Use of the ERD for administrative monitoring of Theta:

Re-implementing xthwerrlog, sedc and related Cray utilities in Go

alexk@anl.gov
ALCF

Argonne NATIONAL LABORATORY

Who we are

The Argonne Leadership Computing Facility (ALCF) is a national scientific user facility that provides supercomputing resources and expertise to the scientific and engineering community to accelerate the pace of discovery and innovation in a broad range of disciplines.

We currently run Theta, a 24-rack XC40 with Knight's Landing CPUs

What are we talking about?

- The ERD (Event router daemon) is the backbone of the XC40
- Most forms of command & control, as well as log data, happen through the ERD
- Console logs, Hardware Error data, environmental data, system state

```
iotasmw1:~ # xtconsumer ec_l1_heartbeat
Included:
    ec_l1_heartbeat (0x0800046f)
----> connection to event router made
Thu Apr  5 16:19:41 2018 - rs_event_t at 0x22211a0
ev_id = 0x0800046f (ec_l1_heartbeat)
ev_src = ::c0-0
ev_gen = ::c0-0c0s0n0
ev_flag = 0x00000002 ev_priority = 0 ev_len = 28 ev_seqnum = 0x00000000
ev_stp = 5ac64c9d.000950d7 [Thu Apr  5 16:19:41 2018]
svcid 0: ::c0-0 = svid_inst=0x0/svid_type=0x0/svid_node=c0-0[rsn_node=0x0/rsn_type=0x6/rsn_state=0x6]
ev_data...
00000000: 01 00 00 00 00 00 00 00 00 00 00 00 0c 06 00 00 *................*
00000010: 00 00 00 00 01 00 00 00 00 00 00 00             *................*
```

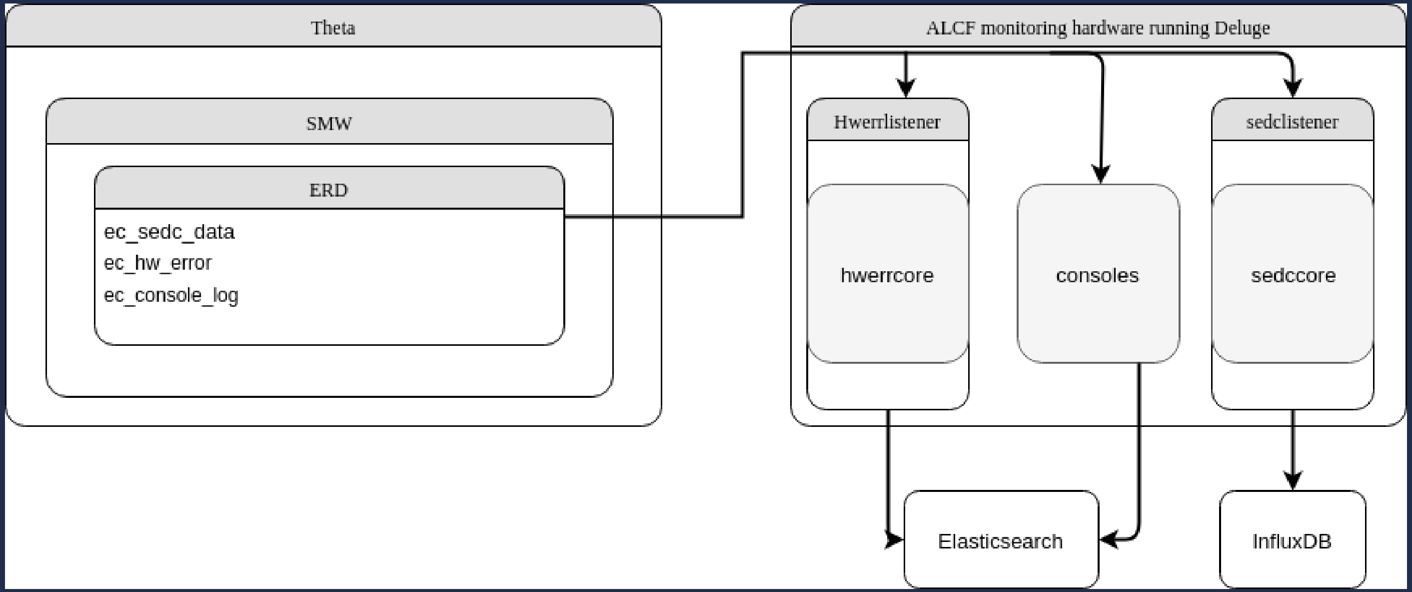
Why?

