a minimal set of requirements for a monitoring codebase before starting Deluge. It should be written in a language with good performance characteristics, memory-safety and easy concurrency primatives, and the libraries should be easily extensible and scalable.

We chose Golang based on these requirements, and it has proven to be a sound choice. Although Go is a relatively young language, with Go 1.0 being released in March of 2012 [3], we felt it was a good candidate based on our needs. It's memory safe, strongly typed, garbage collected, has easy to use concurrency primitives, a rich standard library, and is under active development. The time to release for Deluge was relatively short and free of serious bugs due in part to these characteristics.

Deluge is separated into three major backend libraries: `events`, `hwerrcore`, and `sedccore`. A number of monitoring daemons implement these libraries, named `sedclistener`, `hwerrlistener`, and `consoles`. The Deluge ecosystem also contains a CLI application called `ghal_harness` that assists in the maintenance of `hwerrcore`. A diagram demonstrating how deluge fits into the rest of ALCF's Cray ecosystem can be found in fig. 1.

Deluge takes a very different philosophical approach to handling data compared to Cray's own utilities. Deluge is designed to sit alongside Cray's tools and passively listen for data, and does not replace any Cray functionality. Data produced by Deluge is structured and machine-readable, so it can be handed off to other data analyis software, or inserted directly into a database. Many of Cray's tools provide data as unstructured text, like the events log, or the output produced by `xthwerrlog`. This design choice on Cray's part influenced the decision to make Deluge as comprehensive as it is, as we discovered that if the `xthwerrlog` functionality was redesigned from scratch to produce structured data that could easily be inserted into a database, the resulting library would have superior performance than an otherwise identical library that used regex to parse the human-readable string data returned by the underlying library calls that backed `xthwerrlog`.

Although Deluge is written in Go and none of the monitoring code is built against Cray's libraries, we did make use of the development packages provided by Cray. Many of these development packages contain struct definitions that were used to make sure that Deluge was correctly handling the data it received, and it provided some limited insight into the internal logic of the ERD and the `hwerr` subsystem. We developed Deluge using insights provided by the following packages: `lsb-cray-hss-smw-headers-pub`, `cray-gni-headers`, `lsb-cray-hss-smw-devel`, and `cray-gni-devel`. All of these packages are available to install from the mom nodes or the SMW.

Data flow starts at `events`, which is the only library that directly connects to the ERD on the SMW. As Deluge is a monitoring codebase, and not designed for command and control of the cluster, `events` is an extremely simple library, and its functionality is limited to sending 'subscribe' events to the ERD and reading in the events it receives back. Despite this brevity, `events` is multithreaded and has scaled up to the 500,000 events a minute generated by SEDC on Theta. Consumers can read from `events` either by receiving a buffered channel from the library, or calling a `Read()` method in a loop. In either case, the API returns a struct similar to Cray's own `rs_event_t` struct. It can also optionally parse the service IDs and length headers that are used at the start of `ev_data` in some events. This library is highly generic, and is built so that other libraries can reliably receive and parse network data.

`Sedccore` contains the parsing logic and data definitions for parsing SEDC events. The core parsing logic is extremely compact, and the majority of the code consists of autogenerated constants that turn SEDC scan IDs into strings. The relative simplicity of SEDC for Deluge is due to the configuration of ALCF's Cray machines. Cray introduced a new version of SEDC (referred to internally as SEDCv2) in UP03. Development on Deluge was started shortly after all of ALCF's Cray machines were moved to UP03, negating any need to add in support for both versions of SEDC. The process used to autogenerate the Scan ID maps from Cray's definitions is also relatively simple. The `sedc_scanid_info` table on the PMDB contains the mapping of Scan IDs to string names, and a simple 100-line Go program turns a CSV dump of the table into a Go source file.

