ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 315b

Systems Group, Department of Computer Science, ETH Zurich

Computer platform visualization for Enzian

by

Patrick Wicki

Supervised by

Michael Giardino, David Cock, Timothy Roscoe

September 2020–March 2021

D INFK

Abstract

The Systems Group at ETH Zurich has built Enzian, an experimental server system that combines a main ARM SoC and a Xilinx FPGA on single board. Enzian's research nature means it generates a plethora of low level information that could be visualized for monitoring purposes. This thesis focuses on extraction and visualization of monitoring data, and provides a concrete implementation. In particular, we present a tool for exporting data on the Enzian BMC as well as a monitoring web application that fetches and visualizes said data.

Contents

1 Introduction

Modern computer systems have a large and complex runtime system that sits below the level of the operating system and is concerned with managing power, controlling fan speeds, allowing remote access and much more. Typically, these tasks are handled by a separate processor called Board Management Controller (BMC), which is not visible to the operating system.

Project Enzian, run by the Systems Group at ETH Zurich, has built an experimental server system for research that combines a Cavium ThunderX-1 ARM SoC and a Xilinx FPGA on a single board. Typically, access to the BMC is restricted, but Enzian's BMC runs Linaro Linux and an open-source BMC distribution called OpenBMC. This enables us to use the BMC to acquire and expose real-time information about the system for monitoring.

This project is split into multiple parts. In chapter 2 we'll look at existing work on the topics of monitoring and visualizations for various types of systems and requirements. Afterwards, in chapter 3 we present a set of possible visualizable attributes for single systems and discuss what could be visualized on Enzian specifically. Chapter 4 will then build up on that, as we'll look at how to extract as well as export information on the BMC. We'll talk about the OpenBMC Linux distribution that runs on Enzian's BMC, its infrastructure, integrated web server and the web interface. In OpenBMC, monitoring data is exposed by a service that reads sensor values from a Linux kernel interface. This does not work on the Enzian BMC currently, so we talk about the implications of this and present our alternative approach: A python tool which acquires low level information via the power manager running on the BMC. It structures the data using the JavaScript Object Notation (JSON) format. The JSON is then served by the BMC's integrated web server, making it accessible to external clients via HTTP. In chapter 5 we then present our monitoring application, which uses this JSON data. In particular, we will present a web application, built using HTML and JavaScript. It builds up a local history of samples acquired from the BMC and visualizes the data in time series charts. We'll explain the application logic and data structures, compare different approaches for data decimation and showcase parts of the front end along with the visualizations. Furthermore, we provide instructions for deploying the application. Lastly, we present some ideas for potential future work that could follow this thesis in chapter 6.

2 Background

2.1 Monitoring

This section focuses on topics and related work related to monitoring. We also mention some relevant technologies and terminology.

2.1.1 IPMI and Redfish

Since monitoring is often considered part of managing a system, it is worth talking about the Intelligent Platform Management Interface (IPMI) and its successor, Redfish. IPMI (as well as Redfish) is a specification for an interface that can be used to perform operations on a computer independent of the operating system running on it. It is typically implemented on separate subsystem, that consists of the BMC as well other controllers on the board that connect to the BMC. This enables the BMC to manage and monitor CPU, OS and firmware of the host. Administrators can use this interface to interact with the system in ways that would otherwise typically require physical presence. They can power the system on or off, or reboot it if the host OS crashed, and it also allows them to remotely update the firmware as well as update or even replace the host OS. Apart from all of these management functions, the interface can be used to monitor aspects of the system like power supply, temperatures and fan speeds without having to impact or rely on the host OS.

The successor to IPMI is Redfish [1]. The Distributed Management Task Force (DMTF) released Redfish in 2013 as a standard API for managing computing devices in next generation data centers. Its goal is to counter the increasing amount of issues and inefficiencies arising in older management systems. These include the presence of security risks, protocols not being human-readable, scaling issues and limited models that cannot describe modern systems. It was designed in particular to supersede IPMI due to its security flaws and scalability issues.

Redfish works by using a RESTful API and communication over HTTP, secured through TLS. The specification includes a set of JSON Schema files that are returned by requests to the Redfish service. One can access the so-called Service Root by sending a GET request to `https://<ip:port>/redfish/v1`. It acts as home page for Redfish and returns an overview map of the services. The services and components can be accessed in a tree-like fashion by traversing the different paths. See figure 2.1 for an overview. One such service is the Chassis Service, located at /redfish/v1/Chassis. It provides a physical view of

a system, and traversing its sub-paths eventually leads to individual components like temperature sensors, power supply and redundancy of components.

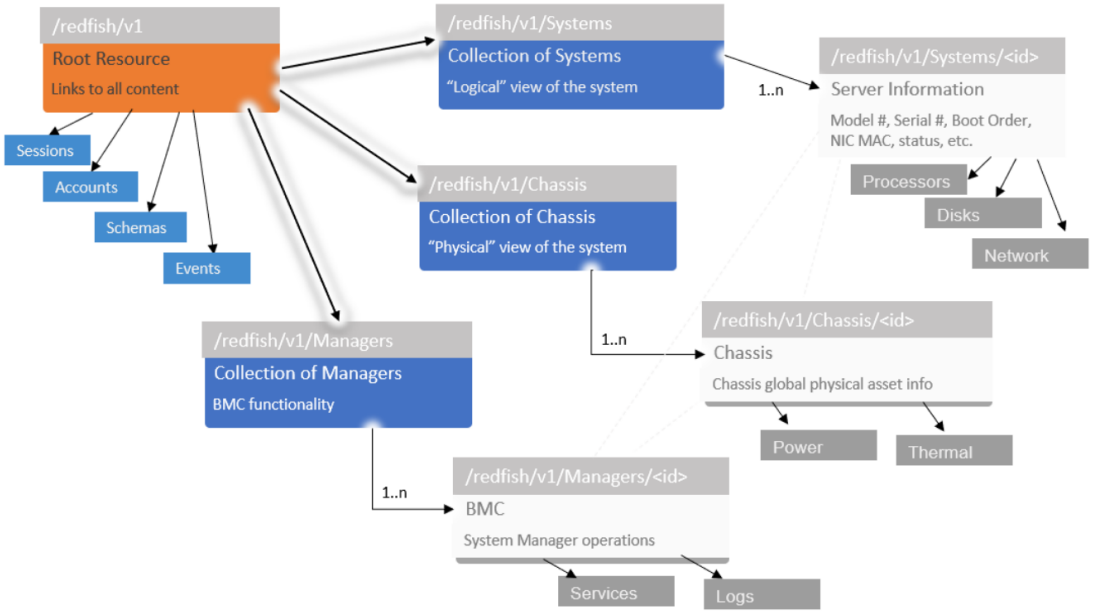

Figure 2.1: Overview of the Redfish data model. Source: [2]

OpenBMC implements Redfish and thus the API is also exposed on the Enzian BMC. We'll go into more detail about this in chapter 4.

2.1.2 In-band and Out-of-Band

Two terms worth mentioning are *In-band* (also called Inline) and *Out-of-Band*, as any type of monitoring can usually be categorized as one of them. One way to decide which type applies is to check where the monitoring tool is placed. If it sits in the critical path of the data, or in other words, if it interferes with the existing pipeline, it can be considered in-band. If the tool however only takes on an observer role and doesn't affect the data path, it is considered out-of-band.

An out-of-band approach was chosen by Libri et al. [3], when they designed their own monitoring infrastructure, called Dwarf in a Giant (DiG). Part of their work was the design of a custom power sensor at the plug, that does high resolution monitoring of the power consumption. This can be characterized as out-of-band, as the power sensor is purely an observer and doesn't directly affect the flow of power. A reason why one might choose such an out-of-band approach is that it is more technology agnostic. Such a power sensor doesn't require any specific features on the board and could therefore be deployed in different systems without any prior adjustment to them.

