SUPERTWIN: DIGITAL TWINS FOR HIGH-PERFORMANCE COMPUTING CLUSTERS

by
FATİH TAŞYARAN

**Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of the requirements for the degree of
Master of Science**

**Sabancı University
December 2022**

ABSTRACT

SUPERTWIN: DIGITAL TWINS FOR HIGH-PERFORMANCE COMPUTING CLUSTERS

FATİH TAŞYARAN

Computer Science MSc. THESIS, DECEMBER 2022

Thesis Supervisor: Assoc. Prof. Kamer KAYA

Keywords: High Performance Computing, Digital Twin, Performance Analysis, Visualization

Computational systems are extremely complex and the composition of their hardware and software components greatly vary from machine to machine. This non-standardized environment can cause up to 100% difference between the best and worst completion times with the same input data. On top of that, the shape of the input data and executed kernels add even more variance to the situation. However, computational systems are not completely hostile environments. These systems are also equipped with diverse observability capabilities. A typical Linux system can report thousands of real-time execution and performance-related metrics from both its hardware and software components.

Digital Twins are knowledge management systems that have vast application areas in the industry, however, digital twins of computational systems remain a gap in the literature. SuperTwin is a knowledge representation generator and manager of the tools and performance data that interact with it. It creates a digital twin of a computational system via detailed probing, configures and listens to performance metric samplers, creates real-time visualizations, links the acquired information, and enables semantic queries for advanced analysis.

In this work, design and implementation choices for SuperTwin are thoroughly presented. The effect of profiling on remote systems is analyzed and the accuracy of the readings is investigated.

ÖZET

SUPERTWIN: YÜKSEK PERFORMANSLI HESAPLAMA KÜMELERİ İÇİN DİJİTAL İKİZ

FATİH TAŞYARAN

BİLGİSAYAR MÜHENDİSLİĞİ YÜKSEK LİSANS TEZİ, ARALIK 2022

Tez Danışmanı: Doç. Dr. Kamer KAYA

Anahtar Kelimeler: Yüksek Performanslı Hesaplama, Dijital İkiz, Performans Analizi, Görselleştirme

Hesaplama sistemleri çok kompleks ve çeşitli donanım ve yazılım bileşenlerinin bir araya gelmesiyle oluşur. Bu standart olmayan ortam, aynı girdi verisiyle en iyi ve en kötü tamamlama süreleri arasında %100'e kadar farka neden olabilir. Bunun yanı sıra, girdi verisi şekli ve çalıştırılan algoritmalar da bu eşitsizlikteki varyansı daha da artırır. Ancak, bu zorlukların yanısıra, söz konusu sistemler aynı zamanda çeşitli gözlemlenme yeteneklerine de sahiptir. Tipik bir Linux sistemi, hem donanımından hem de yazılım bileşenlerinden binlerce gerçek zamanlı çalışma ve performansla ilgili metrik raporlayabilir.

Dijital İkizler, endüstriyel uygulamalarda geniş uygulama alanları bulmasına rağmen, literatürde süper bilgisayar bağlamında araştırılmamış bir konsept olarak durmaktadır. SuperTwin, çeşitli hesaplama ortamı ve performans verisi araçlarını otomatikleştirerek kullanan, bu araçların ve performans verilerinin bilgi temsil oluşturucusu ve yöneticisidir. SuperTwin, detaylı bir taramayla bir hesaplama sisteminin dijital ikizini oluşturup, performans metrik örnekleyicilerini yapılandırma ve dinleme, edinilen farklı tiplerdeki bilgiyi bağlantılı hale getirme kabiliyeti ile gerçek zamanlı görselleştirme ileri analiz için anlamlı sorgulara izin verir.

Bu çalışmada, SuperTwin tasarım ve gerçeklemesi ayrıntılı olarak sunulmuş, performans ölçümü etkisinin hesaplama sistemlerine etkisi araştırılmış ve performans incelemelerinin tutarlılığı incelenmiştir.

ACKNOWLEDGEMENTS

I want to thank my supervisor Prof. Kamer Kaya for all his trust and support during the years we worked together. I was a mediocre student when he accepted me into one of his research projects five years ago. During these years, he always confronted me with understanding and sympathy, and I learned so much from him in both computer science and humanity. None of my dreams about being a researcher come true without his support and trust. I will remain grateful to him throughout my life and always remain his student regardless of how old I become.

Many thanks to Deren, who understood and supported me through many years. Her companion turned the toughest challenges into a peaceful walk in the garden.

I want to thank Şeyma, Anes, and Uğur for the beautiful and cheerful working life they bring.

I also want to thank cats, I have always felt like they are trying to protect me from the grotesque tortures of this universe with their tiny paws.

Finally, I want to thank EuroHPC and TUBITAK for their support:
This research has received funding from the European High-Performance Computing Joint Undertaking under grant agreement No 956213 and The Turkish Science and Technology Research Centre under grant agreement No 220N254.

For my beloved mother & brother

LIST OF ABBREVIATIONS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. Introduction

Today's computing systems consist of overwhelmingly many components whose composition varies in every machine. However, the complexity of these systems is not limited only to this variation. There are many other factors contributing cumulatively to this complexity. The architectures of processor units and capabilities of memory units change with every generation, communication capabilities of these two components are also constantly changing as standards are added each year. Moreover, since servers used for high-performance computation serve multiple users, there may arise problems such as resource contention, network congestion, disk congestion, memory congestion, etc. The type of applications that run on these machines and the type of their inputs also add to this complexity since they may cause significant changes in performance and efficiency. All this variation is caused naturally due to constantly developing technology and evolving algorithms creating the need for fine modelings of these cyber-physical systems using tools such as monitoring, profiling, and forecasting of events that affect application performance in order to optimize it.

Although there are many performance metric sources in computers, such as kernel stats and performance monitoring units (PMUs) which count the number of occurrences of events that took place in different sub-components of the CPUs and the memory subsystem, the implementation of these counters also changes from architecture to architecture. Input shape also changes with respect to the type of application and does not always fit into the system. For example, sparse data, in nature, is not very efficient to run on Von Neumann architectures since memory accesses are done in batches and data points to be used consecutively do not always appear in fetched batches of memory, therefore, causing many redundant data transfers. Although during the development and optimization of frameworks, profiling tools that leverage kernel replays and source code instrumentation are widely used and become handy, it is not always possible to run the code with them since they come with significant overhead. Moreover, they are not meant to work with product releases, in which programs may prune to performance variations caused by several conditions, such as orphan processes, firmware bugs, memory leaks, CPU

throttling, reduced frequency, shared resource contention, and network congestion. These factors can cause up to 100% difference between the best and worst completion times with the same input data (Aksar, 2021). Thus, a need for innovative monitoring tools that can monitor every single component of an HPC system, from kernel statistics to physical hardware performance counters to component features, arises. A digital twin could also be used to create a structural representation of the aforementioned sources of complexity and efficiently model this complex problem.

A Digital Twin is a structured collection of information emitted from a real-world system. Digital twins capture structural and sensory data from their physical counterparts and allow for in-depth historical analysis, real-time monitoring, simulation, and prediction of the entity it models. Albeit there are several digital twin ontologies in the literature for modeling industrial machines (Steinmetz, 2018), cities (Deng, 2021) (Shahat, 2021), smart buildings (LuViVi, 2019) and even earth (Huang, 2022), up to our knowledge, there is little to no work on ontologies to describe computers as cyber-physical systems. A closely related effort on the problem is DTDL. DTDL is developed by Microsoft for IoT frameworks, therefore, provides a more suitable basis for describing computers. In this work, we altered DTDL (Digital Twin Definition Language) to represent computers mainly used for computation purposes and their telemetry sources, as well as their computations, and built a digital twin management system around it. SuperTwin aims to mediate analyzing of the aforementioned complex process via automatically discovering component information, automatically configuring monitoring agents, and providing on-the-fly analysis and monitors for systems and computations run on them while keeping its memory footprint and computational overhead as low as possible.

There are two main domains for performance metrics in common computational systems, hardware-induced, and software-induced metrics. They are generally handled separately in current works, however, they may have implications for each other. For example, a hardware performance counter reads L1 cache bandwidth nonetheless, it is unaware of anything that goes wrong in the system, such as resource contention, or thermal throttle. Our SuperTwin is a digital copy of the "whole system", therefore it contains both software and hardware-related information and past job history. Another important property of the proposed method is; designed digital twins being interactional with each other. Since the metamodel interface defines the shape of the underlying data and how it should be processed, a metamodel class from one digital twin could be co-processed with another twin's class (or this class's content.)

The presentations in this thesis are mostly about design, preliminary results, tests, and verifications of SuperTwin and performance metric readings.

2. Background

2.1 Digital Twins

Digital Twins are used to provide a digital projection of physical systems via describing data-emitting sources and relationships between physical systems components. Collections of methodologies that are used to describe these pieces of information are called ontologies. Some of the well-known ontologies are; SOSA (Sensor, Observation, Sample, Actuator)(Janowicz, 2019), which is used to describe industrial pipelines, and FOAF (Friend of a Friend), which is used to describe relations among people. Moreover, there are several vocabularies used to describe digital twins, such as RDF (Resource Description Framework) and OWL (Web Ontology Language). Ontologies using these vocabularies allow static information to be located and queried using web interfaces via SPARQL endpoints. These frameworks are widely used to represent web-based interactions. Digital twins of HPC systems or computers, in general, have to differ from other physical entity twins due to several reasons; There are (usually) much more sensors for each subdomain of the physical system, each sensor can report up to thousands of metrics, and metrics from the same sensor can be variable even for the same system, and when counting processes as a component of the system, even components change rapidly.

2.1.1 JSON-LD

Linked data is used to generate a network of discrete and distinct data in order to enable semantic queries and complex analysis over different domains of data sources and interoperability. Linked data is widely adopted in web technologies and also used

in different branches of science for knowledge management, such as biology (Xin, 2018) and physics (Chen, 2016).

RDF is a standard for data exchange graph data on the web. In the context of RDF data, an edge is referred to as a triple and consists of a source node (called a subject), an edge name (called a predicate), and a target node (called an object). An RDF graph is defined as a set of RDF triples that follow this structure. On top of this structure, RDFs have identifiers, called IRIs to unambiguously identify and properties to describe the nodes. JSON-LD is a notation used to express RDF data using JSON syntax. This means a JSON-LD document is both a JSON document and an RDF document.

JSON-LD has "ld attributes", on par with RDF, which separates JSON-LD from ordinary JSON. Most relevant of these attribures are; `@context`, `@id` and `type`. With these attributes, we know what a JSON-LD dictionary describes, what kind of datatypes it includes, and how to parse and process them. This is an important aspect since, in turn, it allows create bigger and interconnected systems from building blocks. Albeit very useful in creating knowledge graphs, these ontologies are designed and used to keep static metadata. To add new data points, new triples need to be injected into the graph, which makes these types of ontologies impractical with the use of time-series data without modification (Friedemann, 2019).

2.1.2 DTDL

DTDL is a derivation of RDF, which is made up of six metamodel classes that describe the context of digital twin components. These classes are; `Interfaces, Telemetry, Properties, Commands, Relationships` and `Data Types`. In DTDL, every `Interface` is a digital twin on its own, with its contents describing its `Properties`, `Telemetry`, and `Relationships`. When combined, these enable to capture of the hierarchical structure of a computer and model of every single component (e.g., CPU, GPU, memory subsystem, etc.) as a separate digital twin entity. The idea that every interface is considered a digital twin on its own is heavily exploited in SuperTwin.

Property	*Description*
@type	**Interface**
@id	Unique identifier within digital twin for interface
contents	a set of Interface, 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 2.1 A subset of the DTDL classes and their properties. These particular classes are altered and used in SuperTwin.

2.2 Observability of Computing Systems

2.2.1 Linux Observability

An operating system is a software program that manages a computer's hardware and software resources and coordinates their interactions. It acts as an intermediary between the computer's hardware and the programs that run on it, ensuring that everything runs smoothly and efficiently. The operating system is responsible for managing the hardware and software resources of the computer, including the processor, memory, storage devices, and input/output devices, alongside task management. Therefore, metrics emitted from the operating system are required to ensure that applications run optimally without anomalies. Similarly to hardware, Linux operating system also consists of several components. These components are equipped with thousands of tracepoints, and logs which could be used to generate statistics for each component. A brief breakdown of these components and some of the numerous tools to observe their states are given in 2.1

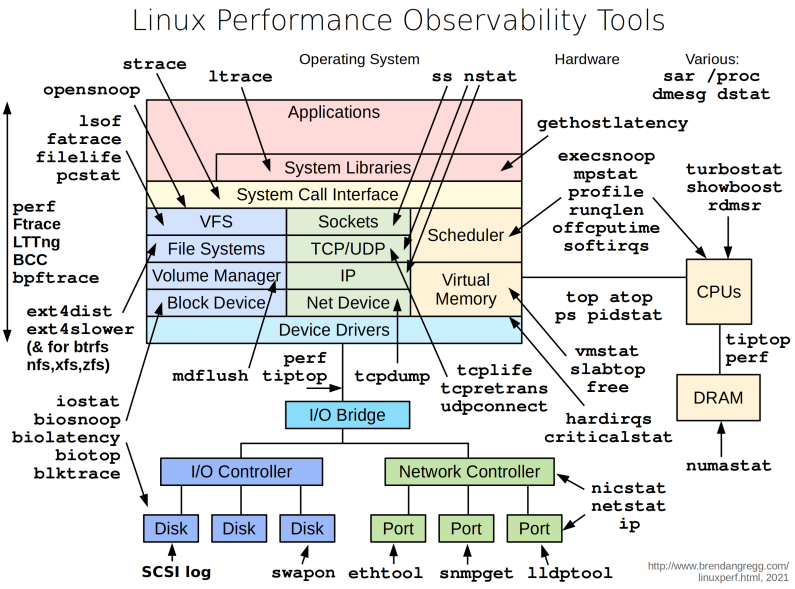

Figure 2.1 Components of the Linux operating system and some of the tools widely used to monitor telemetry arose from these components. (Gregg, 2021)

2.2.2 Hardware Performance Counters

PMUs are programmable interfaces on CPUs that contain built-in special-purpose registers called hardware performance counters that are physical entities on the hardware. Hardware performance counters could be configured to count or sample certain events that took place during the execution of the programs. Linux natively supports PMU inference with `perf` interface, which has been included in the kernel since version 2.6.31. Although there are PMU implementations in almost any processor, their capabilities, events, and event codes change with every new architecture. Therefore they are implemented as MSRs. Even though most of the MSRs on x86 architectures report substantially intersecting sets of events and a small amount of architecture-specific events, they are no generic configurations that will work on different architectures. Apart from MSRs, they are also RAPL PMUs that report components' energy consumption such as CPUs, sockets, and DRAMs. `libpfm4` is a widely available library that can detect PMUs and their events on virtually every architecture, including ARM. Most tools that provide an interface over hardware performance counters rely on libpfm4 to resolve programming parameters and available events. Some of the important software and hardware metrics that correlate highly with execution performance is given in Table 2.2

Source	Source Description	Metric	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 2.2 Kernel performance metrics from /proc and hardware counter metrics that used model sparse computation performance.

2.2.3 Benchmarks & Performance Models

Benchmarks are used to systematically measure the capabilities of a computer system. Since SuperTwin provides an in-depth analysis of computer systems and their applications, benchmarks are crucial information to have in order to have a baseline and insight. Although computer programs could be huge in variety and number of operations they perform to do a calculation, they generally consist of repeating sub-routines which could be classified with respect to the ratio and type of their memory and floating point operations. A class of benchmarks called micro-benchmarks are especially useful for measuring the performance of these sub-routines. Since these

Algorithm 1 STREAM Triad

```
#pragma parallel for
for (i =0; i<N; i++) {
    a[i] = b[i] + c[i] * SCALAR;
}
```

sub-routines are encountered very frequently in scientific computations, they become very useful in measuring system capabilities. Micro-benchmarks are generally written with assembly to refine control over the executed stream of instructions as much as possible. A good example of micro-benchmarks is infamous STREAM triad which is most relevant to highly complex HPC workloads(Intel, 2022; McCalpin, 2007).

These benchmarks are also could be also to build performance models since they have a constant amount of memory and floating point operations. For example, in Algorithm 1, for each iteration of the loop, there are 2 floating point operations, and $8 \times 3 = 24$ bytes are transferred if double precision is used. These values are further augmented into a metric called Arithmetic Intensity (`AI`) which is the ratio of executed floating point operations to transferred. AI for Algorithm 1 is for example $2/24 = 0.083$. Together with the peak number of floating point operations, a system could achieve and maximum memory bandwidth, `AI` value could be used to determine if a kernel's performance is bounded by memory operations or floating point operations. This method is called the Roofline Model (Williams, 2009). For a simple DRAM memory model, Roofline could be calculated as

$$\text{GFLOP/s} = \min \begin{cases} \text{Peak GFLOP/s} \\ \text{Al} * \text{Peak GB/s} \end{cases}$$

peak floating point could be theoretical and calculated with $Peak\ theoretical\ performance = no\ processors \times cores\ per\ processor \times clock\ speed \times (2 \times no\ FMA\ units) \times (max\ vector\ size/64)$, or could be empirical for a more accurate measurement. Peak bandwidth is usually measured with STREAM benchmark or other similar platform-optimized micro-benchmarks. On top of the naive roofline model, it is proposed that when the source of peak memory bandwidth is changed from DRAM to other members of the memory hierarchy, different roofs for maximum attainable performance could be found. Hence individual rooflines for L1, L2, and L3 caches could be found (Marques et al., 2017). An example roofline model using this method is given in Figure 2.2

All of the required values for building a roofline model are collected automatically by SuperTwin and elaborated in Section 4.2. Moreover, SuperTwin configures and

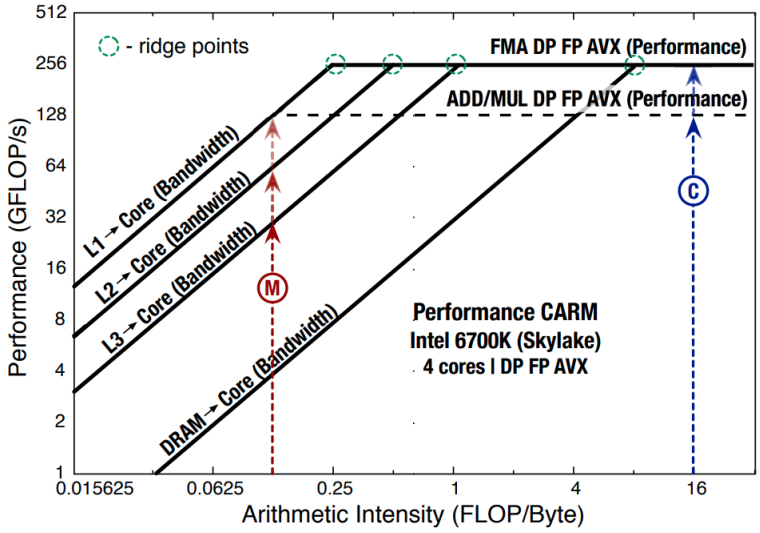

Figure 2.2 Cache Aware Roofline allows performance analysis to be performed w.r. to different memory levels. The ridge point for each roofline separates memory-bounded regions from floating point performance-bounded regions. Spaces between rooflines show in which memory level the application saturates the memory hierarchy. To move to the right, floating point operations per memory operations should be optimized. To move up, efficient use of the cache is required. To be able to surpass the dashed line, on top of the very efficient use of cache memory, the use of fused multiply-add operations is required (Marques et al., 2017).

executes the STREAM benchmark whose main paradigm has been given in Algorithm 1 to provide a baseline for peak memory bandwidth and HPCG benchmark which contains micro-benchmarks SpMV, DDOT, MG, and WAXPBY automatically on the target systems. More detail for this process are also provided in Section 4.2.

