TECHNISCHE UNIVERSITÄT MÜNCHEN

FAKULTÄT FÜR INFORMATIK

Holistic and Portable Operational Data Analytics on Production HPC Systems

Alessio Netti

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:

Prof. Dr. Hans-Joachim Bungartz

Prüfende der Dissertation:

1. Prof. Dr. Martin Schulz

2. Prof. Ozalp Babaoglu

Die Dissertation wurde am 15.09.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 25.01.2022 angenommen.

Acknowledgments

I'd like to use this space to thank all of the people that helped me — in a way or another — along the way to this dissertation. First, I'd like to acknowledge the European Commission for supporting my work with the Horizon 2020/EuroHPC research and innovation programme, in the context of the DEEP-EST (grant agreement No. 754304) and REGALE (grant agreement No. 956560) projects. I would also like to thank all of the people involved in these two projects, with whom I had the chance to collaborate and exchange ideas.

I'd like to thank my supervisor, Prof. Dr. Martin Schulz, who provided me with precious guidance throughout my doctorate, allowing me to grow as a researcher. I would also like to thank Michael Ott, Daniele Tafani and Carla Guillen: I could have never asked for better colleagues, and I'm happy I was able to be a part of the Data Center Data Base project. My thanks also go to all of my colleagues at the Leibniz Supercomputing Centre for the exciting work in these three years, and I'm grateful I was able to stand at the cutting edge of technology with them. I'd like to thank Prof. Ozalp Babaoglu and Prof. Zeynep Kiziltan from the Department of Computer Science and Engineering at the University of Bologna for our valuable collaborations throughout the years, as well as my colleagues at the CAPS chair of the Technical University of Munich for all the insightful discussions we had. Last but not least, I want to thank the members of the Operational Data Analytics team in the Energy Efficient HPC Working Group, with whom I had the pleasure to collaborate.

I'd like to express my gratitude to Julia, thanks to whom all of my days are a bit happier. I'd like to thank my family, that has never failed to support me, as well as all of my friends from Naples, Bologna and Munich alike. At last, my deepest thanks — although not for professional reasons — go to Aaron. You have been a good boy.

Abstract

High-Performance Computing (HPC) systems are one of the pillars of modern science. Their size and complexity continues to increase as we approach the *exascale* era, with systems capable of computing 10^{18} operations per second, raising concerns about long-term manageability and sustainability. For this reason, there is a growing interest in taking control of the reliability and efficiency of a data center's own operations: fine-grained *monitoring* and *Operational Data Analytics* (ODA) are fundamental tools in this endeavor. Monitoring consists of collecting sensor data from instrumented hardware and software components, enabling observability over the behavior of systems and facilities, while ODA leverages sensor data to extract high-level, actionable knowledge that can ultimately be used to drive control decisions and optimize operations. However, while monitoring is an established reality, the gap between ODA research and its use in production environments is still wide and largely unaddressed. This leads to insular, ad-hoc solutions that target only specific aspects of ODA according to a site's needs, hindering the sharing of knowledge across institutions.

With the work in this dissertation we enable the adoption of arbitrary ODA techniques in production data center environments, in a holistic and portable way. We tackle this problem from three fronts: first, we adopt the *Data Center Data Base* (DCDB) monitoring framework as a holistic foundation for the collection of sensor data; we were among the main contributors of DCDB, extending its design and rendering it suitable for large-scale production deployments. Secondly, we propose *Wintermute*, a generic ODA framework that is designed to be integrated into any monitoring system and that is able to cover the vast majority of ODA scenarios, both for visualization and control purposes. In the context of this dissertation, we integrated Wintermute in DCDB. Third, we propose the *Correlation-wise Smoothing* (CS) method to process high-dimensionality monitoring sensor data and produce data representations that are compact, effective, interpretable and portable across systems. The combination of these three components is an end-to-end generic ODA platform: we demonstrate its effectiveness with two distinct deployments on large-scale production HPC systems, each covering different use cases, as well as with a variety of tailored experiments. Moreover, we discuss the analysis of requirements at the base of our designs as well as the lessons learned from our long-term production experiences, providing a cornerstone for ODA design and paving the way for holistic ODA as a foundation of data center operations.

Zusammenfassung

High-Performance-Computing (HPC) Systeme sind eine der tragenden Säulen der modernen Wissenschaft. Auf dem Weg in die *Exascale*-Ära mit Systemen, die 10^{18} Rechenoperationen pro Sekunde ausführen können, nimmt deren Größe und Komplexität stetig zu und Bedenken hinsichtlich der langfristigen Verwaltbarkeit und Nachhaltigkeit solcher Systeme treten in den Vordergrund. Aus diesem Grund wächst das Interesse daran, die Zuverlässigkeit und Effizienz des eigentlichen Rechenzentrumbetriebs besser zu kontrollieren: *Monitoring* und *Operational Data Analytics* (ODA) stellen grundlegende Werkzeuge in diesem Bestreben dar. Beim Monitoring werden Sensordaten von instrumentierten Hardware- und Softwarekomponenten erfasst, um das Verhalten von Systemen und Anlagen beobachten zu können, während ODA Sensordaten nutzt, um auf höhrerer Ebene anwendbares Wissen zu gewinnen, das letztendlich für Steuerungsentscheidungen und die Optimierung des Betriebs verwendet werden kann. Während Monitoring etablierte Praxis ist, klafft zwischen der ODA-Forschung und ihrem Einsatz in Produktionsumgebungen noch immer eine große Lücke. Dies führt zu isolierten Ad-hoc-Lösungen, die je nach den Bedürfnissen des jeweiligen Rechenzentrums nur auf bestimmte Aspekte von ODA abzielen, was den Wissensaustausch zwischen den Einrichtungen behindert.

Mit der Arbeit in dieser Dissertation wird die Einführung umfangreicher ODA-Techniken in Produktionsrechenzentrumsumgebungen ermöglicht, und zwar auf ganzheitliche und portierbare Weise. Das Problem wird von drei Seiten angegangen: Erstens wird das *Data Center Data Base* (DCDB) Monitoring-Framework als ganzheitliche Grundlage für die Erfassung von Sensordaten eingesetzt. DCDB wurde im Rahmen dieser Arbeit stark erweitert und für den Einsatz in großen Produktionssystemen optimiert. Zweitens wird *Wintermute* eingeführt, ein generisches ODA-Framework, das so konzipiert ist, dass es in jedes beliebige Monitoring-Framework integriert werden kann und die große Mehrheit an ODA-Szenarien abdeckt, sowohl für Visualisierungs- als auch für Steuerungszwecke. Im Rahmen dieser Dissertation wurde Wintermute in DCDB integriert. Drittens wird die *Correlation-wise Smoothing* (CS) Methode vorgeschlagen, um hochdimensionale Sensordaten zu verarbeiten und Datenrepräsentationen zu erzeugen, die kompakt, effektiv, interpretierbar und über Systemgrenzen hinweg portierbar sind. Die Kombination dieser drei Komponenten ergibt eine durchgängige, generische ODA-Plattform: Ihre Effektivität wird durch den Einsatz auf zwei großen Produktions-HPC-Systemen demonstriert, wo sie unterschiedliche Anwendungsfälle abdeckt, sowie mit einer Vielzahl von maßgeschneiderten Experimenten. Darüber hinaus werden die Anforderungen, die den Entwürfen zugrunde liegen, sowie die Lehren, die aus langjährigen Produktionserfahrungen gezogen werden konnten, erörtert. Damit wird ein Eckpfeiler für das ODA-Design aufgestellt und der Weg für ganzheitliches ODA als Grundlage des Rechenzentrumsbetriebs geebnet.

Contents

1. Introduction

High-Performance Computing (HPC) is key to many fields of modern science, with large-scale systems that are capable of handling computational problems of extreme complexity and size. Due to the their ever-increasing scale, modern HPC systems pose relevant challenges in terms of energy efficiency and reliability: this chapter opens the dissertation, discussing the complexity of operating modern HPC systems as well as the tools developed by the community to mitigate it. In Section 1.1 we give a brief overview of the HPC field, while in Section 1.2 we introduce the concepts of *monitoring* and *Operational Data Analytics* (ODA). Finally, in Section 1.3 we present the contributions of this dissertation, and in Section 1.4 we describe its organization.

1.1. High-Performance Computing

In this section we discuss the state of the art in the HPC field in terms of systems, trends and technologies, alongside the main concerns and limitations. These provide the main motivations for our scientific contributions.

1.1.1. HPC in Modern Science

Simulation and computation are considered the third pillar of modern science, alongside theory and experimentation [WS16]. A significant contributor to this achievement is HPC, a branch of computer science that uses *supercomputers* to solve computationally complex problems that would otherwise be unapproachable. Today's supercomputers are large-scale, distributed systems that use of thousands of servers to run parallel scientific *applications*. HPC has been applied to a wide variety of fields, ranging from meteorology to particle physics [Ash+10], and it is one of the main drivers behind modern scientific discovery, similarly to microscopes in the previous century [BS18].

The HPC field is renowned for its continuous strife to achieve better computational performance: the current goal of the HPC community is *exascale* performance, which consists in the ability to compute 10^{18}Flop/s, or one billion billion floating point operations per second; it is expected that systems at this scale will be attained by 2022 [Ste+19a]. Figure 1.1 gives an idea of the *arms race* in the HPC sector, showing the performance of the 1st and 500th HPC systems in the Top500 ranking [1] over time, as well as the cumulative performance of all HPC systems on the list. The increase in performance is steady and relentless, starting from 1GFlop/s in 1993 and reaching 0.5EFlop/s in 2020 - an increase corresponding to 9 orders of magnitude. In just 2 years, exascale will bring about a series of transformational changes in many fields of science, due to the sheer

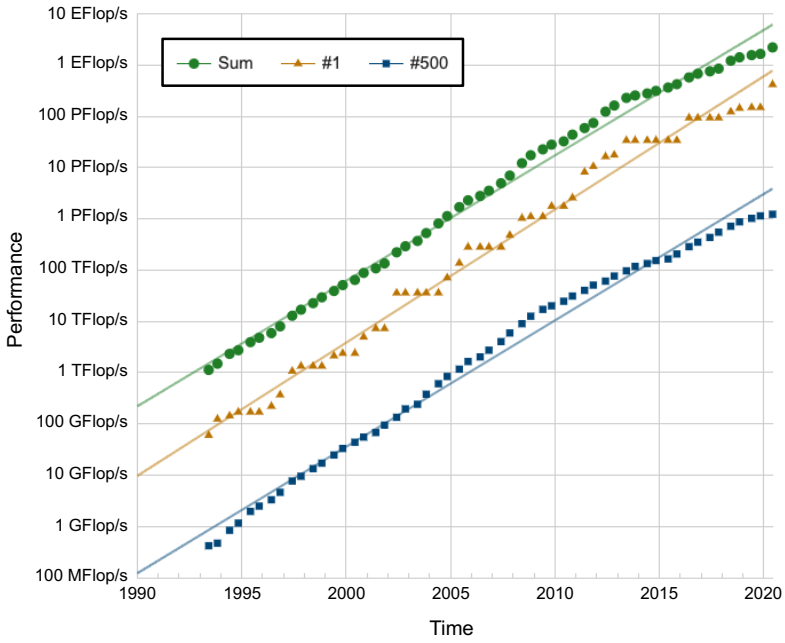


Figure 1.1.: The performance evolution of HPC systems from 1995 to 2020, as registered by the Top500 lists [1].

scale of the predictive simulations enabled by this new technology [Whi11]. In general, HPC centers represent performance-critical, tightly-coupled versions of traditional data centers, sharing many similarities to the latter [Net+18b]. Therefore, most of the challenges that are typical in the HPC world can be generalized and applied to the wider data center domain, which is no less relevant than HPC: with their facilities containing thousands of loosely-coupled servers, data centers fuel modern businesses, the Internet itself and a significant portion of the modern economy.

1.1.2. Architecture of an HPC System

In Figure 1.2 we show a basic overview of an HPC system's operation, alongside the main actors involved. For the sake of simplicity we do not show infrastructure components, such as power supplies or cooling units. At its core, an HPC system is composed of several hundreds (or thousands) of *compute nodes*, which are servers supplying raw computational capabilities. Compute nodes may be equipped with CPUs alone, or may integrate accelerators such as GPUs or FPGAs for increased performance. If an HPC system's nodes include multiple types of computational resources, it is defined as *heterogeneous*, and as *homogeneous* otherwise. Depending on the system's architecture and their physical placement, compute nodes can be organized hierarchically in *chassis*, *racks* and, in the case of large-scale systems, *islands* or *partitions*. Compute nodes are connected to one another by an *interconnect*, which is a high-speed fabric granting network access to the entire system, organized according to a certain topology. In some

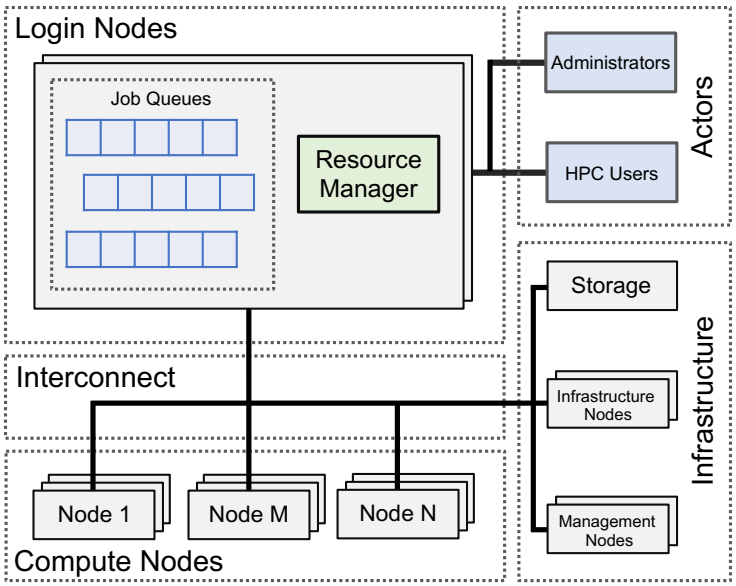


Figure 1.2.: Overview of the main components and actors in an HPC system.

cases, HPC systems may use more than a single interconnect (e.g., for management purposes). *Storage* is also attached to the interconnect and exposed in the form of a distributed file system. Moreover, we find *management nodes*, which are used to carry out ordinary management tasks (e.g., health checking or installation of compute nodes), as well as *infrastructure nodes*, which expose interfaces for control and monitoring of certain components (e.g., network switches or cooling systems).