David et al. [4] present RAPL, a power measurement and limiting architecture for main memory and CPUs. They mention two approaches for measuring memory power, one in-band and one out-of-band. In the first one, they employed a memory power riser with a data acquisition system to directly measure the DIMM power. This method employs a separate physical device that acts as an observer. We therefore count it as an out-of-band solution. The other approach they call Memory Power Model Approach. This consists of implementing activity counters in the memory controller, which, together with a power model and predefined weights, lets them measure memory power at a high resolution and very fine granularity, and with minimal impact on power and area. This latter approach we categorize as in-band, as it affects and sits in the memory controller directly. In-band monitoring is typically more integrated and can allow more detailed and fine-grained access.

Monitoring in this project relies heavily on the telemetry functionality provided by the power manager software that runs on the BMC. Since the power manager is clearly on the critical path and implementing and outputting telemetry affects it as well, our type of monitoring is in-band. This means we need to make sure that monitoring activity does not negatively impact the operation of the BMC and its management functions, as the health of the entire system depends on it.

2.1.3 Distributed systems

While this project only focuses on monitoring a single Enzian system, it's still worth discussing how larger sets of computers can be monitored together, as Enzian might be deployed in clusters in the future.

A distributed system is a set of different networked computers that each make up components of a larger system. Such systems and monitoring thereof come in many scales, and they also differ quite drastically in types of networks. Massie et al. [5] name three main categories, all of which their distributed monitoring system, Ganglia, can run on:

1. Clusters

   Clusters communicate over high bandwidth, low latency interconnects. They are frequently homogeneous, both in terms of hardware and their operating systems. They're also managed by a single entity.

2. Grids

   Grids are heterogeneous, federated systems, but they are still interconnected using high speed, wide-area networks. Unlike clusters, they're often managed by multiple entities.

3. Planetary-scale systems

   Planetary-scale systems are distributed systems that extend over a significant fraction of the planet. They communicate using an overlay network using the existing,

regular internet, which means that, in contrast to clusters or grids, bandwidth is expensive and the network is not as reliable.

The most likely of these use cases for Enzian is a cluster of multiple Enzian boards (therefore being a homogeneous, distributed system) connected by a reliable, low-latency local network. The first category therefore fits the best. In theory though, it could be employed in any setting.

Ganglia was built with a set of design goals in mind, most of which any distributed monitoring system should aim to fulfill:

- Scalability: The system should be able to scale up to many nodes
- Robustness: Distributed nature implies multiple, independent failures, also of the network, therefore a robust, high availability implementation is required
- Extensibility: The scale and heterogeneity mean that the types of data monitored can change in the future, so it should be extendable
- Manageability: Adding additional nodes shouldn't incur much more management overhead. Manual configuration should also be kept to a minimum.
- Portability: The system should be inclusive to a wide variety of architectures and operating systems
- Overhead: The per-node computational and network overhead required to run the system should be kept low

While extensibility, portability and robustness are especially important in systems that are heterogeneous and/or consist of unreliable networks, e.g. Grids or Planetary-scale systems, they might not be as crucial for a cluster that runs a set of similar computers in a highly reliable local network. Scalability, which also correlates with the per node overhead, is an important attribute for any system that tries to monitor a larger set of computers, independent of other factors.

2.1.4 Scalability and High Resolution monitoring

The issue of scalability can be tackled by employing a hierarchical design. Instead of having one central entity responsible for handling the data of each individual node, there are multiple monitoring agents that each are only responsible for a subset of the nodes in the system. Agents can then preprocess, compress or down sample the data they collected, before feeding it to monitoring agents higher up in the hierarchy. This way, agents higher up in the hierarchy can cover a large set of computers, without having to directly interact with each single one of them.

DiG [3] employs a form of hierarchical approach to combat the increasingly high resolution at which monitoring is done nowadays. They list three bottlenecks that occur in modern state-of-the-art systems:

1. Network bandwidth overhead

2. Storage capacity overhead (needed for post-processing)
3. Software processing overhead (both real-time and offline)

All three of those become especially bad if there's just a single, central entity responsible for the monitoring and analysis of the data of all the nodes. Some high resolution monitoring systems produce up to 500kS/s (kilo Samples per second) per node. This is not scalable if a central node has to process all the samples. A super computer with such high resolution monitoring would generate billions of samples each second.

Their proposed solution to this is to bring intelligence to the edge. Every node has its own distributed smart monitoring agent, an embedded computer separate from the node and BMC, that can perform real-time analysis and feature extraction on the high resolution output, and then forward this processed data to a central node at a much lower rate.

They decided to use an embedded computer separate from the node and BMC for various reasons. Among others, because they needed something that can interface with their custom power sensor and because BMCs are usually closed platforms. Even if that latter wasn't the case (e.g. with OpenBMC), they wouldn't want to overwhelm it with edge analytics since the BMC is important to ensure safe working conditions for the node.

This approach represents a hierarchy, albeit with only two levels. Every agent is responsible for a single node, while the central node has a high level overview of the cluster. It scales well, because every added node also comes with its own extra embedded computer, and the additional impact on the central node is small thanks to the preprocessing of every node's data.

2.2 Visualization

While the previous section focused on different types of systems and ways to extract, aggregate and process information, in this section we'd like to address some previous work on how to visualize information on the user end.

Ganglia [5] stores and visualizes time series data using RRDtool (Round Robin Database). They do this per grid, cluster, host and for each metric, and over different time spans. RRDtool was chosen due to its compact databases that are designed for this purpose. A PHP web application presents the graphs to users. Figure 2.2 shows a screenshot of their application, displaying graphs that show the trends of different metrics over the last hour for a few grids.

LLview [6] is another monitoring system for supercomputer grids with two main goals in mind: 1) it should enable administrators to gain a global overview of system utilization and 2) it should allow users to view, control and plan submission of their jobs. To achieve this, they have a set of visualizations. To show an overview of utilization, they use what they call a node display (see figure 2.3).

Figure 2.2: Ganglia [5] web front-end

It shows the power usage of each rack, and the states of the different nodes. Idle nodes are colored white, while other nodes have the color of the corresponding job they're running.

A quick glance at the time series graph in the center of the overview screen (figure 2.4) shows the total system utilization and power usage over the past three days. The overview screen of LLview combines all the different visualizations.

Another visualization, relevant especially to users submitting and monitoring their jobs, is the scheduler prediction view. Along the y-axis the are all the different nodes or mid-planes. Colored rectangles spanning over the x-axis represent currently running jobs and their time window, while the ones in shades of blue represent the predicted time windows of future jobs. This view is located at the bottom right of the overview in figure 2.4.

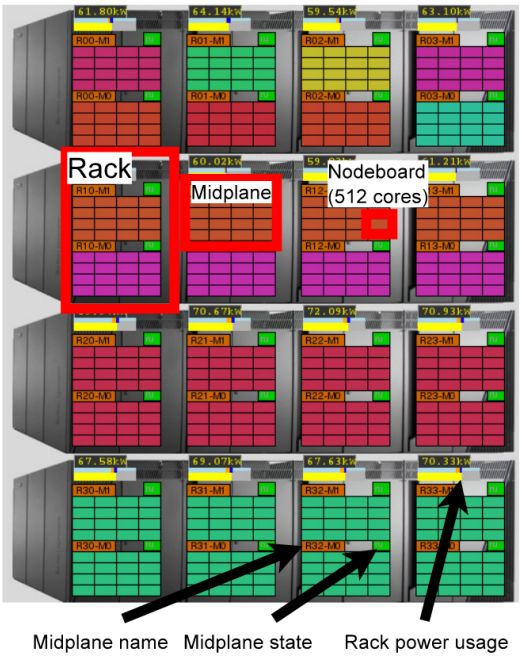

Figure 2.3: Node Display of LLview [6]