2.3 Performance Co-Pilot

Performance Co-Pilot, initially released in 1995 by SGI and currently being developed by Red Hat, is a system performance analysis toolkit. PCP contains two types of components; PCP collectors and PCP monitors. PCP collectors have two components; Performance Metrics Domain Agent (PMDA) and Performance Metric Collector Daemon (PMCD). PCP collectors are responsible for collecting and extracting performance data from various sources. These sources could be remote PMCDs, PMUs, application performance logs, or operating system components which

are given in Figure 2.1. PMDAs are lightweight agents that connect to performance sources and read their values, then report these values to PMCD. There are currently 75 PMDAs available, and apart from existing PMDAs, new PMDAs could be developed to connect any wanted performance metrics source using the PMAPI library. To be able to report metrics from a host machine, there must be a PMCD that listens and control all PMDAs and answer requests of monitoring applications. They are configured to discover available metrics from the system and build a Performance Metric Name Space (PMNS), which includes every metric that the system is configured and available to report. Note that PMNS does not include every performance that the system can report. PMNS grows or shrinks as PMDAs are installed or their configurations are changed. For every metric name in PMNS, there is also an associated instance domain. The instance domain contains identical components which report the exact same value. For example, in a host with 8 threads, a per-thread metric `hinv.cpu.clock` has 8 instances on its domain from `cpu0` to `cpu7`. Or a per-process metric `proc.psinfo.rss` will have an instance for every process there exist at the moment. Upon sampling this pmda, all values will be reported by default, however, instance domains could be filtered and only instances of interest could be sampled. Moreover, PMDAs work in a pull-only manner. They do not report metrics on their own and sample and report performance metrics only when asked to. Therefore they remain silent with negligible overhead when they are not in use. Monitoring tools display, manipulate and store performance metrics extracted from PMCDs.

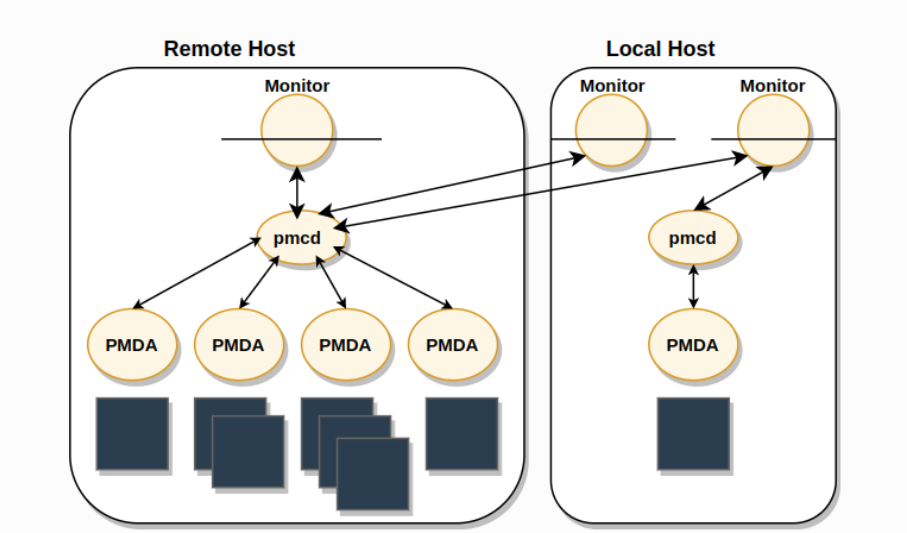

Figure 2.3 Architecture of distributed metric sampling with Performance Co-Pilot. Each PMDA is responsible for a specific domain of performance metrics. And on each host, a PMCD control these PMDAs and/or report their readings to remote machines (Red-Hat, 2021).

Another important component of PCP is PMIE. Upon the discovery of performance metrics, PMIE decides the semantics of the metric to be reported. There are three semantics supported with PCP. The first is counter. A counter is a performance metric source that monotonically increases (or decreases). When PMIE detects a counter value, it automatically configures the sampling PMDA to report the change in the value instead of the actual reading. This is the case with the `pmdaperfevent` which is used heavily in the SuperTwin. Another semantic is instantaneous value, which is more likely to be a singular value that represents a state, and the value at the time of reading is reported directly. The last semantic is a discrete value, which is similar to a counter, however, a discrete metrics value is decided to be independent of previous values, therefore did not change into a rate. PMIE could also define automated reasonings and alarms based on metrics values. Performance Co-Pilot also has monitor interfaces that can order local or remote PMCDs to report values from PMDAs. These monitor interfaces could directly insert reported metrics into the database. As in the case with `pcp2influxdb`. The reason (for especially PMUs) metric collection framework chosen as PCP is to easily generate timelines of executions in contrast to frameworks such as PAPI, perf, likwid-perfctr which report a summation of values at the end of the execution.

2.4 Grafana

Grafana is an open-source visualization tool that provides dynamic dashboards, ad-hoc queries, and alerting functions on time-series data. Since it's initial release at 2014, it quickly become the industry standard and reached 10M+ global users. Due to its massive user-base, Grafana supports every popular database and provides a wide variety of visualization methods. Grafana dashboards are serialized JSON files that could be templated, uploaded, and altered via web API. Since the dashboards are actually only JSON files, they are very easy to manipulate and generate for numerous metrics, and also easy to interact with SuperTwin Description (STD).

An example Grafana dashboard JSON is given in Listing 1. When uploaded and serialized via web API, this JSON turns into an interactive dashboard with a single panel that visualizes metric `perfevent_hwcounters_FP_ARITH_SCALAR_SINGLE_value` for the last 5 minutes. New metrics for comparison on the same could be added to `targets` list, or new panels with different metrics could be added to `panels` list.

Listing 1 A simplified JSON representation for a Grafana dashboard.

```
{
"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"}
}
```

3. Related Work

In order to systematically collect and store information from performance metric sources, several monitoring tools have been developed and widely used in the literature. However, these tools do not create a knowledge representation and automated analysis framework. These systems facilitate intelligent job placement, run-time workload partitioning/adaptation, and HPC hardware procurement planning (Brandt, 2013). Some of these tools are; LDMS (Agelastos, 2014), Ganglia (Ganglia, 2022), Nagios (Nagios, 2022), HPC-Toolkit (Adhianto, 2010) and PerfAugur (Roy, 2015). Among them, Ganglia is proven to be scalable up to 2000 nodes but is used for general system monitoring, requires a considerable number of installation dependencies, targets larger collection intervals (10s of seconds to 10s of minutes), and uses an aging tool for storage. Nagios also targets larger collection intervals (10s of seconds to 10s of minutes) and is mainly used for failure alerts. PerfAugur is used to trace the cause of a system anomaly by finding common attributes that predicate an anomaly (Agelastos, 2014). HPCToolkit is a suite of tools that can provide accurate measurements of program performance on a wide variety of systems, from single host computers to large clusters. However, it involves a binary analysis and re-compilation of the target code. LDMS and Performance Co-Pilot are metric collection, transport, and storage systems that can be configured to sample every performance metric counter on hardware and kernel, including RAPL, PAPI, and perf interfaces. Moreover, they support frequent and variable sampling rates on these performance metrics with negligible overhead and without the requirement of recompile or source code instrumentation. This enables real-time monitoring of HPC systems in the cluster level, node level, and process level in order to provide multiple-aspect insight into application performance. LDMS is part of OVIS, a suite of HPC monitoring, analysis, and feedback tools that is jointly developed by Sandia National Laboratories and Open Grid Computing. LDMS leverages sampler and aggregator daemons called `ldmsd`, which can run on either sampler or aggregator modes. A sampler `ldmsd` daemon is created by running and configuring sampling plugins that sample PMUs. Each sampling plugin combines a specific set of data into a single metric set. An aggregator `ldmsd` daemon is created by

running and `ldsmd` and configuring aggregator plugins. Each aggregator collects metric sets from samplers and/or other aggregators. Higher-level aggregators can listen to many lower-level aggregators, and aggregate and stream data into storage. LDMS can store sampled performance data on CSV files, D-SOS and InfluxDB. Increasing with the number of sampled metrics, LDMS causes very little overhead on the system performance. It causes $\approx 0.01\%$ CPU utilization, $< 2MB$ memory, $< 4MB$ filesystem and $4KB$ network overhead for ≈ 200 metrics @1 second intervals. (Agelastos, 2014) Although it has been repeatedly used in the literature and works well under certain circumstances, LDMS is mostly used by a strictly related group, lacks documentation, and is still under development. Therefore it's hard to deploy, develop and maintain.

E2EWatch (Aksar, 2021) perform system-wide monitoring over a very large system using Linux metrics and focus on anomaly types classification and anomaly detection. ClusterCockpit(Eitzinger, Gruber, Afzal, Zeiser & Wellein, 2019), is a recent and similar tool that reports performance metrics from a distributed system to InfluxDB and uses a pull-only approach. Provide monitoring dashboards and job history queries, also use both Linux and hardware performance counter metrics, however supporting only a number of pre-selected metrics, such as system load, floating point operations, and memory bandwidth and does not implement a knowledge representation or semantic query feature. To our knowledge, there is no monitoring framework that supports both Linux and hardware performance counter metrics, is fully configurable, and facilitates digital twins for analysis.

4. SuperTwin

As mentioned before, a great variety of factors affect application performance for computing systems. SuperTwin is designed to ease the detection of their existence and impact on computation and provide insights about their causes. To provide a detailed performance analysis, build performance models and hints for further optimization. In order to achieve these, SuperTwin needs to wield several qualities.

- First, it needs to be **generic**. Since the main problem was the great variation in the composition of components of computation systems, SuperTwin needs to be able to represent as many of them as possible, if not all, to be useful. In order to achieve this, always the most widely used and available tool or approach is chosen over its equivalents.
- SuperTwin needs to be **unified** in shape, representation, and methodology, even for different architectures and problems.
- SuperTwin needs to be **recursive by nature** in order to be able to represent flexible hierarchies and ownership of the underlying system.
- SuperTwin needs to be **dynamic** and highly **configurable** since the needs and focuses of SuperTwin's users may greatly vary. For example, a user who optimizes a big problem in input size may need to examine the whole memory system in detail while another user with a problem with a much smaller input size can focus on L1 cache efficiency and floating point operation performance.
- SuperTwin needs to be **modular** as it performs numerous distinct operations with different types of tools and different types of data. SuperTwin is designed modular in a couple of different ways. First, it consists of numerous functional modules and their configurator modules in order to adapt its systems to the target system and purpose as much as possible. Second, the SuperTwin description (STD) is not a static document. It is designed to be configurable, extendible, and capture more about the system it represents as time passes. In a manner, it folds on itself to create a more complex structure.

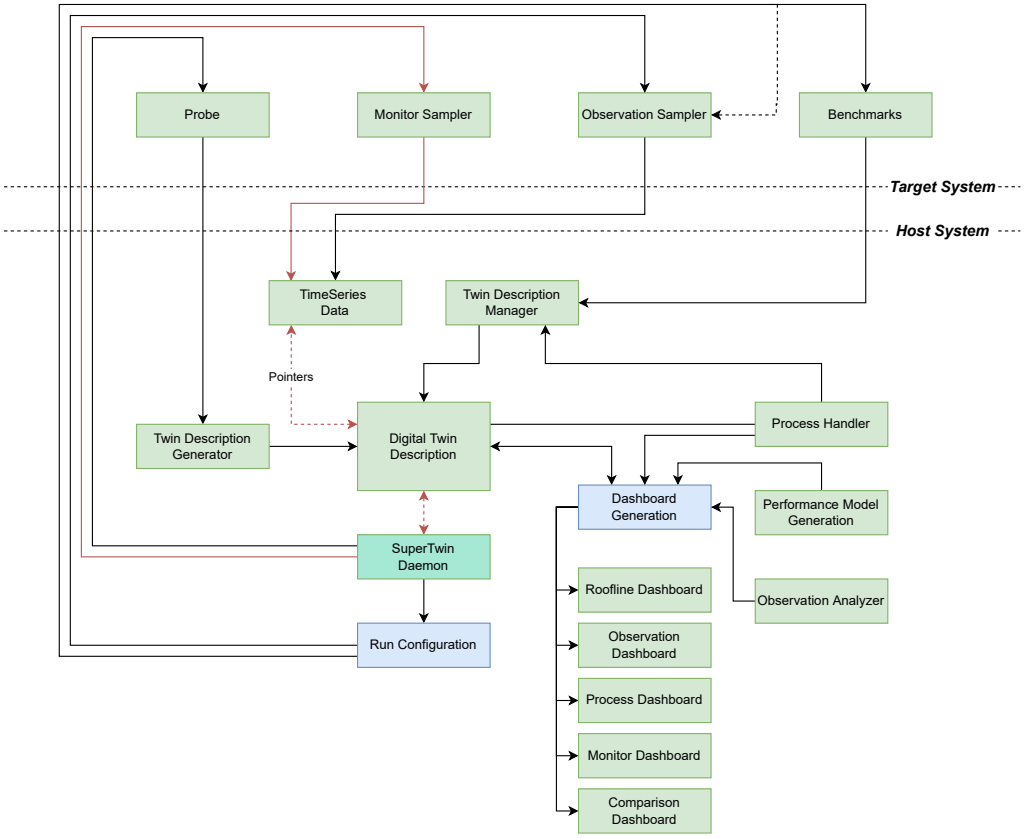

Figure 4.1 Structure of SuperTwin modules. Amber lines highlight the monitoring pipeline that is always on. Green nodes present functional modules, and blue nodes present configuration modules. Every other functional module on this figure is inherently invoked by the SuperTwin daemon.

Since the number of performance metrics a digital twin can report, in theory, is in the order of thousands, it is impractical to sample all of them at all times due to the overhead of the sampling process. Therefore, we defined a distinction between the types of collected metrics. The first type of the collected metrics is "Monitor" metrics, which are system use and system state-related software metrics such as the number of processes, CPU, and memory loads. These metrics are always sampled from the system with a relatively low frequency. The other type of metric is the "Observation" metric. Observation metrics are sampled from PMUs during the execution of computation kernels with high frequency. Some examples of the observation metrics are, floating point operations and cache bandwidths. However, sampling different metrics at different times with varying frequencies creates a need for metadata associated with them alongside the host system's metadata. While time-series databases like InfluxDB are fast and efficient for processing time-series data, they can't keep much metadata for knowledge management. On the contrary, inserting and querying time-series data into a document database such as MongoDB is impractical (Friedemann, 2019; Milenković, 2019). Therefore, the Digital SuperTwin requires two types of databases and a link between them. To this end, in our proposed design, while InfluxDB holds monitor and observation metrics as time-series data, MongoDB, which is a document store, holds STD as JSON-LD, which is extended with observation metadata with each computation. And to associate them with telemetry data, a pointer to InfluxDB is stored, which is the query parameter to recall sampled metrics. This structure allows accessing per-observation data, w.r.t. the job type and component type, via queries on metadata.

Modifiable twin document is one of the key design aspects of SuperTwin since abilities such as querying remote system information, previously observed events, offline analysis, comparison of multiple systems and/or different settings for observations, baseline comparison via recalling of benchmarks and performance models are made available by mutable twin description.

Yet another required quality for SuperTwin is that it needs to be as lightweight as possible to affect monitored systems and observed applications as little as possible. To achieve this, SuperTwin is broken into two parts a host and a target. While the host part of the SuperTwin executes commands on the target system, handles data collectors and data generation, configures samplers, generates dashboards, and runs the analysis. The target system only runs samplers and nothing else, leaving heavier work, such as database injection, analysis, and dashboard generation/presentation to the host system. Also, as mentioned earlier, PCP samplers work in a pull-only manner. Therefore, they report values only when they are asked. A detailed analysis of the sampler overhead is provided in Section 6.

As a medium of transportation between host and remote systems, three methods are used by SuperTwin. First is `ssh`; every command launched by SuperTwin is executed on the remote host via `ssh`. When there is a file transfer -for example, as in probing- `scp` is used. Operations using these types of communications are also facilitated and made scriptable since SuperTwin includes credentials. Other types of communication take place between host collectors and remote samplers via sockets. While scaling to distributed settings, `ssh` connection will be changed with `slurm` scripts or other scheduler interfaces.

SuperTwin is implemented in `Python` as a library. Although SuperTwin could always be kept up, to improve modularity and simplify working with several target hosts, a snapshot of the runtime object is written to the database alongside with STD. When a SuperTwin function wanted to be used, the SuperTwin object could be re-constructed via reading run-time variables from the database. Construction and re-construction of SuperTwin object could be seen in Algorithms 2 and 3.

4.1 Before Deploying SuperTwin

As the previous sections explain, SuperTwin interacts/configures/manages several other tools. For SuperTwin to function, these tools need to be installed on the host and remote systems beforehand SuperTwin is started. On the remote system, Performance Co-Pilot needs to be installed for metric shipment. For the generation of twin description, output from `cpuid, lshw, likwid-tools` (Röhl, Eitzinger, Hager & Wellein, 2017) and `libpfm4` need to be installed.

On a host system, MongoDB and InfluxDB for data management `pcp2influxdb` for metric shipment and Grafana for dashboard generation need to be installed.

