Use of the ERD for administrative monitoring of Theta: Re-implementing xthwerrlog, sedc and related Cray utilities in Go

alexk@anl.gov
ALCF
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

```bash
iotasml:~ # xtconsumer ec_l1_heartbeat
Included:
    ec_l1_heartbeat (0x0000046f)
----> connection to event router made
Thu Apr 5 16:19:41 2018 - rs_event_t at 0x12211a0
  ev_id = 0x6000046f (ec_l1_heartbeat)
  ev_src = ::C-0
  ev_gen = ::C-0-0
  ev_flag = 0x00000002 ev_priority = 0 ev_len = 28 ev_seqnum = 0x00000000
  ev_data = 5ac64c9d.00059d7 [Thu Apr 5 16:19:41 2018]
  svcid 0: ::C-0 = svcid_inst=0x0/svld_type=0x0/svld_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 00 00 0c 06 00 00 ................*
  00000010: 00 00 00 00 01 00 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
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

```go
cn, err := DialErd([]string{"thetasnw1:7004"}, 10000)
    log.Fatal("Error starting listener interface: %s", err)
}
err = cn.StartListen(0x93d) //ec_hw_error

for{
    testPacket, err := cn.Read()
    if err != nil {
        log.Fatal("Error reading test packet: %s", err)
    }
    //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:

```go
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", "0x84000040000000c2", "0x1303101540", "0x0", "0x0", "0x0", "0x0"
],
```

Output:

```
"data": {
  "ADDRV": 1,
  "CORR_ERR_COUNT": 1,
  "IA32_MC1_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.

```c
int rc = qhal.decode_error((hss_all_error_event_t *)test_ev,"c1:lc0s15a0",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(rname);
    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
```

```go
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 SED Cv1

```go
//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_I_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
Usage at ALCF: hardware error data

**KnL_error_nodes**

- Data points for KNL node errors over time, showing counts for different error types.

**Antes_error_nodes**

- Data points for Antes node errors over time, showing counts for different error types.

**Mce_error_codes**

- Data points for MCE error codes over time, showing counts for different error types.

**Antes_error_codes**

- Data points for Antes error codes over time, showing counts for different error types.

**error_categories_chart**

- A chart showing error categories and their counts, with categories like GHAL_ARIES_INFORMATIONAL, GHAL_ARIES_CORRECTABLE_MEMORY_ERRORS, MCE_ERROR, GHAL_ARIES_TRANSIENT_ERRORS, and GHAL_ARIES_TRANSACTION_ERRORS.

**mce_corr_error_count_bar**

- A bar chart showing the count of corrected errors, with categories for different error codes.

**mce_uncorrectable_errors**

- A bar chart showing the count of uncorrectable errors, with categories for different error codes.

**mcaccd_bar**

- A bar chart showing the count of MCACD errors, with categories for different error codes.
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
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?