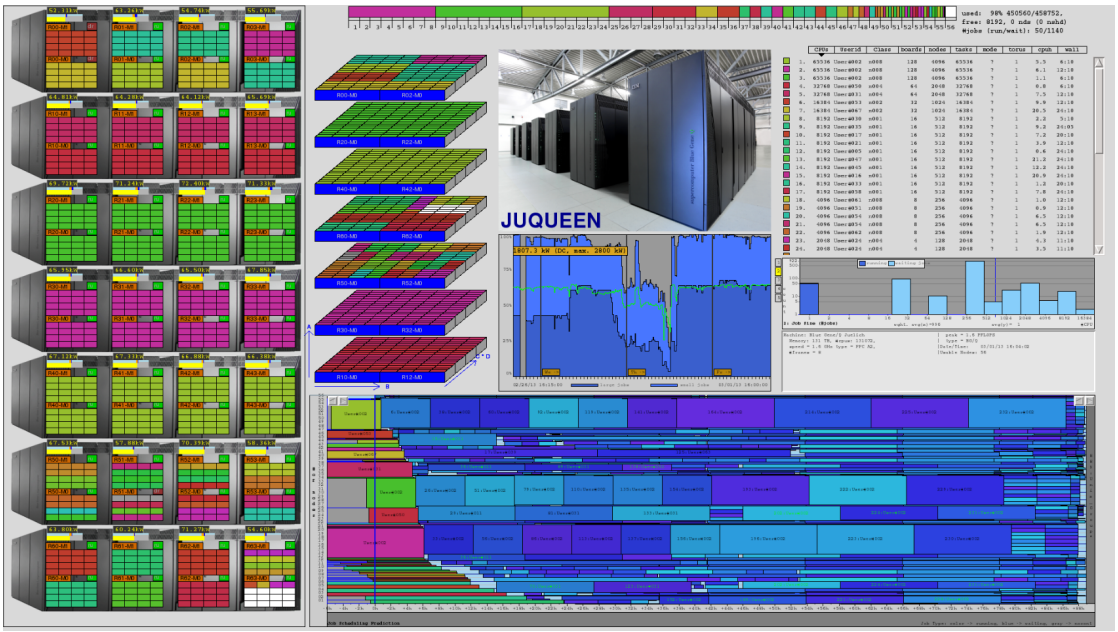

Figure 2.4: Complete screenshot of LLview [6]

2.2.1 Multiple Dimensions

Modern data centers are increasingly complex. To be able to gain not just an overview, but also see correlations between different trends and pin down issues in the temporal/spatial domain, visualizing multiple dimensions can be helpful. Karbach & Valder [7] propose such multidimensional visual approach. Their visualizations are expanded to the following three dimensions:

1. HPC Spatial layout
2. Temporal domain
3. System health metrics like temperature, CPU load, fan speeds, etc.

They leverage the Nagios Core [8] engine to iteratively retrieve the health status of every host using a Redfish. The novel idea is that they don't just show a spatial overview of the entire system, but also includes data in the temporal domain. Figure 2.5 shows this overview. For each rack, hosts are listed top down. Each host contains a time series for power consumption, indicated by colored cells that represent each a 3-minute time window. Clicking on a host opens a popup exposing more detailed information on that host. In the figure 2.5, the popup for host 1-18 is open. The top part of the popup contains a simple time series of CPU temperature of each CPU. A spider chart in the lower panel visualizes a set of health dimensions, including CPU temperature, number of jobs, memory usage, fan speed and power consumption. The spider charts [9] allow detection of patterns in multiple dimensions.

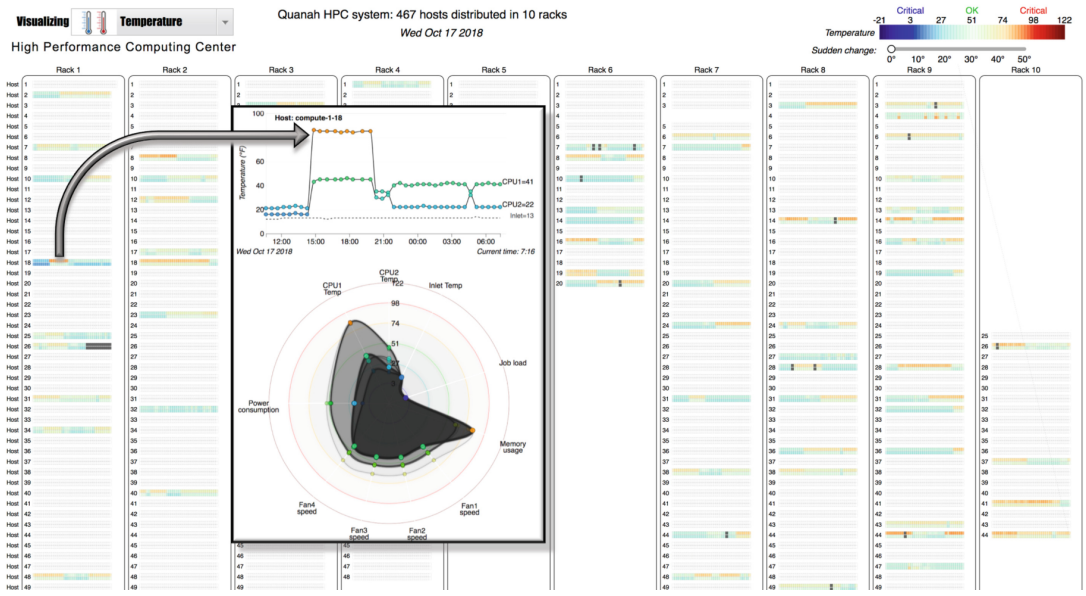

Figure 2.5: Overview screen of [7] visualizing CPU temperature, and demonstrating the pop up for a specific host

The spider charts are used in two main ways. When shown in the popup as before, they show the metrics mentioned, expanded into the temporal domain. In particular, each closed curve represents the observation of all dimensions at one point in time. Figure 2.6 shows a spider chart for one host, showing observations of an eight-hour timeframe.

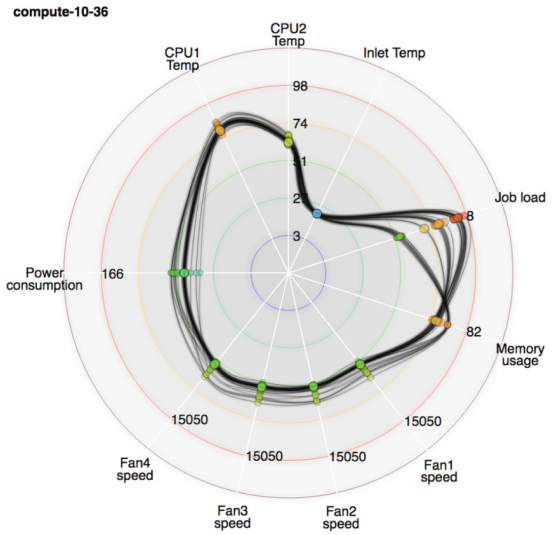

Figure 2.6: A d3.js radar chart [9], used in [7] for visualizing trends of different metrics over time

The second option of using the spider charts is allowing a user to select a set of hosts as well as a timestamp. Now, the different curves no longer represent observations at different times, but instead observations of different hosts at that specific chosen time. An example of this is shown in figure 2.7, where the metrics of many hosts are visible.

In their future work section they discuss an idea that consists doing 2D projections of metrics in order to visualize pairwise correlations. Projecting hosts in a cluster to a 2D plot where one axis represents CPU temperature while the other represents fan speed would enable a user to spot outliers. If a single host has a high temperature but a low fan speed, this would stick out among all other points in the plot and therefore be a good indicator of an anomaly that might require additional inspection.

