ENS 491-492 – Graduation Project
Final Report

Real-Time Supercomputer Monitoring

Muhammed Orhun Gale, Ali Eren Ak, Deniz Batu
Supervisor: Kamer Kaya

January 8, 2023

1 Executive Summary

An increase in architectural and operational complexities of High-performance Computing (HPC) clusters makes bottleneck detection harder in those systems since sufficient monitoring of HPC clusters is challenging. Even though there are various solutions for different bottlenecks, these solutions are applicable only after a diagnosis of the bottleneck. In this project we introduce a Supercomputer monitoring tool *SuperTwin* which can efficiently create a digital twin of a HPC cluster to gather system data in a structured manner for efficient real-time monitoring. In addition to monitoring, SuperTwin uses collected data to train models that can detect bottlenecks in HPC clusters, understand performance characteristics and provide hardware/software insights for efficiency and energy consumption and perform prediction-based simulations of various hardware/software scenarios. SuperTwin defines a Supercomputer Digital Twin Description Language (DTDL) to enable structured representation of Supercomputers and achieves real-time monitoring data gathering via setting data collection daemons on the target system which gather data from data collection agents and other daemons by using Performance Co-Pilot (PCP). To collect informative telemetry data from the system, SuperTwin uses Performance Metrics Domain Agents (PMDA), ReuseTracker and ComDetective tools. After collecting the data from configured Performance Metrics Collection Daemons (PMCD) on the system, SuperTwin visualize the collected data in real-time by using Grafana dashboards. Besides being able to used as a console application, SuperTwin also have a web application to provide better user experience.

2 Problem Statement

If a commercial computer uses the full capacity of its CPU while other resources are not fully utilized during a commonly used process such as gaming, swapping the CPU with a better model will improve that computer's power. On the other hand, bottleneck detection on a supercomputer is a much more complex task due to the complexity of the underlying systems and the high number of computation resources as well as thousands of performance metrics available. Thus, sophisticated investigation tools that utilize performance monitoring tools, techniques, and algorithms must be developed to detect bottlenecks to increase HPC capabilities. Thorough monitoring of said systems can also be utilized to increase power efficiency by using fewer computational resources. Machine learning (ML) is planned to be utilized to detect critical metrics and interpret data.

This project wants to construct a tool for real-time monitoring of supercomputers to allow real-time optimizations to the architectural elements of supercomputers such as Non-Uniform Memory Access(NUMA) configurations to increase the speed of the currently running job on the system. To this end, a digital twin creation and monitoring tool, the SuperTwin, has been under development. A digital twin is a digital representation of a physical system that has established communication channels with the entity to monitor its metrics to create a copy of it that can be digitally analyzed and used to instruct the system for optimization in real time. This tool is designed to be lightweight and user-friendly to enable harmonious integration into existing HPC organizations without obscuring the normal use of the structure and to allow ease of use by interested parties. Simulation capabilities and ML are planned to be added to SuperTwin's capabilities to advance the tool.

There have been many solutions developed for the bottleneck detection problem. The main issue with these solutions is the fact that either the monitoring algorithm slows down the computation of the monitored program significantly or that the monitoring takes place after the execution of the program[SCAU19]. These two problems make current solutions unable to be seamlessly integrated into supercomputing operations and offer real-time optimization possibilities. Meanwhile, simulation-based and sampling-based approaches have shown efficiency increase proofs. For example, Application Heartbeats [HES+10] describes a general user interface to monitor Autonomous Computing Environments performances. Their work proves the performance increasing and optimizing capabilities of Machine Learning and manual observer interference made possible by collecting performance data regularly with intervals called Heartbeats. Most applications on HPCs do not offer high-level data and analyzing performance over low-level data can yield inaccurate results in performance analysis. For example, counting the executed instructions over a period does not yield information about the actual work that was done; considering CPU utilization or cache miss rates causes similar problems. If an application

achieves more performance than it requires, its resources can be allocated to other applications and if an application is underperforming, more resources can be allocated for performance optimization.

2.1 Objectives/Tasks

A comprehensive, lightweight, user-friendly, real-time monitoring and investigation tool stack was aimed to be constructed as a framework that can connect to and probe a supercomputer to create a digital twin which will collect structured data to allow bottleneck detection and performance optimizations. To this end, five main components were to be realized. A digital twin creation module, the real-time gathering of performance metrics, a database for storing and accessing time-series data, a user interface for interacting with the tool and visually displaying and analyzing the metrics, and an ML module for automated analysis, predictions, and simulations.

Digital twin creation was aimed to take place by first probing the target remote computer cluster to gather its structural information such as software, and hardware capabilities, and establishing communication before setting up benchmarks that are used for gathering utilization information and creating the Digital Twin Description(DTD) of the system.

The digital twin was aimed to collect user-specified run-time metrics to detect real-time communication in parallel systems through the framework's various metric collection capabilities that utilize various tools such as Performance Co-Pilot's(PCP) Performance Metrics Domain Agents(PMDA) that are utilized via PCP's daemons.

Another tool to be implemented was ComDetective. ComDetective detects inter-thread communications by configuring Performance Monitoring Units (PMU) to send an interrupt signal on event thresholds such as reads to sample the effective data address of the instructions that exceeded the threshold to keep in a board and arm debug registers of target CPUs to trap previously received addresses. When an address that was in the board is detected again or the traps are activated, communication is detected between the initial thread that owned the instruction and the thread whose instruction just caused a trap activation or interrupt. The detected communication is stored in a matrix to represent inter-thread communications which is the output of the tool. Gathering inter-thread communication information was important as such communication is a major slow-down cause and configuring the target system to minimize such communications would lead to a performance increase.

The last tool to be implemented for the metric collection was ReuseTracker [SCMU21]. It is an open-source tool for analyzing the reuse distance of multi-threaded applications in a way that is both fast and memory-efficient. It achieves this by utilizing hardware counters from performance monitoring units and debug registers, which are widely available in current CPUs. In its design, ReuseTracker uses performance monitoring units to sample memory accesses, which also called uses, in each profiled thread. After that, debug registers are configured to capture subsequent accesses for each sample. Subsequent accesses are called reuses that is wanted to monitor for specific applications. Aim is to monitor reuse distances to analyse memory access of running application on the system, which is a widely used metric that measures data locality. Data locality is a significant point for HPC clusters since the capacity of high-speed links between separate storages is less than the bandwidth of the compute nodes [GFZ12]. Thus, better data locality increases computation power and efficiency in HPC clusters

Attributes	[BH05]	[WLC19]	ReuseTracker
Time Overhead	1.4x<	2.03x	2.9x
Space Overhead	-	2.8x	2.8x
Accuracy	-	90%	92%
Method	PMU	PMU	PMU
Open Source	-	YES	YES

Table 1: Available Reuse Distance monitoring tools

ReuseTracker is built on the HPC Toolkit [AFK+08], a suite of tools for analyzing programs' performance on various computers, from multicore desktop systems to high-performance supercomputers. It uses statistical sampling and hardware performance counters to measure a program's CPU usage, resource consumption, and inefficiencies. It can also monitor GPU operations and gather metrics for GPU-accelerated codes. HPCToolkit supports various programming languages and parallelization approaches, including serial code, threads, MPI, and hybrid MPI/threads, and it works with both statically and dynamically linked applications. It is particularly well-suited for use on large parallel systems and provides the following:

- Tools for analyzing the execution costs.
- Inefficiencies.
- Scaling characteristics of a program within and across nodes.

However, HPC Toolkit is a significantly large dependency of Reuse Tracker that we wanted to remove before adding Reuse Tracker to the SuperTwin monitoring daemon. Additionally, it extends ReuseTracker with other functionalities, such as source-code attribution, which is unnecessary in SuperTwin. That is why it is aimed to integrate Reuse Tracker into SuperTwin's monitoring daemon without using HPCToolkit dependency.

The collected data was to be transferred from the remote computer cluster to a local monitoring machine and then stored in a database suitable for time-series data. There were data conversions, communication channel establishments, and communication protocols necessary to be implemented.

The user interface in the form of a web application was planned to allow the selection of the metrics to be gathered, visualize, and display all the gathered metrics or select specific metrics to be displayed in user-friendly graphs. Also, the interface has data gathering daemon tracing capability as well as sending commands to the remote machine to allow run-time intervention and run experiments. The tool was aimed to have the capability of monitoring not only one but multiple machines to achieve cluster-scale monitoring and management.

2.2 Realistic Constraints

This project requires access to an HPC system whose users accept sharing metadata, job description, and cluster information with our project team for real world development and testing of our monitoring tool and sub-systems. This is a concern since users such as companies might not want to share inside knowledge with us due to security concerns. Changes in HPC architecture, black-box systems, new chip models and the general evolution of HPC systems might render our tools obsolete. For example, a new chip model that does not have hardware counters will render us unable to collect metadata from it. We aim to develop our data acquisition tool such that it checks for available metrics in systems and pulls the metric data to the digital twin.