- Hardware error data is stored in binary logs, making parsing difficult and CPU-time consuming
- A propriety, closed-source software stack
- All other logs are in unstructured text
- I just don't like unstructured data

```
iotasmw1:~ # hexdump /var/opt/cray/log/p0-current/hwerrlog.p0-20180314t145905 | head
0000000 3e1d 5aa9 0000 0000 8957 000b 0000 0000
0000010 416c 4768 0000 0001 0000 0000 ffff ffff
0000020 5ed5 0003 38dc 5aa9 0000 0000 0000 0000
0000030 ffff ffff ffff ffff ffff ffff ffff ffff
*
0000070 bcbb 5aaa 0000 0000 0af6 000d 0000 0000
0000080 5683 587e 6718 0080 0000 0000 0020 0000
0000090 a4e6 000c bcbb 5aaa 0090 0002 0000 00d4
00000a0 1e28 0000 2000 2020 0218 0000 8888 0000
00000b0 0000 0000 0000 0000 0000 0000 0000 0000
```

What we did: Deluge

- Extensible and loosely coupled: Little overhead to support new databases and input streams
Three major backend libraries: events, hwerrcore, sedc
- Highly parallel: makes heavy use of channels and Goroutines
- Scalable: Can use as many cores and as much memory as it's given
- Configurable: CLI options to tweak memory and core usage
- Written in Go: A modern, statically-typed, garbage-collected, memory-safe systems language



Library: events

- The only component that talks to the ERD
- Reads in raw binary data from the network, returns a stream of Go structs to the consumer

```
cn, err := DialErd([]string{"thetasmw1:7004"}, 10000)
    log.Fatalf("Error starting listener interface: %s", err)
}
err = cn.StartListen(0x93d) //ec_hw_error

for{
        testPacket, err := cn.Read()
        if err != nil {
            log.Fatalf("Error reading test packet: %s", perr)
        }
        //do some parsing...
}
```

Library: hwerrcore

-Contains the data structures and logic used to parse cray RAS events
-Has knowledge of all Aries errors and KNL machine-check errors Theta

Deluge data structures mimics that used by Cray:

```
type HwerrEvent struct {
    Magic     uint32 // hwerr magic number: 0x587e5683
    ErrorCode uint16
    Cat       uint16 //Error Category
    Ptag      uint16
    Flag      uint16
    Serial    uint32
    EventTs   uint64
    RsNodeT   uint64    //Cray's rs_node_t struct, consult xtparsename for more info
    InfoMmr   [8]uint64 //Memory-mapped register array
}
```

Library: hwerrcore

-Bulk of the code turns MMR data into JSON with human-readable error data

Input:


```
"info_mmr": ["0x50670100000000", "0x500002b01000000", "0x84000040000800c2", "0x1303101540",
  "0x0", "0x0", "0x0", "0x0"
],
```


Output:


```
"data": {
  "ADDRV": 1,
  "CORR_ERR_COUNT": 1,
  "IA32_MCi_ADDR": 81655764288,
  "MCACOD": 194,
  "MISCV": 0,
  "MSCOD": 8,
  "OVER": 0,
  "PCC": 0,
  "UC": 0,
  "VAL": 1,
  "bank_description": "Integrated Memory Controller 1",
  "bank_name": "IMC1",
  "mcacod_message": "Channel 2 Memory Scrubbing Error",
  "mscod_message": "Correctable patrol scrub error"
},
```

MapDef

- We have a problem: logic that parses memory-mapped registers (MMR) is hard-coded into the Cray Hardware Supervisory System (HSS) libraries
- We need to reliably reproduce it in order to parse hardware RAS events
- Cray's libraries output human-readable strings, we want structured data

Solution: Create a harness that uses Cray's own libraries to re-generate Go code for every hardware error. Why not?