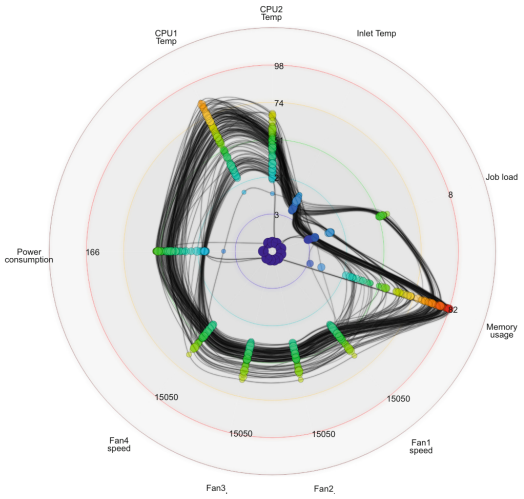

Figure 2.7: A d3.js radar chart [9], used in [7], visualizes different metrics for multiple hosts

3 Visualizable information

3.1 On any system

The following aims to be an exhaustive list of information that could potentially be visualized in modern a PC or server. The focus is on single systems only. It could serve as an inspiration for future work.

- Power usage
  - Total power drawn by the system
  - Power usage per component, e.g. CPU, chipset, PCI devices, network interfaces
- Temperatures
  - Ambient/Inlet temperature
  - Individual component temperature, e.g. CPU/GPU cores, VRAM/DRAM, chipset, storage devices
- System load
  - CPU/GPU load
  - Core frequencies
  - CPU utilization on the BMC
- Memory Usage
  - Total usage of main memory and video memory
  - Usage by process / thread
  - Frequency and bandwidth
  - Cache usage at different cache levels, e.g. hit- or miss-rate
- Storage
  - Topology of storage, e.g. RAID overview
  - Drive health, alerts and failures, e.g. S.M.A.R.T. reporting
  - Usage and capacity of drives and storage arrays
  - Disk Read/Write/IO traffic, both system-wide and per-process
  - File system level information, e.g. overview of sub volumes, compression levels, or ARC size if ZFS is used
  - Usage analysis for files and directories, e.g. similar to Gnome Disk Usage Analyzer (figure 3.1) or WinDirStat
  - Size and time of previous as well as scheduled backups or snapshots

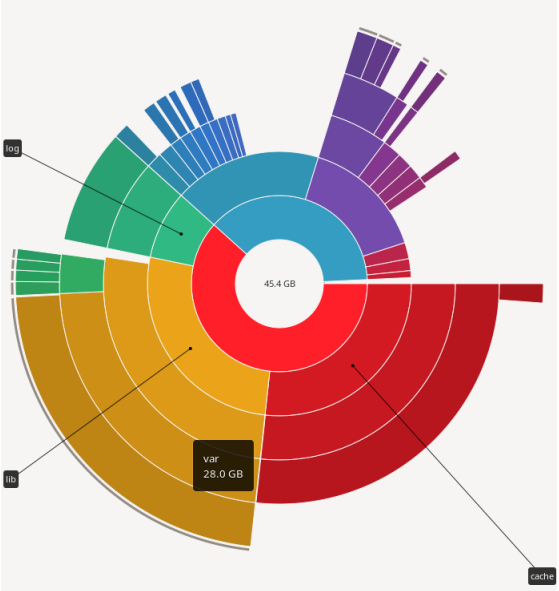

Figure 3.1: A screenshot of the GNOME disk usage analyzer

- Network
  - Overview of network topology, e.g. gateways and routing information
  - Network interfaces and their IP addresses
  - Connectivity information like external IP addresses, LAN and WAN bandwidth
  - In- and outgoing traffic bandwidth, both system-wide and per-process
  - Number of requests and open connections, e.g. Nginx ingress metrics
  - Firewall settings, live monitoring and alerts
- Software
  - List of users currently logged in
  - Running processes
  - Lists of past, running and queued jobs as well as resource utilization by job, if a batch system is employed
  - System uptime and planned restarts
  - System logs like service failures or failed login attempts
  - Versions of relevant software components like firmware, kernel, applications and tool chains
  - Information about past, available or scheduled system and firmware updates

3.2 On Enzian

A lot of the information provided in the previous section's list cannot be acquired directly by the BMC. For example, the BMC cannot directly access the DRAM and information about its caches, it doesn't see the list of running processes and corresponding information like process ownership and per-process resource utilization, it cannot access the storage array of the system, and it doesn't see network traffic apart from the one going to and from the BMC itself.

Monitoring network traffic of the entire system could be done by either aggregating the output of network monitors of each networked device, or by monitoring at the switch or gateway directly and fetching information from it. Acquiring information on processes, DRAM and storage requires some form of communication between the BMC and the primary operating system running on the board. It would require significant work defining interfaces and data structures. We thus decided to limit the scope of monitoring to data that is available on the BMC already. And an additional point to note is, that it is not always clear if exposing a type of information via the BMC is the optimal path. It is useful to have monitoring capabilities built into the BMC, as it doesn't depend on the main operating system and can be used even while the system is powered off. However, it might be more effective to monitor certain components in the main system directly or at least preprocess data first, as transferring and processing it all in the BMC might incur too much overhead and impede its important management functions.

On Enzian the main source of low level information that is currently available to the BMC is the power manager, which runs on the BMC itself. The power manager controls all the voltage regulators and can therefore infer information about the state of many components. This means we can gather information about voltages, currents and thresholds of the CPU cores, FPGA and their DDR memory. Since the power manager is also responsible for fan control, we are also able to read the fan speeds. We'll go into a bit more detail about how we use the power manager in section 4.2.

4 BMC Side

This chapter delves into the BMC side of things. We talk about OpenBMC, the BMC on Enzian, the power manager and how data is extracted and made available to external clients.

4.1 OpenBMC

The Enzian BMC runs the OpenBMC [10] Linux distribution. It is a collaborative project and aims to implement an open-source BMC firmware stack that works across heterogeneous systems. Originally started by IBM, it was adapted by the Linux Foundation in 2018 and is now supported by several big players in the industry [11].

4.1.1 Web server and -interfaces

OpenBMC has an integrated web server called bmcweb, which implements the Redfish API. Authenticated clients can use Redfish to interact with the BMC. Apart from functionality like managing users and firmware updates, it also enables access to low-level hardware information exposed on D-Bus. In particular, OpenBMC already comes with such a client, a web interface that uses Redfish to allow remote management. There are two versions of the web interface. The older one is called phosphor-webui, and is currently in the process of being replaced, as it uses AngularJS, which reaches its End of Life on June 30, 2021. The replacement is called webui-vue, and is, as the title implies, built on Vue.js. The web interfaces, while designed primarily for management, offer a basic monitoring view. There are two sections in user interface, one that contains a list of hardware components and can show corresponding health warnings. The other page lists available sensors and corresponding thresholds and values.

Figures 4.1 and 4.2 show a screenshot of the old and the new web interface respectively, while having the sensors section opened. In both cases, the list is not populated, as no sensors can be found via Redfish. Section 4.2 will explain why.