4.2 Probing Framework

Detailed probing is required to capture the underlying system's structural information. SuperTwin probes the target system in order to describe its components with their specifications and inter-/intra-relations with performance metrics they

Field	*Probed Info*	*Field*	*Probed Info*	*Field*	*Probed Info*
System	arch	**Cache**	L1D associativity	**PMU Event**	PMU name
	os		L1D cache group topology		name
	kernel		L1D cache line size		description
	motherboard		L1D no sets		flags
	uuid		L1D size		Umask-*
	numa domains		L2 associativity		modif-*
Disk	model		L2 cache group topology	**Network**	businfo
	is_rotational		L2 cache line size		firmware
	size		L2 no sets		ipv4
	metrics		L2 size		link
CPU	model		L3 associativity		model
	# of cores		L3 cache group topology		serial
	# of thrads per core		L3 cache line size		speed
	hyperthreading		L3 no sets		vendor
	tlb		L3 size		is_virtual
	flags	**PMU**	name	**Memory**	clock
	nomimal clock rate		# of events		model
	min clock rate		# of counters		size
	max clock rate		max encoding		slot
	topology		type		vendor

Table 4.1 Probed info from the system, categorized for each component.

can report. This approach aims to present each physical hardware component that is monitorable, produces performance metrics, or individually affects the overall system performance. This probing must capture these relationships in a lightweight, easily adaptable, and generic way. To this end, we relied on widely available Linux tools to gather data. The system, network, and memory information are collected via `lshw` utility. The CPU, memory/cache topology metadata are collected by parsing `likwid-topology` and `cpuid` tools. When available, disk info is probed from `/sys/block/*/device` and `SMART` utility. PMU information is collected with `libpfm4` library, which can recognize model-specific registers and their events of virtually every x86 and ARM processor available on the market. Upon probing PMU information via `libpfm4`, every hardware performance metric (core, uncore, offcore) which `perfevent` can report are obtained. Information collected during probing phase is available in the Table 4.1

As repeatedly stated, SuperTwin bears comprehensive analysis and configuration abilities while keeping the overhead and interference to the target host as limited as possible. To this end, probing is ordered to the target system from the host system. When SuperTwin runs on a host system, the probing framework (the probing code itself is copied and executed on the remote) is copied to the target system via `scp`. Modules for the probing are scripted to perform probing operations in place without any further interference. The probing system is designed so that only triggering the main function results in a probing file containing every information of concern.

SuperTwin, when invoked, only asks for an alias and IP of the target system and uses

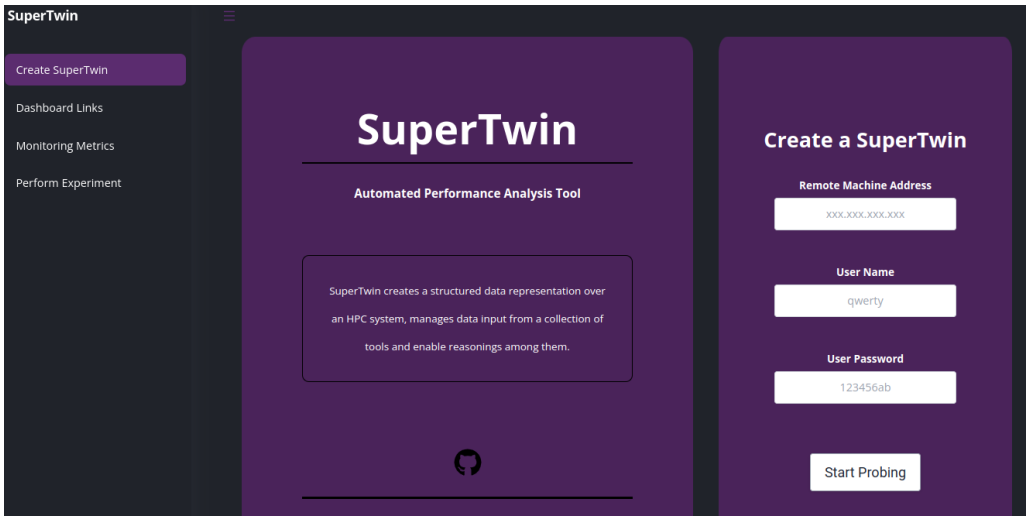

Figure 4.2 Login screen for SuperTwin web app

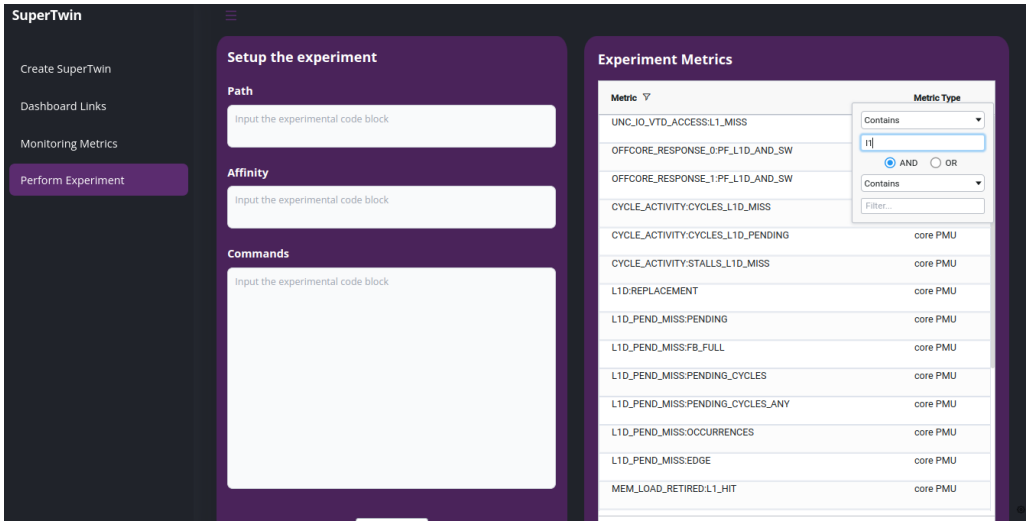

Figure 4.3 Metric selection and parameter entry screen for launching observations from web app

this information to create databases and data sources for the target system, which then will be used to store and recall collected information. Later, the probing module is copied to the remote system, information-collecting tools are launched from the main sapling module, and their results are parsed. Probing is further separated into three phases; the first phase is collecting system feature and topology information. System information collection starts with most general information like OS, kernel versions, motherboard serial number, and architecture. System probing also detects specifications and topology (and hierarchy) of non-processor hardware on the system. For example, network, storage, and memory devices, their specifications, and spatial order -such as individual memory banks- are also detected at phase since they can report individual performance metrics. For example, this particular information yields insignificant importance for the current status of the SuperTwin framework but could be used to detect a faulty memory bank in a cluster environment with hundreds of nodes without extensive investigation. Similarly, it can help to detect if any partition in the cluster differs in kernel version from the rest of the cluster in such a way that, an installation of a software module is incompatible or behave differently on this particular partition. This probing phase takes approximately ≈30 seconds for a single node.

Some specifications in this phase, such as the motherboard serial number, form the "sign" of the system and make sure the particular system is unique. The same system has multiple digital twins, or completely identical systems have separate digital twins. To make the separation, digital twins have their own unique id (which maps to a collection id in the document store), and machines have their motherboard serial numbers.

Collected probing information is written to a JSON file on the remote system and copied back to the host system to generate a twin description.

4.3 SuperTwin Description

As mentioned earlier, Digital Twin Description comprises several classes and relations between them, representing properties and hierarchy in a system. In SuperTwin, STD both; captures a semantic description of the target system and enables a linked time-series structure similar to the framework proposed in (Friedemann, 2019) but with far richer metadata and contemporary metadata instantiations of

Algorithm 2 ConstructTwin($IP, user, password$)

```
name, prob_file = remote_probe(IP, user, password)
mongodb_addr, influxdb_addr, grafana_addr = read_environment()
influx_name, mongodb_name, grafana_name = create_datasources()
mongodb_id = insert_twin_description(generate_twin_description(probe_file))
    ▷ From this moment, SuperTwin description and object is available to interact
SuperTwin.add_cache_aware_roofline_benchmark()
SuperTwin.monitor_metrics = read_from_environment()
SuperTwin.observation_metrics = read_from_environment()
SuperTwin.monitor_pid = start_sampling()
SuperTwin.generate_monitoring_dashboard()
SuperTwin.configure_observation_events(observation_metrics)
SuperTwin.observation(add_stream_benchmark())
SuperTwin.observation(add_hpcg_benchmark())
SuperTwin.generate_roofline_dashboard()
SuperTwin.register_state(SuperTwin.run_time_variables)
```

Algorithm 3 ReConstructTwin(IP)|ReConstructTwin($Name$)

```
db_id = db_lookup(IP)                                          ▷ Or Name
SuperTwin = reconstruct(db[db_id][twin_description],
                        db[db_id][run_time_variables])
```

events. For example, individual components, observations, and processes also have their digital twin descriptions with linked time-series data. This enables a fine-grain analysis of the behavior of applications to run on different systems with different software and hardware. For example, an L1 cache, a network interface, or a process could be isolated from the system, analyzed separately, or compared with its equivalent on a different system.

DTDL, as explained in Section 2.1.2, have a recursive structure that allows components (interfaces in the context of twin description) to be other components' subcomponents which is crucial for describing a cyber-physical system. On top of that, DTDL models `Telemetry, Properties` and `Relationship` have exact correspondence between what they describe in DTDL and STD. However, since DTDL is designed with IOT systems in mind, its descriptions are more physical than cyber-physical and are meant to be static. To this end, DTDL is modified, and new classes and properties are added to describe high-performance computing systems and create linked time-series data. The update made on DTDL to acquire STD ontology can be seen in the following table.

After acquiring system information via probing, using this information, STD is generated. During the generation of STD, every single physical component that performs computations, communications, or I/O operations is presented with an

Property	*Description*
@type	**Interface**
@id	Unique identifier within digital twin for interface
contents	a set of Interface, Process Interface, ObservationInterface, SWTelemetry, HWTelemetry, Benchmark, Properties, Relationships
displayName	Name to be displayed when instantiated
dashboard	dashboard url, optional
@type	**SWTelemetry**
@id	Unique identifier within digital twin for this telemetry instance
name	index in telemetries
instrance	instance name of reported component to be a parameter in queries
samplerName	name of the metric to be referred to during sampler configuration
DBName	name of the metric to be used in the generation of queries
@type	**HWTelemetry**
@id	Unique identifier within digital twin for this telemetry instance
name	index in telemetries
instrance	instance name of the reported component to be a parameter in queries
samplerName	name of the metric to be referred to during sampler configuration
DBName	name of the metric to be used in the generation of queries
PMUName	name of the metric as reported by libpfm4. To be used as parameter in perf event configuration
@type	**BenchmarkInterface**
@id	Unique identifier within digital twin for interface
contents	BenchmarkResult
displayName	Name of the benchmark to be displayed when instantiated
@type	**BenchmarkResult**
@id	Unique identifier within digital twin for this telemetry instance
field	name of field for subkernels, optional
no_threads	number of threads used
involved_threads	involved thread indexes to be used in queries
modifier	modifications in pinning strategy or compilation
result	result of benchmark
unit	unit of benchmark result
sampled_sw_metrics	sampled software metrics during execution, to be used in queries, optional
sampled_hw_metrics	sampled hardware metrics during execution, to be used in queries, optional
dashboard	dashboard url of observed metrics, optional
@type	**ObservationInterface**
@id	Unique identifier within digital twin for interface
displayName	Name to be displayed when instantiated
time	duration of observation
command	executed command
tag	tag of affiliated data in the database
no_threads	number of threads used
involved_threads	involved thread indexes to be used in queries
sampled_sw_metrics	sampled software metrics during execution, to be used in queries
sampled_hw_metrics	sampled hardware metrics during execution, to be used in queries
modifier	any modification made to the environment, optional
dashboard	dashboard url of observed metrics, optional

Table 4.2 New metamodel classes added to DTDL to build STD. There is also a model named `ProcessInterface`, which is in shape identical to `Interface`; however, its content fields are re-assigned every time the corresponding process's pid is changed.

Listing 2 Digital Twin is generated via both contextual and structural information probed from the system.

```
def add_my_metrics(component):
    for metric in available_metrics:
        if(component.type == metric.type):
            add_telemetry(component, metric)

def add_component(component, subcomponent):
    add_to_twin(subcomponent)
    add_my_metrics(subcomponent)
    add_ownership(component, subcomponent)

def add_subcomponents(component, subcomponents):
    for socket in system:
        add_component(system, socket)
        for core in socket:
            add_component(socket, core)
            for thread in core:
                add_component(socket, thread)
                for cache in cache_groups[thread]:
                    add_component(thread, cache)

def add_agents(component, subcomponent):
    for agent in pcp:
        resolve_process_state(agent)
        add_component(system, agent)

def create_twin(system_probing):
    system = create_system()
    add_subcomponents(system, cpus)
    add_component(system, memory)
    add_component(system, disks)
    add_component(system, networks)
    add_component(system, gpus)
    add_component(system, proc)
    add_agents(system, pcp)
```

`Interface`. Every hierarchical relationship between these components is encoded into the contents of these interfaces with a `Relationship` entry. Available metrics from these components are also filtered and encoded via `SWTelemetry` and `HWTelemetry`. Therefore, both precisely pinned executions and advanced semantic queries were made available. These interfaces later use values to configure samplers and locate their values in the database. For example, using generated STD and run configuration module, an execution that will run on 4 threads on each socket that does not share an L1 cache could be launched; similarly, after the execution of a run, performance metrics from threads that share same L2 cache with a given thread could be queried.

Another unique contribution of SuperTwin is that processes also can be modeled as digital twins and monitored via per-process kernel metrics. JSON-LD objects are serialized, which means string values from the JSON object are instantiated with given parameters into a run-time object. In SuperTwin, there are two degrees

of serialization. All models other than `ProcessInterface` are serialized and got their values assigned at the generation time; however, `ProcessInterface` is re-instantiated every time they are invoked and change due to the dynamic nature of processes.

4.4 Sampling Framework

Performance Co-Pilot has metric samplers responsible for a metric domain. As explained in Section 2.3. PMDAs are installed beforehand the sampling takes place, but they do not report values on their own. To sample metrics, a monitor framework needs to send requests to the target system PMCD. SuperTwin use `pcp2influxdb` as a monitor framework. Monitor framework, however, has a configuration that specifies metrics to be collected, instance domains of these metrics, frequency of the sampling, database address, and bucket to write metrics. Software metrics could be sampled by querying their parameters from SuperTwin and running `pcp2influxdb` with a generated configuration file. However, `perfevent` PMDA must be re-configured every time requested metrics are changed to set PMUs report requested metrics. After probing the target system and acquiring MSRs and available events, parameters required to re-configure a remote PMU could be queried from STD. SuperTwin also employs a `perf` reconfiguration module that queries this data from STD and reconfigures remote PMU automatically if a change in hardware telemetry in requested metrics is detected. After the reconfiguration, hardware telemetry could be sampled and recorded for corresponding observation. This process could be seen in Figure 4.4.

4.4.1 Monitoring

Software and hardware telemetry differ in their meaning and effect to the execution performance. Hardware metrics are direct measurements of performance events directly related to performance and could give definite reasoning for performance levels. For example, an exceptionally high L1 miss rate is thought to be primarily responsible for low performance and could not be resolved without source code optimization. On the other hand, software metrics do not give a cause directly related to

the application but provide a picture of the system state during execution and could reveal system-related anomalies such as resource contention, thermal throttle, memory leak, or poor affinity. Therefore, in SuperTwin, software and hardware telemetry are separated, and software telemetry is always sampled with low frequency. This is called the monitoring part of profiling in the SuperTwin context. Monitoring data could be used to model the remote system and predict possible faults mitigate before they happen. A configuration for monitoring is generated just after the generation of STD and monitoring starts. Software telemetry metrics, used for monitoring are given in Appendix A.

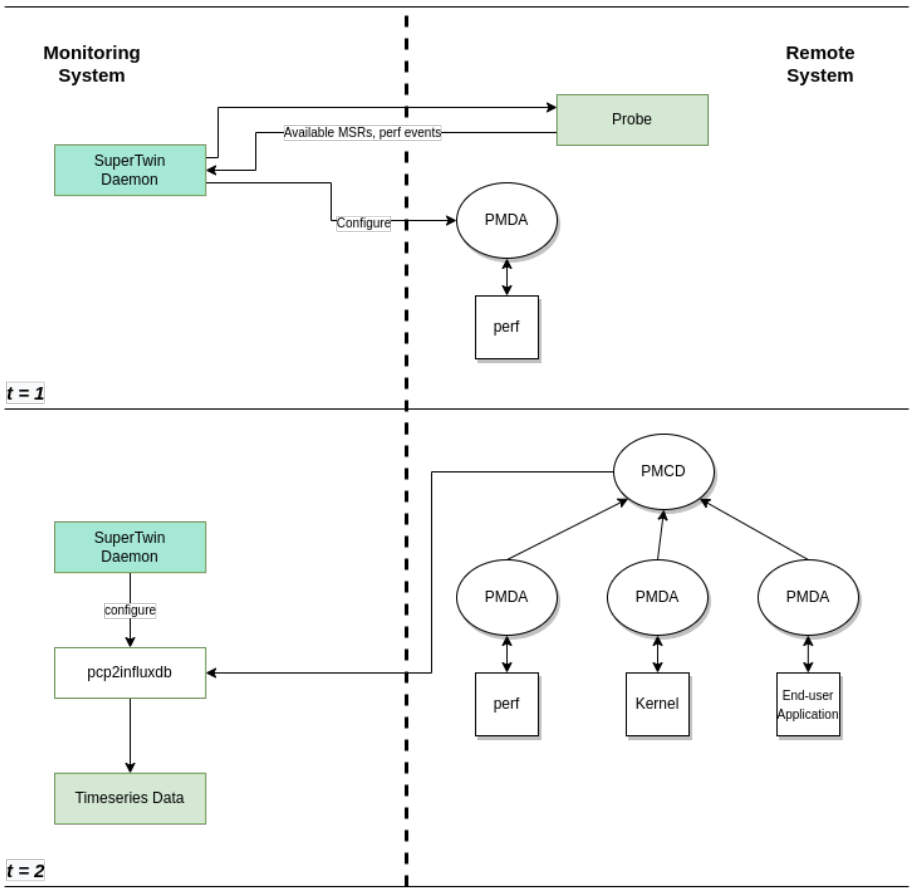

Figure 4.4 To probe hardware metrics with `perfevent` PMDA. PMUs are reconfigured beforehand the observation takes place.

4.4.2 Observation

When kernels will be executed with direct measurements of performance events, SuperTwin uses shell scripts as function wrappers. Whenever there is a need to set a directory, affinity, environment variables, or execute a binary, Observation module,

using Run Configuration module as helper generates a shell script with instructions to execute, copies it to remote, and execute it. An example request and resulting script could be seen in Listings 1 and 2

Listing 1 Bash script generated to execute kernel at desired path alongside with affinity.

```
#!bin/bash

cd /home/sparcity/eu
likwid-pin S0:0-3,22-25@S1:0-3,22-25 ./mkl_spmv mixtank_old.mtx
```

Listing 2 Psuedocode for executing observation at remote host with sampling using SuperTwin.

```
def observation_sampling(SuperTwin):
    config_lines = get_lines(SuperTwin.db_addr,
                             SuperTwin.db_name,
                             SuperTwin.hw_events)
    config_file = generate_configuration(config_lines)
    return config_file

def start_sampling(config_file):
    execute_local(pcp2influxdb -c config_file)
    return process

def observation(SuperTwin, path, command, input, threads):

    config_file = observation_sampling(SuperTwin)
    affinity = generate_binding(threads, "numa compact")
    bash_file = generate_bash_file(path, command, input, affinity)

    copy_to_remote(bash_file)
    sampler = start_sampling
    execute_remote("bash /tmp/st_files/gen_bash.sh")
    sampler.kill()
```

SuperTwin allows profiling of any command executed on the system; new frameworks are easily integrable into the framework via compiled executables.