Access to the entire system is arbitrated by a series of *login nodes*, which are special management nodes that act as frontend machines exposed to the Internet. Here, a *resource manager* software is always running: *HPC users* who wish to run their scientific applications submit *jobs* to the resource manager, usually in the form of shell scripts. Each submitted job is then added by the resource manager to a certain waiting queue, depending on user-specified *Quality of Service* (QoS) constraints. The resource manager serves jobs in the queue by performing *scheduling* (i.e., assigning a start time) and *allocation* (i.e., assigning a set of nodes) for them, keeping track of their status and output. *System administrators* usually enter the system either via the same access nodes, with higher privilege levels, or via dedicated administration nodes. An HPC system is said to be in *production* when it supplies a continuous service to its users, who can run critical applications with certain QoS guarantees and with minimal downtime.

1.1.3. Sustainability Concerns

It is expected that exascale HPC performance will be a reality by 2022, when the first system of this kind, named *Frontier*, will enter production at the *Oak Ridge National Laboratory* (ORNL) [2]. As it is not clear yet which will be the energy demands for such a

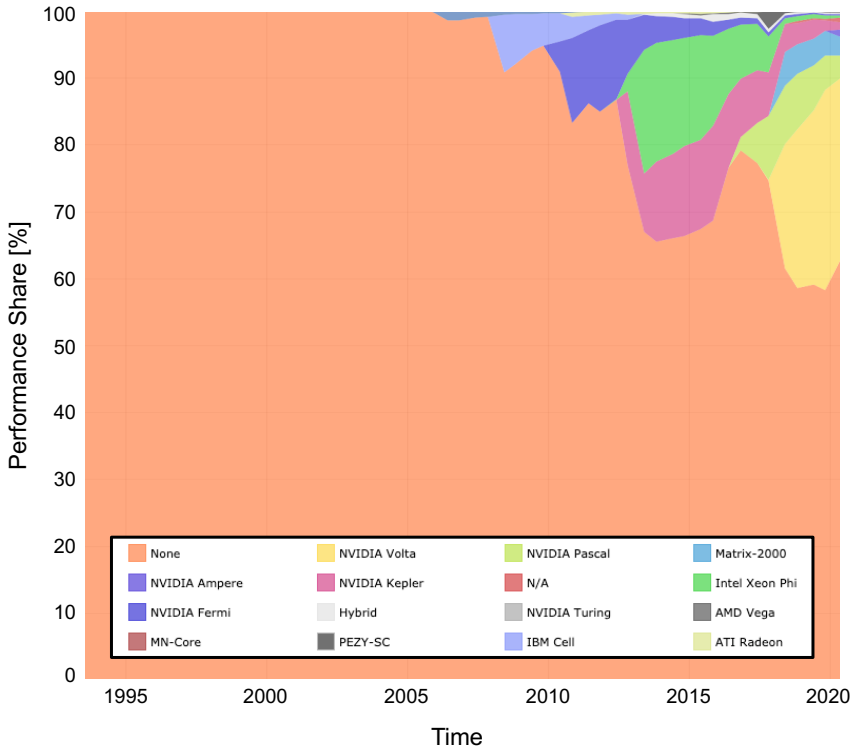


Figure 1.3.: The performance share of HPC systems with certain accelerator types over time, as registered by the Top500 lists [1].

system under sustained operation, we can look at the *Fugaku* HPC system at the *RIKEN Center for Computational Science*, located in Japan: as of June 2021, it is the most powerful HPC system in the world [3] with 442PFlop/s of measured peak performance, thus falling in the same order of magnitude as future exascale systems. Fugaku consumes 28MW of power under peak load, which is a roughly linear increase over *Summit*, the previous owner of the Top500 1st place with 148PFlop/s at 10MW. Energy consumption at this scale is unaffordable for most countries due to limits in the electrical infrastructure and pricing of electricity, as well as due to the general complexity of managing such a demanding system. For these reasons, in 2014 Villa et al. theorized that an exascale HPC system should consume no more than 35MW under peak load [Vil+14]: reaching this goal would require an improvement in the energy efficiency of HPC systems like Summit and Fugaku by a factor of 2.

Energy efficiency is not the sole concern associated with the sustainability of exascale HPC systems. Resiliency is a second key aspect: standing to established failure studies [GC15] exascale HPC systems, comprised of thousands compute nodes and hundreds of thousands CPUs, will exhibit failures every few minutes [Cap+14]. This not only demands for better mechanisms to prevent and mask failures, but also better resiliency of user applications, which will have to rely on running at very large scales in order to achieve exascale performance. On a similar note, performance variation among HPC

components due to manufacturing processes is another concerning aspect [Ina+15] as few, under-performing CPUs or compute nodes might negatively affect the performance of a large-scale application.

The concerns listed above are exacerbated by the fact that, due to limits in current silicon technology and the sunset of Moore's law [Iwa16], most modern leadership-class HPC systems are heterogeneous, employing accelerators such as GPUs, FPGAs or many-core processors in order to boost their performance [Vet+18]. This is shown in Figure 1.3, which depicts the share of HPC systems with certain accelerator types (or none) from 1995 to 2020. As of now, almost 50% of the performance delivered by HPC systems in the Top500 list is supplied by accelerators, adding a new layer of complexity in the management and coordination of such resources from an operational standpoint. The outlook for post-exascale and *zettascale* HPC systems is not clear at this point in time [DH21; Lia+18]; however, it is agreed that sustainability will increasingly become a vital aspect in the development of new HPC systems. Continuous and proactive orchestration and optimization of a system's operation, at all levels of its architecture, is the only way forward [BS18; HN18].

1.2. Monitoring and Operational Data Analytics

Monitoring and Operational Data Analytics (ODA) are useful tools for improving the operation of data center and HPC systems and, in turn, their sustainability. Here, we provide an overview of how they can be used in a production environment.

1.2.1. Monitoring

Monitoring consists in the collection of various types of data about a system's operation, providing information about resource usage and system events, among others. Monitoring data is usually collected from the components of interest (e.g., compute nodes), is propagated throughout a distributed architecture and is then stored in a data base [Age+14]. System administrators can then leverage this data to gain insight over the operation of a system or an entire facility, and take appropriate actions when deemed necessary. Monitoring data can also be useful to the users of a system: HPC users, for example, can exploit monitoring data to understand the behavior of their own applications and thus improve them [GHB14].

Effective and transparent monitoring does not come for free and poses a series of relevant challenges [Bra+16; Ahl+18]. A large-scale distributed system, be it from an HPC site or data center, will be composed of thousands of individual compute nodes, each exposing in turn up to thousands of data sources that can be used for various purposes. This can lead to up to millions of data points being collected every second: this volume of data is difficult to manage from a *storage* perspective (several TBs of data may be collected for a single day of operation [Rao+20]), as well as from the *scalability* (large data volumes require scalable frameworks) and *performance* (monitoring must not

impact the system's operation) standpoints. Because of this, monitoring is a very active research field and not all research questions have been exhausted, especially concerning the monitoring of heterogeneous data sources on large-scale systems at a very fine time granularity.

Features of Monitoring Data

Monitoring encompasses a wide range of data sources, making integration in a single framework difficult [Gim+17; Net+19b]. These provide monitoring data in diverse formats, each having different feature and semantics. The following are the main formats usually associated with monitoring data:

- **Sensor Data**: this is the most common format for monitoring data. Here, each data entry is associated to a sensor (e.g., power or temperature) and is strictly numerical. As such, it can be represented as a time-stamp and value pair, generating a time series [Bra+16].
- **Log Data**: this type of data comes in text format. Log entries are usually recorded anytime a certain event occurs (e.g., an error is detected in a compute node) and, due to the information they carry, they cannot be represented as ordinary sensors [Vaz+17].
- **Scheduling Data**: this type of data is specific to the HPC domain, and describes the scheduling and allocation decisions made by a resource manager for a certain HPC system. Its format is usually structured and comprises a mix of text and numerical fields, describing the start and end time of user jobs, the allocated nodes and other similar information [Wya+18].
- **Application Data**: HPC-specific data that is associated to user applications. Depending on its purpose it can be in a log-like text form (e.g., indicating application phase transitions) or in numerical form (e.g., counting the references to a certain function). In all cases, this data is usually sampled at a very fine time scale (down to microseconds), making it unfit for persistent storage [Boe+16].

In the context of this dissertation we focus on sensor data, since it is the most common type of monitoring data and it has the broadest applicability - hence, our contributions are tailored towards the requirements and characteristics of this data type. As described earlier, sensor data has a numerical format, with each reading being comprised of a time-stamp and value, thus creating a time series. The data associated to a sensor has two main defining features: first, it has a well-defined *sampling interval*, which describes how often the sensor is read and thus the granularity of the data. In data center environments, the sampling interval can range between minutes (e.g., for infrastructure data) down to milliseconds (e.g., for CPU-related compute node data). Additionally, a sensor may have a *unit of measurement* associated to it, which defines the physical quantity being measured (e.g., Joules for an energy sensor). This is not strictly required, and many

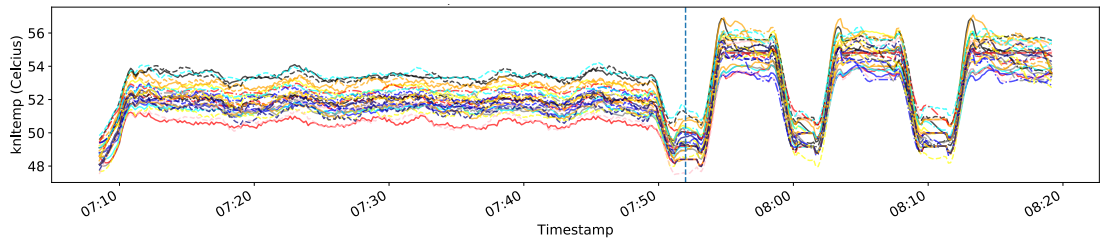


Figure 1.4.: Temperature sensor readings from several HPC compute nodes [Oze+20].

sensor types (e.g., CPU performance counters) are unitless. A visual example of a sensor time series is shown in Figure 1.4, which depicts the temperature of several compute nodes in an HPC system while running the same parallel application.

Within sensor data, an additional distinction can be made regarding the type of sampling: as *in-band*, a sensor is sampled from the same component on which it resides. This could correspond, for example, to sampling the CPU instructions of a compute node from the node itself [Wea13]. As *out-of-band*, instead, a sensor is sampled from a separate server. This is the case, for example, when the power consumption of a compute node is sampled from a separate server by querying the associated *Board Management Controller* (BMC) [LBB18]. In-band sampling is usually more delicate compared to out-of-band sampling, since it affects the performance of the sampled component to a greater extent; however, the direct availability of data guarantees better liveness and lower latency compared to out-of-band sampling, for which data usually has to be sent over a network.

1.2.2. Operational Data Analytics

As the name implies, Operational Data Analytics (ODA) consists in applying *data analytics* techniques to monitoring data in order to improve the *operation* of a system or data center [Bou+19]. For this purpose, a wide range of techniques belonging to the area of data mining and machine learning can be leveraged, with the final objective of transforming raw monitoring data into actionable knowledge. Depending on the actual purpose of the ODA technique at hand, a distinction can be made between two macroscopic classes:

- **Operational Data Analytics for Visualization (ODAV)**: techniques whose main purpose is to provide effective visualization of monitoring data at a large scale or over long time spans, informing system operators and helping them make control decisions. It should be noted that simply visualizing raw monitoring data does not classify as ODAV. This class of techniques can even help in long-term decisions such as the planning of future systems - for example, system operators can process and aggregate long-term power consumption data for an HPC system, correlate it with its workload, and plan the next system's infrastructure based on this knowledge [Bau+19].

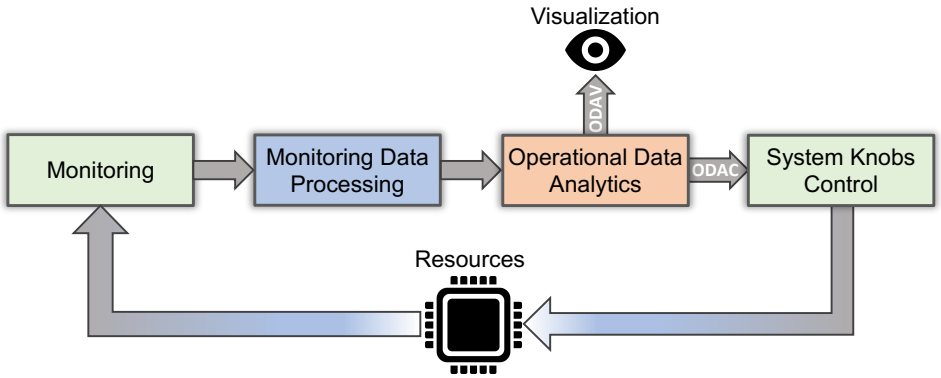


Figure 1.5.: The stages composing a generic ODA pipeline [Net+21c].