MapDef

Step 1: Call Cray's parser function in a loop, skip over non-existent error codes, or codes with no MMR data.

```
int rc = ghal_decode_error((hss_all_error_event_t *)test_ev,"c1-1c0s15a0",event_str, ev_str_size, pflags);
if(rc < 0 || rc > ev_str_size){
    fprintf(stderr, "error trying to get error code 0x%x: got back %d\n", i, rc);
    free(fname);
    continue;
}
```

Example of the data we get back:

```
| HWERR[c0-0c0s10a0n2][5]:0x5225
  NIC_CE_ERR_OP_INFO = 0x0000000000000001
      +    17..0:  OP_DISABLE_SOURCE = 0x1
      +    35..18: BAD_OP_SOURCE = 0x0
      +    53..36: UNXPCT_SCAT_SOURCE = 0x0

  NIC_NETMON_CE_EVENT_CNTR_BAD_REQS = 0x0000000000000002
```

MapDef

Step 2: Take the string output, use regex and code generation libraries to turn it into Go maps

```
| HWERR[c0-0c0s10a0n2][5]:0x5225
  NIC_CE_ERR_OP_INFO = 0x0000000000000001
      +    17..0:  OP_DISABLE_SOURCE = 0x1
      +    35..18: BAD_OP_SOURCE = 0x0
      +    53..36: UNXPCT_SCAT_SOURCE = 0x0

  NIC_NETMON_CE_EVENT_CNTR_BAD_REQS = 0x0000000000000002
```

```
var Error0x5225 = ErrorInstance{ErrorCodes: []uint16{0x5225}, MapDef: map[string]interface{}{
    "NIC_NETMON_CE_EVENT_CNTR_BAD_REQS":
    MaskDef{MMRLoc: 1, Sbit: 0x0, Ebit: 0x3f, Parser: (func(uint64) uint64)(nil)},
    "NIC_CE_ERR_OP_INFO": map[string]MaskDef{
        "OP_DISABLE_SOURCE":  {MMRLoc: 0, Sbit: 0x0, Ebit: 0x11, Parser: (func(uint64) uint64)(nil)},
        "BAD_OP_SOURCE":      {MMRLoc: 0, Sbit: 0x12, Ebit: 0x23, Parser: (func(uint64) uint64)(nil)},
        "UNXPCT_SCAT_SOURCE": {MMRLoc: 0, Sbit: 0x24, Ebit: 0x35, Parser: (func(uint64) uint64)(nil)}}}}
```

Library: sedccore

- Reads data from the ec_sedc_data channel
- was the least documented and hardest to implement
- We didn't have to implement SEDCv1

```
//CrmsSedc2DataT is the header struct for SEDC packets
type CrmsSedc2DataT struct {
    ErrCode    uint32 //Error code for the event
    BaseTs     uint64 //Base timestamp for scanid offsets
    NumItems32 uint32 //number of 32 bit events
    NumItems64 uint32 //number of 64 bit events
}

//ScanEvent contains a single SEDC data point
type ScanEvent struct {
    Scan      string    //Scanid String
    Units     string    //Unit string (V/degC/etc)
    TsOffset  uint32    //Timestamp offset, in milliseconds
    ScanID    uint32    //scanid integer
    ErrorCode uint32    //0=no error
    Value     ScanValue //Interface type representing an error
    Metadata  uint32    //The raw Item header, see CrmsSedc2ItemHeaderEvtT
}
```

Library: sedccore

-SEDC scan IDs are generated based on a dump of the PMDB

```
COPY pmdb.sedc_scanid_info TO '/tmp/scanid_dump.csv' DELIMITER ',' CSV HEADER;
```

```
1184: {
    ScanIDName:  "CC_T_AVRG_AIR_INLET_TEMP",
    SensorUnits: "degC",
}
```