3 Methodology

3.1 Digital Twin Description Language for Supercomputers

Creating the digital twin of a physical device enables continuous data reception from the actual device in a structured manner which is enabled by the digital twin description (DTD) of the device that is created via using the pre-defined digital twin description language (DTDL). Having a structured data extraction capability allows monitoring devices in a more purposive way which prevents redundant data collection and enables accurate bottleneck detection by orienting data collection efforts to the critical parts. Besides monitoring, having a structured data pipeline also facilitates efficient anomaly detection and hardware modification simulations. In this sense, a DTDL is defined to create DTDs of Supercomputers as the first step of the project.

As the basis of the DTDL for Supercomputers, we used the JSON-LD and Azure DTDL ontology. Likewise in the Azure DTDL, Supercomputer DTDL recursively describes components of the actual device contextually by using meta-classes shown in the table [2] "Interface", "Telemetry", "Property" and "Relationships". The DTDL captures the hierarchical structure of the Supercomputers and ensures that all components are represented. Also note that all described components can be seen as an individual digital twin and individually used.

Property	*Description*
@type	**Interface**
@id	Unique identifier within digital twin for interface
contents	Data type, a set of Telemetry, Properties, Commands, Relationships, Components
displayName	Name to be displayed when instantiated
@type	**Telemetry**
@id	Unique identifier within digital twin for this telemetry instance
name	Programming name, to be referred in queries
schema	Data type
@type	**Property**
@id	Unique identifier within digital twin for this property instance
name	Programming name, to be referred in queries
schema	Data type
@type	**Relationship**
@id	Unique identifier within digital twin for this relationship
name	Programming name, to be referred in queries
properties	Data type, a set of properties

Table 2: Property types that are defined in the Supercomputer DTDL.

While creating a Supercomputer's DTD, head node and each compute node are described as a component of the cluster. For each node, starting from the highest abstraction level, major components namely CPUs, GPUs, L1-2-3 caches, memory, disks and internal network, are defined and inserted into the DTD. Moreover, while extracting the component structure, possible software and hardware telemetry components are found and inserted into the DTD for each component of the node.

3.2 Data Collection Tools

Source	Source Description		Metric Description
/proc	Kernel statistics	kernel.all.intr	Context switch metric from /proc/stat
		kernel.all.pressure.cpu.some.total	Total time processes stalled for CPU resources
		kernel.all.pressure.memory.some.total	Total time processes stalled for memory resources
		kernel.all.pressure.memory.full.total	Total time when all tasks stall on memory resources
		kernel.all.pressure.io.some.total	Total time processes stalled for IO resources
		kernel.percpu.interrupts.PMI	Performance monitoring interrupts for each core
		kernel.percpu.interrupts.TRM	Thermal event interrupts for each core
		kernel.percpu.interrupts.line*	Number of interrupts caused by each IO device
/proc/meminfo	System memory statistics	mem.util.used	Used system memory
		mem.util.free	Free system memory
		mem.util.directMap4k	Amount of memory that is directly mapped in 4kB pages
		mem.util.directMap2M	Amount of memory that is directly mapped in 2MB pages
		mem.util.directMap1G	Amount of memory that is directly mapped in 1GB pages
		swap.pagesin	Pages read from swap devices due to demand for physical memory
		swap.pagesout	Pages written to swap devices due to demand for physical memory
	NUMA statistics	mem.numa.util.free	Per-node free memory
		mem.numa.util.used	Per-node used memory
		mem.numa.alloc.hit	Per-node count of times a task wanted alloc on local node and succeeded
		mem.numa.alloc.miss	Per-node count of times a task wanted alloc on local node but got another node
		mem.numa.alloc.local_node	Per-node count of times a process ran on this node and got memory on this node
		mem.numa.alloc.other_node	Per-node count of times a process ran on this node and got memory on another node
/proc/vmstat	Virtual memory statistics	mem.vmstat.kswapd_low_wmark_hit_quickly	Count of times low watermark reached quickly
		mem.vmstat.kswapd_high_wmark_hit_quickly	Count of times high watermark reached quickly
/proc/net/dev	Network interface statistics	network.interface.in.bytes	Network recv read bytes per network interface
		network.interface.out.bytes	Network send bytes per network interface
/proc/diskstats	Disk statistics	disk.dev.read	Per-disk read operations
		disk.dev.write	Per-disk write operations
		disk.dev.read_merge	Per-disk count of merged read requests
		disk.dev.write_merge	Per-disk count of merged write requests
/proc/<pid>/*	Per process statistics	proc.psinfo.ngid	NUMA group identifier
		proc.psinfo.threads	Number of threads
		proc.psinfo.nvctxsw	Number of non-voluntary context switches
		proc.psinfo.processor	Last CPU the process was running on
		proc.psinfo.cmaj_flt	Count of page faults other than reclaims of all exited children
		proc.psinfo.maj_flt	Count of page faults other than reclaims
		proc.io.wchar	write(), writev() and sendfile() send bytes
		proc.io.rchar	read(), readv() and sendfile() receive bytes
Hardware Counter		Metric	Metric Description
CPU_CLK_UNHALTED.THREAD		Cycles	Counts the number of core cycles while the logical processor is not in halt state
MEM_INST_RETIRED.ALL_LOADS		Loads	Counts the number of retired loads
MEM_INST_RETIRED.ALL_STORES		Stores	Counts the number of retired stores
L1D.REPLACEMENT		L1 Data Misses	Counts the data line replacements that occur on L1D cache
LLC_REFERENCE - L2-RQSTS.CODE_RD_MISS		L2 Data Misses	Number of data requests that miss L2D cache. Corresponds to the difference between every core request that references a cache line in LLC and the L2 code misses
CAS_COUNT.RD+CAS_COUNT.WR		LLC Misses	Sum between all DRAM reads and all DRAM writes
CYCLE_ACTIVITY.STALLS_L1D_MISS		L1 Data Stalls	Stalls that occur due to outstanding loads that miss L1D cache
CYCLE_ACTIVITY.STALLS_L2_MISS		L2 Stalls	Stalls that occur due to outstanding loads that miss L2 cache
CYCLE_ACTIVITY.STALLS_L3_MISS		L3 Stalls	Stalls that occur due to outstanding loads that miss L3 cache
CYCLE_ACTIVITY.STALLS_MEM_ANY		Memory Stalls	Stalls that occur due to outstanding loads in the memory subsystem
CYCLE_ACTIVITY.CYCLES_L1D_MISS		Cycles with misses on L1 Data	Cycles while there are outstanding loads that miss L1D cache
CYCLE_ACTIVITY.CYCLES_L2_MISS		Cycles with misses on L2	Cycles while there are outstanding loads that miss L2cache
CYCLE_ACTIVITY.CYCLES_L3_MISS		Cycles with misses on L3	Cycles while there are outstanding loads that miss L3 cache
CYCLE_ACTIVITY.CYCLES_MEM_ANY		Cycles with outstanding loads	Cycles while there are outstanding loads in the memory subsystem
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE		FP Scalar Double	Double-precision scalar FP instructions
FP_ARITH_INST_RETIRED.SCALAR_SINGLE		FP Scalar Single	Single-precision scalar FP instructions
FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE		FP 128-bit SIMD Double	Double-precision 128-bit packed FP instructions
FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE		FP 128-bit SIMD Single	Single-precision 128-bit packed FP instructions
FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE		FP 256-bit SIMD Double	Double-precision 256-bit packed FP instructions
FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE		FP 256-bit SIMD Single	Single-precision 256-bit packed FP instructions
FP_ARITH_INST_RETIRED.512B_PACKED_DOUBLE		FP 512-bit SIMD Double	Double-precision 512-bit packed FP instructions
FP_ARITH_INST_RETIRED.512B_PACKED_SINGLE		FP 512-bit SIMD Single	Single-precision 512-bit packed FP instructions

Table 3: Kernel performance metrics from /proc and hardware counter metrics that used model sparse computation performance.

3.2.1 Performance Co-Pilot (PCP)

Performance Co-Pilot (PCP) is a system performance analysis toolkit which enables real-time system level data gathering and monitoring. PCP manages data collection by utilizing two main units which are Performance Metrics Domain Agents (PMDA) and Performance Metrics Collection Daemons (PMCD). PMDAs are data collection agents each of which listens a specific system level data sources and gathers data while PMCDs are data gathering daemons which collect data from PMDAs and other PMCDs. A PMCD is configured to constantly pull the collected data from PMDAs that are assigned to itself and it directs the data to an end-point such as an InfluxDB database. PCP discovers the system and generates a Performance Metric Name Space (PMNS) for structured data colelction. Also, has an internal tool Performance Metrics Inference Engine (PMIE) that evaluates various assertions

to provide automated filtering and automated reasoning of performance metrics.[1] [PCP]