- **Operational Data Analytics for Control (ODAC)**: proactive techniques that aim at using the knowledge extracted from monitoring data to optimize system operations autonomously, based on certain models. This is usually done in a streaming, online fashion, tuning system knobs at regular time intervals. For example, a model running in the compute nodes of an HPC system could leverage real-time information about its workload (e.g., number of CPU instructions) to optimize power consumption by tuning CPU frequencies [Eas+17].

In general, ODAC techniques are much more powerful than ODAV ones, allowing for significant efficiency improvements. At the same time, they are much more difficult to implement and integrate successfully into the operation of a system, since they must operate in a real-time, large-scale and heavily constrained production environment. ODAV techniques, on the other hand, can be easily employed using historical monitoring data and without interfering with a system's operation.

Structure of an ODA System

In Figure 1.5 we show the typical steps that are required to implement ODA in a data center environment. The first step consists in *monitoring* of certain system resources; this is followed by the *monitoring data processing* step, which consists in transforming the raw monitoring data into a polished representation that can be comprehended by ODA techniques. A representation of this type can be defined as a *signature*, which can be seen as a set of coefficients describing the status of an hardware or software component. Aggregation and dimensionality reduction of monitoring data are two common approaches to solve this task. This is followed by the actual ODA step: here, a certain model (e.g., obtained via machine learning) is applied to the data representations computed by the monitoring data processing step, producing in turn knowledge that can be used to improve a system's operation. This could consist, for example, of a health diagnosis for a compute node, or a prediction of a system's workload in the upcoming few minutes.

Certain specific and highly complex ODA techniques may implement their own processing steps in addition to (or in place of) those in the monitoring data processing step. This, however, should be avoided in order to minimize computation redundancy when multiple ODA techniques are being used. At this point we can observe the distinction between ODAV and ODAC: in the former, the output of the ODA process is propagated outside of the system and is visualized immediately by interested parties, aiding them in their management tasks. In the case of ODAC techniques, on the other hand, the data produced by the ODA process remains within the system and is further propagated to a *system knobs control* step, which applies the ODA process's output as a new setting for a certain system knob using a specific interface (e.g., CPU frequency in a compute node). This new setting affects the operation of the resource being monitored, effectively producing a *feedback loop*.

1.2.3. Models for Operational Data Analytics

Monitoring and ODA are broad research spaces which may be difficult to navigate, mainly due to the diversity of the approaches employed by HPC centers. Therefore, a clear understanding of the main areas of action and the involved actors is fundamental to identify the most beneficial opportunities for sustainability improvement in a data center. To this end, here we discuss two models for categorizing monitoring and ODA techniques both from a *scope* and *complexity* perspective.

The 4-Pillar Framework

The *4-pillar framework for energy efficiency* was proposed by Wilde et al. in 2014 [WAS14], aiming at providing an overview of the main components and areas in a data center or HPC environment whose operation can be improved. While originally targeted at the energy efficiency aspect, the 4-pillar framework can be generalized and applied to the wider problem of sustainability: in particular, the *pillars* constituting the framework can be interpreted as areas of a data center from which monitoring data can be gathered, and to which ODA can be applied. The framework is summarized in Figure 1.6 and, as the name implies, it comprises four pillars: one the left side, the *building infrastructure* pillar provides control over the machinery driving a data center as a whole, including electrical and cooling equipment. This can be optimized both at a structural level (e.g., by physically changing components), as well as from an operational standpoint. At the second level we find the *HPC system hardware* pillar: this includes the HPC-specific hardware, including compute, management and storage nodes. This is the area where most actions can be performed, once again either from a technological standpoint (e.g., by integrating new accelerator technologies) or from an operational one (e.g., by applying energy-saving or resiliency techniques).

The remaining two pillars are strictly related to the software aspect of a data center. The third pillar, in particular, corresponds to *HPC system software*, and includes a system's resource manager as well as all other software for management tasks. At

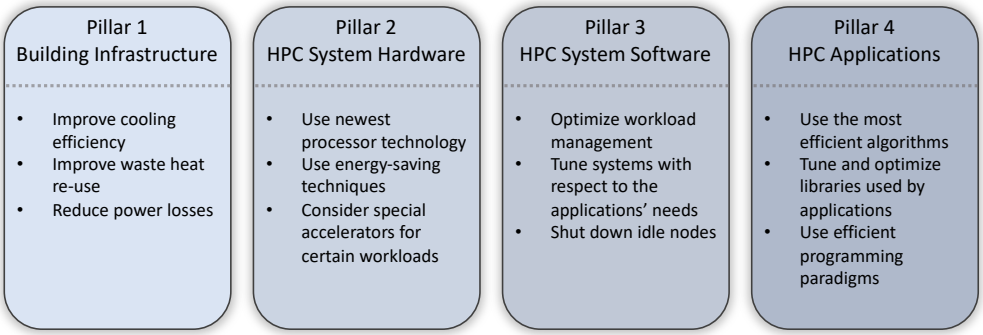


Figure 1.6.: A representation of the 4-pillar framework for energy efficiency [WAS14].

this level, actions to improve the quality of service (e.g., by reducing the waiting times of user jobs) can be undertaken, as well as those associated with tuning of system components (e.g., fan speeds or CPU frequencies). The last pillar, named *HPC applications*, falls outside of the domain of system administrators and involves the end users: it requires development actions such as using efficient algorithms and libraries, as well as programming paradigms that are fit to the scientific problem at hand. Since they do not actively contribute to maintaining the HPC system, users might not be as motivated as administrators to optimize their applications; however, this can be incentivized in a variety of ways, for example by prioritizing jobs belonging to virtuous users at scheduling time [Geo+15]. The authors of the 4-pillar framework argue that the only way forward for data center sustainability is to consider the pillars to be fluid and inter-connected, using holistic design, rather than isolated *silos*.

Gartner's Taxonomy of Data Analytics

The 4-pillar framework describes the main areas of interest for ODA in a data center, but it lacks a categorization of the functionality and complexity of the techniques being used. To address this, we complement it with the general-purpose model introduced by the Gartner consulting firm [4] to categorize data analytics techniques in business intelligence scenarios [Lep+20; SS16]. This model, which is summarized in Figure 1.7, is widely used by large firms due to its generality and applicability. It comprises four different types of analytics: the *descriptive* type includes any form of basic processing (e.g., aggregation of sensors) and visualization of data (e.g., dashboards) that can be used by system operators to understand system events, aided by their domain knowledge. In the case of HPC, descriptive analytics can be mostly associated with traditional monitoring, and any form of advanced knowledge extraction goes beyond its scope. The *diagnostic* type goes a step further, introducing automated models (e.g., machine learning classifiers or heuristic techniques) to draw insight that is not obvious to system administrators, usually carrying out root cause analysis to identify the reason behind specific events. A typical example of this in the HPC domain is anomaly detection [Tun+18].

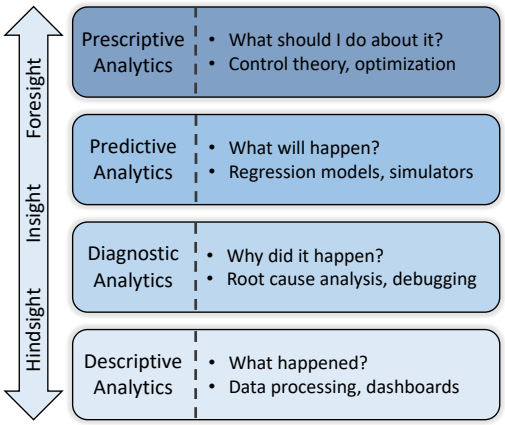


Figure 1.7.: The four types of data analytics commonly used for business intelligence [4].

Predictive analytics is associated with any form of forecasting applied to monitoring data: this includes prediction of sensor time series or of data center-level KPIs, as well as prediction of failures in components and of other relevant events. Predictive analytics, like the diagnostic type, can leverage arbitrarily complex techniques belonging to domains such as machine learning or data mining; in addition, it is usually implemented to support other types of analytics, with the purpose of anticipating a system's state transitions (*proactive* behavior) rather than simply reacting to them (*reactive* behavior). Finally, the *prescriptive* type is tightly associated with control of system knobs, as it revolves around identifying optimal parameters (in function of a performance objective) for certain components, given a certain input state. Models for automated control belong to this class (e.g., for automatic tuning of CPU frequencies), as well as recommendation systems to improve the various aspects of a data center's operation.

The 4-pillar framework and Gartner's data analytics model can be combined to form a bi-dimensional representation that describes both the scope and complexity of ODA techniques in data center environments [Net+21a]. This derived model comes in the form of a 4x4 matrix, with each of the 16 cells associated with a specific HPC pillar and data analytics type. While being a powerful vehicle to categorize and understand ODA techniques, this model focuses on high-level specification and does not capture the low-level details behind the implementation of a generic ODA system. For this reason, in the context of this dissertation we instead adopt the conceptual model introduced in Section 1.2.2, which provides a clear-cut separation between the functional components of an ODA system.

1.2.4. Role in a Production Environment

Monitoring and ODA can be arranged in a variety of ways in a data center environment, with different placement of the associated software components. In Figure 1.8 we

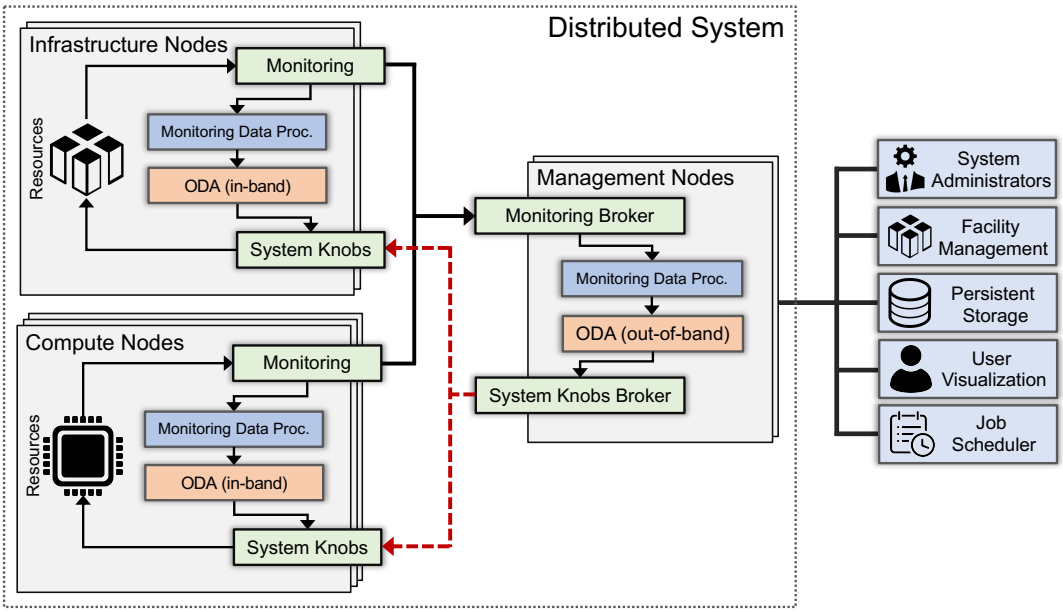


Figure 1.8.: Typical flow of data in a distributed system using ODA (and specifically, ODAC), showing the most common components and actors involved.

show an overview of the main processes involved in ODA, as introduced in Figure 1.5, in the context of a distributed system - this can be an HPC system or a more generic data center cluster. On the right we show the main actors who are usually interested in using monitoring and ODA: these are system administrators and facility managers, whose purpose is to improve the data center's operation in general. We then the find the users of the system, who aim to understand the behavior of their own applications and improve their performance. Among the additional destinations of the ODA processes we find the job scheduler (i.e., the system's resource manager), which aims to optimize resource utilization, throughput and efficiency. Persistent storage is the last, passive destination for ODA and monitoring data.

We now look at the physical placement of ODA-related processes in a system and introduce the difference between *in-band* and *out-of-band* ODA. On the left of Figure 1.8 we find the machines supplying the main sources of data, which are the compute nodes of a system and the surrounding infrastructure. At the center of the figure we find instead a set of management servers running *monitoring brokers*, which are capable of ingesting remote monitoring data. In addition, the management servers may implement *system knobs brokers* to transmit new settings back to the monitored machines in ODAC scenarios. In the case of *in-band* ODA, the monitoring data processing and ODA steps are carried out within the monitored machines themselves, implementing a self-contained system and requiring no network communication. In the case of *out-of-band* ODA, on the other hand, raw monitoring data is transferred to one or more management servers, where monitoring data processing and ODA take place. In ODAC scenarios,

new settings are either applied directly on the monitored machine (in-band) or remotely using a system knobs broker (out-of-band).

It should be noted that out-of-band ODA does not imply out-of-band monitoring: an out-of-band ODA process could analyze monitoring metrics that were originally collected in-band in a compute node. The concept of in-band and out-of-band ODA will be expanded in Section 2.3.3. Furthermore, this scheme can be re-arranged in different ways: for example, both monitoring and monitoring data processing may be performed in-band, transmitting the resulting data representations to a management node, where the ODA step is performed out-of-band. This scheme can be applied to ODAV as well, by simply removing the system knobs control block and by transmitting, if required, the output of any in-band ODA computation to a visualization frontend.