Usage at ALCF: hardware error data

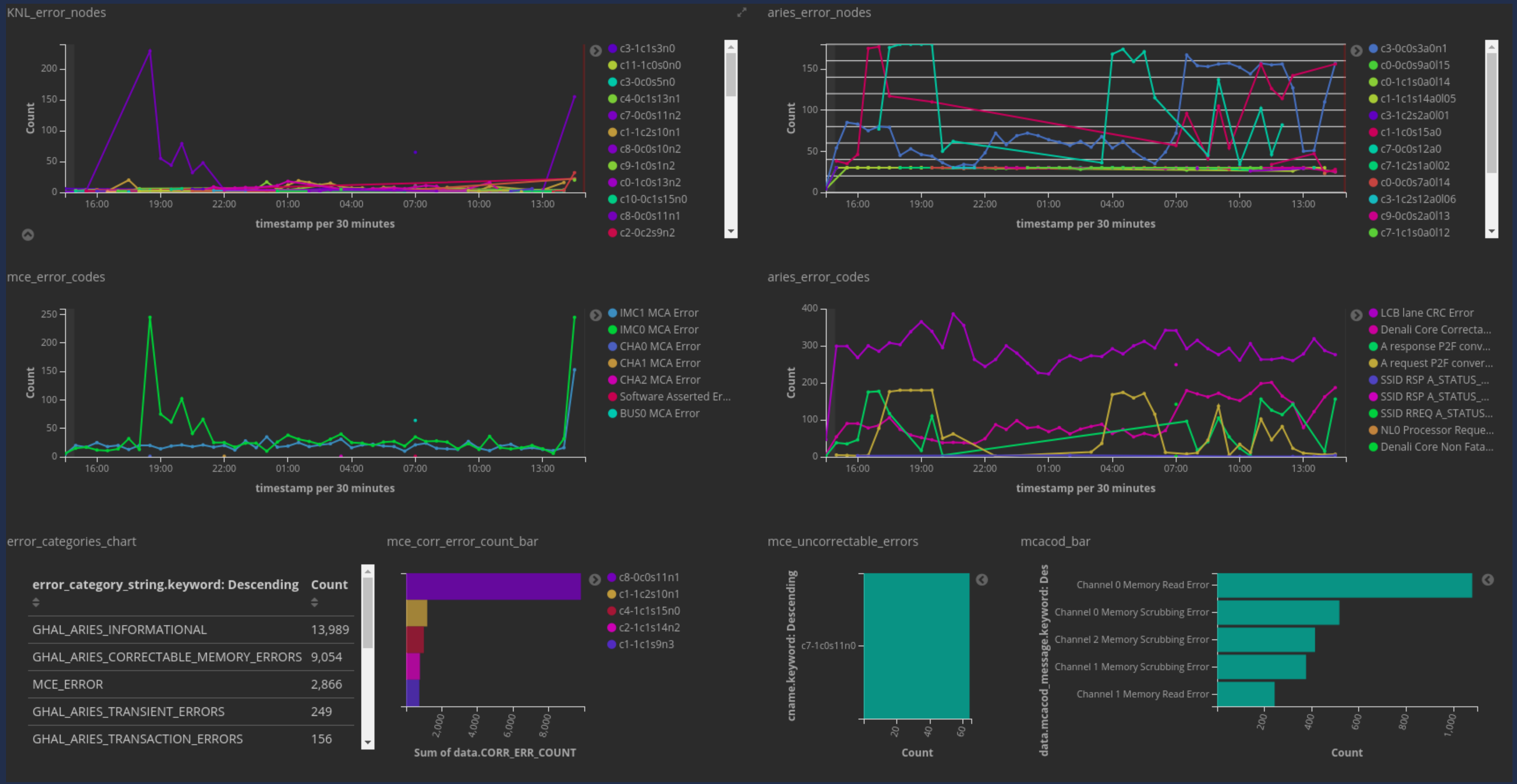
- Elasticsearch is used to store all hardware error data