4.4.3 Generation of Dashboards

SuperTwin can generate several types of dashboards on the fly after STD is generated. SuperTwin's dashboard module exploits the fact that Grafana dashboards are serialized JSON files and easily generate Grafana queries parameters stored in STD interfaces. Generated dashboards are later uploaded to the local Grafana server and

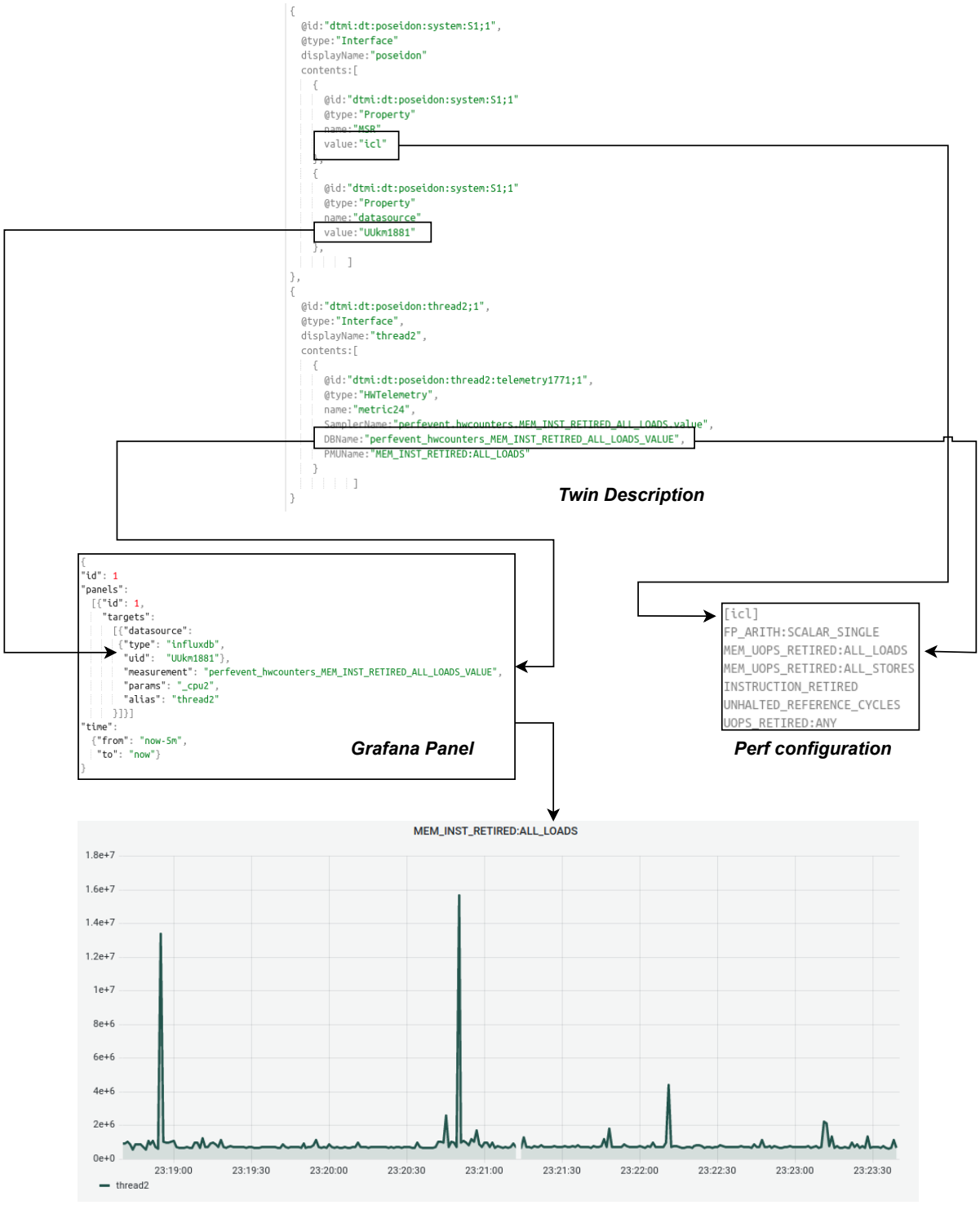

Figure 4.5 A summarized version of SuperTwin dashboard generation pipeline. STD is generated with all components having their metrics, and their metrics representation in different frameworks as content. Structured queries then capture these values, create configuration files and run tools.

Figure 4.6 Generated Monitor dashboard for host Dolap. In the socket panels, threads sharing the same L1 cache are plotted consecutively, leveraging STD. At the time of the screenshot, a computation that is just launched in NUMA socket 0 but anomalously allocates memory from NUMA socket 1.

their addresses are encoded in the corresponding interface entry in STD. A brief example of JSON generation using STD could be seen in Listing 1.

Since the metadata and benchmark results are stored in the digital twin description, and the generation of performance models, dashboard panels, and dashboards are functions of SuperTwin, generated performance models and charts could be recalled at any time after the digital twin is created. This allows any new observation to be ready for comparison with executed micro-benchmarks.

4.5 Benchmarks

Widely used benchmarks STREAM and HPCG are also copied to the target system with their architecture optimized and most recent versions, their make files configured with respect to available maximum vector extension capabilities in the target system and compiled in place in order to ensure system performance is ideally measured. Benchmarks are counted as a part of probing but at the same time interpreted just as any other observations. Therefore monitoring metrics and observation metrics are also sampled during the execution of benchmarks and made available

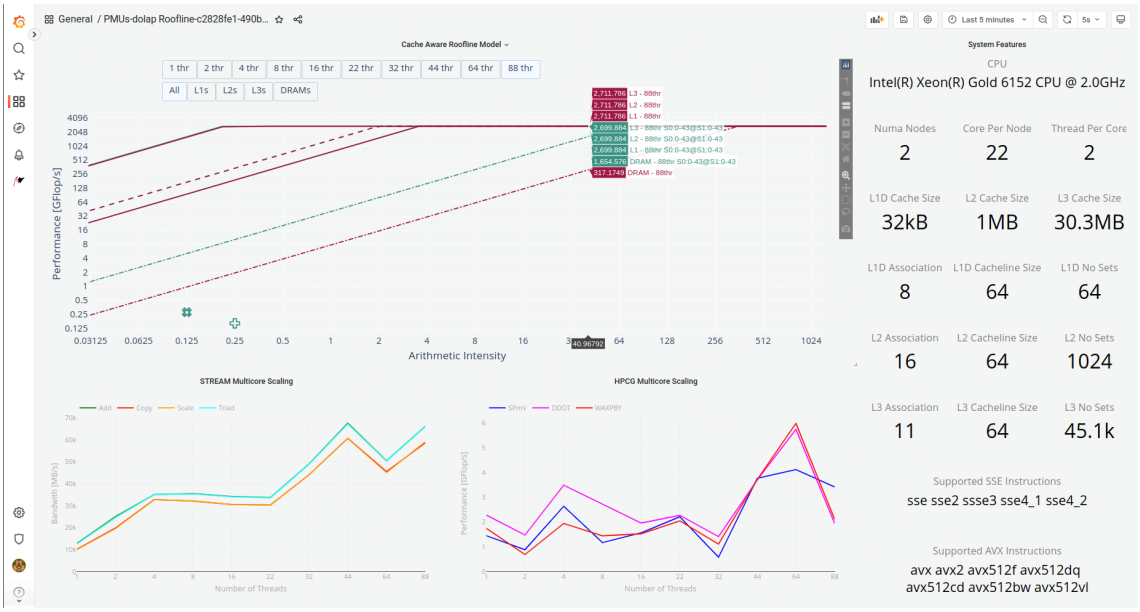

Figure 4.7 Performance model dashboard generated by SuperTwin showing CARM model generated for threads that are multiples of two plus cores per socket, threads per socket, and total threads, along with STREAM and HPCG multicore scaling and architecture information. This information is also readily available and comparable to any other observation made by SuperTwin.

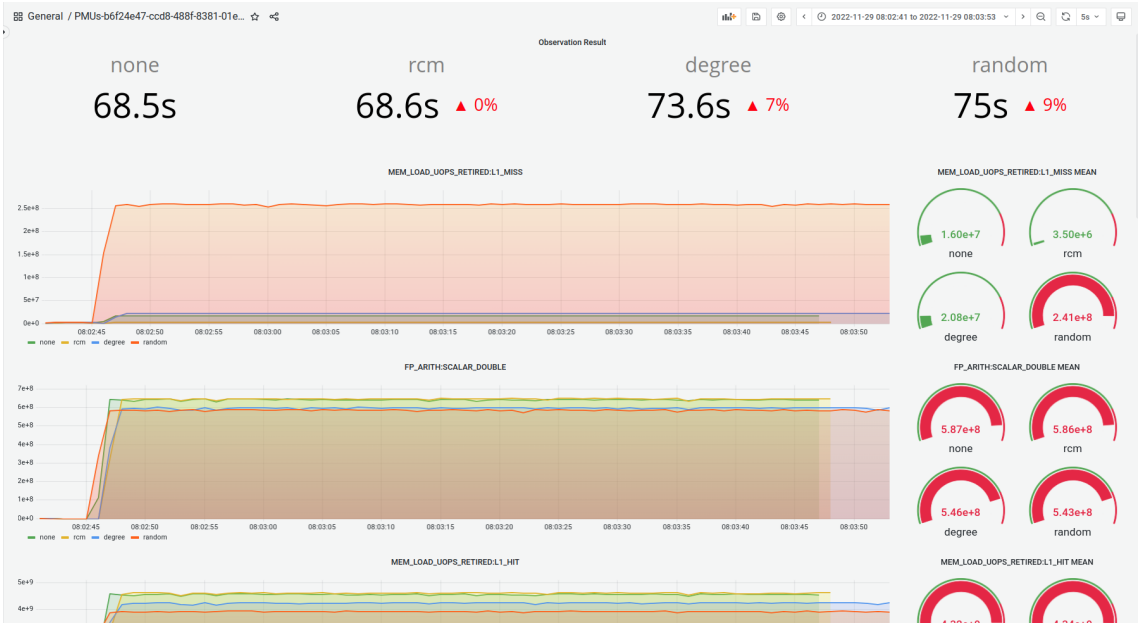

Figure 4.8 Comparison dashboard generated with SuperTwin. After the execution of several distinct events at distinct times (these events could also be executed at different hosts), metric timestamps are overlapped and presented to reveal common program phases in different settings. Augmented statistics are also provided alongside time-series data.

for future comparisons. Benchmarks are encoded in STD with dedicated `Benchmark` and `Benchmark Result` models to facilitate semantic queries. Benchmark entries for STD could be seen in Table4.2. A minimal example for an executed benchmark could be seen in Algorithm 1, whose result is given in Listing 3.

Listing 3 A minimal subset from SuperTwin digital twin description, recording results for STREAM benchmark.

```
{
  {dtmi:dt:dolap:system:S1;1:
    {@type: "Interface",
     @id: "dtmi:dt:dolap:system:S1;1"
     @contents:
       [{@id: "dtmi:dt:dolap:benchmark:B1;1"
         @type: "benchmark",
         @name: "STREAM",
         @contents:
           [{@id: "dtmi:dt:dolap:benchmark_res:B1;1",
             @type: "benchmark result",
             @field: "triad"
             @threads: 1,
             @modifier: "likwid-pin -c 0",
             @result: 12816.9,
             @unit: "MB/s"},
            {@id: "dtmi:dt:dolap:benchmark_res:B2;1",
             @type: "benchmark result",
             @field: "triad"
             @threads: 2,
             @modifier: "likwid-pin -c 0-1",
             @result: 25071.5,
             @unit: "MB/s"}]
        }]
    }
  }
}
```

5. Experimental Results

SuperTwin aims to present and monitor every software and hardware component, with statistics of the past executions on a target system. It creates linked data, performs semantic queries, and generates live and historical dashboards and analyses. However, measurements need to be performed for all of the former to be meaningful, accurate, and lightweight. Measuring the performance of a system creates extra work to perform the measurement, affecting the correctness of measurements or, worse, decreasing the performance of the measuring system and/or execution. Due to this, a measurement on a target system should be as lightweight as possible and made sure not to affect the measured events.

SuperTwin performs performance measurements on target systems via Performance Co-Pilot. To prove Performance Co-Pilot's suitability to SuperTwin use cases, an in-depth analysis of Performance Co-Pilot's performance, correctness, and effect on the target system, using SuperTwin configurations, is performed. To provide a complete picture of SuperTwin scenarios, a comprehensive analysis including system resource usage, remote report efficiency, the maximum resolution of monitor/performance events, correctness, and overhead of the Observation events are studied.

Dolap		**Deren**	
OS	Ubuntu 20.04.3 LTS x86_64	**OS**	Ubuntu 22.04.1 LTS x86_64
Kernel	5.4.0-135-generic	**Kernel**	5.15.0-47-generic
CPU	Intel Xeon Gold 6152 @3.7GHz x2 (44c/88t)	**CPU**	Intel i7-9700F @4.7GHz (8c/8t)
MSR	skx	**MSR**	skl
Mem.	1TB DDR4 @ 2666MHz	**Mem.**	64GB DDR4 @ 2133MHz
Env.	pcp 5.3.7-1	**Env.**	pcp 5.3.6-1
Poseidon		**Luna**	
OS	Ubuntu 20.04.4 LTS x86_64	**OS**	Ubuntu 18.04.6 LTS x86_64
Kernel	5.15.0-56-generic	**Kernel**	5.4.0-135-generic
CPU	Intel i9-11900K @5.1GHz (8c/16t)	**CPU**	Intel Xeon E5-1650 v2 @3.9GHz (6c/12t)
MSR	icl	**MSR**	ivb_ep
Mem.	16GB DDR4 @2666MHz	**Mem.**	16GB DDR3 @1866MHz
Env.	pcp 6.0.1-1	**Env.**	pcp 4.0.1-1

Table 5.1 System speciations of hosts used in experiments.

To provide a wider depiction, increase the confidence interval for the results, and assure the previously mentioned genericness of SuperTwin, a test set including reasonably different systems, all different in capabilities with different MSRs, is used. `Dolap` is a recent and remarkably powerful high end server with 2 CPUs, 88 threads and 1TB of RAM. `Poseidon` is a performant and recent server, `Deren` is an upper-middle tier desktop for general use which have the desktop version of `Dolap` MSR. `Luna` is a 10-year-old and fairly weak machine included in a test set to analyze consistency in extreme cases. The specifications of the host machines used are summarized in Table 5.1.

5.1 Resource Use of Sampling

Since PCP employs several agents who collectively perform metric shipment operations, resource usage on the remote system may become overwhelming with the increasing number of sampled metrics and resolutions. To this end, CPU and memory usage of individual PCP agents that are used by SuperTwin are measured for the different number of sampled metrics and sampling frequencies. Measurements are performed for 10 minutes while the target systems are empty, and results are averaged. Results for sampling 50 metrics with varying frequencies are given in Figures 5.1, 5.2, 5.3, and 5.4 for Dolap, Deren, Poseidon and Luna, respectively. The network is monitored as a whole for each system. I/O use of PCP agents was found to be negligible (<1 KB). Therefore, they are not included in the results. During the measurements, the host system had 100 Mbit cabled connection with each system. The host system's disk performance was measured at 182 KB/s, and 1.2 MB/s for 512B and 8K block-sized writes, respectively.

For recall, from PCP agents, `pmcd` manages other agents and reports their readings to remote requesters. `perfevent` samples PMU readings via Linux `perf` interface, `pmdalinux` reports software sourced system state metrics such as CPU load, `pmdaproc` reports per process metrics, such as io and memory usage for each process on the system. Measurements for CPU usage are made using `proc.psinfo.utime` and `proc.psinfo.stime` metrics, for memory `proc.psinfo.rss` metric. For each host, the measurement with the most metrics is reported, and the rest of the test set is given in Appendix A. Metrics used for measurements are given in Appendix B. The first observation that could be made instantly is, apart from the number of reported metrics or frequency of sampling, all agents are found to use a constant

amount of memory. Higher memory usage of `pmdaproc` is due to the size of a much bigger instance domain. Other higher usages of system resources in Dolap, albeit having much more powerful component composition, is also due to much bigger instance domains in Dolap. For example, a `pmdaperfevent` metric has 8 instances in Deren while the same metric has 88 instances on Dolap. Similarly, a much higher number of running processes and system components results in higher system resource uses for Dolap. Apart from `pmdaproc`, all agents are found to be thrifty in system usage resources. These measurements were made without filtering on instance domains; instance domains could be filtered to reduce thousands of instances to a couple of instances of interest. Also, the monitoring framework of SuperTwin uses 0 per-process metrics and uses ≈20 `pmdalinux` metrics and ≈2 `pmdaperfevent` metrics at 1 second intervals.

An interesting observation is that even though Dolap has more processes (therefore instances in `pmdaproc` instance domain) than other baseline systems, `pmdaproc` uses slightly lower memory w.r.t. Poseidon and Deren and the memory consumption is similar to that of Luna. This could be due to both servers bearing Xeon CPUs, but it requires further investigation.

5.2 Throughput and Integrity of Reported Metrics

PCP agents and network usage are scaled almost linearly for increased sampling frequency. They use resources consistently, as almost no deviation was observed during measurements, as seen in the error bars. This proper scaling also exists with metrics. Every system in the experiment set, except for Luna, scales proportionally when the number of collected data points is increased. On Luna, the reason for poor scalability is most probably due to a constant overhead for running the framework, since there is not much reporting loss, which will be explained later. However, one case in the test set hints that the PCP framework does not scale perfectly. In Figure 5.1, there is almost no difference in 4 and 8 reports per second, and the network traffic varies during measurement contrary to the rest of the entire test set. This behavior is also observed in other Dolap measurements with the exception of 10 metrics.