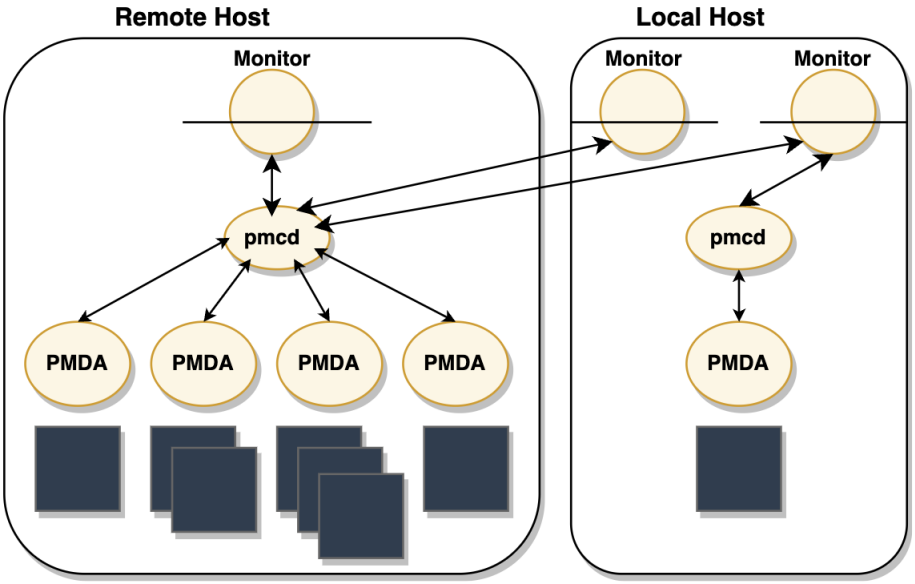

Figure 1: [PCP]'s remote data collection structure

SuperTwin configures following PMDAs for collecting monitoring metrics: perfevent PMDA, proc PMDA, linux PMDA and lmsensors PMDA. Since there exist 75 available PMDAs, monitoring extent of the system with PMDAs can be broaden for specific uses. An example set of metrics that can be collected by using PCP can be seen at table [3]. [PCP]

3.2.2 ReuseTracker

Reuse Tracker has two main parts; calculating use distances with PMUs and trapping reuses with debug registers. In order to calculate use distances, it is necessary to sample memory addresses accessed by each thread in the monitored program. Additionally, to identify reuses, it is needed to determine whether the memory access immediately following a sample address access is also accessing the same address.

To sample memory accesses, performance monitoring units are used. They are components of CPUs capable of counting various hardware events, such as retired instructions, memory loads, and memory stores. PMUs can be configured to trigger an interrupt, called a counter overflow interrupt, after a certain number of events, which is adjustable, have occurred. When the interrupt is triggered, a signal handler within ReuseTracker handles the operating system signal and collects data about the event that caused the interrupt, including the memory address.

Listing 1: PMU Signal Handler

```
static void pmu_signal_handler(int signum, siginfo_t *info, void *uc) {
    // disable file descriptor to read data
    ret=ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);

    // read data and config debug register
    prev_head=config_debug_register(our_mmap,MMAP_DATA_SIZE,prev_head,
            sample_type,read_format,
            0, /* reg_mask */
            NULL, /*validate */
```

```
            quiet,
            NULL, /* events read */
            0);
    count_total++;
    // send signal to pmu to refresh
    ret=ioctl(fd, PERF_EVENT_IOC_REFRESH, 1);
    (void) ret;
}
```

Hardware debug registers are used to detect whether a memory address is being reaccessed. Hardware debug registers are also hardware components that trap the execution of a program when the program counter reaches a specific instruction address (a breakpoint) or when an instruction accesses a monitored address (a watchpoint). When a thread sets up a debug register to trap future memory access to a particular address, we refer to this as "arming a watchpoint" on that address. Debug registers can be configured to trap different addresses, memory access widths, and types of accesses (such as stores or both loads and stores). Processors from x86 architectures generally have four debug registers available.

Performance monitoring units and hardware debug registers can be configured with Linux perf_events tool. Linux provides a system call and ioctl calls that can be used to configure PMUs and debug registers. This capability has been available for debug registers since Linux 2.6.33 and multiple PMUs since Linux 2.6.39. When a PMU interrupt or a debug register trap occurs, the Linux kernel can deliver a signal to the specific thread that encountered the event. User code can use the mmap function to create a circular buffer where the kernel can append the sampled data on each sample and collect the signal context on each watchpoint trap.

Listing 2: perf_event_open config

```
struct perf_event_attr pe;

// handle signal
struct sigaction sa;
sa.sa_sigaction = pmu_signal_handler;
sa.sa_flags = SA_SIGINFO;

// allocate memory for file descriptor
memset(&pe,0,sizeof(struct perf_event_attr));
sample_type=PERF_SAMPLE_IP|PERF_SAMPLE_WEIGHT|PERF_SAMPLE_ADDR;
read_format=0;

// set performance event monitor attributes
pe.type=PERF_TYPE_RAW;
pe.size=sizeof(struct perf_event_attr);
pe.config = 0x82d0;
pe.sample_period=SAMPLE_PERIOD;
pe.sample_type=sample_type;
pe.read_format=read_format;
pe.disabled=1;
pe.pinned=1;
pe.exclude_kernel=1;
pe.exclude_hv=1;
pe.wakeup_events=1;
pe.precise_ip=2;

fd=perf_event_open(&pe, monitored_pid, -1, -1, 0);
```

3.2.3 ComDetective

Similar to Reuse Tracker, ComDetective is built on HPCtoolkit and utilizes PMUs and debug registers to detect inter-thread communication by sampling store and load addresses to try to detect if different threads access the same cache lines and also arm debug registers to trap addresses that were sampled by other threads. ComDetective works by running the program to be monitored with the ComDetective command to profile it using HPCtoolkit's capabilities. On the other hand, the digital twin works by profiling jobs sent to the system automatically. Process IDs of the jobs are used to determine which CPUs are to be monitored. The PMUs and debug register are configured according to the process to send signals to a profiling process that will handle the signals and create the inter-thread communication matrix. Since the digital twin has the capabilities of the HPCtoolkit that are used by the ComDetective tool, the tool was reconstructed without using HPCtoolkit to avoid clogging the system with unnecessary components. The hardware configuration and signal processing modules of the ComDetective were isolated and recreated to be able to monitor given processes. Utilizing the digital twin's abilities, whenever a new job enters the system, a profiling program instance is created to configure the hardware and handle the signals to monitor the job and create the inter-thread communication matrix. Since the system calls and algorithms used for the Reuse Tracker and ComDetective are practically the same, the methodology explained above in the Reuse Tracker subsection will not be repeated. The main difference is how the information received through the signal is used. While the information regarding the data locality is used in the former, the ComDetective approach uses the thread IDs and sampled address' information such as the sample date, operation type(load/store), and cache line.

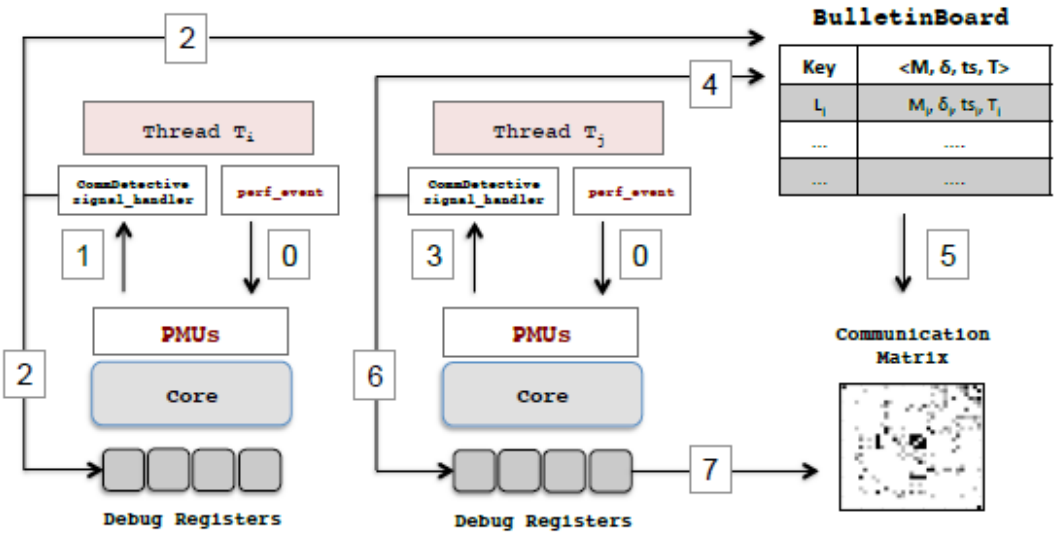

Figure 2: One possible execution scenario: 0) Every thread configures its PMU to sample its stores and loads. 1) Thread T_i 's PMU counter overflows on a store. 2)T_i publishes the sampled address to BulletinBoard if no such entry exists and tries to arm its watchpoints with an address in the BulletinBoard (if any). 3) Thread T_j ' PMU counter overflows on a load. 4) T_j looks up BulletinBoard for a matching cache line. 5) If found, communication is reported. 6) Otherwise, T_j tries to arm watchpoints. 7) T_j accesses an address on which it set a watchpoint, the debug register traps, communication is reported. [SCAU19]

3.3 Supercomputer Monitoring Tool (SuperTwin)

SuperTwin is designed to achieve comprehensive monitoring by collaboratively using digital twin description and data extraction tools. To this end, a "SuperTwin" class is implemented whose instances can generate DTD of a target system by probing it via using proposed system probing tools, set data

collection daemons, configure local time-series database, generate monitoring dashboards and perform anomaly detection.[3] The reason for implementing a class structure rather than a set of scripts for monitoring tool is to enable the tool to efficiently scale from one node to a cluster since every component of the digital twin is also another digital twin which can be monitored independently. Also, having this class makes it possible to include this tool in various projects for using it as a data source or a control mechanism with its remote access facilities such as executing Bash commands on the code. For example, one can create a SuperTwin instance in a Python script with a remote computer's IP and user credentials to collect this computer's real-time energy consumption as a time-series data and use the data by querying local InfluxDB database to train a machine learning model.