ODA System Example

In the following we present a practical application of ODA to demonstrate the concepts in the previous sections. The ODA application we consider was proposed in a recent work [Jia+19] and aims at improving the efficiency of *warm-water cooling* in HPC systems with fine-tuning of the activity of *cooling units*. This, in turn, determines the inlet temperature of the cooling water entering the compute nodes. In this context, a cooling unit is a hardware component managing heat removal from compute nodes, usually by tuning the amount and temperature of cold water. This ODA approach operates in the following steps:

1. **Monitoring**: in each compute node, the CPU temperature is sampled in-band and transmitted to a management server.
2. **Monitoring Data Processing**: in the management node, each data point associated to a CPU is flagged as a *hotspot* if temperature exceeds a certain threshold. The averages of the CPU temperatures for the nodes associated with each cooling unit, defined as the *cooling demand*, are also computed.
3. **Operational Data Analytics**: in the management node, the required *cooling capacity* for each cooling unit is determined as a function of the percentage of hotspots from its nodes, as well as its cooling demand and previous states.
4. **Visualization (ODAV)**: the cooling demands, as well as a heatmap of the hotspots in the system, can be visualized in real-time by system operators to evaluate the effectiveness of cooling.

OR

4. **System Knobs Control (ODAC)**: in the management node, the cooling capacity is translated into a new setting for the inlet cooling water temperature of each cooling unit. It is then transmitted back to an infrastructure node that is in charge for tuning the associated system knobs.

As it can be seen, this is an out-of-band ODA application: even though the CPU temperatures are monitored in-band, they need to be sent to a management server in order to have an overview of the entire system's status and tune the warm-water cooling system. If a simpler, albeit less effective, cooling system had been used (e.g., air cooling) an in-band ODA approach could have been used, local to each node.

1.3. Contributions

Having established the vital importance of monitoring and ODA for next-generation HPC systems, we now lay out the contributions of this dissertation: we provide directions, frameworks, techniques and practical experiences in order to simplify the adoption of monitoring and ODA, finally closing the gap between research use and production operation that has historically been the bane of this field. While our specific area of application is HPC, our contributions apply to the more generic data center area equally. Specifically, our scientific contributions are the following:

- We conduct an in-depth analysis of functional and operational requirements, identifying the constraints by which the components of a monitoring, ODA and control system should abide in order to achieve effective and efficient operation in production data center environments. We use these requirements as a foundation for the remaining contributions of the dissertation.
- We design and develop *Wintermute*, a generic and holistic ODA framework enabling analysis of data (ODAV) and control (ODAC) at all levels of a data center's operation. Wintermute is designed to be integrated into any monitoring system: here, we integrated it into the *Data Center Data Base* (DCDB) framework, to which we extensively contributed in order to extend its design and make it suitable for large-scale deployments. Furthermore, we present our insights based on several case studies carried out on production HPC systems.
- We propose the *Correlation-wise Smoothing* (CS) method to extract knowledge from multi-dimensional time-series data. It is tailored for the features of ODA processes, producing low-dimensionality image-like signatures that are interpretable to the human eye and enable portability of ODA models across systems and data centers. We apply the CS method to *HPC-ODA*, a dataset collection that we acquired and released publicly, demonstrating its effectiveness in several ODA use cases.
- We present our long-term experiences and insights on the use of ODA on production HPC systems, providing a quantitative evaluation of their effectiveness, as well as documenting the deployment of ODA pipelines from a process perspective. In particular, we treat the themes of performance data visualization for user jobs and machine learning-based cooling control on HPC systems. We extract a series of action items for the ODA community, aiming to make approaches of this kind not only usable, but also maintainable in the long term.

The work in this dissertation provides a comprehensive collection of requirements, frameworks and techniques, as well as real-world experiences in the field of ODA. By covering the entirety of the ODA development process starting from the design, continuing with the implementation and integration, and down to the deployment and evaluation, we provide a cornerstone that will enable and simplify the adoption of such techniques by a larger part of the community, as well as by commercial data center operators. This work was carried out internally at the *Leibniz Supercomputing Centre* (LRZ) in Munich, Germany, profiting from its production data center environment and its large-scale HPC systems, in the context of several projects and with a clear outlook of the necessities that characterize a leadership-class HPC center.

1.4. Organization of the Dissertation

The dissertation is structured as follows. In Chapter 2 we review literature in the area of data center monitoring and ODA, as well as the experiences shared by several data centers, extracting a series of requirements and guidelines that will be used as a basis for the remaining contributions. In Chapter 3 we describe the open-source DCDB monitoring framework together with the Wintermute ODA framework, discussing their design and implementation, coupled with an extensive experimental evaluation and a series of case studies. In Chapter 4 we then introduce the CS method to extract knowledge from raw monitoring sensor data, and we apply it to a series of ODA use cases leveraging the HPC-ODA dataset collection. In Chapter 5 we present our experiences in deploying and evaluating ODA approaches on several production HPC systems, while in Chapter 6 we discuss these processes from a software engineering perspective. In Chapter 7 we finally conclude the dissertation.

Own publication acknowledgement. The description of ODA processes in Section 1.2.2 was originally introduced in [Net+21c]. The models for describing and categorizing ODA approaches in Section 1.2.3, instead, were discussed in [Net+21a]. A full list of all publications associated with this dissertation, including unrelated prior work, can be found in Appendix E.

2. Analysis of Requirements

In this chapter we review the state of the art in monitoring and ODA and present a requirements analysis that is used as the foundation for the rest of the dissertation. In Section 2.1 we discuss common use cases and experiences from several data centers, while in Section 2.2 we review relevant work in the literature. In Section 2.3, we propose a series of functional and operational requirements for effective monitoring and ODA on production HPC and data center environments.

2.1. Background

Monitoring is a well-established practice in the management of data centers, while ODA is still in its early adoption stages. Here, we discuss the experiences shared by several data centers, introduce the main performance metrics that are usually affected by ODA and at the end give an overview of the main ODA efforts carried out at LRZ.

2.1.1. Long-term Data Center Experiences

Many sites have shared their long-term experiences with a variety of monitoring approaches as well as frameworks. Brandt et al. [Bra+16] discuss their two-year deployment of the *Lightweight Distributed Metric Service* (LDMS) monitoring framework [Age+14] on *Blue Waters*, a large-scale HPC system at the *National Center for Supercomputing Applications* (NCSA) with more than 27,000 nodes, using a coarse-grained sampling interval of 1 minute for data collection. The authors describe a series of technical challenges associated to reliability, overhead, data consistency and ownership, as well as dive into system-specific issues such as binary sizes (with respect to the compute nodes' images) or clock skew effects. Finally, the authors propose a list of recommendations for monitoring frameworks used in production environments.

Ahlgren et al. [Ahl+18] collected a series of experiences associated to monitoring, its deployment and the main usage scenarios from several HPC centers. This work highlights the fact that most data centers rely on similar data sources (e.g., CPU performance counters or memory-related metrics) but at the same time employ highly different collection and storage solutions, mostly because of the lack of standard ones, as well as because of vendor constraints. On top of asserting the necessity to give more importance and resources to monitoring at the time of system design, procurement and everyday operation, it is evident from this work that monitoring is never performed without purpose: administrators wish to detect performance variation and anomalies,

as well as optimize the operation of systems in a timely manner, usually inspecting monitoring data via graphical dashboards.

Like the works mentioned above, also the ones by Bourassa et al. [Bou+19] and Bautista et al. [Bau+19] discuss the practical use of monitoring data: here, the term *Operational Data Analytics* (ODA) in the HPC context is used for the first time. The authors describe several cases of successful infrastructure optimization, fault detection and future system planning by means of visual inspection of data collected by the *Operations Monitoring and Notification Infrastructure* (OMNI) framework at the *National Energy Research Scientific Computing* (NERSC) center. For example, the authors demonstrate how, by using monitoring data, they were able to identify voltage-related issues in the data center's infrastructure, which appeared only after large power consumption variations of the HPC systems (e.g., after the end of very large jobs) and that in turn caused the unexpected shutdown of compute nodes. These case studies belong to the ODAV category due to their visual nature.

The works listed above highlight several key issues in the current use of monitoring in data centers: first, there is a lack of frameworks that encompass the totality of data sources that can be monitored in a holistic way, from the infrastructure down to application data, together with their varying requirements and granularities. This often leads to multiple tools being used, in turn resulting in a complex software stack that is hard to maintain and, finally, in a wide disarray of monitoring data that is very difficult to correlate and use effectively [Gim+17]. This limitation complicates the knowledge extraction process, and with it the implementation of an ODAC feedback loop for proactive system control. The latter statement is confirmed in a survey conducted by the *Energy Efficient HPC Working Group* (EEHPCWG) in 2019 [Ott+20] regarding the use of ODA in several HPC centers: sites employ highly different sets of monitoring, storage and analysis solutions which often rely on commercial products that are not tailored for data center monitoring. Moreover, in-house and self-maintained systems are often used, making ODAC feedback loops hard to achieve and limiting data center operators to ODAV solutions for visual inspection.

2.1.2. Metrics for System Performance

The ultimate goal of ODAV and ODAC techniques is to optimize the effectiveness and efficiency of data center operations, which can be quantified with a series of metrics at different system levels. At the topmost level, the main metrics are the *Operating Expenses* (OpEx), which captures the costs of daily operation in a data center including energy, rent costs and employee salaries, and the *Capital Expenses* (CapEx) [ZMW14], which instead captures the one-time costs associated with procuring systems and infrastructures, usually not affected by ODA. The *Total Cost of Ownership* (TCO) [Cui+17] encompasses both of these metrics, providing a more comprehensive picture.

At the facility and infrastructure levels, the *Power Usage Effectiveness* (PUE) metric [YM13] quantifies the energy efficiency of a data center's infrastructure (e.g., cooling equipment) and is formulated as the ratio between the total energy consumption of

a data center and that of its IT equipment (e.g., compute nodes or power supplies). The PUE has a coarse granularity, hence more detailed metrics are sometimes required. Among these, the most common are the *IT-power Usage Effectiveness* (ITUE) and the *Total-power Usage Effectiveness* (TUE) [Pat+13]: the former, a PUE-like metric, is the ratio between the energy consumption of the IT components (i.e., the PUE's denominator) and that of its strictly computational resources (e.g., CPUs, GPUs, or the network fabric). The TUE, instead, can be seen as a combination of the ITUE and PUE. Other similar metrics exist for evaluating the efficiency of cooling equipment: the *Energy Efficiency Ratio* (EER) and the *Coefficient of Performance* (COP) [Pat+06] are similar metrics, both defined as the ratio between the cooling output of a device (i.e., the amount of heat removed) and its energy consumption.

As we descend to the HPC system and compute node levels, we find metrics able to quantify the impact of ODA on the performance of user applications. Among these, the most important are the *Energy-To-Solution* (ETS), which is the total energy consumed by a user application to produce its output on a certain number of nodes, and the *Energy Delay Product* (EDP) [Auw+14], which is a variant of the former with a stronger focus on the application's total runtime. Further, certain metrics capture the effects of ODA on user experience and are mainly affected by scheduling and allocation policies: among them we find the the *slowdown* [Fei01], which equals to the total round-trip time of a user job (i.e., its waiting time plus the execution time) divided by the execution time, quantifying the perceived impact of high waiting times on users. Finally, a system's *throughput* [Reu+18] is proportional to the number of jobs started on it per unit of time: a high throughput implies low waiting times, as well as good utilization of computational resources.

2.1.3. Operational Data Analytics at LRZ

LRZ has a rich history as a data center and service provider with a strong focus on energy efficiency, and many ODAC efforts were carried out throughout the years to this end. Among these, one of the most relevant is the use of the IBM *LoadLeveler* [Kan+01] resource manager on the now decommissioned *SuperMUC Phase 1* HPC system [5] as well as on its successor, *SuperMUC Phase 2* [Auw+14]. On this system, LoadLeveler performed automatic CPU frequency optimization at the start of each job with the aim of minimizing the ETS of applications: the effectiveness of this approach derives from the fact that applications showing strong memory activity (*memory boundedness*) are less sensible than those exhibiting intense computational activity (*compute boundedness*) to CPU frequency and other system knobs, which therefore can be optimized without adverse effects on performance. To this end, LoadLeveler used a simple regression model to characterize the scaling of application energy in function of frequency, given a single observed point at a default frequency and a set of coefficients characterizing the behavior of compute nodes under different compute kernels. However, LoadLeveler required users to supply *energy tags* to identify applications and recall previous configurations. Despite this, LoadLeveler helped reduce the energy consumption of SuperMUC by

roughly 6%, corresponding to EUR 200,000 per year of operation.

The *SuperMUC-NG* [6] system builds upon its predecessor and uses a more advanced solution. The *Energy-Aware Runtime* (EAR) [CB19] framework runs together with each user job on the system and optimizes CPU frequencies on a per-compute node basis, leveraging monitoring data and machine learning models to characterize the performance profiles of applications. Unlike LoadLeveler, EAR is decoupled from the resource manager and performs tuning not only at the start of user jobs, but throughout their duration according to a configurable optimization policy. Due to the lack of historical data, quantifying the impact of EAR on system performance is difficult at this point in time. However, EAR has been shown to produce energy savings of up to 10% on several large-scale HPC applications during its experimental validation phase. While LoadLeveler and EAR are effective ODAC solutions, they were designed ad-hoc for a specific purpose and hence lack generality for different ODA use cases.