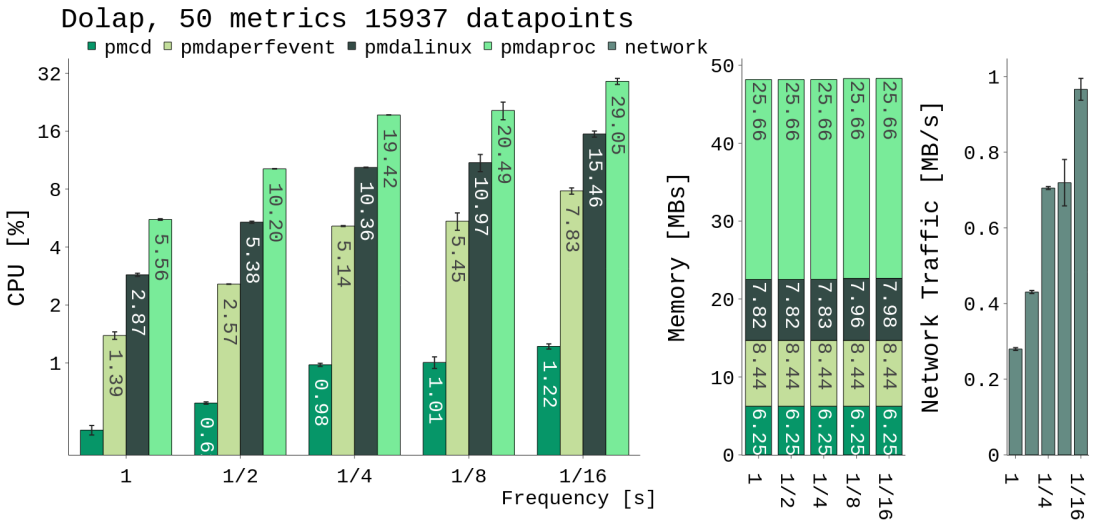

Figure 5.1 System resource usage of metric shipment with kernel and PMU metrics on Dolap. Metric agents `pmdaperfevent`, `pmdalinux`, and `pmdaproc` report 24, 20, 6 metrics and 2112, 285, 13572 data points, respectively.

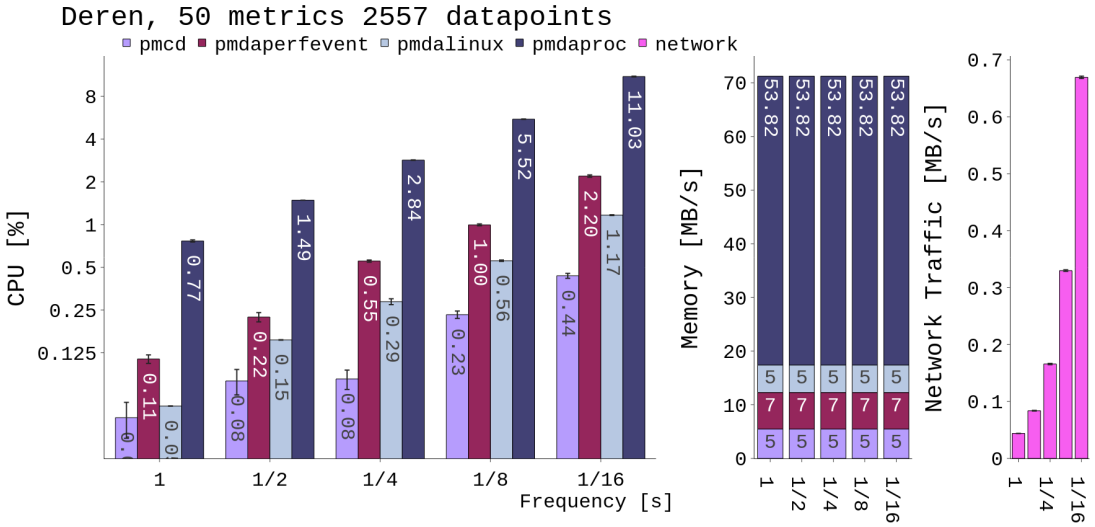

Figure 5.2 System resource usage of metric shipment with both kernel and PMU metrics on Deren. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` report 24, 20, 6 metrics and 192, 40, 2325 data points respectively.

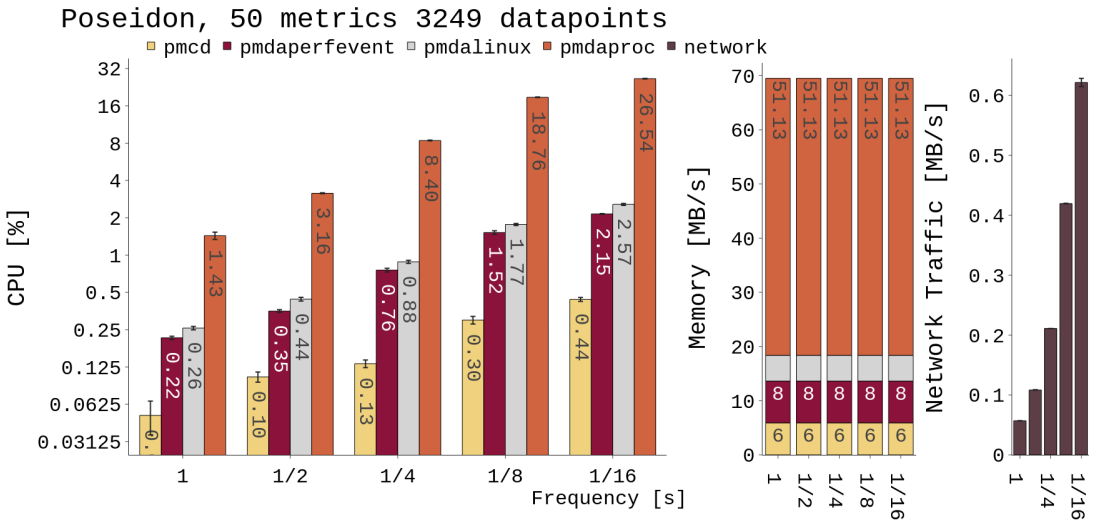

Figure 5.3 System resource usage of metric shipment with both kernel and PMU metrics on Poseidon. Metric agents `pmdaperfevent`, `pmdalinux`, and `pmdaproc` report 24, 20, 6 metrics and 384, 60, and 2805 data points, respectively.

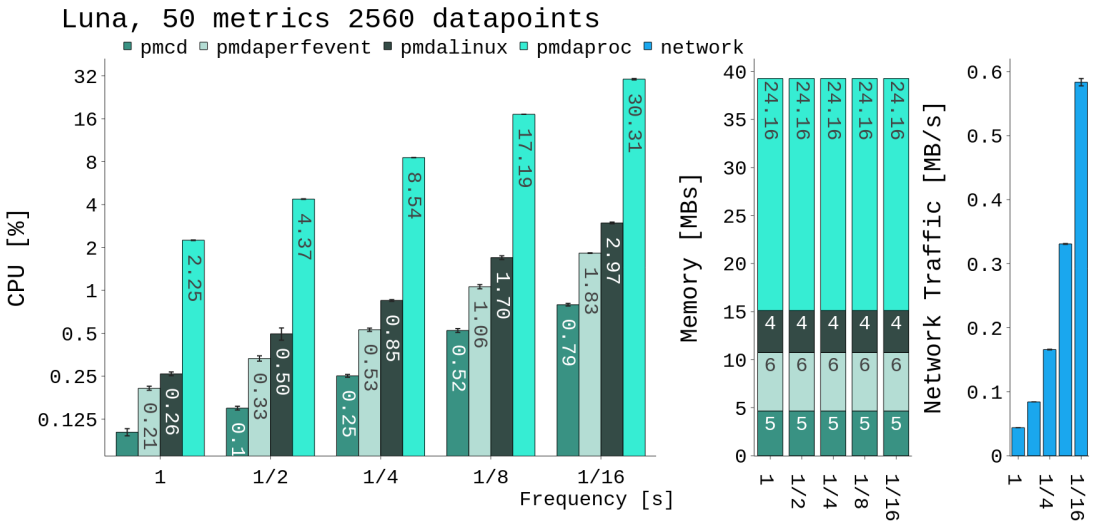

Figure 5.4 System resource usage of metric shipment with both kernel and PMU metrics on Luna. Metric agents `pmdaperfevent`, `pmdalinux`, and `pmdaproc` report 24, 20, 6 metrics and 288, 52, 2205 data points, respectively.

The underutilization of the network, together with the underutilization of the CPU, suggests that the framework is stalled and neither samples nor reports the performance metrics with the desired frequency. This is possible since communication is over the network, and there is no mechanism to buffer and resend missing metrics once more. Due to high frequency, at the time of resending, missing metrics are outdated by hundreds or thousands of new reports.

To further investigate this situation, we performed high-frequency readings and measured the actually reported data points and the ratio of loss with `procpmda`

and perfeventpmda. On top of the missing values, we observed batched zero values in our database with very high frequencies of samplings. Since procpmda reports for all processes, and there will be many correct zero values, we measured the wrong zero values with pmdaperfevent. With perfevent, we sampled metrics that are highly unlikely to report zero; UNHALTED_CORE_CYCLES, INSTRUCTION_RETIRED, UOPS_DISPATCHED etc. Then, we count the number of zeros in the database after the measurement is completed.

Tabe 5.2 reports throughput of procpmda. On Dolap, there are ≈1100 processes running while the server is empty; therefore, pmdaproc has ≈1100 instances per metric. On Deren, there are ≈400 processes running while the server is empty. On Dolap, a loss jump was observed after 8 reports per second when the number of individual data points exceeds 30K per second, and another considerable jump is observed with 16 reports per second. After this point, despite slight increases with increasing demand, losses also increased and reported individual data points remained between 30K and 40K data points per second. On Deren, losses exhibited a similar jump after 30K data points with 16 reports per second. Although 48K data points per second are achieved, with increasing frequency, losses are also increased, and maximum throughput remains around ≈40K data points. It is concluded that losses are affected by both the frequency and number of instances, and reports that include fewer data points are slightly less prone to losses. The maximum throughput of procpmda is around 30K-40K data points per second, despite being subject to small changes w.r.t. host and number of instances.

The throughput achieved with pmdaperfevent can be seen in Table 5.3. Instead of sampling and reporting of operating system files, pmdaperfevent samples PMUs, another bottleneck for maximum throughput. Therefore, we expect a lower value than pmdaproc. Similar to pmdaproc, with 16 reports per second, a massive jump in losses is observed with both systems, with the contribution of false zeros. Furthermore, the loss amount is correlated with the size of the instance domain. While being much more significant in Dolap, an increase in batch zeros and losses with correspondence with pmdaproc are observed. This further strengthens the previous conclusion that larger numbers of instances both in reports and high frequency are effective in losses.

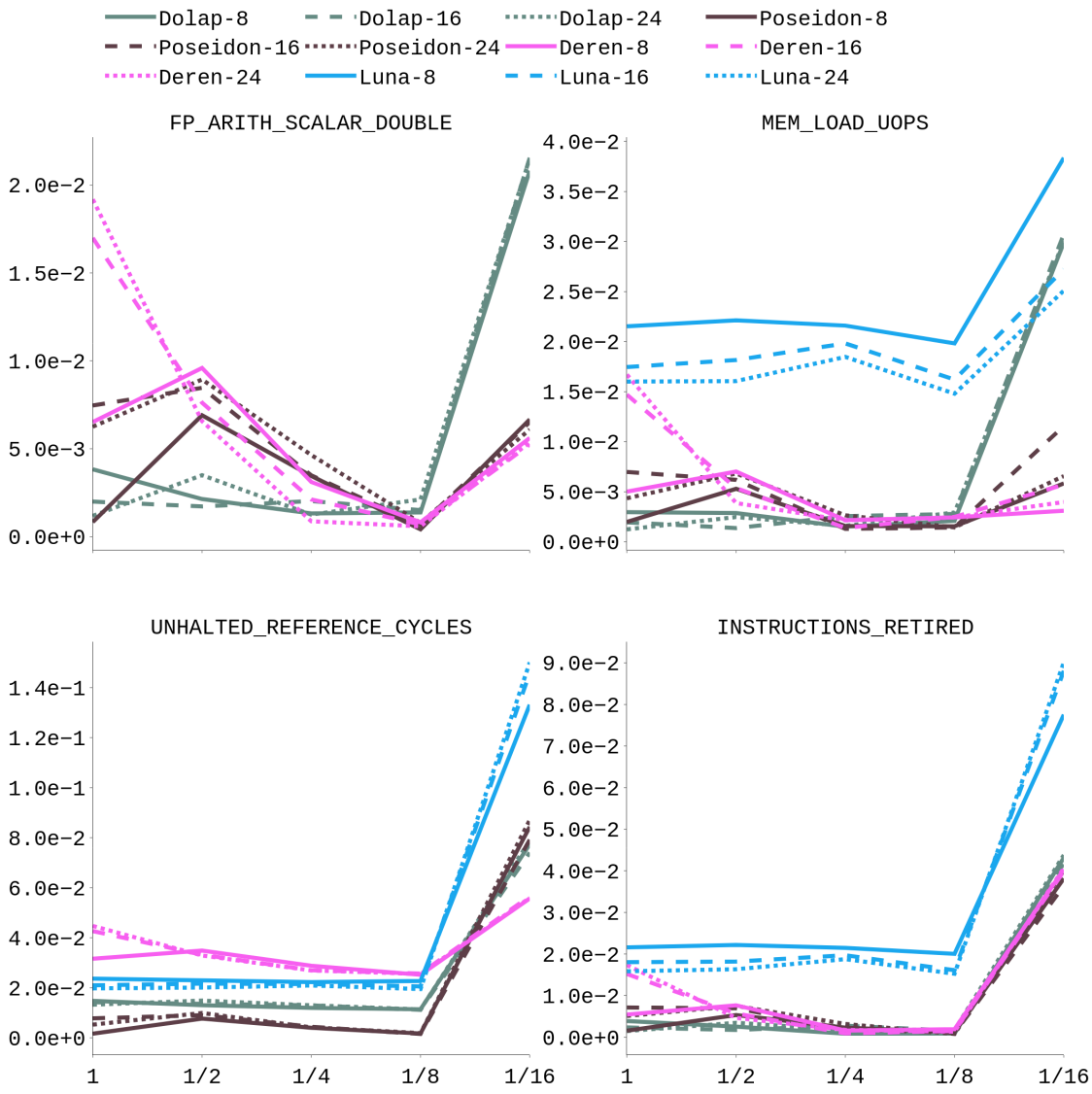

Figure 5.5 Accuracy (y-axis) in terms of relative error of 4 different events counted and compared against values reported by `likwid-bench` kernels triad, stream, sum, peakflops, ddot, daxpy on 4 different systems. Calculated individual errors are further averaged into a single value. The x-axis shows the sampled values per second.

Host	*Frequency*	*# of metrics*	*Expected*	*Inserted*	*% Loss*	*Throughput*
Dolap	2	4	8.56E+04	8.45E+04	**1.3**	8447.8
		5	1.07E+05	1.06E+05	**1.0**	10592.6
		6	1.29E+05	1.28E+05	**0.8**	12802.3
	4	4	1.72E+05	1.68E+05	**2.4**	16793.7
		5	2.13E+05	2.09E+05	**2.3**	20860.3
		6	2.57E+05	2.50E+05	**2.5**	25040.7
	8	4	3.49E+05	3.05E+05	**12.5**	30536.0
		5	4.22E+05	3.61E+05	**14.5**	36084.8
		6	5.16E+05	3.79E+05	**26.5**	37909.5
	16	4	6.92E+05	3.19E+05	**53.9**	31917.6
		5	8.45E+05	3.38E+05	**60.0**	33787.5
		6	1.03E+06	3.85E+05	**62.8**	38466.1
	32	4	1.37E+06	3.14E+05	**77.2**	31368.6
		5	1.69E+06	3.63E+05	**78.5**	36321.7
		6	2.07E+06	3.90E+05	**81.2**	38962.9
	64	4	2.72E+06	3.01E+05	**88.9**	30109.7
		5	3.38E+06	3.62E+05	**89.3**	36213.2
		6	4.11E+06	3.78E+05	**90.8**	37783.2

Host	*Frequency*	*# of metrics*	*Expected*	*Inserted*	*% Loss*	*Throughput*
Deren	2	4	3.16E+04	3.14E+04	**0.4**	3144.2
		5	3.99E+04	3.97E+04	**0.4**	3974.0
		6	4.76E+04	4.76E+04	**0.0**	4761.6
	4	4	6.32E+04	6.21E+04	**1.7**	6209.4
		5	7.98E+04	7.91E+04	**0.9**	7914.5
		6	9.54E+04	9.27E+04	**2.8**	9274.9
	8	4	1.26E+05	1.24E+05	**2.3**	12352.2
		5	1.60E+05	1.57E+05	**2.0**	15672.2
		6	1.91E+05	1.88E+05	**1.5**	18829.7
	16	4	2.54E+05	2.49E+05	**2.0**	24871.5
		5	3.18E+05	3.11E+05	**2.1**	31147.7
		6	3.82E+05	3.14E+05	**18.0**	31364.0
	32	4	5.08E+05	4.18E+05	**17.6**	41844.4
		5	6.35E+05	4.81E+05	**24.3**	48116.4
		6	7.64E+05	4.56E+05	**40.4**	45579.0
	64	4	1.02E+06	3.95E+05	**61.1**	39483.4
		5	1.26E+06	4.07E+05	**67.7**	40738.7
		6	1.53E+06	3.84E+05	**74.9**	38367.2

Table 5.2 Number of data points expected and observed at the host database w.r.t. the number of metrics and sampling frequency. *Throughput* is inserted datapoints per second.