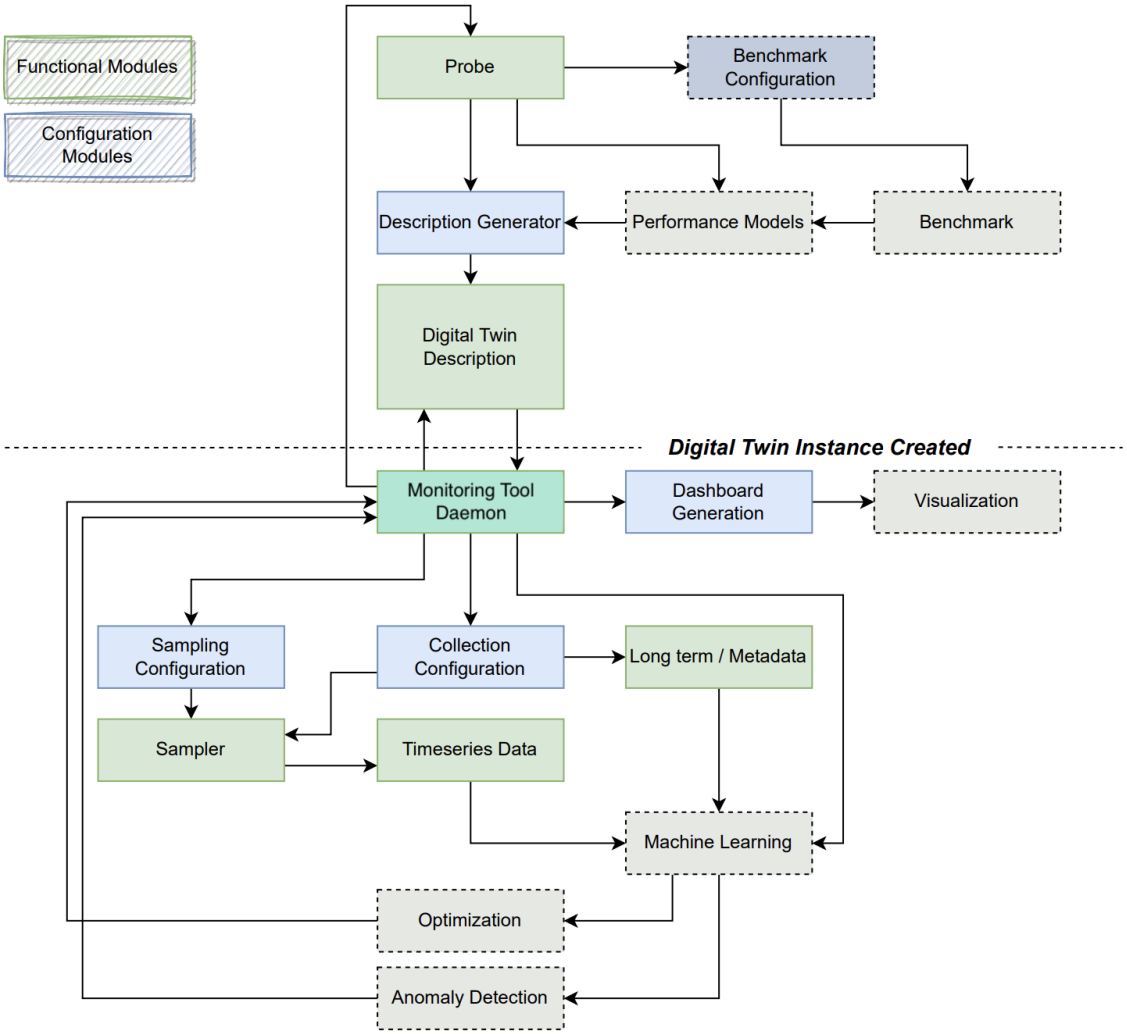

Figure 3: SuperTwin Design

Monitoring of a node with an SuperTwin instance starts with creation of the DTD of the target system. To do that, SuperTwin instance establishes an SSH connection with the remote computer with "sudo" user credentials. Once the connection is established, SuperTwin executes various shell commands on the remote machine to configure PMDCs and PMDAs, setup probing tools and start these services. As the system probing ends, STREAM and HPCG benchmarks are executed for better profiling of the system.[17][18]After the probing and benchmarking processes are finalized, SuperTwin instance copies the generated data to the local machine and passes the data to the digital twin description generator module ("generate-dt") to create the DTD of the system.[4][5][6] DTD of the remote system is stored in a MongoDB database as an entry that is uniquely coupled with the SuperTwin instance and SuperTwin instance stores the metadata to reach the DTD of the system. For clusters,

probing and benchmarking strategy is executed on each node and DTD is created by adding every node as a component to the cluster digital twin.[7]

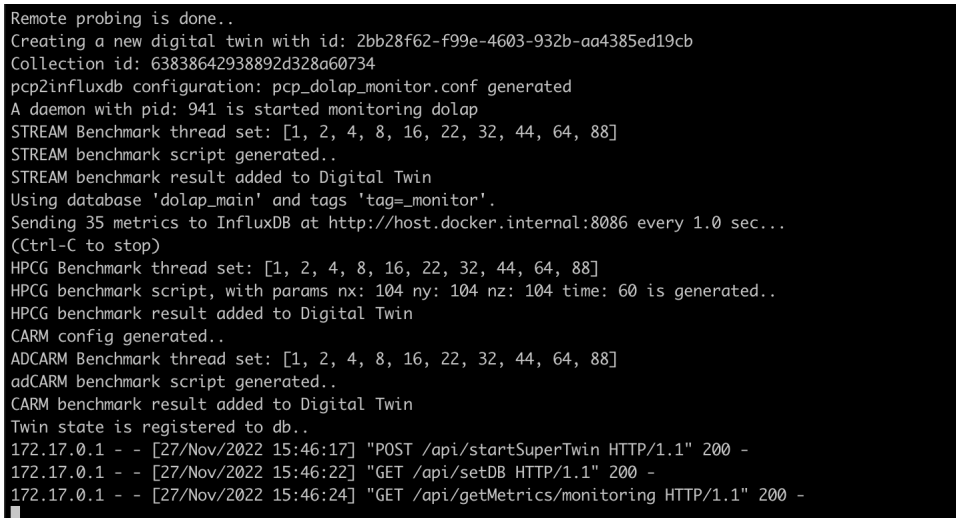

Figure 4: Daemon generation and benchmark execution

Figure 5: Digital twin description (DTD) of Dolap server

Figure 6: core and thread structure of the Dolap as a DTD

Upon creation of the DTD, SuperTwin instance initiates the telemetry data collection with the configured PMCD which gathers and labels the data from PMDAs on the node in accordance with the

DTD. At each heartbeat, which defines the sampling interval, PMCD sends PMDAs' data to the local machine's InfluxDB end-point. Local machine stores the time-series data to provide it to monitoring dashboards and machine learning module. For clusters, SuperTwin instance configures PMCD on the head node and recursively configures PMCD on each compute node. Compute node PMCDs are configured in a way that data that they gathered from PMDAs are directed to the head node's PMCD and head node PMCD sends each nodes' data to the local machine.[8]