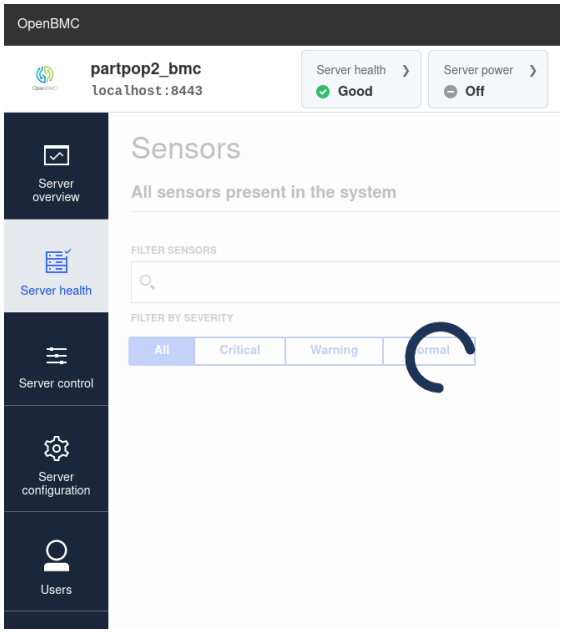

Figure 4.1: phoshor-webui hosted on one of the Enzian boards' BMC

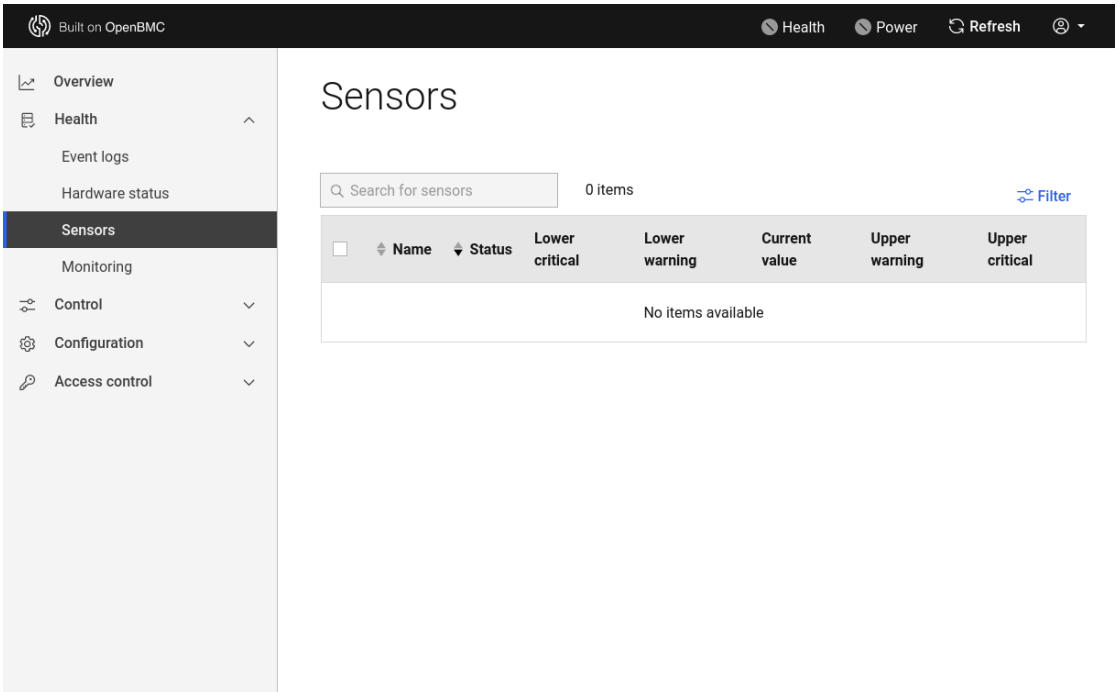

Figure 4.2: The new webui-vue running on a local development environment

4.1.2 Building webui-vue locally

At the time of writing, the BMC versions on the Enzian boards all still ran the phosphor-webui. The Vue.js interface was therefore built and hosted locally using npm. For reference, here are the steps required to reproduce this:

1. Acquire the source tree for webui-vue. We cloned it from OpenBMC's GitHub repository: github.com/openbmc/webui-vue
2. Run `npm serve` in the root of the source tree to build and host the web interface locally on port 8000.
3. API requests need to go through to an actual OpenBMC instance running the bmcweb web server. If the BMC is located in a different network, an SSH tunnel can be used to forward for example port 8443 on localhost to the BMC's port 443, which bmcweb listens on. webui-vue will then automatically proxy API requests to bmcweb.

Tunneling to the BMC, assuming it sits in a network behind a bastion host, can be achieved with the command

```
ssh -L 8443:<address_of_BMC>:443 <domain_or_address_of_bastion>
```

If webui-vue doesn't automatically discover the BMC, or if the BMC can be directly accessed in the network, webui-vue can be directed to the BMC's address (`localhost:8443` or `<address_of_bmc>:443` respectively) by creating the file `.env` in the root of the source tree, containing the following line:

```
BASE_URL=https://<address>:<port>
```

Specifying the protocol as HTTPS is important, as bmcweb will not respond otherwise. Also note that the web server uses self-signed certificates by default, so it might be necessary to instruct the browser to make an exception for the page, before it allows a connection.

4.1.3 Monitoring architecture

OpenBMC is compliant with the Linux HWMon sensor format. This means that any sensor that has a corresponding driver for the Linux kernel running on the BMC can be automatically detected and mapped to D-Bus objects. To do that, OpenBMC contains a service called phosphor-hwmon. Linux exposes sensor data via sysfs in the directory `/sys/class/hwmon`, where phosphor-hwmon detects and reads them, and exposes them on D-Bus under the path `/xyz/openbmc_project/sensors/<type>/<label>`. These objects are then in turn made available to clients via the Redfish interface.

4.2 BMC on the Enzian

Moving on to Enzian, the situation is slightly different. The kernel running on the BMC doesn't contain the necessary drivers for the different sensors, so no data is exposed via the HWMon interface. Instead, a user space power-manager written in Python runs on the BMC. It manages I2C, SMBus and PMBus devices. It also exposes Logging-, Power- and GpioObjects on D-Bus, that allow polling GPIO pins, controlling devices over D-Bus and enable periodic logging of monitor values. However, no sensor values are exposed on D-Bus in the format that phosphor-hwmon would expose them. In turn this means that no monitoring information is available via Redfish, which is why the lists of sensors in figures 4.1 and 4.2 are empty. We mentioned in section 2.1.1 that the Redfish interface can be queried using simple GET requests. We can do this by sending a request using *curl*:

```
curl --user root:<password> --insecure \
  https://<bmc_adress:port>/redfish/v1/Chassis
```

The insecure flag is required due to the BMC's self-signed certificates. Redfish returns the following object:

```
{
"@odata.context": "/redfish/v1/$metadata#ChassisCollection...,
"@odata.id": "/redfish/v1/Chassis",
"@odata.type": "#ChassisCollection.ChassisCollection",
"Members": [],
"Members@odata.count": 0,
"Name": "Chassis Collection"
}
```

The members array is empty, which means the interface currently does not know about any hardware components that it could expose for monitoring. Typically, the structure of that chassis object would be similar to 4.3.

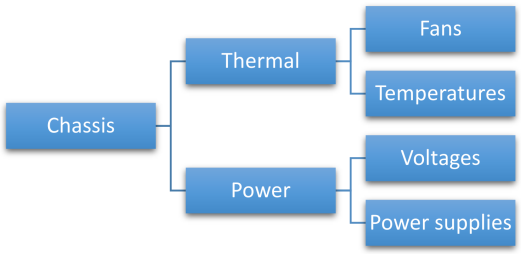

Figure 4.3: Chassis member objects exposed by Redfish. Source: [2]

This turned out to be a hurdle for this project, as the first idea for bringing visualizations to the user front was to build on top of the existing webui-vue. We initially spent some effort building and extending the Vue.js web app with an additional section for monitoring. This way, users could have visualizations for monitoring along with the other management functionality in on single web application.

Apart from including (and possibly writing) the needed Linux drivers, a long term solution would be either writing a service that acts in a way similar to phosphor-hwmon, but instead of taking the route through the kernel, it would use the power manager's telemetry functionality to acquire sensor values to expose on D-Bus. Alternatively, the power manager could be extended to expose such values itself.

These options however are out of scope for this thesis. We therefore took on a different direction. We use the telemetry functionality of the power manager, but instead of exposing data on D-Bus, we collect it and write a JSON file. Additionally, instead of trying to extend webui-vue, we decided to build a separate independent web application that builds on data acquisition via this JSON.