2.2. State of the Art

Here we discuss relevant literature work in the topics of monitoring, monitoring data processing and ODA, corresponding to three blocks of the feedback loop introduced in Section 1.2.2. For the last of the three, we focus on works proposing frameworks to enable ODA in production environments, leaving the discussion of ODAC and ODAV techniques themselves for a later section.

2.2.1. Monitoring Work

Most monitoring frameworks for HPC and data center management that are currently available are insular in nature and offer varying feature sets. *Examon* [Ben+17] employs a push-based monitoring model, allowing for accurate fine-grained sampling of data, and using *Internet of Things* (IoT) technologies such as the *Message Queuing Telemetry Transport* (MQTT) [Loc10] and *Apache Cassandra* [WT12], which respectively act as communication protocol and data store. This framework, however, has been mostly employed as an ad-hoc solution for certain production environments: it lacks a modular and plugin-oriented architecture, making the integration and configuration of new data sources, and in turn adoption by other sites, difficult. The *MetricQ* framework [Ils+19] is similar in nature, providing in addition a storage model designed to reduce the volume of fine-grained monitoring data, by aggregating it in a hierarchical fashion.

The LDMS monitoring framework [Age+14] is one of the most renowned open-source monitoring solutions in the HPC domain and it has been shown to be suitable for deployment on large-scale systems, as shown in Section 2.1. However, while plugin-based, the LDMS architecture is not designed for customization and development of new plugins for specific data sources, as well as configuration of existing ones, requires considerable effort. Moreover, storage options for sensor data in LDMS are limited, requiring external software layers to ingest monitoring data and store it into data stores

such as Apache Cassandra. LDMS also uses a custom communication protocol between its sampler and aggregator processes, making integration with other frameworks and environments cumbersome. Finally, the pull-based model upon which LDMS is built is problematic for fine-grained monitoring, which requires high sampling accuracy and precise timing. *Ganglia* [MCC04], which is one of the first open-source monitoring solutions ever proposed, faces similar issues as LDMS. The open-source *Elastic Stack* [7] framework is also employed by several data centers, but it is not inherently designed for large-scale HPC configurations at fine time scales.

The *Persyst* framework [GHB14] specializes in collecting performance monitoring data, transforming and aggregating raw data into performance patterns as it is collected. However, the collected data lacks the detail of the raw performance metrics and the framework itself is not designed for extensibility. The *ClusterCockpit* [Eit+19] and *MPCDF* [SR19] frameworks provide similar job-oriented feature sets. The *ScrubJay* tool [Gim+17] instead allows to automatically derive semantic relationships from raw monitored data, which is aggregated to generate performance indicators. Performance profiling tools such as *HPCToolkit* [Adh+10] and *Likwid* [THW10], as well as *TAU* [SM06] and *perf* [Wea13] offer extensive support for the collection and manipulation of node-level performance metrics at a fine granularity, useful for application behavior characterization, but do not support the transport and storage of monitored data nor the integration of facility data. Finally, *TACC Stats* [Eva+14] is a comprehensive solution for monitoring and analysis of resource usage in HPC systems at multiple resolution levels, whereas tools such as *Caliper* [Boe+16] provide generic interfaces to enable application instrumentation. The aforementioned tools are not designed to perform system-wide and fine-granularity monitoring, but rather application analysis, often in an offline context, and have a different focus compared to the style of monitoring that is required for the purposes of ODA. Nonetheless, these tools can provide useful, complementary knowledge if properly integrated with system-wide monitoring.

In the space of data transport solutions, frameworks like the *Multicast Reduction Network* (MRNet) enable high-throughput and efficient multicast and reduction operations in distributed systems [RAM03]. In particular, MRNet relies on a tree-based overlay communication network, according to which retrieval of data is performed from the leaves to the root of the tree, whereas packet aggregation can be implemented via customized filters. While MRNet could be integrated into any monitoring system, it is often preferable to employ widespread solutions (e.g., MQTT from the IoT space) due to their higher acceptance among system administrators and their loosely-coupled setups, which in turn simplifies the deployment process.

There are many commercial and closed-source products for system-wide monitoring. Among these, the most popular are *Nagios* [Bar08], *Zabbix* [Olu16] and *Splunk* [Car12], which are deployed in many data centers across the world. *Icinga* [8] provides a similar feature set, while being tailored specifically for HPC systems. These frameworks, however, are oriented towards producing alerts for system administrators and focus on the analysis of *Reliability, Availability and Serviceability* (RAS) metrics to provide insights

into system behavior, which is one of the most common usage scenarios as seen in Section 2.1. They supply conventional monitoring features, but these usually focus on infrastructure-level data and cover only a very small part of the vast amount of metrics that could be monitored in a system (e.g., CPU performance counters in compute nodes or application-level event data).

2.2.2. Monitoring Data Processing Work

The problem of transforming data center monitoring data into insightful representations for ODA purposes (i.e., signatures) has been dealt with in a variety of ways. Most relevant works belong to the ODA domains of anomaly, fault and failure detection: Tuncer et al. [Tun+18] propose a method that relies on computing statistical indicators (e.g., mean or standard deviation) of recent data from each available sensor, which are then used to build signatures and subsequently fed to a machine learning classifier. Bodik et al. [Bod+10] propose a similar approach, focusing instead on characterizing the distribution of each sensor's data by computing percentiles. Lan et al. [LZL09] use the raw time-series data of each sensor concatenated to form a signature, thus preserving time information; the signatures are then compressed using *Principal Component Analysis* (PCA) [AW10], and distance metrics are used to identify anomalies. Laguna et al. [Lag+13] use the pairwise correlation matrix associated with the set of sensors as a signature, under the assumption that faulty behavior will alter their natural correlations, while Cohen et al. [Coh+05] use an ensemble of statistical models to characterize the meaningfulness of each sensor against a reference metric (e.g., a health indicator), from which the signature coefficients are extracted. Hui et al. [HPE18] apply *Singular Value Decomposition* (SVD) to sensor data and compute a single entropy coefficient aiming to characterize system state transitions. Finally, Borghesi et al. [Bor+19] use an autoencoder-based system to compress and subsequently extract sensor data, leveraging the reconstruction error to detect anomalies.

Many classical techniques belonging to the wider domain of data mining and dimensionality reduction have been applied to monitoring data with varying degrees of success. PCA [AW10] and *Independent Component Analysis* (ICA) [HO00] compress a multi-dimensional dataset to a lower-dimensionality space in which each dimension is a linear combination of the original ones, with higher weight associated to those that contribute the most to the total variance. Guan et al. [GF13] use a variant of this approach, named *Most Relevant Principal Components* (MRPC), that aims at preserving some of the original dimensions intact. Techniques like *Multidimensional Scaling* (MDS) [CC08] or *Vector Quantization* (VQ) [Gra84] are also often employed. However, many traditional dimensionality reduction techniques have been proven to not work well in HPC and data center-specific ODA problems: when performing fault detection, for example, the most critical status indicators are often not found in the metrics that contribute to most of the variance [Tun+18].

While many other generic approaches to process multi-dimensional time-series data are available, their fitness for system monitoring data is not clear. Among these we find

Symbolic Aggregate Approximation (SAX) [Lin+07] and *Trend-value Approximation* [Esm+12], which aggregate time-series data both on the time and value axes, and the *Matrix Profile* [YKK17], which computes data structures to aid in the identification of time-series motifs. Other techniques belonging to the domain of spectral signal analysis [CF99] are also commonly used.

2.2.3. Operational Data Analytics Work

The problem of enabling ODAV and ODAC on data center and HPC systems in a generic, holistic way is still an open research question. The LDMS framework has been recently enhanced to support ODA features on top of standard HPC monitoring [Iza+18]: however, due to its pull-based architecture, it is not suitable for in-band, fine-grained ODAC use cases that require live data with minimal overhead and latency. Moreover, LDMS currently lacks global configuration abstractions to simplify the instantiation of models at a very large scale. The OMNI framework has a similar architecture, being mostly based on LDMS itself, but it is more oriented towards ODAV and also misses the abstractions necessary for control.

Examon is shown to be suitable for ODA applications, being based on the MQTT protocol and thus compatible with out-of-band tools such as *Apache Spark* [9], which is leveraged for most of the analysis. This reliance on the use of external tools to process data results in a complex software stack that needs to be tuned for each specific use case, as mentioned in Section 2.1, as well as in non-optimal data retrieval performance. The *GUIDE* [Vaz+17] framework combines monitoring and ODA features, but it is mostly log-oriented and the semantics of its data analytics features are not clear. Elastic Stack supports the post-processing of data ingested from external sources on top of standard monitoring, thus enabling data analytics for legacy monitoring frameworks such as Ganglia. The analysis, however, is centralized at the server level, which limits scalability for large installations and is only suitable for ODAV purposes.

Many other tools propose basic applications of ODAC tailored for HPC and implement simple feedback loops between the monitoring component and the resource manager: among these are the EAR [CB19] or LoadLeveler [Auw+14] frameworks, described in Section 2.1.3 and both focused on energy consumption optimization via tuning of CPU frequencies. Similarly, tools like *SPar* [Gri+18] provide user-friendly interfaces for runtime tuning, by specifying high-level QoS parameters. These efforts, however, tackle specific issues of HPC resource management and customization for other purposes is not trivial. Further, due to the lack of coordination mechanisms, using multiple systems of this kind to address multiple resources can even be counterproductive, due to their concurrent access to data sources and system knobs. The *Global Extensible Open Power Manager* (GEOPM) [Eas+17] provides a plugin-oriented and extensible ODAC interface for resource and power management in HPC systems, but its monitoring capabilities are currently very limited. Alongside the open-source solutions discussed above, there are also many commercial and closed-source products, such as Splunk or *Zenoss* [10], offering extensive ODA capabilities. As mentioned earlier, however, these products are

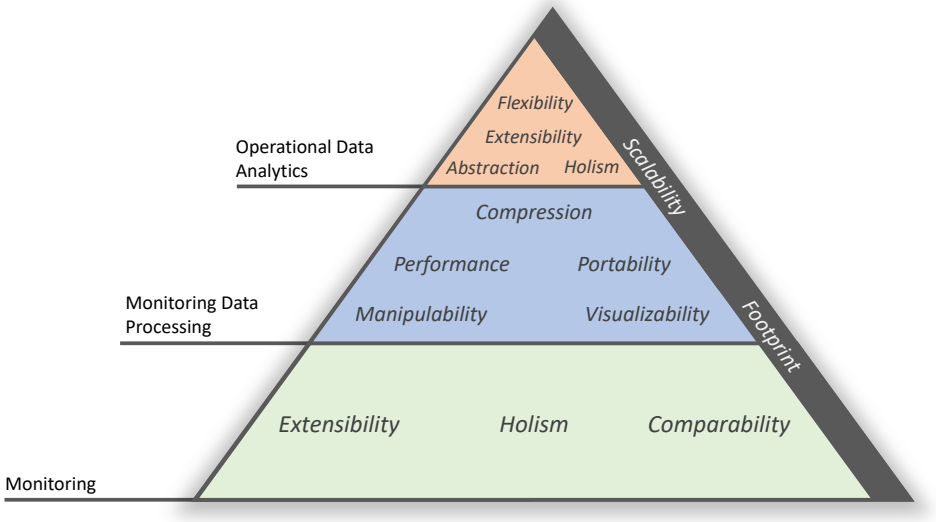


Figure 2.1.: A diagram of the main functional and operational requirements of ODAV and ODAC for data center management.

often designed solely for loosely-coupled data center environments and are not suitable for fine-grained monitoring and ODAC.

2.3. Analysis of Requirements

Based on the works reviewed in Section 2.2, we are able to derive a series of functional and operational requirements that should be satisfied by all blocks of an ODA pipeline in order to achieve both ODAV and ODAC in production data centers. The requirements, which will be discussed in detail in the following, are summarized in Figure 2.1 for each of the three blocks, highlighting common characteristics. In addition, we propose a taxonomy of ODA techniques found in the literature, focusing on ODAC, which we use to further refine the constraints driving an ODA framework as a whole.

2.3.1. Requirements of Monitoring

As the employment of ODA techniques to improve the efficiency of data center operations becomes progressively common, different algorithms and system components might require access to the same data sources and system knobs: in a typical scenario, ODAC systems for fault tolerance [JDD12], runtime tuning and optimization [Eas+17; BPG10; ŢCH02] as well as visualization (ODAV), among others, will require concurrent access to different types of data, such as performance counters, power meters, facility sensors or even application-level metrics. This leads to the necessity of a unified and controlled access to all data sources. For this reason, a system monitoring solution

appropriate for use in next-generation data centers must be *holistic* (i.e., comprising of data sources of the facility, the system and the applications), *thorough* (i.e., storing as many data points as available), and *continuous* (i.e., storing the data from all sensors at all times). Achieving these three characteristics is extremely challenging [Ahl+18; Bra+16]. To aid in this process, we propose the following list of requirements that should be satisfied by a modern monitoring solution:

- **Scalability**: traditional sensor data (e.g., health status, temperature or power draw) is comparatively easy to collect as it is typically acquired on a per-node basis and does not require high sampling rates. Even for large-scale systems, this type of data will only consist of a few thousand sensors, which can be managed by a monolithic monitoring solution. Metrics that are descriptive of application behavior (e.g., executed instructions or branch misses), however, typically need to be collected on a per-CPU core basis and at high sampling rates (e.g., 1Hz), resulting in thousands of individual sensors per-compute node and millions of data points per second. This requires the use of scalable and high-performance techniques for monitoring, to prevent the appearance of bottlenecks.