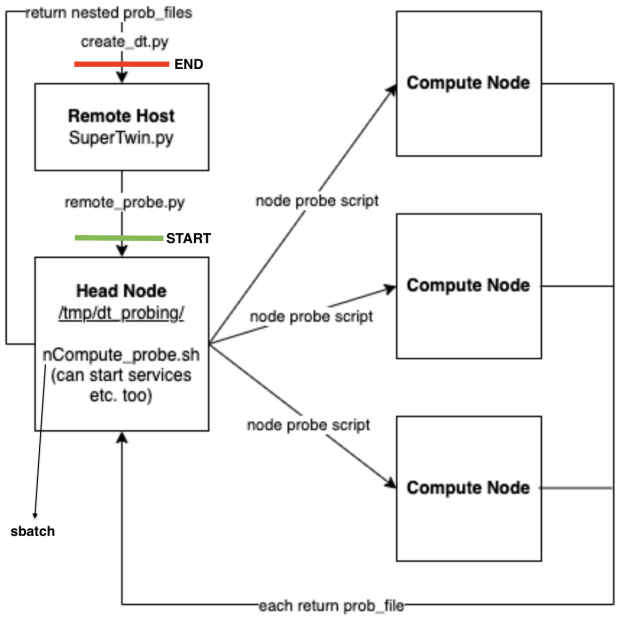

Figure 7: SuperTwin creation for a cluster

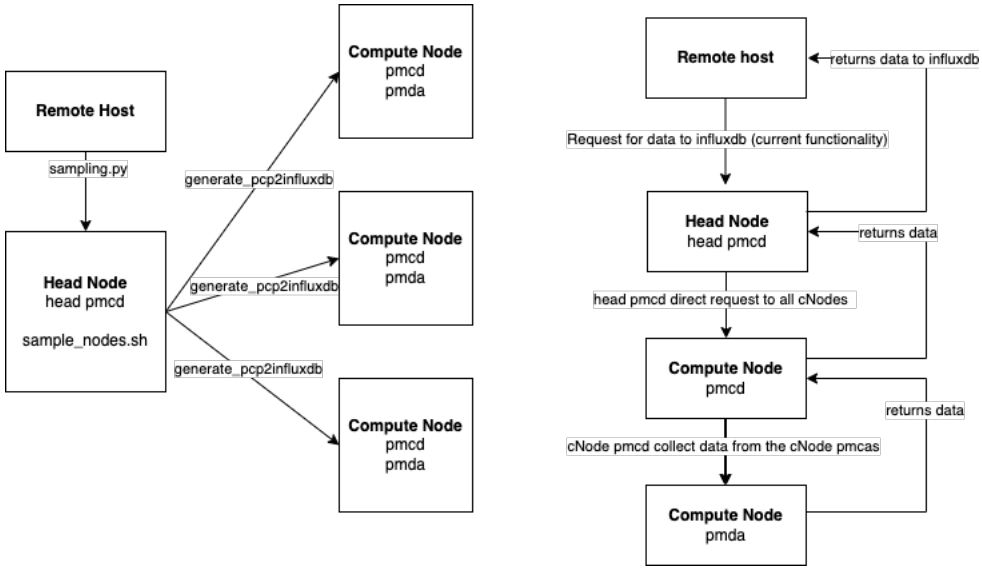

Figure 8: Cluster data collection strategy with PCP

Dashboards that enables monitoring of the target system are created on Grafana whose dashboards are serialized JSON files that can be easily configured according to DTD by SuperTwin instance. By querying the InfluxDB database every 100-1000 milliseconds, Grafana retrieves monitoring data for each defined dashboard to allow them to display the data in real-time.[9][19][20][21][22]

Figure 9: Dolap's monitoring dashboard

3.4 Integration and Deployment

Development process of the tool is a complex process since it requires to implement various modules each of which constantly interact with each other and more than one people is involved in to the process. Moreover, the tool has numerous dependencies that are fundamental for the execution of the tool. To be able to cope with these challenges and make the tool more reliable by regularly testing, a continuous integration and continuous deployment (CI/CD) pipeline is implemented for the project.

A CI/CD pipeline was developed using GitHub Actions, which triggers tests through web hooks when push and merge actions occur.[10]

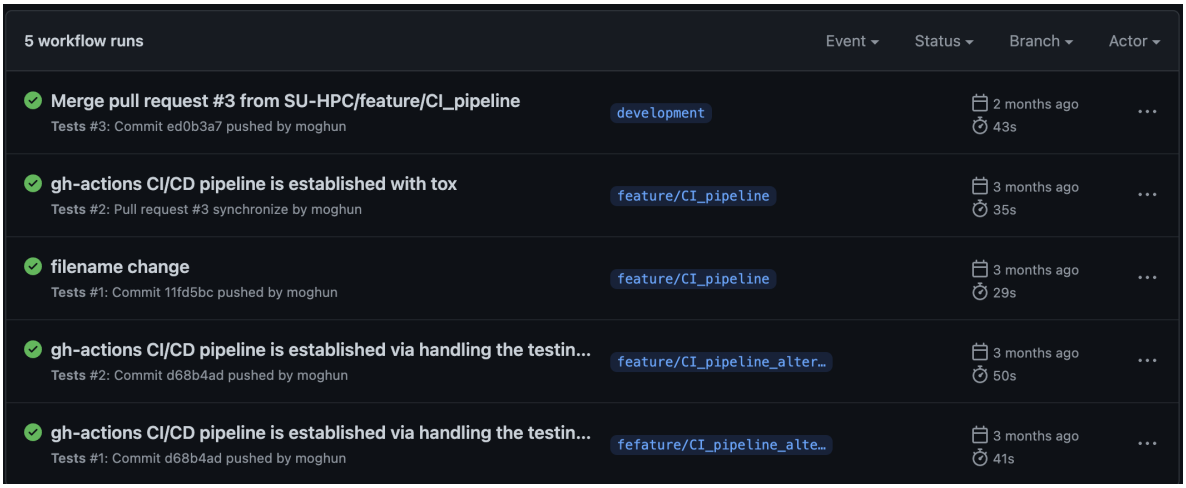

Figure 10: Workflows runs in the GitHub Actions

Another issue encountered during the deployment of the SuperTwin was the dependencies that are only functioning on Linux operating system. The tool is designed to monitor Linux machines (remote machine), but it must be able to run any operating system (OS) to monitor a remote machine. To

achieve this, Docker containers running Linux were developed to host the tool and its native dependencies. Additionally, the back-end of the web application was also containerized.[11]

```
FROM ubuntu:22.04

#Optional - rather can clone repo in the container
COPY ./*/ root/Digital-SuperTwin/

#Run with "docker run -it --privileged --network="host" <image-name>" settings

RUN apt update
RUN apt-get -y install sudo
RUN apt-get -y install python3
RUN apt-get -y install pip
RUN pip install influxdb
RUN pip install pymongo
RUN pip install paramiko
RUN pip install scp
RUN pip install flask
RUN pip install grafanalib
RUN pip install plotly
RUN pip install flask_cors
RUN pip install pandas

RUN apt-get -y install pcp
RUN apt-get -y install ssh
RUN apt-get -y install firewalld
RUN apt-get -y install pcp-export-pcp2influxdb
RUN apt-get -y install vim

EXPOSE 5000 8086 27017 3000

CMD /etc/init.d/pmcd start && /etc/init.d/pmlogger start && /etc/init.d/pmie start && /etc/init.d/pmproxy start /etc/init.d/ssh start && bash
```

Figure 11: Dockerfile that generates container for the back-end.

3.5 Web Application

The tool can be run as a console application however, it is often the case that it extracts more than 1000 monitoring metrics, which can be overwhelming for users to scan and select the desired metrics to monitor. To address this issue, the tool has a web application that is built with React.js and Flask to improve the user experience.

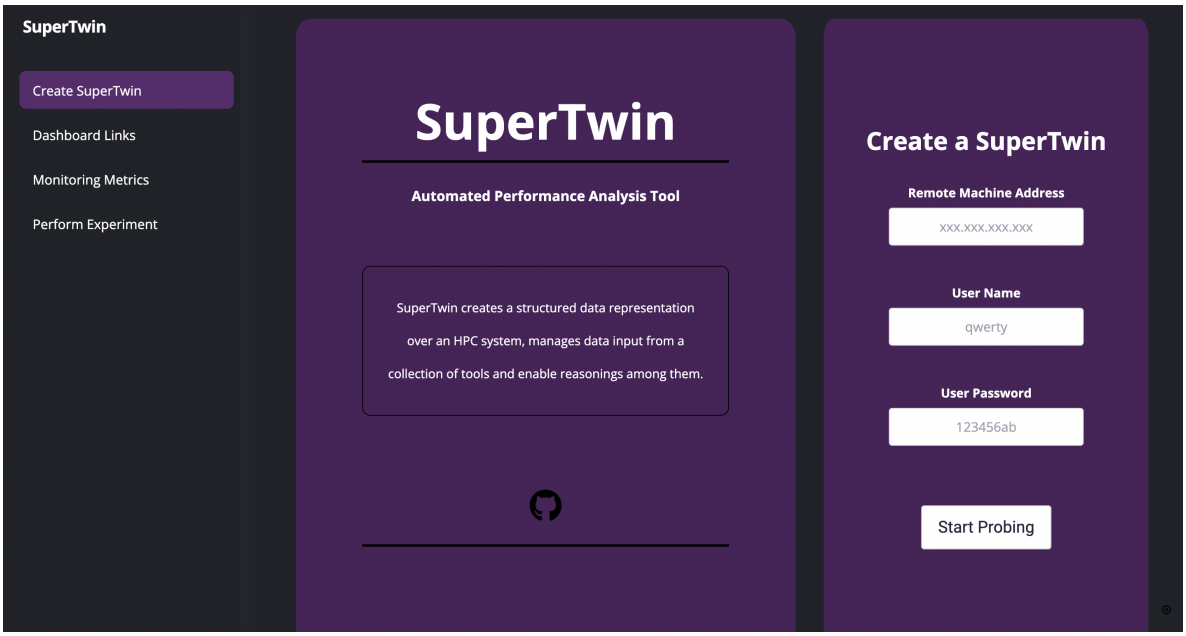

Figure 12: Index page of the web application of the tool

The web application has four pages which are “Create Digital Twin”, “Monitoring Metrics”, “Dashboard Links” and “Perform Experiment”. Create Digital Twin is the index page of the web application which takes required information for SuperTwin instance to establish SSH connection with the remote machine.[12] By clicking the “Start Probing” button, user creates a SuperTwin instance on the back-end via an API call.[13]

```
 * Running on http://172.17.0.2:5000
Press CTRL+C to quit
172.17.0.1 - - [27/Nov/2022 15:45:30] "OPTIONS /api/startSuperTwin HTTP/1.1" 200 -
Remote host name: dolap
Executing command on dolap : sudo rm -r /tmp/dt_probing
Executing command on dolap : mkdir /tmp/dt_probing
Copying probing framework to remote system..
Executing command on dolap : sudo rm -r /tmp/dt_probing/*
Probing framework is copied to remote system..
Executing command on dolap : sudo rm /tmp/dt_probing/pmu_event_query/out.txt
Executing command on dolap : sudo rm /tmp/dt_probing/pmu_event_query/out_emp.txt
Executing command on dolap : make -C /tmp/dt_probing/pmu_event_query
Executing command on dolap : sudo /tmp/dt_probing/pmu_event_query/./showevtinfo -L -D &>> /tmp/dt_probing/pmu_event_query/out.txt
Executing command on dolap : sudo /tmp/dt_probing/pmu_event_query/./showevtinfo &>> /tmp/dt_probing/pmu_event_query/out_emp.txt
Executing command on dolap : sudo python3 /tmp/dt_probing/system_query/probe.py
```