4.3 Exporting via JSON

We use a Python script to connect to the power manager over D-Bus and enable a set of monitors for telemetry. A handler function receives a collection of device/value pairs. Each device either has value *unknown* or *uninitialized*, or contains a list of monitors with values. We built up a JSON using a very similar hierarchy. It contains at least the following entries:

- A timestamp, in particular the UNIX time at which telemetry data is received
- Various system information like the BMCs name
- An array of devices, which in turn contain monitors and values

This is a sample of such a JSON file:

```
{
  "timestamp": {
      "unix": 1613384456.961348
  },
  "system_info": {
      "hostname": "partpop2_bmc"
  },
  "errors": [],
  "devices": [{
          "value": "UNINIT",
          "name": "max20751_mgtavcc_fpga",
          "altname": "MGTAVCC_FPGA_I"
```

```
        },
        {
            "name": "max15301_vadj_1v8",
            "monitors": [{
                    "name": "IOUT",
                    "altname": "VADJ_1V8_FPGA_I",
                    "value": 0.0
                },
                {
                    "name": "VOUT",
                    "altname": "VADJ_1V8_FPGA_V",
                    "value": 0.248779296875
                }
            ]
        }
    ]
}
```

The web application, which we'll present in chapter 5, creates visualizations dynamically, based on the list of devices and monitors in the JSON. Therefore, to adjust the set of devices or monitors to be displayed, we can simply add or remove entries in the Python script, which are responsible for enabling telemetry on the corresponding monitor. Any data output by the telemetry service will be included in the JSON by the handler function.

Fresh telemetry data is acquired once a second, and we overwrite the JSON file on every update. The file therefore only contains the most recently measured state. This decision was made for two reasons. The first one relates to the problem discussed in the previous section. Currently, our application continuously acquires the most recent data to build up a live history on the client side, and it doesn't rely on the BMC storing historical data. This makes it easy to update the application to use the Redfish API in the future, once a solution for exporting monitoring data over Redfish is implemented. That would not be feasible for an application that relies on the ability to fetch historical data. Reason two is, that if past states are to be kept, using a simple JSON file is not a scalable solution, as it would simply grow over time, and fetching it would require more bandwidth. Unless the application fetches the JSON for the first time on startup, it will only need to extract the most recent system state from it. A lot of the bandwidth used to fetch the file on every refresh would therefore be wasted. In order to mitigate that, it would be better to maintain a database on the BMC, that allows a client to query only the required information. This might be worth exploring the future.

To make the JSON file easily accessible from outside the BMC, we use OpenBMC's existing infrastructure. The integrated web server, bmcweb, automatically serves all files located in its web root `/usr/share/www/`. We therefore store the JSON in the file `/usr/share/www/monitoringData.json`. If the file doesn't exist at the time the service

is started, bmcweb might need to be restarted before it picks up the file and starts serving it via HTTPS on port 443.

5 Web Application

5.1 Intro

This chapter is dedicated to the user-facing web application. The purpose of this application is to fetch monitoring data that is exposed by the BMC in the form of a JSON file as described in section 4.3, and visualize this data to allow a user to easily get an overview of the current state of the Enzian system.

5.1.1 Technology Stack

The three most basic technologies used in the monitoring application are HTML, CSS and JavaScript. Visual styling is done with the help of the Bulma CSS framework [12], which allows us to keep efforts spent on CSS to a minimum and enables the app to be adaptive to different screen sizes and to run even on a mobile device. jQuery [13] is also used, specifically it's Ajax function in order to retrieve and parse the JSON file from the BMC. To turn the datasets into visual representations like charts, the library Chart.js [14] is used, and lastly, to convert the timestamps from UNIX time to a format supported by Chart.js, we include Moment.js [15]. The application logic itself is written in pure JavaScript.

5.2 Core Logic

5.2.1 Overview

The overall behavior of the application is best is described by explaining the most important functions. The applications' "heart beat" is determined by an interval timer that runs every second to call the Ajax fetch function. This function fetches the JSON file from the BMC. On success, it parses the JSON and passes the parsed data to an update function. This one will first check if the newly acquired data has a newer timestamp than the one from the previous update to discard duplicates. It will then iterate through all the data in the JSON. This entails checking if there is a corresponding chart object for each of the devices, making sure there's a dataset for each of the monitors, and then finally adding all the new samples to the datasets and update all visualizations. If a device is encountered for the first time, a separate function is called that creates the chart, the HTML containers and canvas, and adds everything to a device wrapper object, which

can then be stored in a global device map. We'll cover these data representations in the next section.

As explained in chapter 4, one of the design considerations was that the BMC only exposes the most up-to-date state of the system and doesn't keep any older information. This has a major implication on the design of the web app. For one, it is not possible to show system information from before the moment the application has been opened. And secondly, it is now part of the application's job to keep the information it fetched and continuously build up a history to be able to visualize trends. Ways to store data on the side of the web application (e.g. the client side) across multiple sessions have been considered. One could for example make use of the browser store, or let the user save the current session to a file and later load it again. However, there would be a gap between the sets of data from different sessions, so this would not prove very useful and only increase the complexity of the application.

5.2.2 Data structure

A central part of the application is the monitoring data. We'll dedicate this section to explain how this data is being handled and stored.

Samples

Samples make up the core of all the data. They simply contain a timestamp and a value:

```
{
  t: unix_time,
  y: value
}
```

Device map and wrappers

A global map called *deviceMap* is the key to storing all data. It maps from device names to objects, which we call device wrappers. Every wrapper has a set of fields. Arguably the most important one of these fields is called *allData*. It holds an array that contains all the data that has so far been accumulated on that device. The array in turn contains more arrays, one for each monitor that the device owns. For each monitor, we then have an array of objects. The number of these objects is determined by the number of modes the application has. Modes will be explained later in this chapter. Every one of these objects now has two fields. The *samples* field contains the final array that contains actual samples. The other field stores the UNIX time, representing the time of when we last added a sample to this mode's array. The following bit of JavaScript hopes to serve as an illustration of the structure:

```
deviceMap = new Map(); // Maps device names to their wrapper objects
wrapper = deviceMap.get(device.name); // Device wrapper object
```

```
wrapper.allData[i][j].samples[k]; // Sample k of monitor i in mode j
```

We also create a chart for every device, so the device wrapper further contains the chart object created by the Chart.js library, as well as references to various HTML elements that contain the chart, like the surrounding div container and the title element.

5.2.3 Data Decimation

One big question that arose when creating the first time series charts for the monitors was: What time span should the charts cover? This is not easy to answer, since ideally, the visualizations should be useful both for short- and long term monitoring. In addition to that, there is also a limit to how many data points can be displayed in a chart, without it becoming too cluttered. New samples are being fetched in one-second intervals. If we show a time span of one minute, showing all 60 data points is reasonable and the chart shows a detailed trend. However, if we want to display a time span of one hour in a chart, we have 3600 samples for that time window, and showing a data point for each one of them would make the chart too dense to be useful. We therefore need to use data decimation.

Down sampling by skipping samples

The initial approach entailed keeping two arrays per monitor. One that contained every single sample that we collected so far, and another, shorter one, which's samples are to be displayed in the chart. The short array would be freshly generated every time the chart is updated by taking every *nth* sample from the full array such that we don't end up with more data points than we want to display maximally. It turned out that using this method led to a very erratic behavior of the lines in the charts. One might observe a peak in a certain spot on the axis and expect it to move along over time, but instead it just disappears, while new peaks suddenly arise in other locations. This is due to the fact that if there's a short peak or outliers, the corresponding samples don't always end up in the short array, since they might be skipped in one iteration, and not in another. For a monitoring application, such behavior is very unintuitive and unhelpful as one does not expect past data points to appear and disappear.