- **Footprint**: many sensors must be sampled *in-band* (i.e., on the compute nodes themselves) as opposed to *out-of-band* (i.e., via a dedicated management network). For the former, monitoring solutions need to be highly efficient in order not to interfere with running applications, both in terms of *overhead* and *resource footprint*. Achieving good efficiency is especially pressing when performing fine-granularity monitoring of several thousand sensors per-node, which could have a significant impact on running applications and in turn offset the benefits of monitoring.

- **Holism**: a monitoring framework should supply interfaces that are uniform and generic, abstracting from the location at which data is stored and even from its specific type, allowing users to extract useful knowledge by querying and correlating metrics belonging to different levels of a system.

- **Extensibility**: very often, new hardware and software devices need to be added to an already-running monitoring system, potentially requiring additional protocols and interfaces to access their data. This is the case, for example, when an HPC system is extended with new resources (e.g., GPUs) during its life time, or when the compute nodes' operating system is upgraded. Being able to easily add new data sources, either by deploying existing implementations or developing new ones from scratch, is therefore important for production use of any monitoring solution. On the other hand, extensibility is also a fundamental prerequisite for a community-driven monitoring effort.

- **Comparability**: data coming from different sources is often recorded at varying sampling rates, using varying units of measure and exposing different metadata characteristics. A series of mechanisms to translate data from different sensors,

derive comparable metrics, and ultimately enable cross-source correlations are necessary, supporting in turn holism.

Most of the state-of-the-art tools discussed in Section 2.2.1 propose solutions which address separately the challenges previously mentioned; furthermore, they often provide monitoring capabilities that are targeted only to specific types of data center installations (e.g., HPC centers or cloud operators) or, sometimes, even to specific layers (e.g., application profiling in HPC). From a practical perspective, tools that are specialized into gathering performance counter data very often do not support monitoring of sensor data at the system level (e.g., node temperature or energy consumption). However, a comprehensive solution which satisfies all of the requirements listed above, and that takes into account all possible data sources in a truly holistic way, is still clearly missing and is necessary more than ever to make ODAC practical.

2.3.2. Requirements of Monitoring Data Processing

Despite the existence of many effective ODA algorithms in the literature, there is a severe lack of generic monitoring data processing methods to transform raw time-series sensor data into signatures (see Section 1.2.2 for a definition of signature). In fact, as monitoring data is inherently high-dimensional, architecture-dependent and hard to interpret even for domain experts [Gim+17; Ahl+18], most existing solutions are created ad-hoc for the specific ODA problem at hand and lack any kind of generality or applicability to different scenarios. This aspect is complicated by the fact that most traditional data mining methods [MR05] are not applicable to the ODA domain due to its strict operational constraints in large-scale and low-overhead environments [FBB08]. Focusing on the aspect of processing raw sensor data into actionable signatures, we identify the following set of requirements:

- **Scalability**: meaningful analysis for ODA purposes will often result in very large volumes of data, stemming from large amounts of data points (e.g., when analyzing long time spans for historical analysis) or high dimensionality (e.g., when considering thousands of sensors from an HPC system). A signature algorithm must be able to cope with such conditions, ensuring good performance as well as numerical stability.
- **Footprint**: in order to be suitable for in-band ODAC use cases, where low overhead on running applications is a strict requirement, the algorithm's resource footprint must be minimal and possibly adaptable to changing system conditions.
- **Compression**: in order to be useful, signatures must be significantly more compact than the original sensor data they are computed from, both in terms of dimensionality and number of data points. Moreover, the compression ratio should be configurable, to ensure adaptability with respect to the nature of the data as well as the system's constraints. Finally, effective compression of the signatures simplifies their use in training of models as well as in inference.

- **Performance**: the signatures must capture most of the information contained in the original data, thus leading to acceptable and predictable ODAC performance (e.g., classification accuracy) that is comparable to that obtained when using the raw, uncompressed sensor data.
- **Visualizability**: signatures must be easily visualizable and interpretable regardless of their size and source data, so as to grant a natural way to visualize sensor data and enable intervention of human operators in the ODA processes.
- **Manipulability**: it must be possible to manipulate, scale, resize and compare the signatures with simple and established mathematical methods. On top of improving flexibility, this feature potentially allows to leverage techniques from other branches of science (e.g., image and signal processing) and thus widen the scope and applicability of signatures to ODAC models.
- **Portability**: the signatures must be formulated in a space of parameters with clear properties, that is generic and not system or architecture-dependent, as well as robust against the addition or removal of sensors over time. This enables the portability of ODAC models across systems and institutions, which is especially important in use cases where the generation of effective models is not trivial (e.g., fault detection), stimulating in turn the sharing of competences.

Most of the methods discussed in Section 2.2.2 exhibit very good performance in specific ODA use cases, but their generality, visualizability and scalability, among other aspects, are not proven for other use cases and for large-scale datasets with hundreds of dimensions. Moreover, as most existing methods do not satisfy our *portability* requirement, they cannot be manipulated in simple ways and are extremely sensible to data properties such as scaling and order: this hinders re-use of models in time and across systems, while causing even minor changes in the monitoring infrastructure to be detrimental. This aspect is critical for future exascale HPC platforms, as their sustainable operation will require the coordinated use of ODA at all system levels and in turn a vast set of competences to train and operate the underlying models. Approaches to enable the sharing of knowledge and models across institutions are therefore necessary to simplify this daunting task. In general, a monitoring data processing method that satisfies all of the requirements listed above and is thus tailored for ODA applications in data center environments is missing in the literature.

2.3.3. Taxonomy and Requirements of Operational Data Analytics

Due to its visual inspection nature, ODAV can be usually carried out without particular constraints and non-intrusively in a data center environment. ODAC, on the other hand, poses a much more complex challenge: although it is an established research field, the associated techniques have not been systematically classified and typical functional and operational requirements are still not clear. Identifying these parameters is paramount

for the design and development of an effective ODA framework targeted at data center operation. To this end we propose a non-exhaustive *taxonomy*, depicted in Figure 2.2, identifying the most common use cases associated with ODAC in data centers and HPC sites. This list is based on recent and relevant works, reflecting current trends and the sustainability concerns discussed in Section 1.1.3; we also factor in our experiences at LRZ. It should be noted that, in this context, we limit our analysis to ODAC techniques that leverage scheduling or time-series sensor data, and do not consider techniques that exploit different types of data (e.g., log streams). In particular, we identify the following main usage scenarios for ODAC applied to data center operation:

- **Infrastructure Management**: these techniques optimize the operation of infrastructure and facility-wide machinery in a data center, potentially involving multiple systems, as well as adapting to environmental changes. For example, an ODAC algorithm could optimize the inlet temperature of a warm-water cooling infrastructure depending on current workload conditions [Con+15; Jia+19; GPG15; Jha+18; Sho+17; SK20; Wil+15; Abd+18; Ste+19b; Kjæ+16; BNS19].

- **Scheduling and Allocation**: algorithms of this type improve the placement of user jobs on an HPC or data center system by supplying additional information (e.g., system energy budgets, thermal limits or I/O features) to the scheduler. For example, a power-aware scheduler could arrange jobs in the most power-efficient way by leveraging information about predicted workloads from compute nodes [BF07; IHH18; SB16a; VAN08; Bri+16; ZF13; GRC09].

- **Prediction of Job Features**: the use of heuristic or learning techniques to predict the duration of user jobs and their submission patterns can improve the effectiveness of scheduling policies and reduce queuing times. A technique of this kind could exploit scheduling-time information (e.g., user ID and job name) to estimate the duration of a job and use this in place of its wall time for scheduling [Gal+17; NS18; MF10; Eme+15; Eja+10; Li+09; Zhu+19; PC19].

- **Application Fingerprinting**: certain management decisions can be optimized by predicting the behavior of user jobs and correlating this to historical data to characterize features such as power consumption and network usage. For example, an algorithm could predict whether a certain job will run rogue applications (e.g., cryptocurrency miners) based on historical monitoring data associated to previous runs, thus enabling appropriate corrective actions [Ate+18; Gal+15; Zha+12; Wya+18; McK+16; DSB13].

- **Fault Detection**: detecting and predicting anomalous states in hardware and software components can prevent unmasked failures and other catastrophic events, in turn improving the resiliency of a distributed system. An algorithm that is able to predict a failure event before it happens can be used, for example, to trigger checkpointing of system states and thus prevent waste of resources [GF13; Tun+18; SB16b; SV18; FBB08; Net+19a; LZL10; Bod+10; Lag+13].

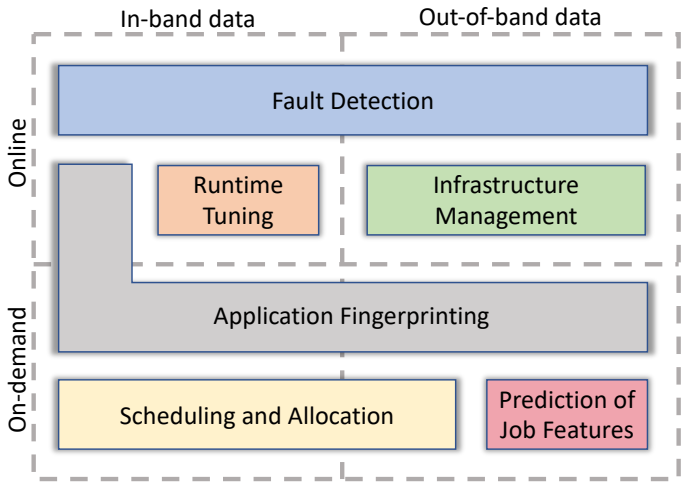


Figure 2.2.: A taxonomy of common ODAC applications. Some applications can be employed in different ways depending on their requirements [Net+20].

- **Runtime Tuning**: techniques of this kind are able to predict the behavior of applications and components in compute nodes for dynamic tuning using system knobs, usually at very fine time scales. An algorithm that is able to predict the upcoming instruction mix for a certain application in compute nodes can be used, for example, to tune CPU frequencies every few milliseconds accordingly [Eas+17; CB19; Wan+17; LWP16; Sar+14; Mic+12; Bor+16].

Taxonomy of Operational Data Analytics

The list of use cases above show that ODAC can be applied at all levels of a data center's operation, as well as at different time scales and with varying volumes of data. These techniques, moreover, rely on the same type of data sources, which in this case fall under the sensor monitoring data umbrella. Some specific applications, such as those associated with job analysis, may require additional non-numerical data (e.g., job name or user ID). Based on our observations, we are able to further divide ODAC techniques into four classes, according to their specific location with respect to the data and their mode of operation. On one hand, we identify two location types:

- **In-band**: data is sampled and consumed within a specific component in an HPC system, usually a compute node. This scheme, which limits factors of unpredictability such as network latency, is often used by techniques that operate at a fine time scale (i.e., greater than 1Hz) and require low analysis overhead and latency in gathering data. Techniques belonging to the domain of runtime tuning usually operate in this fashion and hence require this type of data.
- **Out-of-band**: the data potentially comes from any available source in the system, including historical, job-related or asynchronous facility data. For techniques using

this type of data, operation often is at coarse scale (e.g., in the order of minutes); moreover, since the data may come from different machines and be processed in others, it must be explicitly synchronized (e.g., through time-stamps) to enable correlation. However, latency and overhead are less of a concern.

The concepts of in-band and out-of-band ODAC were originally introduced in Section 1.2.4. On the other hand, we are able to group ODAC techniques according to the two following modes of operation, which capture most common production use cases:

- **Online**: continuous, streaming operation in which the ODAC processes are invoked periodically, producing an output resembling a time series. This information can then be re-used to drive management decisions and tune system knobs, thus producing a feedback loop.
- **On-demand**: operation triggered at specific times (e.g., job submission), requiring a real-time analysis to steer management decisions that benefit from certain information about the system's status. ODAC techniques associated to scheduling and allocation usually fall in this class.

Using these features we are able to classify the use cases presented above and produce a taxonomy, as shown in Figure 2.2, which we use as guide for our analysis of requirements. It should be noted that, practically, most ODAC algorithms complying to the on-demand class can also operate as online algorithms: however, this is usually extremely inefficient, since most of the data produced by the ODAC process would not be used and the associated computation would thus be wasted. As mentioned earlier, ODAV techniques do not exhibit the same level of complexity as their ODAC counterparts: in the vast majority of cases, visualization is carried out on dedicated machines (out-of-band) using historical data (on-demand). This is easily attainable as long as a robust monitoring infrastructure is available.

Requirements of Operational Data Analytics

In light of our taxonomy of ODA techniques we are able to extract a series of functional and operational requirements, which we present in the following, that must be taken into account when designing any generic ODA framework tailored for HPC and data center operation. These requirements are built on top of those established for monitoring (Section 2.3.1) and monitoring data processing (Section 2.3.2):

- **Scalability**: an ODA framework must allow models to scale up to thousands of inputs at very fine time scales so as to be suitable for system-wide operation, adapting automatically to current conditions and requiring minimal tuning of configuration parameters.
- **Footprint**: the framework must exhibit a light resource footprint, to not interfere with running applications in compute nodes when used in-band, as well as with other management processes when used out-of-band.

- **Holism**: an ODA framework must provide a holistic view of the available sensor space, exposing data in the way that is most fitting to the current scenario and independently from its locality. An in-band ODAC model will benefit from in-memory processing of local sensor data, for optimal latency and overhead. Conversely, an out-of-band model performing coarse-grained analysis may require large amounts of data (e.g., historical or from an entire HPC system) that cannot be maintained within local memory and thus must be fetched from remote storage.