Host	*Freq.*	*metrics*	*Expected*	*Inserted*	*Zeros*	*% Loss*	***% L+Zeros***	*Throughput*	*A. Throughput*
Dolap	2	4	7.04E+03	6.62E+03	0.00E+00	6.0	**6.0**	661.8	661.8
		5	8.80E+03	8.71E+03	0.00E+00	1.0	**1.0**	871.2	871.2
		6	1.06E+04	1.06E+04	0.00E+00	0.0	**0.0**	1056.0	1056.0
	4	4	1.41E+04	1.31E+04	2.22E+02	7.0	**8.6**	1309.4	1287.2
		5	1.76E+04	1.76E+04	0.00E+00	0.0	**0.0**	1760.0	1760.0
		6	2.11E+04	2.05E+04	0.00E+00	3.0	**3.0**	2048.6	2048.6
	8	4	2.82E+04	2.60E+04	5.84E+02	7.8	**9.8**	2597.8	2539.4
		5	3.52E+04	3.42E+04	7.72E+01	2.8	**3.0**	3423.2	3415.5
		6	4.22E+04	4.22E+04	0.00E+00	0.0	**0.0**	4224.0	4224.0
	16	4	5.63E+04	4.49E+04	1.34E+04	20.3	**44.0**	4491.5	3151.5
		5	7.04E+04	6.88E+04	1.73E+04	2.3	**26.8**	6881.6	5155.0
		6	8.45E+04	8.25E+04	2.00E+04	2.4	**26.0**	8247.4	6248.5
	32	4	1.13E+05	6.97E+04	3.04E+04	38.1	**65.1**	6969.6	3927.9
		5	1.41E+05	1.14E+05	5.32E+04	19.4	**57.2**	11352.0	6030.3
		6	1.69E+05	1.20E+05	5.02E+04	28.8	**58.5**	12027.8	7012.1
	64	4	2.25E+05	3.57E+04	1.61E+04	84.2	**91.3**	3569.3	1962.6
		5	2.82E+05	1.14E+05	5.32E+04	59.6	**78.5**	11387.2	6063.7
		6	3.38E+05	1.31E+05	5.91E+04	61.3	**78.8**	13073.3	7163.6
Host	*Freq.*	*metrics*	*Expected*	*Inserted*	*Zeros*	*% Loss*	***% L+Zeros***	*Throughput*	*A. Throughput*
Deren	2	4	6.40E+02	6.40E+02	0.00E+00	0.0	**0.0**	64.0	64.0
		5	8.00E+02	8.00E+02	0.00E+00	0.0	**0.0**	80.0	80.0
		6	9.60E+02	9.60E+02	0.00E+00	0.0	**0.0**	96.0	96.0
	4	4	1.28E+03	1.24E+03	0.00E+00	3.0	**3.0**	124.2	124.2
		5	1.60E+03	1.53E+03	0.00E+00	4.5	**4.5**	152.8	152.8
		6	1.92E+03	1.83E+03	0.00E+00	4.5	**4.5**	183.4	183.4
	8	4	2.56E+03	2.49E+03	0.00E+00	2.8	**2.8**	249.0	249.0
		5	3.20E+03	3.10E+03	1.60E+00	3.0	**3.1**	310.4	310.2
		6	3.84E+03	3.72E+03	0.00E+00	3.0	**3.0**	372.5	372.5
	16	4	5.12E+03	5.00E+03	4.61E+02	2.3	**11.3**	500.5	454.4
		5	6.40E+03	6.40E+03	5.27E+02	0.0	**8.2**	640.0	587.3
		6	7.68E+03	7.48E+03	5.89E+02	2.6	**10.3**	747.8	689.0
	32	4	1.02E+04	9.67E+03	3.69E+03	5.6	**41.6**	967.0	597.8
		5	1.28E+04	1.25E+04	4.67E+03	2.6	**39.1**	1246.4	779.9
		6	1.54E+04	1.49E+04	5.50E+03	2.9	**38.7**	1490.9	940.8
	64	4	2.05E+04	1.99E+04	1.05E+04	2.8	**54.3**	1991.0	936.2
		5	2.56E+04	2.19E+04	1.09E+04	14.4	**57.2**	2190.4	1095.6
		6	3.07E+04	2.70E+04	1.36E+04	12.2	**56.4**	2696.6	1338.1

Table 5.3 Number of data points expected and observed at the host database w.r.t. the number of metrics and sampling frequency. Throughput is inserted data points per second. ***L%+Zeros*** is the ratio of false zeros subtracted from inserted values to the expected value. *A.throughput* is the number of correct data points inserted to the database per second.

5.3 Accuracy of Hardware Performance Counter Sampling

To measure the accuracy of hardware performance counting, we employed `likwid-bench` micro-benchmark, which executes the generated assembly code with adjustable size and time and reports performance events that have correspondence with hardware performance counters. From the reported values of `likwid-bench`, `Cycles` is calculated with `UNHALTED_REFERENCE_CYCLES`, the number of FLOPs is calculated with `FP_ARITH:SCALAR_DOUBLE` on Dolap, Deren and Poseidon and `FP_COMP_OPS_EXE:X87` on Luna (Weaver, Terpstra & Moore, 2013). Data volume is calculated as `MEM_UOPS_RETIRED:ALL_LOADS+MEM_UOPS_RETIRED:ALL_STORES` on Dolap, Deren and Luna, and `MEM_INST_RETIRED:ALL_LOADS+MEM_INST_RETIRED:ALL_STORES` on Poseidon. The number of total instructions is calculated as `INSTRUCTION_RETIRED` on all hosts. UOPs is calculated with `UOPS_RETIRED_SLOTS` on all hosts except on Poseidon calculated with `UOPS_RETIRED_SLOTS`. Finally, `AI` is then calculated total FLOPs/total bytes for corresponding metrics. Micro-benchmark kernels `triad`, `sum`, `stream`, `peakflops`, `ddot` and `daxpy` executed on all hosts with varying frequencies and additional metrics other than previously mentioned in order to provide a deep analysis of accuracy. To focus on L1 bandwith on memory operations, kernels are executed with 100KB size, which completely fits in cache on all hosts.

To present the setting with the highest accuracy, averaged relative errors for all kernels are examined and presented in Figure 5.5. It's found that Luna performs slightly worse than other hosts in terms of accuracy for all metrics, except for `UNHALTED_REFERENCE_CYCLES`. This is due to the fact that benchmarks are performed without fixing core frequency, and Deren, which is a much newer architecture, had much more fluctuations. However, results are presented this way due to still having a low error with varying core frequency, which is much more realistic for any other daily scenario. Moreover, Luna is found to be unable to report correct floating point operations. Luna's floating point operation reports seem to be unaffected by the executed kernels and always have $\approx 100\%$ error. Nevertheless, since Luna is older than the other baseline, this error level is acceptable and a deeper analysis is not performed.

It's found that on every host frequency rating, 1/8 achieved the best accuracy. This is on par with findings from throughput, and higher errors coming with higher frequencies are thought to be a result of losses in reported data points. However,

Dolap	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	1.01E-03	1.18E-02	8.65E-04	1.33E-03	1.35E-03	0.0625	0.0625
sum	1.16E-03	1.10E-02	1.50E-03	1.70E-03	1.80E-03	0.1249	0.1250
stream	2.32E-03	1.17E-02	1.15E-03	1.43E-03	2.26E-03	0.0833	0.0833
peakflops	1.61E-03	1.17E-02	4.27E-04	4.30E-05	4.87E-03	1.9935	2.0000
ddot	1.06E-04	1.08E-02	4.09E-04	2.66E-04	6.29E-04	0.1249	0.1250
daxpy	2.07E-03	1.16E-02	8.89E-04	1.74E-03	1.68E-03	0.0834	0.0833
Deren	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	5.80E-05	2.57E-02	2.65E-03	2.27E-03	1.80E-03	0.0624	0.0625
sum	5.63E-04	2.45E-02	4.05E-03	4.45E-03	3.90E-03	0.1246	0.1250
stream	1.76E-03	2.61E-02	3.59E-04	8.56E-04	5.79E-04	0.0831	0.0833
peakflops	8.28E-04	2.44E-02	6.15E-04	1.25E-03	4.45E-03	1.9896	2.0000
ddot	1.15E-03	2.57E-02	3.03E-03	2.85E-03	2.46E-03	0.1248	0.1250
daxpy	5.54E-04	2.48E-02	8.27E-04	2.70E-03	1.50E-03	0.0832	0.0833
Poseidon	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	1.18E-03	2.00E-03	1.08E-03	3.99E-01	1.19E-03	0.0624	0.0625
sum	9.91E-04	1.81E-03	1.20E-03	4.43E-01	1.18E-03	0.1247	0.1250
stream	1.89E-04	1.48E-03	1.55E-03	3.06E-01	2.31E-03	0.0831	0.0833
peakflops	4.41E-04	1.69E-03	1.52E-04	9.53E-04	2.48E-03	1.9942	2.0000
ddot	9.85E-04	2.10E-03	4.73E-04	2.21E-01	1.31E-03	0.1247	0.1250
daxpy	9.40E-04	2.21E-03	1.74E-04	3.07E-01	1.97E-04	0.0832	0.0833
Luna	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	1.00E+00	2.30E-02	1.53E-02	1.67E-02	1.93E-02	0.0000	0.0625
sum	1.00E+00	1.99E-02	1.56E-02	1.47E-02	1.53E-02	0.0000	0.1250
stream	1.00E+00	3.02E-02	2.80E-02	2.59E-02	2.78E-02	0.0000	0.0833
peakflops	1.00E+00	1.73E-02	1.46E-02	1.43E-02	9.14E-03	0.0000	2.0000
ddot	1.00E+00	2.34E-02	2.88E-02	2.86E-02	2.94E-02	0.0000	0.1250
daxpy	1.00E+00	2.25E-02	1.79E-02	1.69E-02	1.80E-02	0.0000	0.0833

Table 5.4 Best case scenario observed in Figure 5.5 Likwid-bench kernels are sampled with 8 metrics with frequency $8/s$, executed 10 times, and average values are reported.

to show best and worst case scenarios and to show the impact of losses in high-frequency reporting, cases yielding the best and worst accuracies broke down to kernels are presented in Table 5.4 and 5.5 respectively. It's found that, as mentioned in (Weaver et al., 2013), architectures differ in accuracy for different events. While Poseidon chronically has the highest error with `UOPs` event in Table 5.4, it exhibits the lowest error in the floating point event. Among other hosts, Dolap and Deren are found to perform consistently with high accuracy, while Luna is found to have acceptable errors on all events other than the floating point. It's also found that the accuracy of measurements could be affected by the type of kernel, as in the case of `peakflops UOPs`. Poseidon achieves a thousand times less error than other kernels. Beside, Dolap and Luna also achieve the lowest error for `UOPs`. That may imply that PMUs are more accurate when counting the same type of events for a given metric. Another important finding is that, even with the worst-case accuracy, the calculated `AI` values are accurate enough to build roofline models. Still, the losses in sampling increased all errors of all hosts 4 to 10 times.

5.4 Overhead of Measurements

To measure the overhead of the PMU profiling, `likwid-bench` kernels are executed for 10 times. The runtimes without profiling and with a different number of metrics are reported for different sampling frequencies in Figure 5.6. The only system that consistently experiences overhead from PMU sampling is found to be Luna. This is understandable since Luna is an old architecture and has poor performance. Apart from Luna, negative overheads are observed which means the overhead added by sampling is smaller than the natural variance observed between different runs of the same kernel. A similar negative overhead is also reported in (Nowak & Bitzes, 2014), even in a much bigger distributed setting. However, a meaningful skew towards positive overhead is observed with increasing frequency. That hints that, in previously presented integrity results, PCP did not just skip samplings for high frequencies. It tries to sample events; however, it could not catch up with high frequency. Still, the overhead is still very low, and negative overheads are present with runs where the frequency is 16/sec.

Dolap	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	2.05E-02	7.55E-02	3.81E-02	1.29E-01	2.62E-02	0.0628	0.625
sum	2.01E-02	7.98E-02	4.35E-02	1.31E-01	3.26E-02	0.1266	0.125
stream	2.28E-02	7.81E-02	4.93E-02	1.31E-01	3.68E-02	0.0845	0.833
peakflops	1.93E-02	7.98E-02	4.29E-02	1.33E-01	2.39E-02	2.0094	2.000
ddot	2.21E-02	7.20E-02	4.19E-02	1.34E-01	3.03E-02	0.1260	0.125
daxpy	2.01E-02	7.90E-02	4.31E-02	1.33E-01	2.97E-02	0.0841	0.833
Deren	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	5.55E-03	5.57E-02	4.08E-02	2.48E-03	2.09E-03	0.0620	0.625
sum	5.54E-03	5.51E-02	4.04E-02	2.75E-03	1.53E-03	0.1241	0.125
stream	3.94E-03	5.74E-02	4.26E-02	6.32E-03	5.58E-03	0.0825	0.833
peakflops	5.60E-03	5.40E-02	3.84E-02	6.97E-04	4.76E-03	1.9795	2.000
ddot	8.13E-03	5.33E-02	3.79E-02	9.21E-04	1.85E-03	0.1242	0.125
daxpy	4.91E-03	5.89E-02	4.19E-02	6.85E-04	2.86E-03	0.0826	0.833
Poseidon	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	8.23E-03	8.74E-02	3.95E-02	4.02E-01	3.34E-03	0.0618	0.625
sum	7.28E-03	8.17E-02	3.40E-02	4.47E-01	5.48E-03	0.1247	0.125
stream	5.08E-03	8.52E-02	4.02E-02	3.10E-01	3.10E-05	0.0829	0.833
peakflops	5.98E-03	9.07E-02	4.37E-02	5.91E-03	1.63E-02	1.9570	2.000
ddot	6.45E-03	8.13E-02	3.57E-02	2.25E-01	1.44E-03	0.124	0.125
daxpy	7.02E-03	7.97E-02	3.58E-02	3.11E-01	8.42E-03	0.0834	0.833
Luna	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	1.00E+00	1.35E-01	9.51E-02	5.57E-02	3.00E-02	0.0000	0.625
sum	1.00E+00	1.36E-01	8.68E-02	3.43E-02	3.11E-02	0.0000	0.125
stream	1.00E+00	1.24E-01	7.87E-02	2.71E-02	3.03E-02	0.0000	0.833
peakflops	1.00E+00	1.30E-01	8.35E-02	3.93E-02	3.10E-02	0.0000	2.000
ddot	1.00E+00	1.16E-01	2.97E-02	1.40E-02	8.58E-02	0.0000	0.125
daxpy	1.00E+00	1.56E-01	9.14E-02	3.84E-02	2.19E-02	0.0000	0.833

Table 5.5 Worst case scenario observed in Figure 5.5. Likwid-bench kernels are sampled with 24 metrics with frequency $16/s$, executed 10 times and average values are reported.

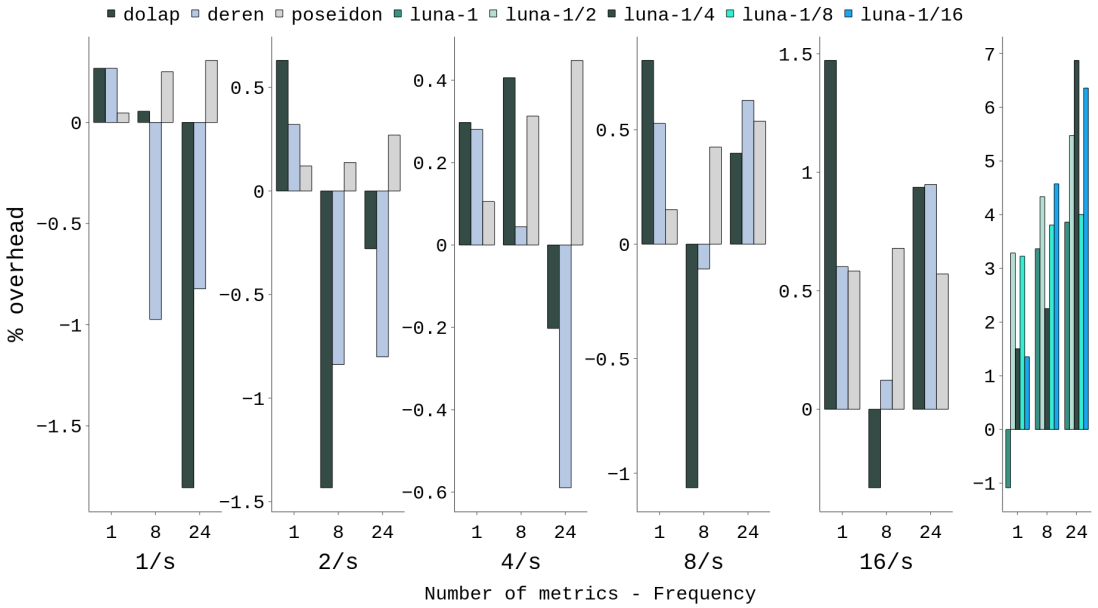

Figure 5.6 Overhead of PMU sampling on 4*systems* using PCP via SuperTwin. Values represent likwid-bench kernels triad, stream, sum, peakflops, ddot, daxpy executed 10 times each and averaged together with 1,8 and 24 metrics sampled. Comparison is againist baseline which no sampling take place.

6. Conclusions

In this work, we present the design, implementation, and tests of a high-performance computing environment digital twin called SuperTwin. During the implementation of SuperTwin, digital twin and linked time-series data approaches prove worthy of attention since, with the added augmentation and semantical query abilities, they allow for automatically generated interlinked dashboards for every individual component for a target system whose data structure is promising to be scaled to much larger systems seamlessly. SuperTwin also stays promising for straightforward integration of external tools since integrating several benchmarks such as cache-aware roofline model, STREAM, and HPCG into SuperTwin during this development process proves this capability. Moreover, the reason for these promising integration is alongside the capabilities they added to the SuperTwin, the capabilities SuperTwin added to them. For example, while cache-aware roofline adds SuperTwin's ability to generate performance models, SuperTwin adds cache-aware roofline model ability of an automated generation of performance models for different threading settings and with respect to NUMA domains, alongside with capability to mark executed kernels on generated rooflines on-the-fly. Moreover, the ability to automatically and quickly make a high volume of observations with minimal configuration and compare them, although yet to be realized, will prove worthy of architecture research and algorithm research.

On the more practical side, data structures are designed and implemented for SuperTwin, several templates of live dashboards are crafted, and data links between different modules responsible for different functionalities of SuperTwin are initiated. To this end, the plausibility of SuperTwin, and other possible digital twins that may follow similar approaches is proved. Moreover, measurements made by SuperTwin via PCP are analyzed in-depth, limitations and bottlenecks are determined, and measurements are found accurate. The generality and usefulness of the SuperTwin are verified.

7. Future Work

Although implemented and analyzed for single-node systems, SuperTwin is actually designed to work on high-performance clusters, especially for kernels that work with sparse data. To this end, SuperTwin is a project that still has much to go: Global databases, which collect observations made on different hosts systematically, automated anomaly detection, benchmark validation, and statistical studies on widely used kernels such as SpMV are the first things to complete in SuperTwin in near time. As mentioned, SuperTwin, in the future, will be scaled to cluster size systems and will become a more powerful tool with its aspects designed for clusters gaining more importance and usage, such as recursive and interlinked dashboards, component networks, and comparison modules.

BIBLIOGRAPHY

Adhianto (2010). Hpctoolkit: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput.: Pract. Exper.*, *22*(6), 685–701.

Agelastos (2014). The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. Technical Report SAND2014-19868C, Sandia National Lab. (SNL-NM), Albuquerque, NM (United States); Sandia National Lab. (SNL-CA), Livermore, CA (United States).

Aksar (2021). E2ewatch: An end-to-end anomaly diagnosis framework for production hpc systems". In *Euro-Par 2021: Parallel Processing*, (pp. 70–85)., Cham. Springer International Publishing.

Brandt (2013). Lightweight Distributed Metric Service (LDMS): Run-time Resource Utilization Monitoring. Technical Report SAND2013-6521C, Sandia National Lab. (SNL-CA), Livermore, CA (United States); Sandia National Lab. (SNL-NM), Albuquerque, NM (United States).

Chen, X. (2016). Cern analysis preservation: A novel digital library service to enable reusable and reproducible research. In *Research and Advanced Technology for Digital Libraries*, (pp. 347–356)., Cham. Springer International Publishing.

Deng (2021). A systematic review of a digital twin city: A new pattern of urban governance toward smart cities. *Journal of Management Science and Engineering*, *6*(2), 125–134.

Eitzinger, J., Gruber, T., Afzal, A., Zeiser, T., & Wellein, G. (2019). Clustercockpit — a web application for job-specific performance monitoring. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, (pp. 1–7).