Average and Max/Min

To try and keep the trends more visually consistent, a slight variation of the previous method was explored: Instead of taking every *nth* sample, we instead split the full array into chunks of size n. The short array is then created by taking the average of each chunk. Instead of the average one can also take the minima or maxima. The latter two to seemed to be a good solution for the issue of consistency. Peaks were now predictably moving along the axis over time. However, this only works for high peaks when the maximum function is used, or low peaks with the minimum function respectively. For example, if

the minima are taken and the chunks cover a time span of one minute each, a positive peak that lasts for 20 seconds would simply be "swallowed" by surrounding samples and therefore not show up in the chart at all. A similar problem arose when using the method of averaging. Peaks no longer disappear entirely, but especially short ones get attenuated to the point where it is impossible to see a conclusive trend. As an example, if CPU load was to be sampled in a system that is mostly idle, it would be impossible to distinguish between cases 1) CPU load reaches its maximum for a very short time and 2) CPU load is slightly elevated for an extended period of time. Both cases lead to a very similar looking trend if the overall time span of the chart and therefore the chunks are sufficiently large.

Both approaches result in potentially valuable monitoring information not being seen by the user. Furthermore, they both have yet another issue. Iterating through the full array of samples to generate a shorter array is computationally expensive. This has to be done every time the charts are updated, and for every monitor. We therefore decided to come up with yet another approach: Modes.

Modes

The idea is simple: Instead of trying to find a compromise between covering a long time span and not losing too much information by down sampling, we allow the user to choose the time granularity of samples to be visualized. For example, a mode to show just the last minute of samples does not require any down sampling, since we can reasonably display all 60 samples. If the time span is larger, we down sample again in a way similar to the first approach by skipping certain samples.

To solve some of the issues we encountered in the first approach, we implement the data decimation differently this time. No longer do we keep an array that contains the complete set of samples the application has stored so far. Instead, we maintain one array for each mode, as described in the chapter about the data structure. For every mode, we have a certain resolution at which we want to display samples. If we display at most 60 samples in a chart and the chart covers a time span of 24 hours, the resolution is one sample per 24 minutes. Whenever a new sample is fetched, we go through each of the modes and depending on the timestamp of the previously added sample, we add the new sample to the corresponding array. For the 24-hour mode, this means that we add a new sample only if the previously added sample is at least 24 minutes old. Now, we can simply feed the chart the array corresponding to the mode that is currently active. Arrays are kept at a desirable length by shifting them whenever they become too long, removing the oldest sample.

Comparison

Apart from not having to compromise and giving the user choice, the last approach has further advantages compared to the previously discussed methods. For one, it is much

more efficient. It might not sound like it initially, since we now have to keep track of even more arrays. All of these arrays however are comparatively small. If we have five modes and show at most 60 data points in a chart, we store at most 300 samples per monitor. To put this into perspective: In the previous approaches we kept one array containing all recorded samples, up to 24 hours, which means up to 86400 samples. In addition to that, we needed to iterate through that large array every time when down sampling.

Another advantage is that we no longer observe the erratic behavior of the data points as mentioned in chapter 5.2.3. Since for every mode, we simply add new samples to the front of the array and remove only elements at the back, past data points no longer suddenly change. They simply move along the axis as time progresses.

5.3 Front End

The user interface consists of a navigation bar at the top, and a main body that takes up the rest of the browser window. The bar contains the title with an Enzian logo, a dropdown menu allowing the user to choose between different modes and two buttons. One is to clear all the samples that have been accumulated so far and clear all the charts. The other button allows pausing updates of the charts, in case the users wants to inspect a chart without it continuously changing and its data points moving along the axis. The application will still continue to fetch new data in the background. The navigation bar and its buttons can be seen in figure 5.1, while the dropdown with the different modes is open in screenshot 5.3.

The main body contains all the visualizations. They are each contained in a box with some padding and shadows to bring them out of the background. On a wide screen monitor or tablet, two columns of boxes will be displayed, while on a smaller display or shrunken browser window, the interface will automatically adapt to show only a single column, as shown in figure 5.2.

Each device has an associated chart, which contains a dataset for each of the devices' monitors. Every dataset is displayed in the chart by default, but by clicking on the name of the dataset it can be toggled on or off.

For devices that are uninitialized or unknown, the chart will still be displayed, but its box will have a red border and a text indicating its state. A device can also be turned off by the power manager while the web application is running and already has some samples. In that case, the application will also indicate the updated state with a red border. It will keep the samples it already collected before and cede to update the chart until the device is turned on again and new samples are fetched.

5.4 Deployment

The application can be hosted either on a local development machine or on the BMC. It is relatively straight forward, as the HTML, CSS and JavaScript files of the application

can simply be placed in a directory and served by a web server. For development, we ran a python web server to serve the source directory of the project to be able to open it on the local machine. One the BMC, the application can be placed in the web root of the bmcweb web server, which is located at `/usr/share/www/`.

If hosted locally for development purposes, fetching the JSON file from an actual BMC might not always be feasible. For such cases, a python script is used to generate a JSON file containing the same structure, filled with a few example devices and monitors, one uninitialized device, and randomly generated values. To work with traces of an actual BMC, a connection to it must first be established either directly or via a proxy. This is done in the same way as described in section 4.3. Once a connection is available, the address to the JSON file must be specified in the application's source code.

As mentioned in section 4.3, HTTPS must be use for requests to the web server, as it will not respond otherwise, and since the bmcweb uses self-signed certificates by default, it might be necessary to first visit the URL for the JSON file directly and instruct the browser to make an exception. Otherwise, the browser might disallow the web application to fetch the file.

Also note that if the web application and the JSON file are not hosted on the same origin, the browser likely restricts the request to fetch the file, due to the enforcement of the same-origin policy of CORS (Cross-Origin Resource Sharing). CORS is a mechanism that allows web servers to add HTTP headers that indicate other origins from which browsers are permitted to load its resources. The exception to this is loading of external scripts that might be included in web pages. If the web application is hosted on a local machine, fetching the JSON file from a remote server counts as a cross-origin request, and for security reasons, this is blocked by browsers unless the other origin server includes the right CORS headers. In our case, the bmcweb web server would have to add headers that allow resource loading either from any other origin, or specifically from the address where the developer loads the web application. A workaround for during development is to use one of the browser extensions available that allow one to selectively override CORS restrictions. Once the web application is hosted on the BMC itself, CORS is not a problem, as both the application and the JSON file will have the same origin.

5.5 Showcase

This section contains screenshots showcasing the application.