- **Extensibility**: as system knobs in data centers are often controlled via a set of common protocols (e.g., IPMI or SNMP), an ODA framework must be modular and able to integrate a wide range of external interfaces, in a specular way to its underlying monitoring system. Further, as ODA techniques often rely on similar processing steps (e.g., regression), pipelining of analysis and control stages to maximize code re-use is a desirable quality.

- **Flexibility**: multiple modes of operation, online and on-demand being the most relevant, must be supported to address the necessities of different techniques targeting the various components and levels of a data center.

- **Abstraction**: manual configuration is prohibitive when a large amount of independent ODAC models (e.g., one per CPU core of an HPC system) must be deployed together, either in-band or out-of-band. This highlights the need for abstraction constructs to simplify and automate the configuration of ODAC models at scale.

Careful system design is needed to address the requirements above in a generic and comprehensive way and thus render both ODAV and ODAC feasible in practice; this, to the best of our knowledge, has not been accomplished by any currently available ODA solution. Most of the recent efforts in ODA for data center management presented in Section 2.2.3 rely in fact on the use of off-the-shelf analysis tools such as Apache Spark or Elastic Stack: while these are sufficient for specific ODAC and ODAV configurations, guaranteeing scalability and holism, they lack the needed flexibility, abstraction and footprint. Further, since tools of this kind are designed to run continuously on dedicated server instances, their support of in-band models is severely limited, thus cutting out a large portion of literature ODAC techniques.

To overcome the limitations above and to address all requirements, ODA must be tightly integrated with the components of a holistic monitoring system (as described in Section 2.3.1) for HPC and data center environments: monitoring interacts naturally with most system components addressed by ODA, providing efficient and convenient interfaces for control as well as streaming data access. On one hand, integration with monitoring daemons in compute nodes enables in-band operation, close to where data is sampled; on the other hand, management nodes can be used for out-of-band operation. In the latter case, the ODA framework can interact with a monitoring data broker, gaining access to streamed system-wide data, as well as remote persistent storage. This approach covers all requirements laid out in this section.

2.4. Research Opportunities

The ODA research field thrives with techniques and optimization algorithms targeting all levels of a data center's operation, as we have shown with our taxonomy in Section 2.3.3. At the same time, however, most of these algorithms have been validated using specific datasets and test conditions, and have never been employed in a production environment: we attribute this to the severe lack of clear guidelines and frameworks for monitoring and ODA, as well as for monitoring data processing techniques. The requirements that we laid out in this chapter, which serve as a reference to fill this gap and guide the design and implementation of ODA frameworks, are unfortunately not satisfied by most currently available solutions and, as a result, ODAC is rarely employed beyond its more common ODAV counterpart.

The lack of appropriate end-to-end frameworks renders the long-term adoption of ODAC cumbersome and hard to justify, which is made worse by the fact that the added value of many ODAC techniques (e.g., in terms of OpEx or TCO) is made clear only after months or even years of continuous operation. This sentiment is reflected in the experiences of other data centers that we described in Section 2.1: due to the intricacy of deploying ODA on production systems, which sometimes extends to monitoring itself, visual inspection via ODAV is the only possible vehicle of analysis.

The fact that the most novel ODAC approaches rely on machine learning and data mining techniques adds further complexity to their use: system operators are often not well-versed in these disciplines, calling for approaches to train ODAC models in a resilient and system-independent way, thus enabling the sharing of knowledge across institutions. Finally, the sustainability constraints of future HPC machines require the use of ODA at all levels of a facility in a tightly orchestrated fashion, exacerbating these issues. The purpose of this dissertation is to tackle these research gaps, motivating the work behind the set of contributions laid out in Section 1.3.

Own publication acknowledgement. The discussions of related work and requirements for monitoring, monitoring data processing and ODA (including the taxonomy for the latter) were originally carried out in [Net+19b], [Net+21b] and [Net+20] respectively, and were expanded for the purposes of this dissertation.

3. The Data Center Data Base and Wintermute Frameworks

In this chapter we discuss the *Data Center Data Base* (DCDB) monitoring framework and its ODA extension, *Wintermute*. After describing the system's architecture, we present a series of experiments and case studies carried out on production HPC systems. The chapter is structured as follows: in Section 3.1 we present an overview of the project, in Section 3.2 we present its architecture and in Section 3.3 its implementation. In Section 3.5 we discuss a performance evaluation of DCDB, and in Sections 3.6 and 3.7 we present several monitoring and ODA case studies. Section 3.8 concludes the chapter.

3.1. Overview of DCDB

The DCDB project [Net+19b] was started in 2013 to address the monitoring necessities of LRZ, in particular regarding power consumption and infrastructure data: at the time, a framework tailored for fine-grained HPC monitoring and able to capture data at all levels of a facility, from the infrastructure down to the compute nodes, was missing. DCDB strives to address these challenges, and by extension the requirements listed in Section 2.3, offering an open-source monitoring solution that addresses the complexity of managing modern HPC installations for administrators and users alike: its key features are modularity, which allows for easy integration in existing environments, as well as a flexible plugin-based design (*Extensibility*). DCDB is also highly scalable, thanks to its distributed architecture (*Scalability*), and efficient, due to its low-overhead implementation (*Footprint*). DCDB is deployed on the production HPC systems at LRZ, which are listed in Section 3.5.1: its deployment on the large-scale SuperMUC-NG HPC system, in particular, will be described in Section 5.1. DCDB has also been employed in a series of European Union-funded projects, offering monitoring for the prototype HPC systems in the *Mont-Blanc* projects (1 and 2) [11], as well as in the *DEEP* projects [12].

Contributing to the entirety of the DCDB code base and its polishing for production use was an integral part of the work in this dissertation. The main and original scientific contribution, however, lies in *Wintermute*: it is a novel framework aimed at online and holistic ODA on production HPC and data center systems, capable of processing data and taking decisions at any level [Net+20]. Wintermute tackles the research gap highlighted in Section 2.2.3, satisfying in turn the requirements laid out in Section 2.3. It is designed to be integrated into arbitrary monitoring systems, enabling them to perform ODA tasks: here, Wintermute was implemented and subsequently integrated in DCDB, granting to it ODA capabilities that would otherwise have been out of reach.

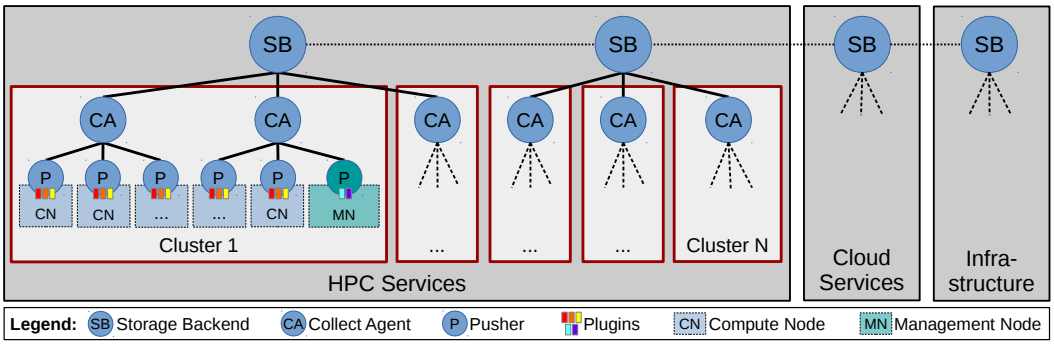


Figure 3.1.: A possible deployment scenario for DCDB, highlighting its modular and distributed nature [Net+19b].

Wintermute's workflow accommodates most real-world ODA applications, belonging to both the ODAV and ODAC categories, while its small resource footprint renders it suitable for applications in which low overhead and latency are critical.

3.2. Software Architecture

In this section we describe the overall software architecture of DCDB, and the design principles upon which it is based. After providing a general overview we discuss the architecture of Wintermute, which is agnostic with respect to the surrounding monitoring framework, and then proceed with describing DCDB as a whole, together with the integration of Wintermute into its components.

3.2.1. Overview of the Architecture

In order to address the requirements described in Section 2.3, DCDB has been designed following a highly modular and distributed architecture. This is depicted in Figure 3.1: DCDB consists of three major classes of components, each with distinct roles, which are the *Pushers*, the *Collect Agents* the *Storage Backends*. These components are distributed across the entire system and facility, including compute nodes, management nodes and other infrastructure components. In the following we briefly describe the role of each of the hierarchy's components:

- **Pusher**: A Pusher is a plugin-based component responsible for sampling monitoring data. It is designed to run as a daemon process either on compute nodes to collect in-band data, or on management servers to gather out-of-band data.
- **Collect Agent**: A Collect Agent is responsible for receiving the sensor readings from a set of associated Pushers and writing them to a Storage Backend. For that purpose, it assumes the role of data broker according to the semantics of *publish/subscribe* protocols.

- **Storage Backend**: A Storage Backend is a distributed database providing persistent storage for the monitoring data forwarded by Collect Agents. Due to the high volume and velocity associated with monitoring data, *noSQL* databases are a natural fit for this purpose.

3.2.2. Design Principles

Having introduced the main components comprising the DCDB architecture, we now describe the design principles according to which they communicate and operate.

Data Flow. Data collection follows the push principle: instead of a central server that pulls data from monitored entities, a distributed set of Pushers close to the data sources acquire data and push it to the Collect Agents. Collect Agents receive monitoring data from their associated Pushers and forward it to their respective Storage Backend for persistent storage, forming the distributed DCDB data storage. Communication between components is performed via established protocols and well-defined APIs, such that each component could be easily swapped for a different implementation, leveraged for other purposes or integrated into existing environments.

Components. The modular and hierarchical design of DCDB facilitates its horizontal scalability as all components are designed to be distributed: depending on the system's size and the number of metrics to be monitored, there may be a large number of Pushers (e.g., tens of thousands), many Collect Agents (up to tens), and one or more Storage Backends. While Collect Agents do not exhibit any cross-talk, Storage Backends need to communicate in order to maintain a distributed database: this, however, can be optimized at the configuration stage. In terms of extensibility, Pushers provide a flexible plugin-based interface that allows for easily adding different data sources via various interfaces. DCDB provides plugins for the most common protocols, but additional ones can be implemented with ease.

Transport Protocol. DCDB employs the *Message Queuing Telemetry Transport* (MQTT) protocol [Loc10] for the communication between Pushers and Collect Agents. We adopted MQTT for a variety of reasons: first, it is a lightweight protocol that was introduced specifically for the exchange of telemetry data between devices in the IoT space, which makes it an ideal fit for sensor monitoring. Moreover, it has a generic design and is well-established, enabling DCDB to be integrable with a vast amount of available tools for data transmission, processing and visualization that support MQTT.

Sensors. In the context of DCDB, each monitored entity producing data points is called a *sensor*. This could be a physical sensor measuring temperature, humidity or power, but can also be any other source of monitoring data such as a CPU performance counter event, the measured bandwidth of a network link or the energy meter of a *Power*

Distribution Unit (PDU). Each sensor's data is in time-series format, in which readings are represented by a time-stamp and a numerical value. The sensors themselves, on the other hand, are identified via unique keys. This format is enforced across DCDB, ensuring uniformity of sensor data.

Sensor Space. Following the MQTT protocol, each sensor in DCDB is identified by a unique MQTT topic, which is a string organized similarly to a file system path. For this reason, topics implicitly define a hierarchy, which we leverage to organize the space of sensors: individual hierarchy levels can be defined by users, but we commonly specify them reflecting the location of monitored entities (e.g., rooms, systems, racks, chassis, nodes, and CPUs). Defining an appropriate hierarchy for sensors is fundamental in a holistic monitoring tool such as DCDB, since the amount of available sensors can potentially become enormous due to the system's size and to the monitoring requirements of different parties (e.g., developers, vendors, operators). By enabling separation of the sensor space, both the navigability and usability of DCDB are greatly improved. We return on this concept in Section 3.4.

3.2.3. Architecture of Wintermute

Following the design guidelines laid out in Chapter 2 we now describe Wintermute, a novel ODA framework with generic interfaces designed to be integrated into any HPC or data center monitoring system as an additional software component.

Overview of the Architecture

In Figure 3.2 we show Wintermute's modular architecture: it is based on *operator plugins* supplying analysis capabilities, which follow an agnostic code interface and are used to instantiate *operators*. Operators represent the actual computational entities performing ODA tasks asynchronously, by relying on a flexible local thread pool. Each operator works on a set of *blocks*, which are container data structures representing physical components (e.g., compute nodes or racks) or logical entities (e.g., user jobs) in a system: a block has a set of sensors that are used as inputs for the analysis (*input* sensors), as well as a set of outputs, which store the results of the ODA operation and are, again in the form of sensors, to be consumed by the monitoring system or by other operators (*output* sensors). We will return to the concept of blocks in Section 3.4.