`Hwerrcore` is the largest and most sophisticated library in Deluge, with auto-generated code totaling 76,000 lines. It supports all Aries errors, as well as Sandy Bridge and Knight's Landing (KNL) machine check errors. These limitations, obviously, were designed around ALCF's Cray installation. The most interesting part of `hwerrcore` is the autogeneration: a CLI tool called `ghal_harness` builds a simple C application against Cray's `libxthwerrdecode` library. This code generates a hardware error event for every known Aries error, and parses it using the verbose output of `ghal_decode_error()`. This produces the same output as if one passed `--decode` to `xthwerrlog`. Another component, written in Go, parses this output, and uses it to generate maps containing the variables needed to parse each individual hardware error. On an XC40, hardware error events consist of, among other things, a 16 bit error code and an array of 8 unsigned 64 bit memory-mapped

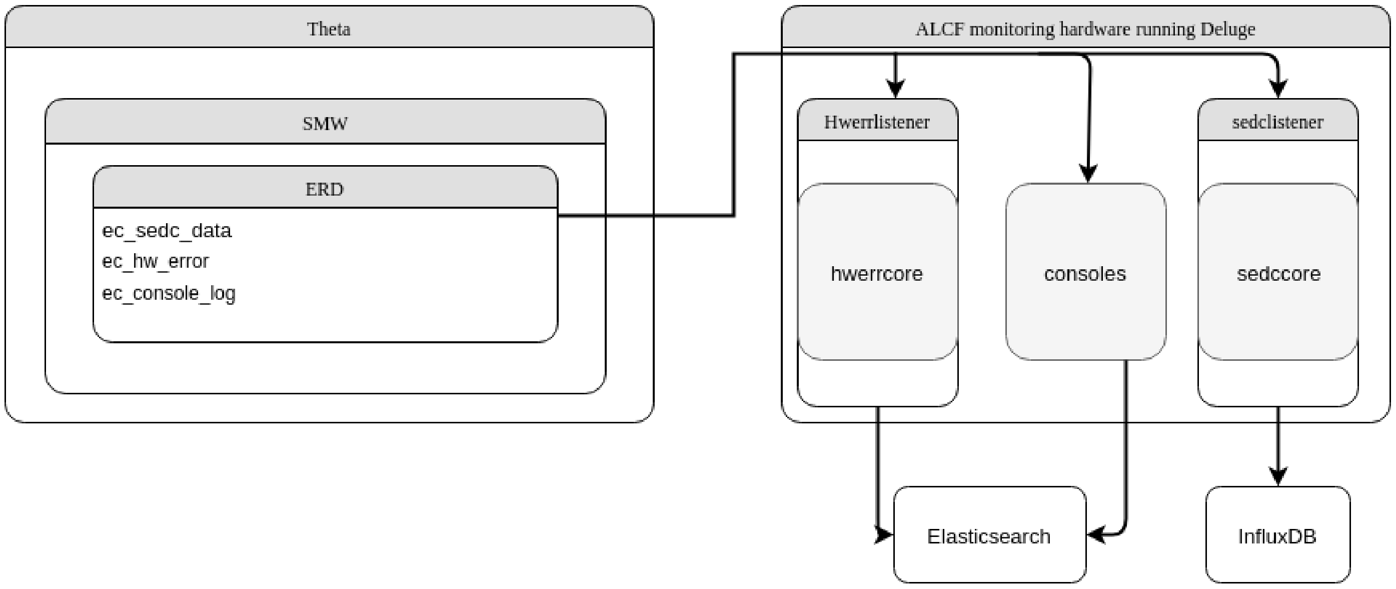

Fig. 1. Deluge's place within the rest of the Cray ecosystem

registers (MMR). The resulting data structures are 2 dimensional maps, with the parent map corresponding to a single hardware error code, and the inner maps corresponding to individual fields of each MMR. This autogeneration harness significantly reduces the maintenance and upkeep of the Deluge codebase, as updating our code in response to changes on Cray's part requires running a single bash script. We added support for KNL and Sandy Bridge machine check errors by hand rather than using the harness to autogenerate them, due to their complexity. Although this did take time, it proved to be worth the effort, as we did find bugs in Cray's parsing of KNL machine check errors as a result. Both bugs were reported and Cray provided patches.