Friedemann (2019). Linked data architecture for assistance and traceability in smart manufacturing. *MATEC Web of Conferences*, *304*, 04006.

Ganglia (2022). Monitoring system.

Gregg, B. (2021).

Huang, J. (2022). Nvidia to build earth-2 supercomputer to see our future.

Intel (2022). Optimizing memory bandwidth on stream triad.

Janowicz (2019). Sosa: A lightweight ontology for sensors, observations, samples, and actuators. *Journal of Web Semantics*, *56*, 1–10.

LuViVi (2019). *Developing a Dynamic Digital Twin at a Building Level: using Cambridge Campus as Case Study*, (pp. 67–75).

Marques, D., Duarte, H., Ilic, A., Sousa, L., Belenov, R., Thierry, P., & Matveev, Z. A. (2017). Performance analysis with cache-aware roofline model in intel advisor. In *2017 International Conference on High Performance Computing Simulation (HPCS)*, (pp. 898–907).

McCalpin, J. D. (1991-2007). Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia. A continually updated technical report. http://www.cs.virginia.edu/stream/.

Milenković, K. (2019). Enabling knowledge management in complex industrial processes using semantic web technology. In *Proceedings of the 2019 International*

Conference on Theory and Applications in the Knowledge Economy. 2019 International Conference on Theory and Applications in the Knowledge Economy, TAKE 2019 ; Conference date: 03-07-2019 Through 05-01-2020.

Nagios (2022). Nagios. `https://www.nagios.org/`. Accessed: 2022-12-12.

Nowak, A. & Bitzes, G. (2014). The overhead of profiling using PMU hardware counters.

Red-Hat (2021). 1. programming performance co-pilot.

Roy (2015). Perfaugur: Robust diagnostics for performance anomalies in cloud services. In *2015 IEEE 31st International Conference on Data Engineering*, (pp. 1167–1178).

Röhl, T., Eitzinger, J., Hager, G., & Wellein, G. (2017). Likwid monitoring stack: A flexible framework enabling job specific performance monitoring for the masses. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, (pp. 781–784).

Shahat (2021). City digital twin potentials: A review and research agenda. *Sustainability*, *13*(6).

Steinmetz (2018). Internet of things ontology for digital twin in cyber physical systems. In *2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*, (pp. 154–159).

Weaver, V., Terpstra, D., & Moore, S. (2013). Non-determinism and overcount on modern hardware performance counter implementations. (pp. 215–224).

Williams, S. (2009). Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, *52*(4), 65–76.

Xin (2018). Cross-linking biothings apis through json-ld to facilitate knowledge exploration. *BMC Bioinformatics*, *19*.

APPENDIX A

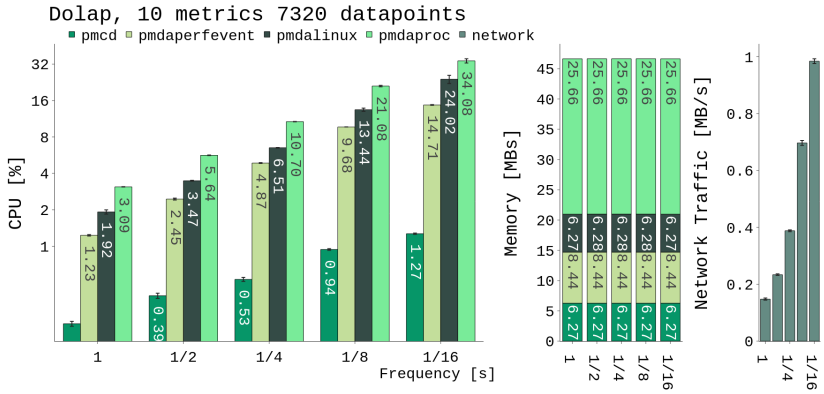

Figure A.1 System resource usage of metric shipment with both kernel and PMU metrics on Dolap. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 4, 3, 3 metrics and 264, 180, 6876 data points respectively.

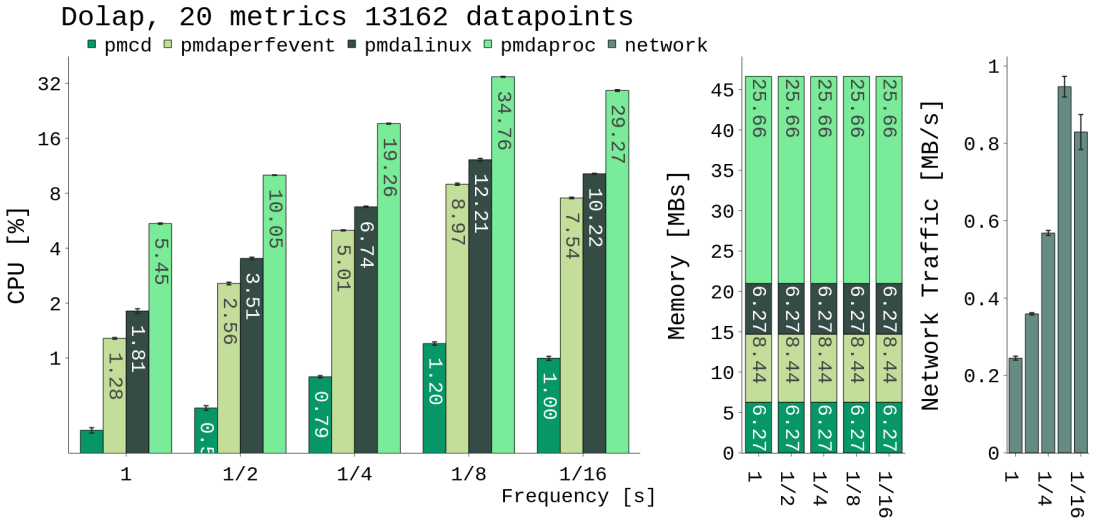

Figure A.2 System resource usage of metric shipment with both kernel and PMU metrics on Dolap. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 8, 6, 6 metrics and 528, 185, 12449 data points respectively.

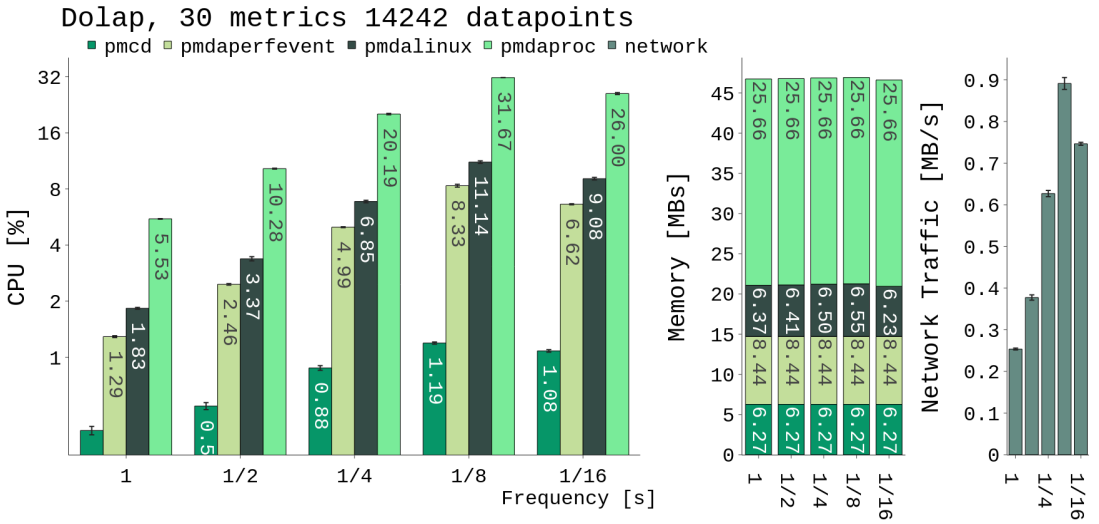

Figure A.3 System resource usage of metric shipment with both kernel and PMU metrics on Dolap. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 12, 12, 6 metrics and 1056, 189, 12997 data points respectively.

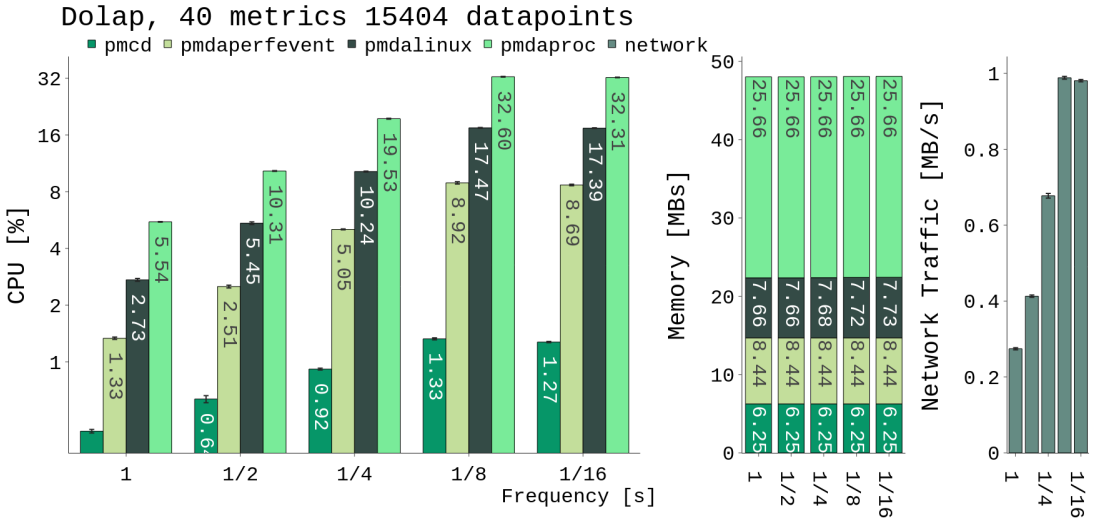

Figure A.4 System resource usage of metric shipment with both kernel and PMU metrics on Dolap. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 18, 16, 6 metrics and 1584, 281, 13539 data points respectively.

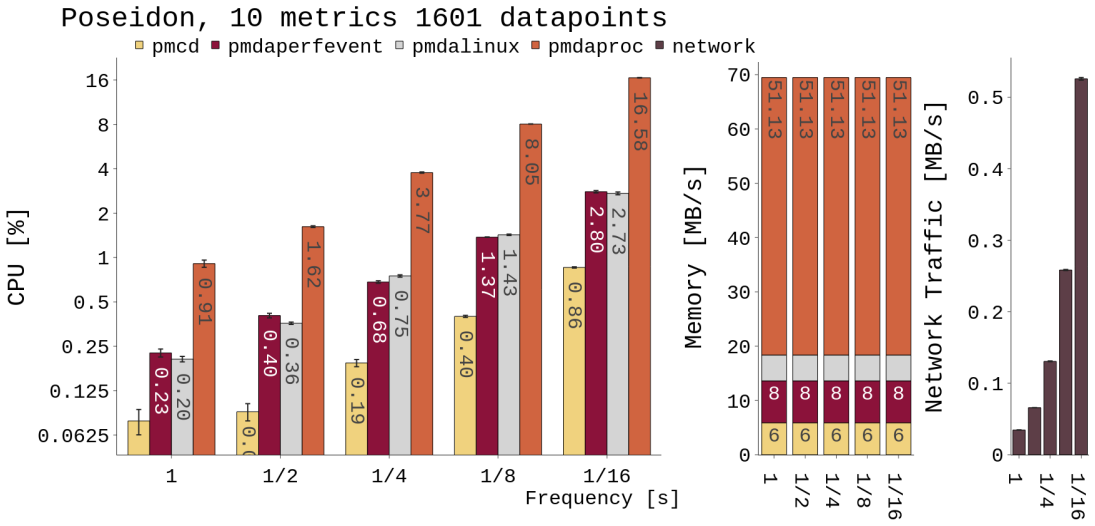

Figure A.5 System resource usage of metric shipment with both kernel and PMU metrics on Poseidon. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 3, 3, 4 metrics and 48, 35, 1528 data points respectively.

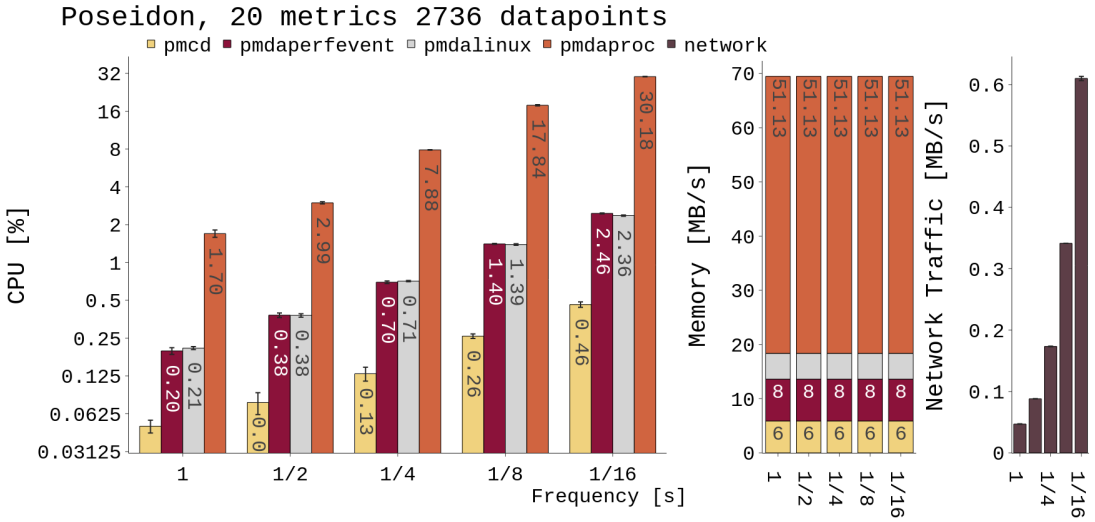

Figure A.6 System resource usage of metric shipment with both kernel and PMU metrics on Poseidon. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 6, 8, 6 metrics and 96, 40, 2600 data points respectively.

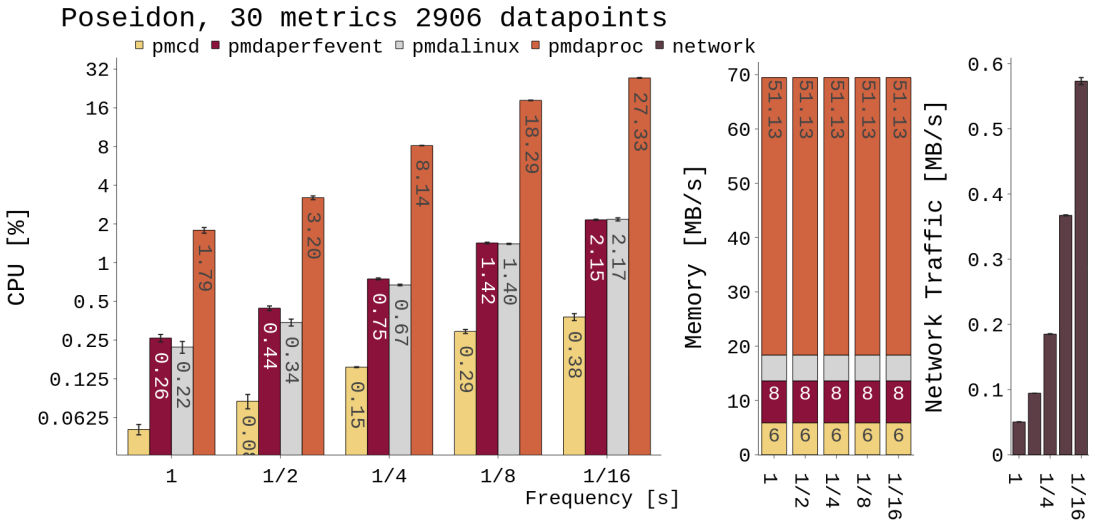

Figure A.7 System resource usage of metric shipment with both kernel and PMU metrics on Poseidon. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 12, 12, 6 metrics and 192, 44, 2670 data points respectively.

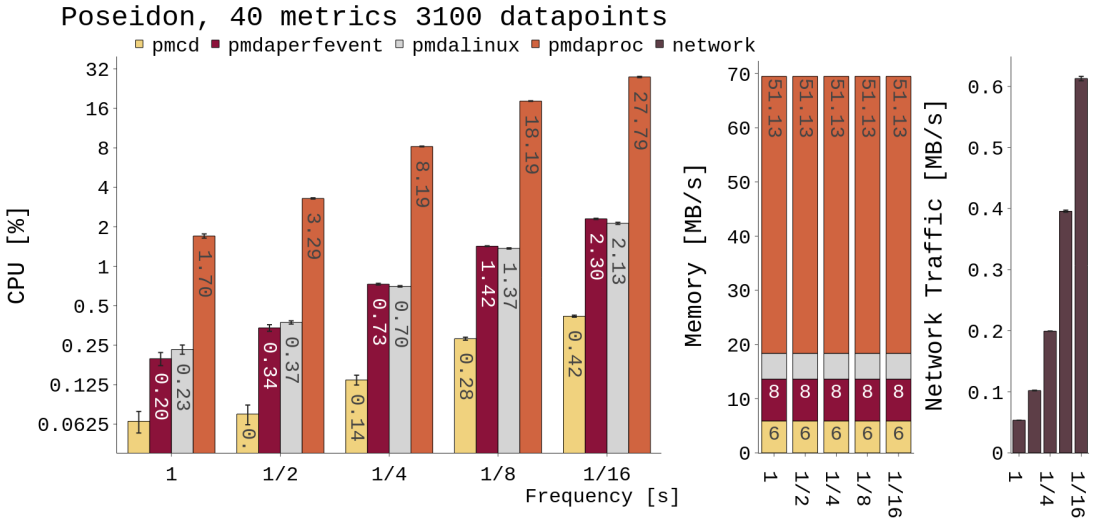

Figure A.8 System resource usage of metric shipment with both kernel and PMU metrics on Poseidon. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 18, 16, 6 metrics and 288, 56, 2756 data points respectively.