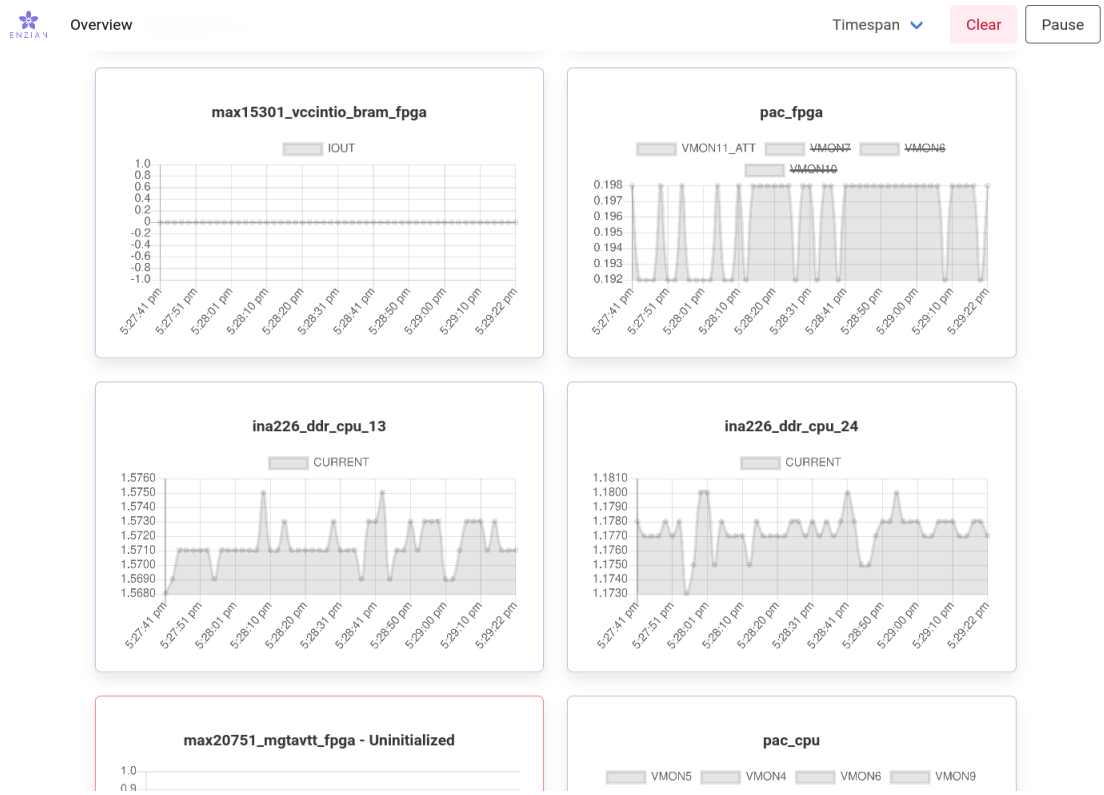

Figure 5.1: A screenshot of the application, running on a desktop browser. Some devices are still uninitialized, as indicated by the red border on one of the charts.

Figure 5.2: A screenshot of the application, running on a mobile browser. Entries in the navigation bar are accessible via a hamburger menu and only a single column on visualizations is shown. All devices are currently uninitialized.

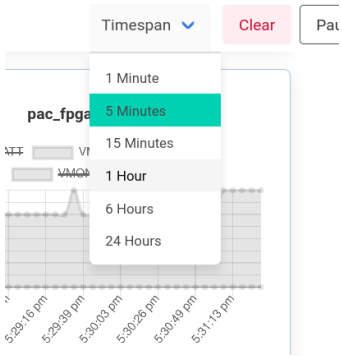

Figure 5.3: A screenshot of the dropdown menu showing the different time granularity a user can choose for the charts.

6 Future Work

6.1 Storing historical data

As outlined in section 4.3, we decided to go for a simple approach and export the system state using a JSON file that only contains the most recent state. The monitoring application therefore builds its own history continuously from an online stream of data, without having access to any prior data. If in the future, it becomes a requirement to make past data available, this data would likely need to be stored on the BMC. A database tool like RRDtool might be worth considering. It stores data in a database based on circular buffer, thus keeping the database size constant being well suited for time series data.

6.2 Exporting to D-Bus / Redfish

Currently, sensor values are not exposed on D-Bus, and in turn made accessible via Redfish. We stated some possible solutions already in section 4.2. It would enable integration with the existing web interfaces like webui-vue and allow them to populate the health section, showing basic information about the state of the hardware and sensors. Furthermore, the monitoring web application could be adapted to acquire its data over the same unified, standard Redfish interface.

6.3 FPGA

The on-board FPGA is the piece that's sets Enzian apart from other server platforms. However barely any information on it is exposed for monitoring so far. And instead of monitoring it, there is also potential to use the FPGA itself to monitor other parts of the system. Aggregating data produced by the system and doing preprocessing and even anomaly detection could be a possible use case for the FPGA. Baumeister et al. [16] present such an FPGA-based approach for evaluation of real-time data.

6.4 Cluster monitoring

Eventually, Enzian systems might be deployed and used in the form of a cluster. The monitoring application built as part of this project is for monitoring an individual Enzian system and currently there is no way to easily monitor multiple systems at one. This is

therefore another direction that can be explored in the future, possibly once monitoring data can be exposed over Redfish.

6.5 Visualizations

The current set of visualizations is still limited and there is potential for adding more features. A view for showing correlations of different metrics, or different overviews like showing for example how power is distributed in the system could be useful and would be possible with the current infrastructure. Information about the network or main memory on the other hand would require more work, as there is no simple way to acquire this on the BMC yet.

7 Conclusion

This thesis addressed the topics of data monitoring and visualization. We talked about technologies and previous work in this field. We created a catalog of information that could be monitored and visualized, and discussed what can be visualized on the Enzian BMC specifically. Furthermore, we described the architecture of OpenBMC and discussed different approaches of extracting and exporting data on the Enzian BMC. Some issues were encountered, when we learned that the default OpenBMC web interface builds on Redfish and the BMC on Enzian cannot expose monitoring data via Redfish yet. However, we worked around it and created a tool to acquire data from the power manager and export it via JSON. To visualize this data, we then built a web application based on HTML and JavaScript. It works by fetching the JSON from the BMC and building up a local time series of samples to display to the user. We discussed internal representations of samples and compared different ways of data elimination, before showcasing the front end and providing instructions on how to deploy it.

Bibliography

[1] G. Gonçalves, D. Rosendo, L. Ferreira, G. L. Santos, D. Gomes, A. Moreira, J. Kelner, D. Sadok, M. Wildeman, and P. T. Endo, "A standard to rule them all: Redfish," *IEEE Communications Standards Magazine*, vol. 3, no. 2, pp. 36–43, 2019.

[2] G. Mills, *Redfish on OpenBMC*, `https://2019.osfc.io/uploads/talk/paper/41/Redfish_on_OpenBMC.pdf`.

[3] A. Libri, A. Bartolini, and L. Benini, "Dig: Enabling out-of-band scalable high-resolution monitoring for data-center analytics, automation and control," en, 2nd International Industry/University Workshop on Data-center Automation, Analytics, and Control (DAAC 2018); Conference Location: Dallas, TX, USA; Conference Date: November 12, 2018, Dallas, TX: Data-center Automation, Analytics, and Control (DAAC), 2018.

[4] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, 2010, pp. 189–194.

[5] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: Design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[6] V. J. Karbach C., "System monitoring: Llview," in *Introduction to the programming and usage of the supercomputer resources at Jülich November 2015 (Course no. 78a/2015 in the training programme of Forschungszentrum Jülich)*, 2015.

[7] T. Dang, "Visualizing multidimensional health status of data centers," in *Programming and Performance Visualization Tools*, A. Bhatele, D. Boehme, J. A. Levine, A. D. Malony, and M. Schulz, Eds., Cham: Springer International Publishing, 2019, pp. 273–283.

[8] W. Barth, *Nagios: System and network monitoring*. No Starch Press, 2008.

[9] N. Bremer, *A different look for the d3.js radar chart*, `https://www.visualcinnamon.com/2015/10/different-look-d3-radar-chart`, Accessed: 2020-10-11, 2015.

[10] The Linux Foundation, *OpenBMC home page*, `https://www.openbmc.org`.

[11] J. Zemlin, *OpenBMC Project Community Comes Together at The Linux Foundation to Define Open Source Implementation of BMC Firmware Stack*, https://www.linuxfoundation.org/blog/2018/03/openbmc-project-community-comes-together-at-the-linux-foundation-to-define-open-source-implementation-of-bmc-firmware-stack/.

[12] J. Thomas, *Bulma CSS framework*, https://bulma.io/.

[13] J. Resig and the jQuery Team, *jQuery home page*, https://jquery.com/.

[14] E. Timberg and contributors, *Chart.js library*, https://www.chartjs.org/.

[15] M. contributors, *Moment.js library*, https://momentjs.com/.

[16] J. Baumeister, B. Finkbeiner, M. Schwenger, and H. Torfah, "Fpga stream-monitoring of real-time properties," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019.

List of Figures