III. Performance

Measuring the performance characteristics of Go applications is a trivial process, as the Go toolchain includes benchmarking, profiling and tracing tools built in. We used these tools throughout the development of Deluge, and they optimize memory usage and runtime performance. However, we must put these performance numbers in context, which is more difficult. Many of the functions performed by Deluge are also performed by Cray's own libraries, however, Cray's libraries and Deluge take very different approaches to solving the same problem. For example, the function `ghal_decode_error()` takes a hardware error data structure (like the data in the binary-encoded `hwerrlog` files) and returns a string buffer with formatted human-readable text that is printed by the CLI utility `xthwerrlog`. The equivalent Deluge function, `ParseHwerr` takes the same data structure, but returns a Go struct that contains the same detailed information about the MMRs. This is an important caveat to keep in mind as we discuss performance and compare difference between Cray's software and our own.

For those not familiar with Go's benchmarking tools, a short overview may be helpful. Benchmarks in Go take the form of a function with the signature `BenchmarkBENCHNAME(b *testing.B)`. The data structure passed into the function contains a counter, `b.N`, that acts as an iterator for the test. Tests are run multiple times with different values of `b.N`, to find an interval where the benchmark runs long enough to be timed reliably, because of this, not all benchmarks run with the same number of iterations. After the test is completed, it prints the sum of `b.N`, and the average time, in nanoseconds, that each iteration took. For example, the main benchmark for `ParseHwerr` looks like this:

```
for i := 0; i < b.N; i++ {
  _,_ := ParseHwerr(testEvent,0x591a1577)
}
```

For the sake of equalizing the test environment, Cray's libraries were also tested using Go's Benchmarking tools, by way of the language's Foreign Fuction Interface (FFI). Effort was taken to eliminate any additional overhead of the wrappers used to call C within Go.

The parsers for `ec_hw_error` data were the first to be completed and put into production, and so will be discussed first. The `ParseHwerr` benchmark was performed by passing a raw hardware error event (You can view a human-readable form of this data by passing the `-R` flag to `xthwerrlog`) for an Link Control Block (LCB) Transmit Lane Error and a KNL Integrated Memory Controller (IMC0) machine-check error. Both of these error messages were taken from real machine data, and represent two of the more common errors on the system. Deluge takes an entirely different code path for CPU machine check errors, and the available development headers from Cray suggests they are doing something similar. This function returns a Go struct that contains all the data that would be produced by passing the `--decode` flag to `xthwerrlog`, but in a data structure.

TABLE I

PARSING TIMES FOR HARDWARE ERROR MESSAGES

Function	LCB CRC error average time in NS (iterations)	IMC0 error average time in NS (iterations)	LCB CRC error iterations	IMC0 error iterations
ParseHwerr()	1590.6	2730.0	5,000,000	2,500,000
ghal_decode_error()	2347.0	1778.0	2,500,000	5,000,000

TABLE II

PARSING TIMES FOR SEDC MESSAGES

Function	Average parse time in NS	Number of iterations
ParseSedcErd()	4206.8	1,700,000
crms_print_sedc2_data_data()	27627.8	250,000

You can view the results of these tests in Table I. Deluge was much faster at parsing the LCB Transmit Lane error. However, Deluge performed much worse when parsing the IMC0 error. Using Go's built-in CPU performance profiler, we can see that this has nothing to do with the different parsing logic, but instead with the choice to use a hashmap to carry the detailed debugging MMR data. Although this may seem like a sound choice from a design perspective, as MMR data varies widely from error to error, there is additional overhead involved in using hashmaps. In the case of this test, the MMR data for the IMC0 error has more information than the MMR data for the LCB error, requiring more calls to the underlying mapassign function. According to the performance profiler, parsing the IMC0 error required spending 12.92% of runtime in the `mapassign_faststr` library call, while parsing the LCB error required 3.05% of runtime in `mapassign_faststr`. For Cray's code, we benchmarked the `ghal_decode_error()` function, using the same input data sets. Although Go's performance profiling does not provide insight into C functions, we can use it to calculate the overhead of the FFI/wrappers used for the benchmark. In this case, the call into the C function was 98.85% of the total benchmark runtime. Using this number, we can calculated the runtimes used on the table.