- ES a good fit for hwerr data

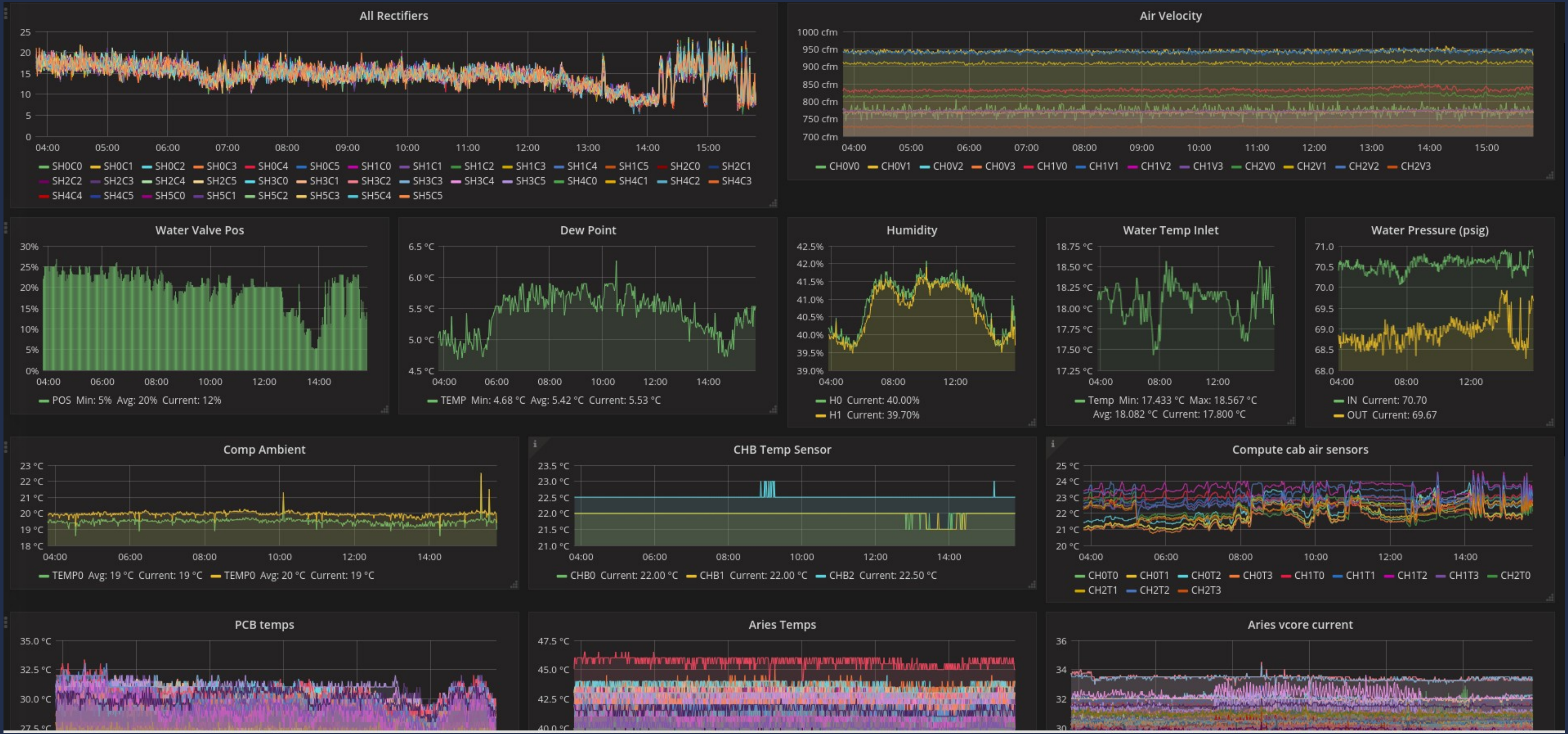
- Nested JSON makes data analysis easy, compared to string parsing


```
{
  "_index": "ras",
  "_type": "error",
  "_id": "AWLAkLIZ-vWk1L-XltT3",
  "_version": 1,
  "_score": null,
  "_source": {
    "magic": 1484674691,
    "error_code": 64784,
    "error_category": 0,
    "error_category_string": "MCE_ERROR",
    "ptag": 0,
    "flag": 0,
    "serial_number": 0,
    "timestamp": "2018-04-13T19:52:11Z",
    "component_id": 172843400064272130,
    "error": "IMC1 MCA Error",
    "cname": "c10-1c2s6n2",
    "cname_type": 0,
    "node_type": "rt_node",
    "data": {
      "ADDRV": 1,
      "CORR_ERR_COUNT": 5,
      "IA32_MCi_ADDR": 202050353216,
      "MCACOD": 145,
      "MISCV": 0,
      "MSCOD": 32,
      "OVER": 1,
      "PCC": 0,
      "UC": 0,
      "VAL": 1,
      "bank_description": "Integrated Memory Controller 1",
      "bank_name": "IMC1",
      "mcacod_message": "Channel 1 Memory Read Error",
      "mscod_message": "Correctable error",
      "pdc": {
        "PDC_CORE": 48,
        "PDC_REV": 0,
        "PDC_SOCKET": 0,
        "PDC_THREAD": 0,
        "PDC_TYPE": 5
      }
    },
    "mmr": [
      "0x5067010000000",
      "0x5000030000000000",
      "0xc400014000200091",
      "0x2f0b23b840",
      "0x0",
      "0x0",
      "0x0",
      "0x0"
    ]
  },
  "fields": {
    "timestamp": [
      "2018-04-13T19:52:11.000Z"
    ]
  },
  "sort": [
    1523649131000
  ]
}
```