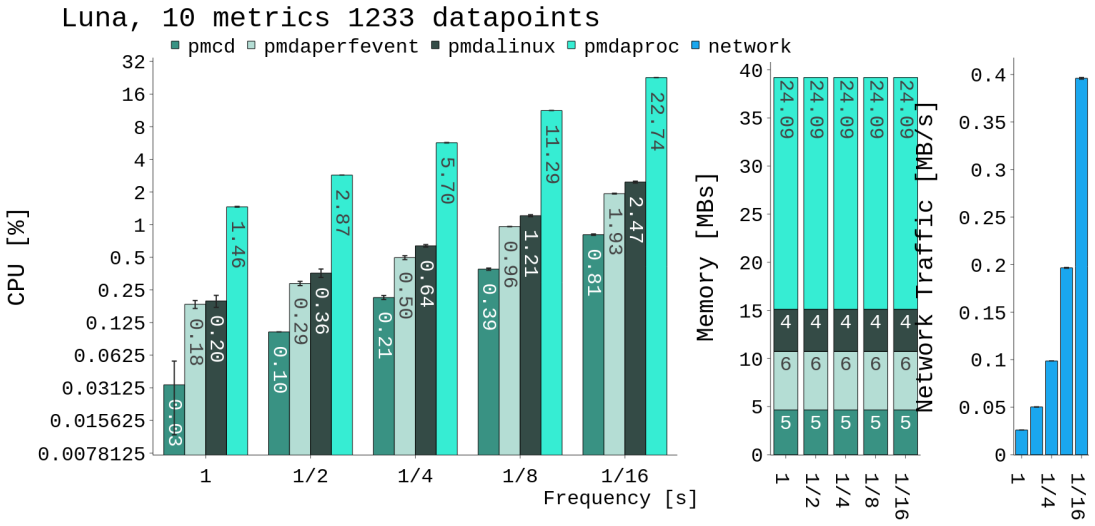

Figure A.9 System resource usage of metric shipment with both kernel and PMU metrics on Luna. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 3, 3, 4 metrics and 36, 31, 1152 data points respectively.

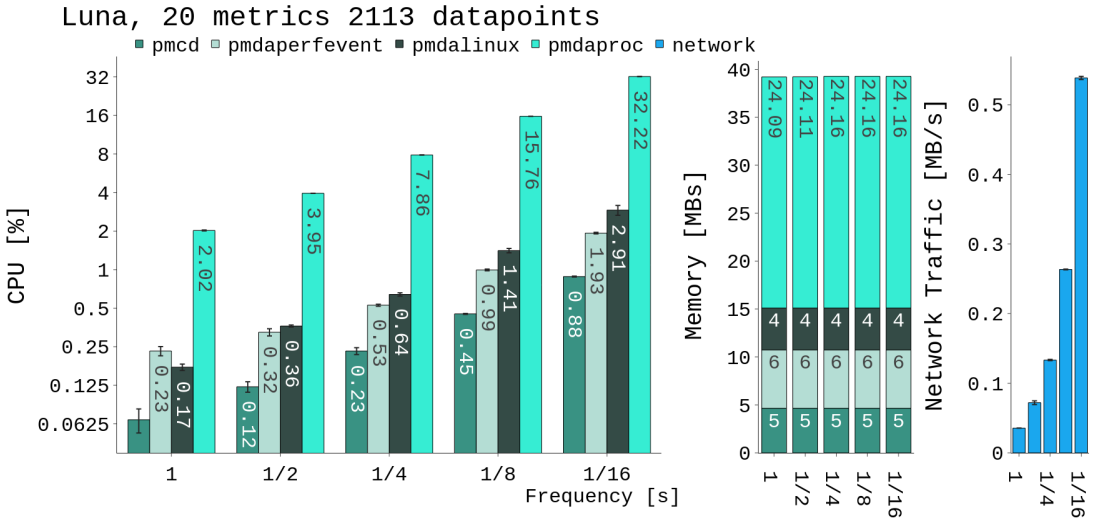

Figure A.10 System resource usage of metric shipment with both kernel and PMU metrics on Luna. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 6, 8, 6 metrics and 72, 36, 1990 datapoints respectively.

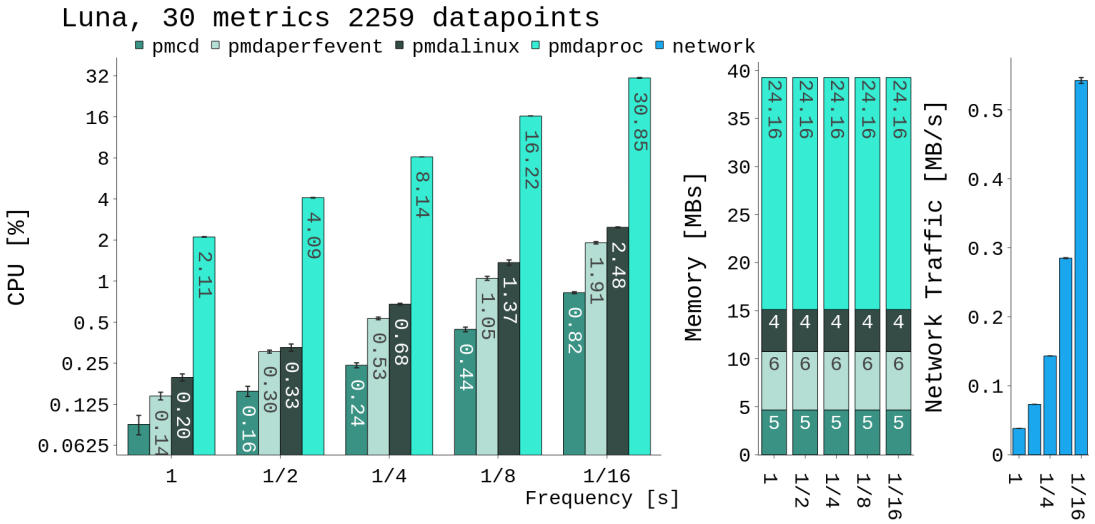

Figure A.11 System resource usage of metric shipment with both kernel and PMU metrics on Luna. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 12, 12, 6 metrics and 144, 40, 2065 data points respectively.

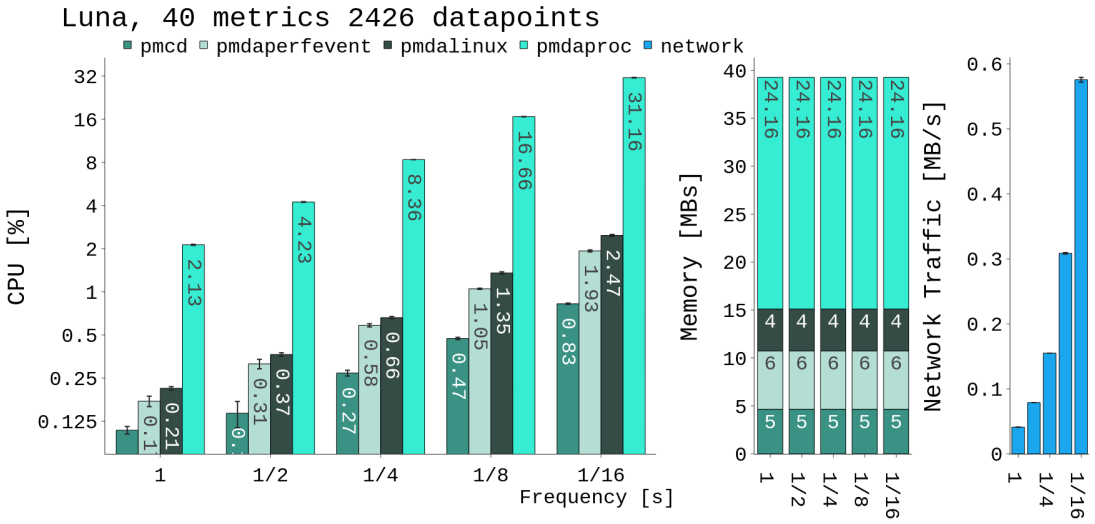

Figure A.12 System resource usage of metric shipment with both kernel and PMU metrics on Luna. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 18, 16, 6 metrics and 216, 48, 2130 data points respectively.

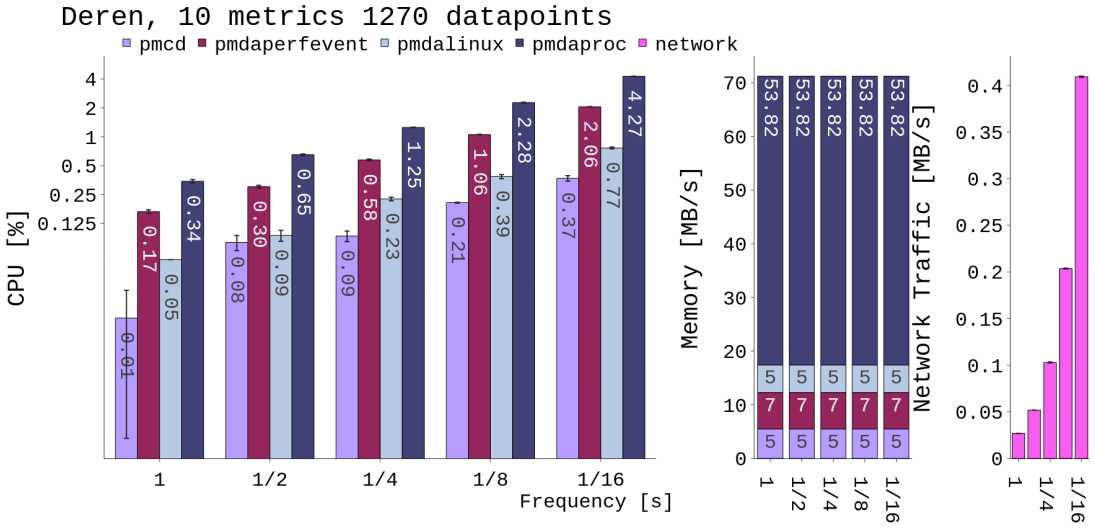

Figure A.13 System resource usage of metric shipment with both kernel and PMU metrics on Deren. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 3, 3, 4 metrics and 24, 19, 1227 data points respectively.

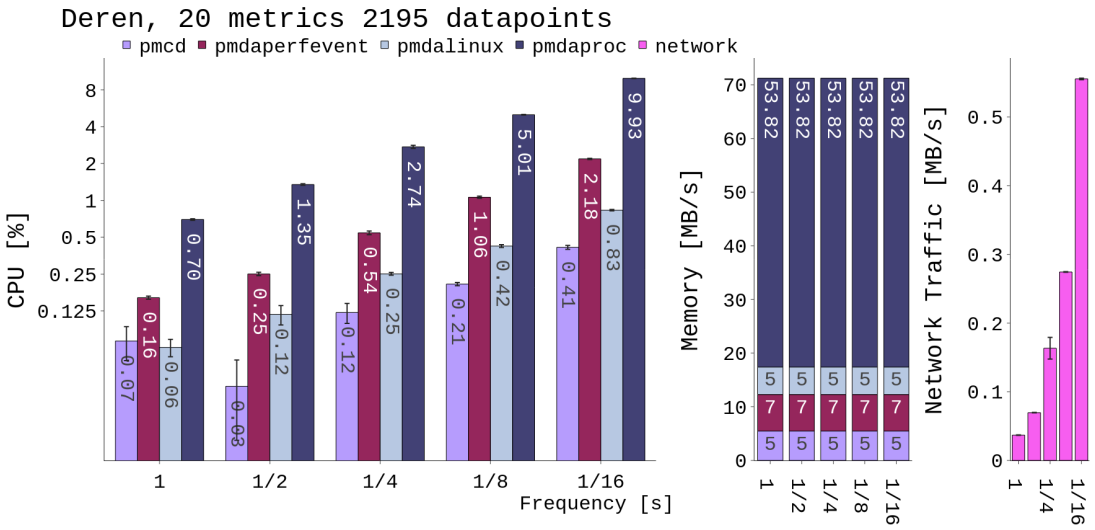

Figure A.14 System resource usage of metric shipment with both kernel and PMU metrics on Deren. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 6, 8, 6 metrics and 48, 24, 2123 data points respectively.

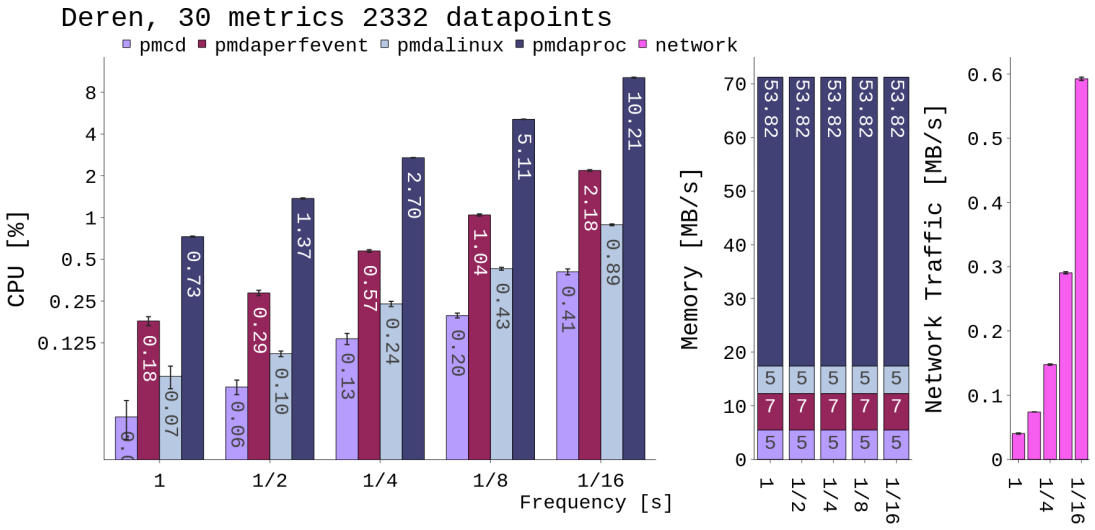

Figure A.15 System resource usage of metric shipment with both kernel and PMU metrics on Deren. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 12, 12, 6 metrics and 96, 28, 2208 data points respectively.

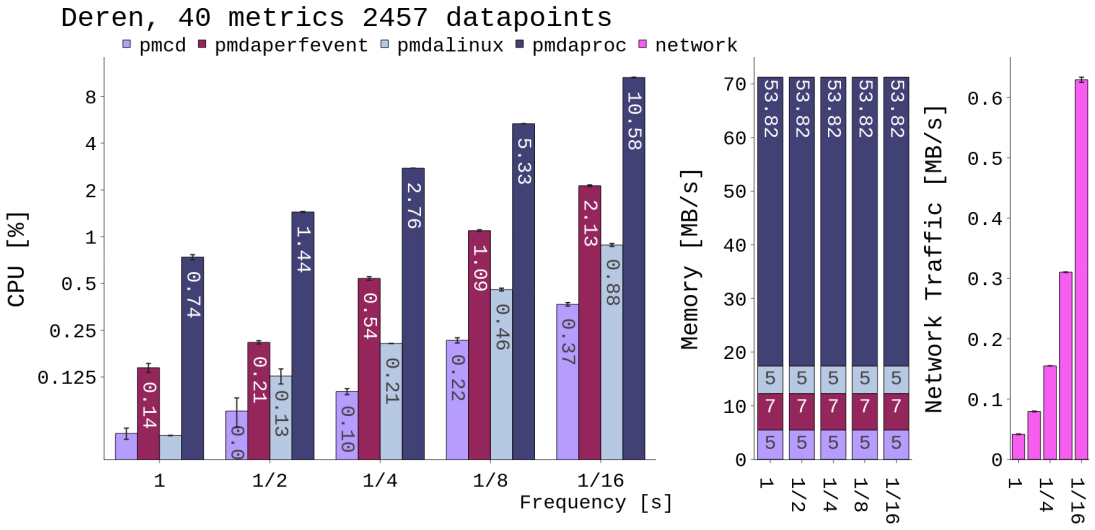

Figure A.16 System resource usage of metric shipment with both kernel and PMU metrics on Deren. Metric agents `pmdaperfevent`, `pmdalinux` and `pmdaproc` reports 18, 16, 6 metrics and 144, 36, 2277 data points respectively.

APPENDIX B

Metrics used for resource usage experiments. Lists are cumulative.

10 Metrics
#perfevent
UNHALTED_CORE_CYCLES
UNHALTED_REFERENCE_CYCLES
INSTRUCTION_RETIRED

#proc
proc.psinfo.rss
proc.psinfo.utime
proc.psinfo.stime
network.all.out.bytes

#linux kernel.percpu.cpu.idle
hinv.cpu.clock
disk.dev.read

20 Metrics
#perfevent
LLC_REFERENCES LLC_MISSES
MISPREDICTED_BRANCH_RETIRED

#proc
proc.psinfo.nvctxsw
proc.psinfo.vctxsw

#linux mem.util.used
mem.util.free
mem.util.bufmem
mem.util.cache
mem.util.other

30 Metrics

#perfevent
BACLEARS_ANY
L1D_REPLACEMENT
L2_LINES_IN_ALL
L2_TRANS_L2_WB
MEM_LOAD_RETIRED_L1_HIT
MEM_LOAD_RETIRED_L1_MISS

#linux mem.util.active
mem.util.inactive
mem.util.swapTotal
mem.util.swapFree

40 Metrics
#perfevent
UOPS_DISPATCHED_PORT:PORT_0
UOPS_DISPATCHED_PORT:PORT_1
UOPS_DISPATCHED_PORT:PORT_5
FP_ARITH:SCALAR_DOUBLE
FP_ARITH:SCALAR_SINGLE
FP_ARITH:128B_PACKED_SINGLE

#linux
network.interface.in.bytes
network.interface.in.packets
network.interface.in.errors
network.interface.in.drops

50 Metrics
#perfevent
FP_ARITH_128B_PACKED_SINGLE
FP_ARITH_128B_PACKED_DOUBLE
FP_ARITH_256B_PACKED_SINGLE
FP_ARITH_256B_PACKED_DOUBLE
MEM_INST_RETIRED_ALL_LOADS.
MEM_INST_RETIRED_ALL_STORES
UOPS_ISSUED_ANY

#linux
network.interface.in.drops
mem.vmstat.kswapd_low_wmark_hit_quickly
mem.vmstat.kswapd_high_wmark_hit_quickly
mem.vmstat.nr_active_file
mem.vmstat.nr_dirty_background_threshold

APPENDIX C

Metrics used for accuracy, overhead and throughput experiments. First x metrics are used in experiment with x metrics

FP_ARITH_SCALAR_DOUBLE
MEM_UOPS_RETIRED_ALL_LOADS
MEM_UOPS_RETIRED_ALL_STORES
INSTRUCTION_RETIRED
UNHALTED_REFERENCE_CYCLES
UOPS_RETIRED_ANY
LLC_REFERENCES
LLC_MISSES
UNHALTED_CORE_CYCLES
MISPREDICTED_BRANCH_RETIRED
BACLEARS_ANY
L1D_REPLACEMENT
L2_LINES_IN_ALL
L2_TRANS_L2_WB
MEM_LOAD_RETIRED_L1_HIT
MEM_LOAD_RETIRED_L1_MISS
UOPS_DISPATCHED_PORT_PORT_0
UOPS_DISPATCHED_PORT_PORT_1
UOPS_DISPATCHED_PORT_PORT_5
FP_ARITH_SCALAR_SINGLE
FP_ARITH_128B_PACKED_SINGLE
FP_ARITH_128B_PACKED_DOUBLE
FP_ARITH_256B_PACKED_SINGLE