The setup for benchmarking the SEDC parser was slightly more complicated. We used `events` to dump a raw `ec_sedc_data` packet, then formatted it and declared it as a constant. The Deluge function under test was `ParseSedcErd()`, which takes an ERD packet, and returns a Go struct that contains an array of fully parsed SEDC events, each item in the array being a Go struct that represents a single SEDC datapoint. The results can be viewed in Table II. The biggest contribution to the runtime was the function that iteratively walks over every event in the packet and returns an array of parsed datapoints, followed by the map lookups that translate SEDC scan IDs into string names. Finding a Cray equivalent to this was challenging. Eventually the `crms_print_sedc2_data_data()` function, provided as part of the development packages on the mom nodes, was chosen, as the data it outputs is most similar to the data provided by `ParseSedcErd()`. Once again, the Deluge function outputs structured data, while Cray's writes human-readable output to a string buffer. According to the profiling data, the time spent in the C code represented 99.88% of runtime, making for an adjusted runtime of 27626.8 ns.

It should be emphasized that these numbers are not provided to prove the superiority or inferiority of any codebase, but more to lend legitimacy to Deluge's approach, and justify the time spent developing it, as opposed to simply attaching wrappers around existing code. To emphasize how these different design requirements relate to the performance numbers above, we modified the `ParseHwerr` LCB Benchmark, passing the resulting struct to a `printf` statement using Go's `%v` format specifier, which prints a detailed, human-readable string of the passed variable. In this test, across 1,500,000 iterations, the average time was 5542.4 ns, which is significantly slower than the Deluge or Cray implementations. In addition to this, we can't seem to find any production Cray CLI tools/daemons that make heavy use of `crms_print_sedc2_data_data()`.

IV. USAGE

The deployment and usage of Deluge at ALCF is relatively straightforward. Hardware error data and console data are shipped to Elasticsearch and visualized with Kibana. There is one main hardware error Kibana dashboard, which provides a 24-hour view of Machine Check Errors (MCE) and Aries error counts divided by node/link, and charts that break the data down by individual Aries errors and KNL machine check errors, as well as a view of any uncorrectable errors. This visualization has proven immensely valuable for providing a quick view of the machine state, as well as quickly identifying anomalous behavior. Without this dashboard, trying to get a similar idea of machine state would normally be accomplished by searching through multiple log files. With Elasticsearch and Kibana, it's easy to identify issues that need to be immediately addressed, as well as view trends over time.

We are also using the hardware error data in Elasticsearch to calculate the Bit Error Rate (BER) for Aries links, which