Usage at ALCF: hardware error data



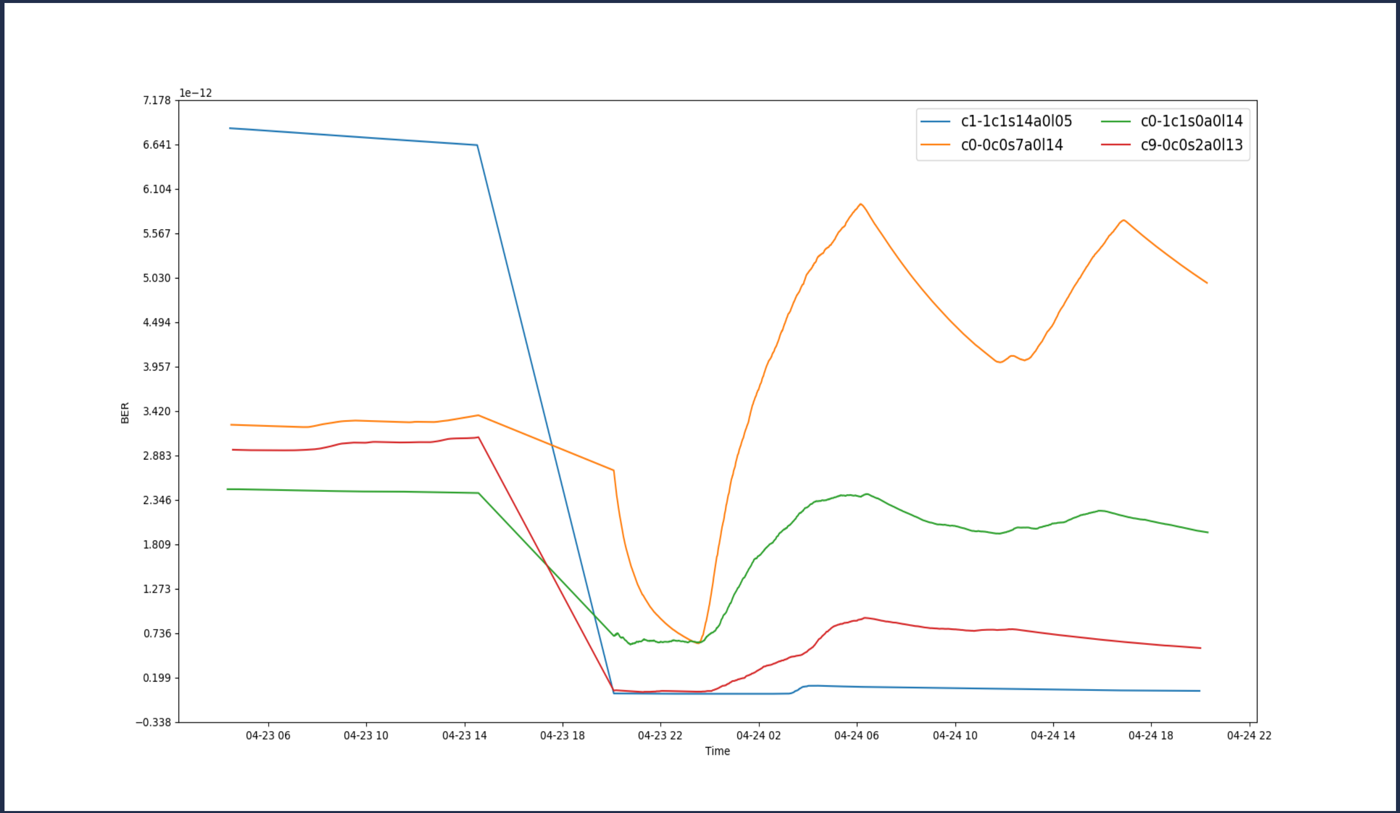
Usage at ALCF: SEDC data



Usage at ALCF: BER data

- Generated from hardware error data

- Used to track health of Aries links and in some cases predict failure



Argonne NATIONAL LABORATORY

Future work

- Data science and machine learning to find trends

- More correlation with job data

- Ingest more data from the ERD:
  – History of Admin commands
  – ERFS metadata
  – Some ALPS data
  – Track system state

Tack! (Thank you!)

- Questions?