Figure 13: Digital twin instantiation via using the web application

After probing and benchmarks are complete, user is directed to the “Monitoring Metrics” page for selecting the metrics that would be used to monitor the system. In this page, possible monitoring metrics are shown in a table which allows sorting, filtering and grouping of entries. Also, pre-defined “recommended” metrics sets to monitor specific aspects of the system are presented to user. Once the user selects the metrics that she wanted to monitor, clicking to “Add Metrics” button adds selected metrics to the monitoring metrics list.[14] The SuperTwin instance generates Grafana dashboards for the selected monitoring metrics on the back-end and directs the user to the “Dashboard Links” page upon completion.[15]

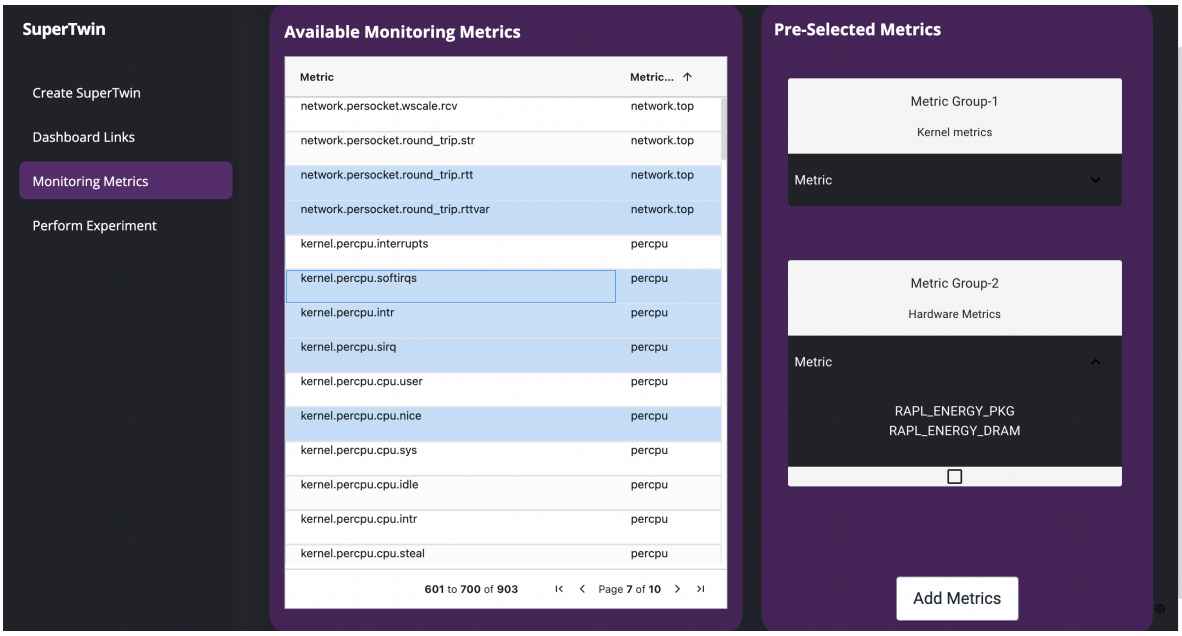

Figure 14: Monitoring metric selection page

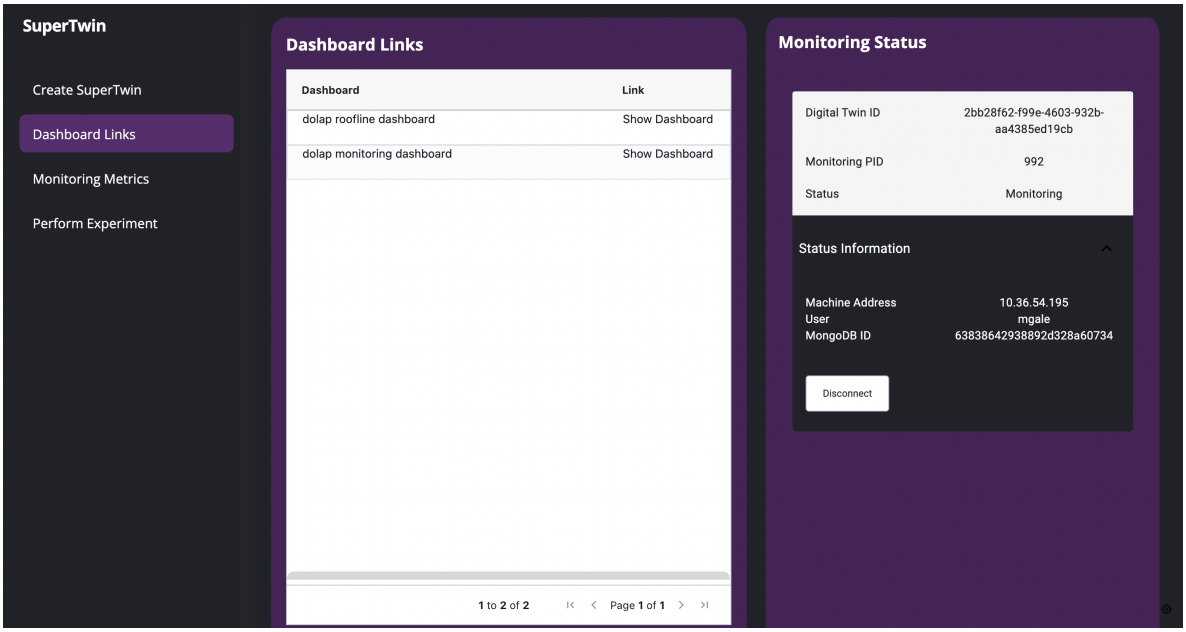

Figure 15: Dashboard links and monitoring status page

Perform Experiment page allows users to use SuperTwin instance's executing commands on the remote machine feature. With this way, a user can setup and run experiments to monitor different behaviours of the machine on different conditions. Directory that the experiment will take place, affinity, and the commands to be executed to perform the experiment are taken as input. User can also add new monitoring metrics to gather during an experiment.[16]

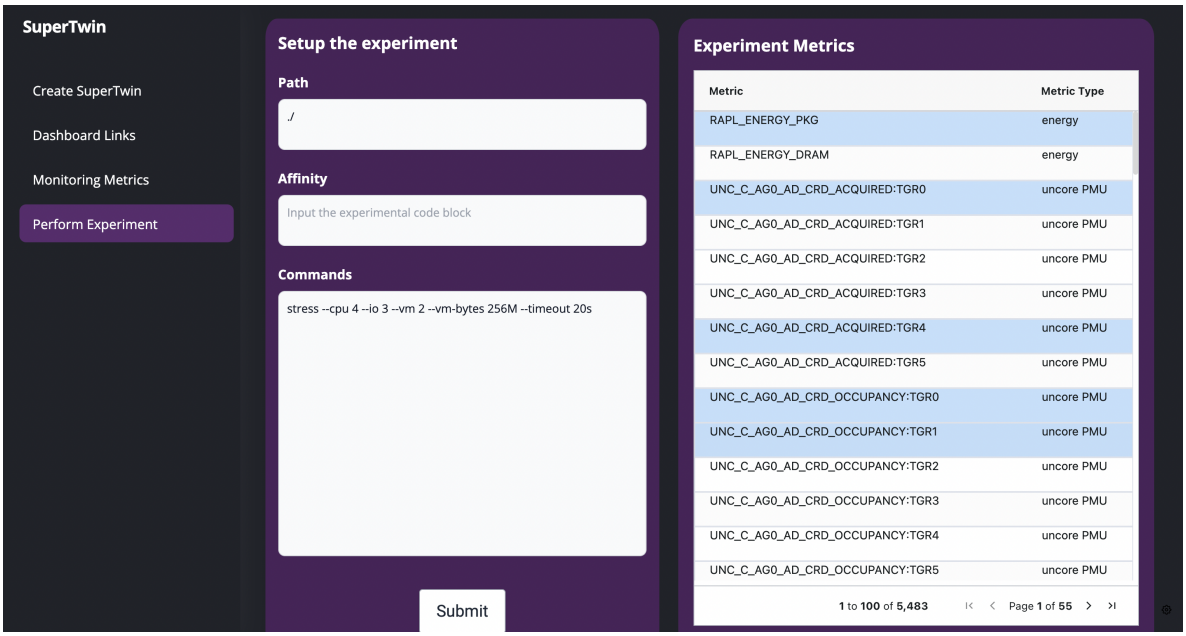

Figure 16: Perform experiment page

4 Results & Discussion

The initial objectives of the project contain four different parts. Back-end system development, called "Probing, Benchmarks and Data Gathering" module, to gather all hardware metrics related to HPC overall performance, frontend system development, called" Visualization module," to visualize collected data in the user interface, a machine learning pipeline, called Learning and Reasoning module to detect anomalies, and the digital twin to construct a digital representation of a supercomputer. During the project, initial objectives are followed, and the backend system, frontend system, and digital twin representation of a system are developed. A backend system and digital twin creation are highly related and designed accordingly. Probing and benchmarks are used to create the digital twin representation of the system that is going to be monitored. Since those different parts depend on each other, development started with probing and benchmarks, continued with data gathering, and lastly, visualization. The aim was to develop those four parts iteratively. However, the data gathering module is extended with two essential metrics: Reuse Tracker and ComDetective. Before the last part, the development of the learning and reasoning part is not started.