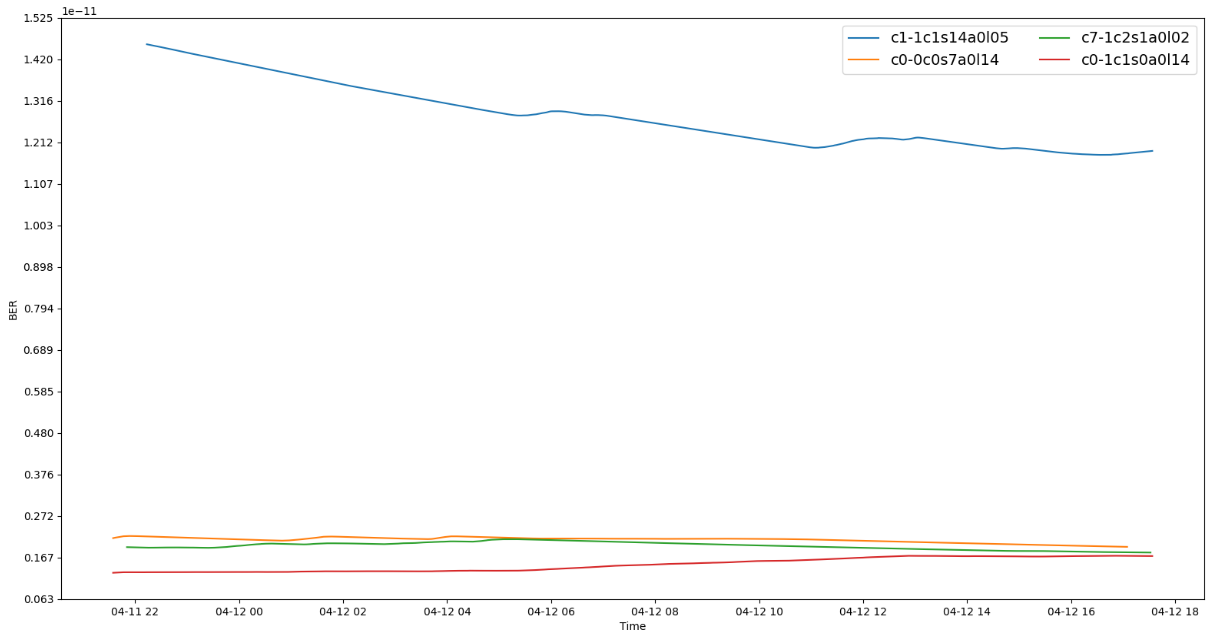

Fig. 2. A 10 hour graph of the top 4 Aries links on Theta by BER, showing one link (blue) with a high error rate compared to the rest of the links, demonstrating how we can visually track Aries links

is normally performed by the netwatch log. In our case, it made more sense to extract the BER data directly from the hardware error logs, rather than ingesting the netwatch logs separately, as as it provided us with detailed historical data that could be used to graph trends over time and in some cases, predict link failures due to an excessive BER.

To calculate the BER, a python script queries Elasticsearch for a list of all LCB CRC lane error messages. The script filters and reduces the data, and calculates the BER using the following formula:

$$\frac{\left(\frac{\text{TOTAL\_CRC\_ERR\_CNT}}{\text{GOOD\_UP\_CNT}+\text{TOTAL\_CRC\_ERR\_CNT}}\right)}{480}$$

This data is then graphed, allowing us to view trends over time. An example of this graph can be found in Fig. 2.

By default, if an Aries link reaches a sustained BER of 1.0e-10, it will be disabled, and the Aries will be quiesced for a re-route. It should be noted that this isn't the only failure mode for an Aries link, and that sometimes the BER can grow at a rapid or unpredictable rate. However, this graph has proven to be informative and useful.

SEDC data is stored in InfluxDB, and visualized using Grafana. The wealth of data available through SEDC has allowed us to create a number of real-time visualizations, using everything from air temps to CPU-level views of temperature and current.

We store all SEDC metrics going back since the Deluge deployment. We can discover trends in humidity, air temps, and water temps with a level of detail that often exceeds the upstream datacenter monitoring. This data has proven invaluable in discovering upstream datacenter environmental issues. On multiple occasions, these visualizations have been used to discover brief or subtle anomalies in inlet temperatures or humidity that would have been easy to overlook if one was just viewing the results of a SELECT statement in postgresql. An example of this can be found in Fig. 3.

V. Conclusions and Future Work

There are a number of ways in which we would like to expand Deluge with time, and also new avenues for making use of the data we already have. The most obvious would be to ingest more data from the ERD itself, as the three events we are collecting now (`ec_sedc_data`, `ec_console_log` and `ec_hw_error`) represent only a fraction of what the ERD is used for.

In addition, because Deluge allows us to store hardware error data and SEDC error data in a structured format, we can easily export data for use by outside researchers. Normally, the PMDB only retains SEDC data for a short period of time, and hardware error data is in a binary format, making these data sources much harder for researchers to use.

Code optimization is also a fruitful area of exploration. Although it would require significant refactoring, there is the possibility that moving from maps to autogenerated structs for carrying parsed MMR data could provide significant performance improvements. However, Go is a strongly typed language, and it's possible that the extra runtime overhead of

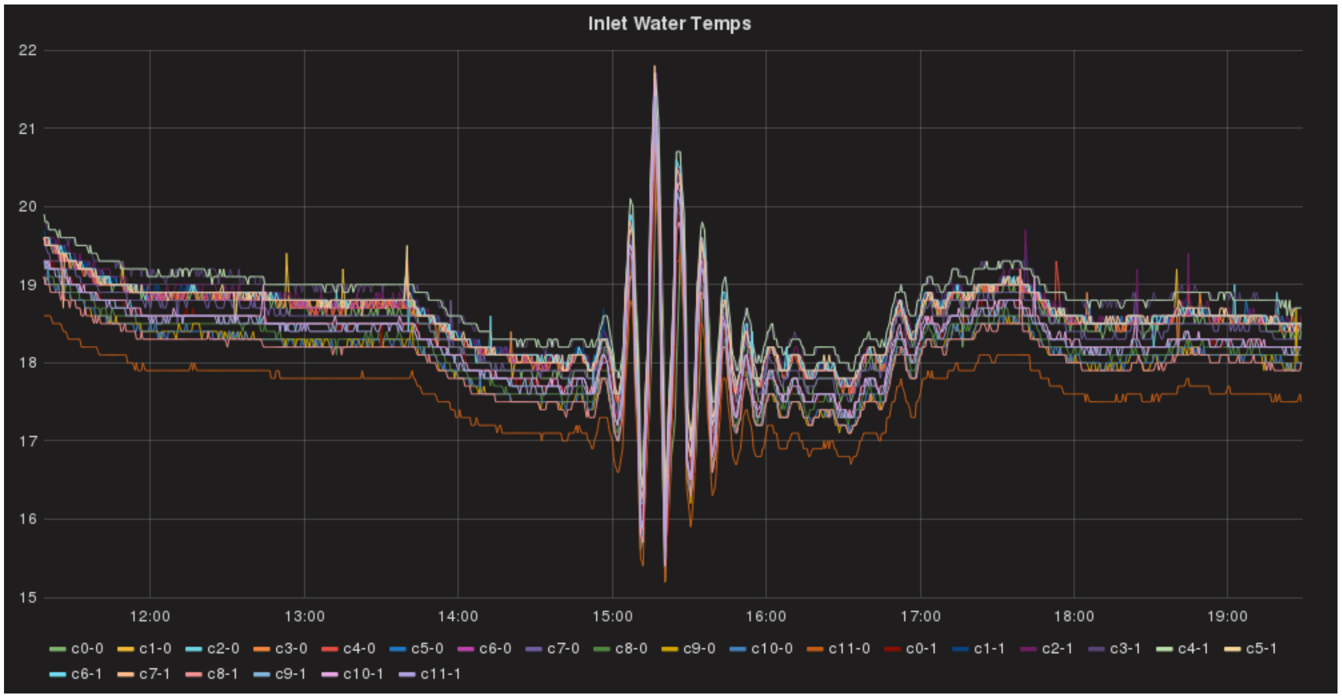

Fig. 3. An example of brief 'hunting' behavior in inlet water temperatures due to problems with upstream water chillers

managing thousands of different custom types could mitigate some of these performance boosts.

Data analysis and visualization is probably the most fruitful area of exploration. With some additional data science, more interesting trends may be discovered in the hardware error and SEDC data. There's also the work of correlating hardware error and SEDC data to user jobs, which could be useful to both administrators and end users.

In this paper we have presented our work on gathering useful data directly via the ERD, using our own codebase. Deluge was started because we wanted to build a database-backed monitoring system that avoided storing and parsing human-readable text. As we have demonstrated, the development and research time put into Deluge has proven to be fruitful, and the resulting data is a valuable resource for the ALCF.

VI. Acknowledgements

This research was supported by and used resources of the Argonne Leadership Computing Facility, which is a U.S. Department of Energy Office of Science User Facility operated under contract DE-AC02-06CH11357.

References

[1] ALCF, *XC40 Machine Overview — Argonne Leadership Computing Facility*. [Online]. Available: https://www.alcf.anl.gov/user-guides/xc40-machine-overview. [Accessed: 03-Apr-2018].

[2] IBM, *IBM System Blue Gene Solution: Blue Gene/Q Application Development*. [Online]. Available: http://www.redbooks.ibm.com/redbooks/pdfs/sg247948.pdf [Accessed: 03-Apr-2018].

[3] *Release History*. [Online]. https://golang.org/doc/devel/release.html [Accessed: 03-Apr-2018].

[4] K. Harms, T. Leggett, B. Allen, S. Coghlan, M. R. Fahey, C. Holohan, G. McPheeters, P. Rich, *Theta: Rapid Installation and Acceptance of an XC40 KNL System,* Concurrency and Computation: Practice and Experience, DOI: 10.1002/cpe.4336, Article accepted on 23 August, 2017. Published 5 Dec 2017 and proceedings of the 2017 Cray User Group Conference, May 2017, Redmond, WA.