Learning and Reasoning come with various challenges, and before this part, the data-gathering module should be finished. That's why our team focused on Reuse Tracker, ComDetective, and cluster data extraction. Additionally, removing HPC Toolkit dependency comes with serious challenges. First of all, it is built in purpose to make it easy to develop monitoring projects like Reuse Tracker and CommDetective. Removing these two submodules depends on various low-level APIs, which should be developed ourselves. In addition, creating these tools requires knowledge of low-level Linux modules, performance monitoring units, and debug registers. This process slowed down the development and delayed the development of the Learning and Reasoning module.

Besides the learning and reasoning module, the project is completed. The current version of SuperTwin provides a user-friendly interface to create digital twins of the supercomputers, gather significant metrics from the system, store data in the backend using InfluxDB and Grafana, and visualize data in the user interface.

Supercomputer monitoring and optimization is a significant problem, and there are several studies. One of them can be Application Heartbeats [HES+10]. It presents a technique for monitoring the performance of autonomous computing environments using regular data collection intervals called "heartbeats." This data can be used to improve performance through machine learning and manual intervention. Low-level data, such as counting executed instructions or looking at CPU utilization, may not provide accurate information about an application's performance.

Another study that can be stated is PIKA [DWKN20]. It is similar to SuperTwin in that it collects and visualizes runtime metrics, but only a few select metrics are chosen. PIKA uses InfluxDB databases to store temporary data, the NVIDIA management library to acquire GPU metrics, the collected daemon to collect runtime data, and LIKWID to capture IPC, FLOPS, main memory bandwidth, and power consumption. Scheduling is handled by SLURM, and the SPANK plug-in architecture may be used for this purpose. Data is written to InfluxDB using HTTP. The main difference between PIKA and the SuperTwin is the number of metrics analyzed and the fact that the SuperTwin is a digital twin monitoring system.

Besides highlighted studies, SuperTwin gathers more of data about the system with low overhead. Additionally, it creates a Digital Twin representation of the system and allows for future simulations and predictions. It will forecast system health and optimize the supercomputer with learning and reasoning modules in the future. SuperTwin is a supercomputer monitoring tool that uses state-of-the-art methods and extends currently available monitoring tools with a digital twin approach. It is a good use-case for digital twin applications on supercomputers and real-time monitoring.

5 Impact

One of the important aspects of Supercomputer monitoring is gathering energy consumption intel from the system. Considering that greenhouse gas emissions from the energy consumption of Information and Communication Technologies could account for 23% of total emissions in 2030, optimizing the energy consumption of computing systems that consumes massive amount of energy such as clusters would have an utmost priority. [PEGL+22] In this sense, SuperTwin can be used to monitor and improve energy consumption of clusters.

As mentioned before, besides monitoring the a Supercomputer, SuperTwin aims also detect bottlenecks and simulate what if hardware scenarios. With achieving these goals, SuperTwin allows vendors to use their machines at their peak performance by resolving bottlenecks. Moreover, vendors can enhance their systems by simulating effects of hardware modifications on the system in terms of performance and energy consumption. In this sense, SuperTwin may help vendors to increase their revenues by using their systems more productively and not wasting money to unnecessary hardware modifications. Note that, since SuperTwin is an open-source tool, vendors can freely benefit from it.

6 Ethical Issues

The project was planned to be developed in a way that it can be used in any supercomputer in the world. It is open-source licensed and the findings were transparently released. It is a non-profit software, and no patent-protected designs and software are used. During this project, other open-source projects/concepts were used or custom software was developed. In sum, there are not any ethical issues concerned.

7 Project Management

The first step of the project was to research literature on the supercomputer performance monitoring approaches. Many papers were investigated and discussed to ensure the validity of and enhance the pillars of the digital twin approach. The research topic included state-of-the-art performance collection techniques, the suitability of different database architectures, and visualization tools. One of the most important points of focus was finding a way to collect the metrics with creating minimal overhead on the target system. As the project does not have a concrete finish criterion, the research on metric acquisition is an ongoing process.

Second step was to create the digital twin. To this end, a digital twin creation module using DTDL was developed. It was decided that a probing approach that gathers the structural information of the target system was the most suitable. By probing the remote target system through the local system's terminal, a digital twin was to be created. The created digital twin had to have communication capabilities for real-time monitoring and intervention for optimization.

The collected metrics needed to be stored. Various databases were to be investigated for this end. The decided database was to be constructed for the twin. The database was to later be used to access the collected metrics for visualization and ML.

As the metric acquisition is an ongoing process. It was decided that as new approaches were discovered, they were to be discussed and integrated into the twin. To this end, built-in Linux performance monitoring capabilities, CPU/GPU manufacturer-specific system calls, and tools such as PCP, ComDetective, and Reuse Tracker were some of the tools that decided to be used or integrated. The collected data would be sent to the local host and database for further processing. Integration of ComDetective and Reuse Tracker turned out to be challenging as these tools were built on the HPC-toolkit which was not an appropriate tool for the twin as it introduced too many overheads. Instead, these tools were reconstructed for the twin using existing capabilities with the help of the team that built these tools.

The next step after collecting metrics was the user interface development. The interface would be used to offer ease of use. With the interface, a user could monitor different supercomputer nodes, gather specific metrics, display selected metrics in a suitable visual presentation, export the metrics to built-in or stand-alone ML applications, and control the twin.

The project has been successful with all the steps except the Machine Learning module. The hardest part of the project was controlling the code complexity. As more and more capabilities were added to the project, the different tools would collide with each other, and it would be harder to debug. Modularization and thorough investigation of existing code would take place to deal with this issue. Also, during the implementation, it was realized that operating system compatibility would be an issue for the tool. The tool had to be interactable using different operating systems so, Docker containerization was researched and integrated. The tool was meant to scale to computer clusters, but as of now, only one node(computer) can be monitored.

8 Conclusion and Future Work

The detection of bottlenecks in high-performance computing systems is a challenging problem that has garnered significant attention from researchers and practitioners in recent years. One promising approach to addressing this issue is using digital twins, virtual representations of physical systems that can be used to simulate and analyze their behavior.

One key aspect of this approach is the collection of data from the physical system, which is necessary to create an accurate digital twin. There has been a great effort in recent years to develop low-overhead methods for data acquisition. On top of the current state-of-the-art, we developed a SuperTwin monitoring tool that creates virtual representations of high-performance computer clusters and monitors running applications in the computer. With SuperTwin, which is easily extendible to monitor new metrics, it will be possible to orchestrate clusters and optimize them to increase the efficiency of those computers. There is still much work to be done in the area of bottleneck detection in HPCs. This includes implementing more advanced data collection techniques and developing better visualization tools for analyzing and interpreting the collected data.

In addition to improvements in the current system, it is also important to integrate machine learning modules to facilitate the automated identification of bottlenecks and other issues. Machine learning has the potential to revolutionize the way we detect bottlenecks in high-performance computing (HPC) clusters. Traditional methods for identifying bottlenecks rely on manual analysis of system logs and other forms of data, which can be time-consuming and prone to errors. In contrast, machine learning algorithms can automatically analyze large volumes of data and identify patterns that may indicate the presence of a bottleneck. In addition to detecting bottlenecks, machine learning can also be used to predict future performance issues and recommend actions to prevent them. This can be particularly useful in large-scale HPC environments, where it is hard to monitor every aspect of system performance and detect anomalies.

Overall, the digital twin approach holds great promise for addressing the challenges of bottleneck detection in HPCs, and it is likely to continue to be a significant focus of research and development in the coming years.

9 Appendix

```
########## Performance Summary (times in sec) ##########=
Benchmark Time Summary=
Benchmark Time Summary::Optimization phase=1.36904e-07
Benchmark Time Summary::DDOT=0.00185527
Benchmark Time Summary::WAXPBY=0.00205307
Benchmark Time Summary::SpMV=0.00207577
Benchmark Time Summary::MG=0.0422289
Benchmark Time Summary::Total=0.0482362
Floating Point Operations Summary=
Floating Point Operations Summary::Raw DDOT=1.26157e+06
Floating Point Operations Summary::Raw WAXPBY=1.26157e+06
Floating Point Operations Summary::Raw SpMV=1.01229e+07
Floating Point Operations Summary::Raw MG=5.55949e+07
Floating Point Operations Summary::Total=6.8241e+07
Floating Point Operations Summary::Total with convergence overhead=6.69029e+07
GB/s Summary=
GB/s Summary::Raw Read B/W=8.74691
GB/s Summary::Raw Write B/W=2.02321
GB/s Summary::Raw Total B/W=10.7701
GB/s Summary::Total with convergence and optimization phase overhead=10.3628
GFLOP/s Summary=
GFLOP/s Summary::Raw DDOT=0.67999
GFLOP/s Summary::Raw WAXPBY=0.61448
GFLOP/s Summary::Raw SpMV=4.87671
GFLOP/s Summary::Raw MG=1.31651
GFLOP/s Summary::Raw Total=1.41472
GFLOP/s Summary::Total with convergence overhead=1.38699
GFLOP/s Summary::Total with convergence and optimization phase overhead=1.36122
User Optimization Overheads=
User Optimization Overheads::Optimization phase time (sec)=1.36904e-07
User Optimization Overheads::Optimization phase time vs reference SpMV+MG time=4.70998e-05
Final Summary=
Final Summary::HPCG result is VALID with a GFLOP/s rating of=1.36122
Final Summary::HPCG 2.4 rating for historical reasons is=1.38698
Final Summary::Reference version of ComputeDotProduct used=Performance results are most likely suboptimal
Final Summary::Reference version of ComputeSPMV used=Performance results are most likely suboptimal
Final Summary::Reference version of ComputeMG used and number of threads greater than 1=Performance results are severely suboptimal
Final Summary::Reference version of ComputeWAXPBY used=Performance results are most likely suboptimal
Final Summary::Results are valid but execution time (sec) is=0.0482362
```

Figure 17: Dolap's HPCG benchmark results

```
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
Total memory required = 228.9 MiB (= 0.2 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Number of Threads requested = 88
Number of Threads counted = 88
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 2027 microseconds.
   (= 2027 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           60491.1     0.002774     0.002645     0.003209
Scale:          55229.1     0.003174     0.002897     0.003927
Add:            57915.7     0.004262     0.004144     0.004661
Triad:          58836.5     0.004156     0.004079     0.004559
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

Figure 18: Dolap's STREAM benchmark results

```
{
"id": 1
"panels":
  [{"id": 1,
    "targets":
      [{"datasource":
       {"type": "influxdb",
        "uid":  "UUkm1881"},
        "measurement": "perfevent_hwcounters_
         FP_ARITH_SCALAR_SINGLE_value",
        "params": "_cpu0"}]}]
"time":
  {"from": "now-5m",
   "to": "now"}
}
```

Figure 19: JSON representation of a Grafana dashboard

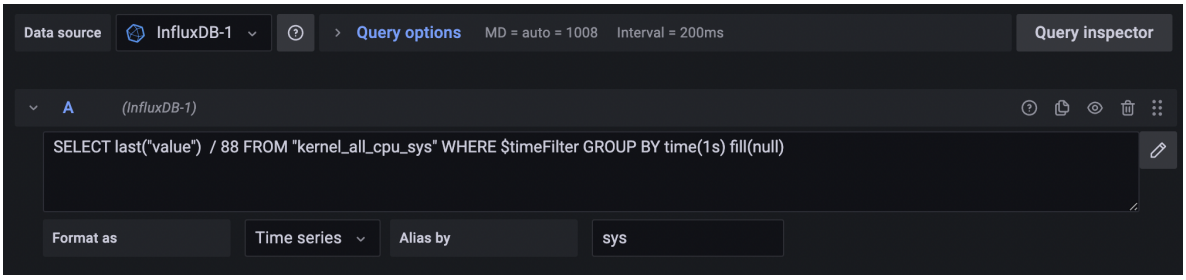

Figure 20: A query to retrieve information for a Grafana dashboard from InfluxDB

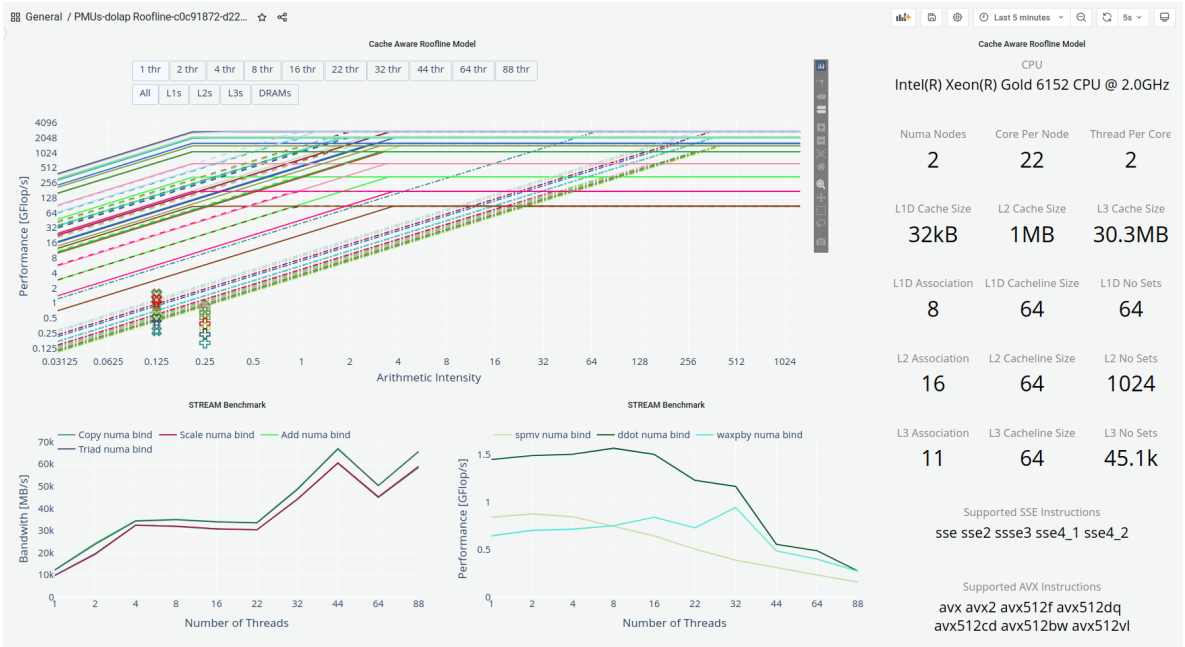

Figure 21: Dolap's roofline dashboard

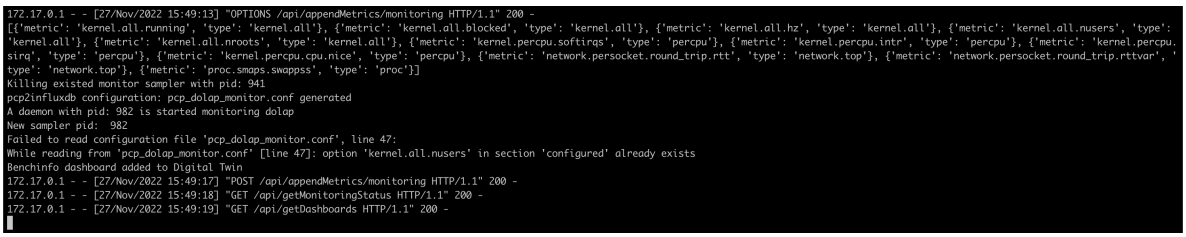

Figure 22: Dashboard generation on the web application

References

[AFK+08] Laksono Adhianto, Michael Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan Tallent. Hpctoolkit: Performance measurement and analysis for supercomputers with node-level parallelism. In *Workshop on Node Level Parallelism for Large Scale Supercomputers*, 2008.

[BH05] Erik Berg and Erik Hagersten. Fast data-locality profiling of native execution. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, page 169–180, New York, NY, USA, 2005. Association for Computing Machinery.

[DWKN20] Robert Dietrich, Frank Winkler, Andreas Knüpfer, and Wolfgang Nagel. Pika: Center-wide and job-aware cluster monitoring. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 424–432, 2020.

[GFZ12] Zhenhua Guo, Geoffrey Fox, and Mo Zhou. Investigation of data locality in mapreduce. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 419–426. IEEE, 2012.

[HES+10] Henry Hoffmann, Jonathan M. Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *ICAC '10*, 2010.

[PCP] Programming Performance Co-Pilot process structure for distributed operation. https://pcp.readthedocs.io/en/5.2.3/PG/ProgrammingPcp.html. Accessed: 2022-11-27.

[PEGL+22] Beatriz Prieto, Juan José Escobar, Juan Carlos Gómez-López, Antonio F. Díaz, and Thomas Lampert. Energy efficiency of personal computers: A comparative analysis. *Sustainability*, 14(19), 2022.

[SCAU19] Muhammad Aditya Sasongko, Milind Chabbi, Palwisha Akhtar, and Didem Unat. Comdetective: A lightweight communication detection tool for threads. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[SCMU21] Muhammad Aditya Sasongko, Milind Chabbi, Mandana Bagheri Marzijarani, and Didem Unat. Reusetracker: Fast yet accurate multicore reuse distance analyzer. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(1):1–25, 2021.

[WLC19] Qingsen Wang, Xu Liu, and Milind Chabbi. Featherlight reuse-distance measurement. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 440–453, 2019.