Operator plugins are supported by two central components, the *query engine* and the *operator manager*, which provide input data to operators and expose their output respectively. These are designed to isolate the plugins from the location in which they are instantiated, meaning that a plugin can be deployed to the different locations of a monitoring system (or different monitoring systems altogether) without alterations. The last set of components, depicted on the right side of Figure 3.2, belongs to the monitoring system in which Wintermute is integrated: the *sensor input* and *sensor output* components identify the interfaces through which Wintermute obtains sensor data and

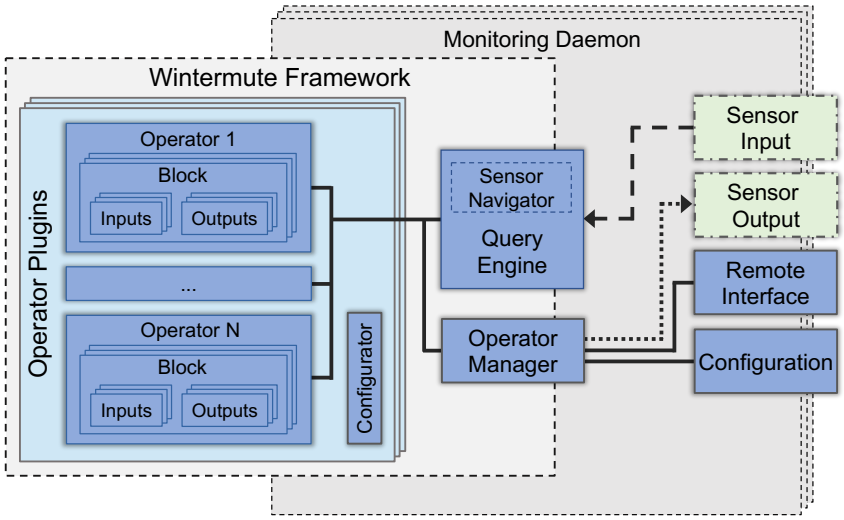


Figure 3.2.: Architectural overview of Wintermute, abstracting from its integration in a monitoring system. We only show the external components with which the framework interacts.

exposes analysis results respectively. The *configuration* component is responsible for initialization and will grant Wintermute access to its designated configuration files, to which the global monitoring system's configuration should point. The *remote interface* component, finally, represents the interface exposed by the monitoring daemon, through which Wintermute can in turn expose its remote control and data retrieval features. While a RESTful API is our preferred interface type, Wintermute can easily be adapted to work with other interface types (e.g., remote shells).

Components of the Architecture

In the following we describe the core components that compose the Wintermute architecture, as well as their interactions with the surrounding monitoring system.

Operator Manager. The operator manager is the central entity responsible for loading Wintermute plugins, exposing the associated configuration files to them and managing their life cycle. As such, it is the main interface between Wintermute and the monitoring system and allows users to specify which sensors to read. Additionally, the operator manager acts as a frontend for all remote interface requests (e.g., via a RESTful API), exposing available actions implemented within the framework. For example, these requests can be used to start, stop or load plugins dynamically, as well as trigger specific actions on a per-plugin basis (e.g., training a machine learning model) and retrieve recent sensor data.

Query Engine. The query engine is a *singleton* component that exposes the space of available sensors to operator plugins. In particular, it gives access to a *sensor navigator* object, which maintains a tree-like representation of the current sensor space using the block system described in Section 3.4, allowing Wintermute plugins to discover which sensors are available and where in the hierarchical structure they stand. The query engine's uniform interface enables queries based on sensor names and time-stamp ranges. Access to low-level sensor data structures is achieved by means of a callback function, which is set at startup by the monitoring entity in which the Wintermute framework is running - access to job data and other metadata can be enabled by setting similar callback functions. Hence, Wintermute plugins do not need to be aware of low-level details about sensor data availability or access.

Operator Plugins. Operator plugins implement the specific logic to perform analysis processes of a certain kind, complying to the Wintermute plugin interface. The analysis carried out by default operator plugins can only take sensor data as input (or its metadata), but other classes of operator plugins can be created. Among them we find job operator plugins: these are an extension of normal operator plugins that can also use job-related data (e.g., user id or node list), producing output associated to a specific job. Plugins consist of the following main internal components:

- **Operator**: operators are objects performing analysis tasks. Each operator has assigned a set of blocks, each referencing a set of input and output sensor objects. Whenever computation is invoked for an operator, it will iterate through its blocks and perform an analysis for each of them, querying the respective input sensors through the query engine, processing the obtained readings, and storing the result in the output sensors. When performing analysis for a certain block, access to the operator's other blocks is allowed for correlation purposes.
- **Configurator**: a configurator is responsible for reading a plugin's own configuration file, exposed by the operator manager, and instantiating operators together with their blocks: the process to generate the latter is controlled by a series of template-based constructs that allow users to easily instantiate a large number of blocks (e.g., one per CPU core in a large-scale HPC system), each with their own unique sets of inputs and outputs. This mechanism is discussed in Section 3.4.4.

3.2.4. Architecture of DCDB

Here we describe in detail the architecture of the DCDB framework as a whole, highlighting the integration of Wintermute in its components and the resulting functionality.

Overview of the Architecture

In Figure 3.3 we show the architecture of the Pusher, Collect Agent and Storage Backend components of DCDB, which are ordered from left to right according to the flow of

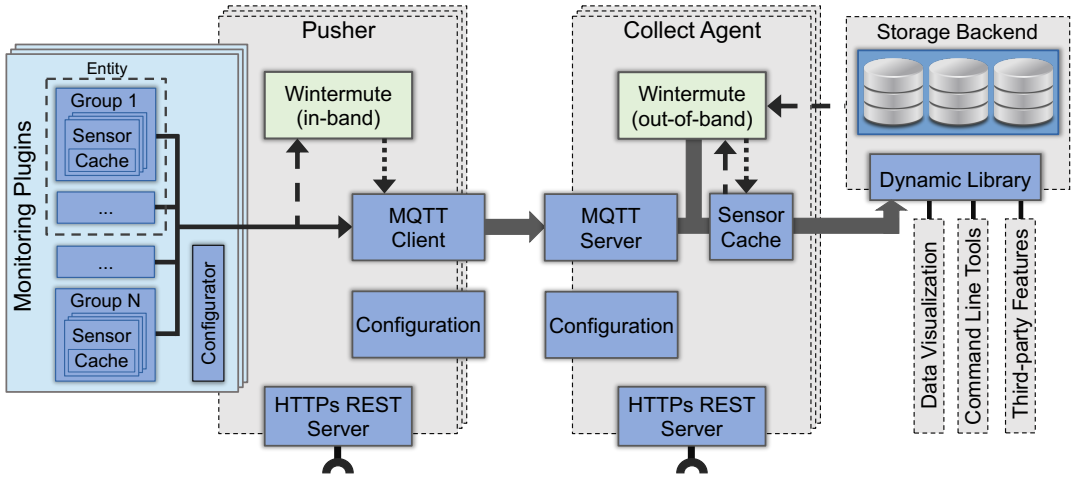


Figure 3.3.: Architectural overview of DCDB highlighting the flow of sensor data, as well as the integration of Wintermute.

data. Starting from the left, Pushers employ *monitoring plugins* to collect data from a variety of sources: a series of *sensor groups* sample certain data sources periodically and asynchronously, storing the associated readings into *sensor* objects. The latter are exposed to an *MQTT client*, which periodically sends out recent sensor readings to a designated Collect Agent as MQTT messages. Specularly, the Collect Agent implements an *MQTT server* which collects incoming sensor readings and forwards them to the designated Storage Backend. In order to hide the low-level database implementation, access to the latter is regulated by a dynamic library which can also be leveraged by users to access stored sensor data conveniently.

It can be seen that the Pusher and Collect Agent share a series of common components: these are *configuration*, which is responsible for reading and parsing DCDB configuration files, and the *HTTPs REST server*, which exposes a RESTful API for remote control. Moreover, both the Pusher and Collect Agent provide configurable *caching* of recent sensor data: in the Pusher this only includes local sensors, while in the Collect Agent the sensor readings received by all Pushers are cached. Wintermute is integrated into both daemons and has access to the resources they provide, such as the RESTful APIs: in the Pusher, in particular, Wintermute has only access to local sensor data and uses the MQTT transport to publish its output; in the Collect Agent, it can access both the local cache and the Storage Backend for sensor data, publishing its output to the latter. More details about Wintermute's integration can be found in Section 3.3.5.

Components of the Architecture

We now describe in detail the single components that are part of the DCDB architecture, focusing on the structure of the Pusher due to its complexity.

Pusher. A Pusher comprises a set of *plugins*, an *MQTT client*, an *HTTPs Server*, and a *configuration* component. The latter is responsible for configuring the Pusher at start-up and instantiating the required plugins; this process is controlled by a set of configuration files that define the data sources for each plugin as well as the global Pusher configuration, using an intuitive *property tree* format - several examples of this format can be found in Appendix B. The HTTPs server provides a RESTful API to facilitate configuration tasks and to access the sensor caches. The MQTT client component periodically extracts the data from each sensor and sends it to the designated Collect Agent. The key components of a Pusher, however, are the plugins that perform the actual data acquisition: these are implemented as dynamic libraries, which can be loaded at initialization time as well as at runtime, allowing to sample different types of data sources. Pusher plugins consist of four template components, each with a specific code interface to be implemented according to a plugin's needs; common logic to all Pusher plugins (e.g., to synchronize sampling) is already implemented and is transparent to developers. The components are the following:

- **Sensors**: the most basic unit for data collection. A sensor represents a single data source that cannot be divided any further. It may represent, for example, the number of L1 cache misses of a CPU core or the power consumption of a device, which are sampled as time-series data. A sensor has to be part of a *group* and has a unique MQTT topic, which is used for all associated MQTT messages.
- **Groups**: the first aggregation level, combining multiple sensors. All sensors that belong to one sensor group share the same sampling interval and are read collectively at the same points in time. Groups are intended to tie together logically-related sensors, such as all power outlets of one power distribution unit and cache-related performance counters of a CPU core.
- **Entities**: an optional hierarchy level to aggregate groups or to provide additional functionality to them. For example, for a plugin reading data from a remote server (e.g., via IPMI or SNMP), a sensor entity may be used by all groups reading from the same host such that a single connection to it can be maintained.
- **Configurator**: the component responsible for reading plugin configuration files and instantiating data collection components. The configurator provides the interface between the Pusher and its plugins, giving access to their entities, groups, sensors and the associated data.

We use the *Network Time Protocol* (NTP) [Mil91] to synchronize sensor read intervals not only within groups, but also across plugins and even Pushers. Moreover, DCDB's push-based monitoring approach allows for more precise timings compared to pull-based monitoring, especially at fine-grained (i.e., sub-second) sampling intervals. This allows for easily correlating different sensors without having to interpolate readings to account for different sampling time-stamps. Additionally, synchronization minimizes

jitter on compute nodes of HPC systems, as parallel multi-node applications will be interrupted at the same time, minimizing load imbalance [FBB08]. Although the data collection intervals of Pushers are synchronized, these will send their data at different points in time in order not to overwhelm the Collect Agents and Storage Backends.

Collect Agent. The Collect Agent is a data broker built on top of a custom MQTT implementation that only provides a subset of features necessary for its tasks. In particular, it only supports the publish interface of the MQTT standard, but not the subscribe interface. As Storage Backends are currently the only consumers of sensor readings, this avoids additional overhead for filtering MQTT topics: this means that, as Pushers publish their sensor readings, Collect Agents always forward them to a Storage Backend, which is the only subscriber to all MQTT topics. However, it is foreseeable that additional subscribers may want to receive certain sensor readings as well for other purposes. This may include, for example, different visualization systems or frameworks for on-the-fly processing of data: in such cases, a third-party MQTT broker in place of the Collect Agent can be used, although with reduced scalability. Upon retrieval of an MQTT message, a Collect Agent parses its contents and inserts into the Storage Backend all sensor readings contained therein using as key the message's MQTT topic.

Storage Backend. The Storage Backend is a distributed database that provides persistent storage of sensor data received from Collect Agents. By its nature, monitoring data is time-series data that is typically acquired and consumed in bulk: it is usually streamed into the database and retrieved for longer time spans, and rarely for single points in time. Due to their time-series nature, the data points for a sensor are organized logically as a tuple of <*sensor ID, time-stamp, reading*>. These properties make monitoring data a perfect fit for a noSQL database in general and wide-column stores in particular, due to their high ingest and retrieval performance for this kind of data.

3.3. Implementation

DCDB and Wintermute are written in C++11 and are freely available under the GNU GPL license via GitLab [13]. In this section we discuss the implementation of DCDB's components as well as the open-source solutions we use.

3.3.1. Software Stack

The current DCDB implementation relies on a series of open-source off-the-shelf solutions for storage and transport, among other features. In order to implement a reliable MQTT client we use the *Mosquitto* library [Lig17] in the Pusher for communication, which proved to be the most suitable for our purposes in terms of scalability, stability and resource footprint. Moreover, being a single-thread implementation, Mosquitto's impact on system performance is predictable and in general low. The Collect Agents, on

the other hand, rely on a custom MQTT broker implementation in order to maximize performance and scalability, while sacrificing certain parts of the MQTT standard which were deemed superfluous for HPC monitoring.

We rely on Apache Cassandra [WT12] to implement Storage Backends. Due to the modularity of our implementation, Cassandra could easily be swapped for a different database, such as *InfluxDB* [14], *KairosDB* [15] or *OpenTSDB* [16]: however, we chose Cassandra not only because it fits well the semantics of monitoring data, but also because of its built-in data distribution mechanism that allows us to distribute a single database over multiple server nodes either for redundancy, scalability, or both. This feature works in synergy with the distributed architecture of DCDB and allows us to scale our deployments to arbitrary size effectively. As Cassandra may be distributed across multiple servers (a *cluster* in Cassandra terminology), any of those servers may be used to insert or query data. The distribution of data within the cluster can be controlled via a so-called *partition key* and a *partitioning algorithm*. We exploit this feature by leveraging the hierarchical MQTT topics as Cassandra partition keys, with a corresponding partitioning algorithm: we map different portions of the MQTT topic space (i.e., sub-trees in the sensor hierarchy) to different database servers, thus improving data locality and reducing network traffic: a practical example of this is storing the monitoring data of different HPC systems to separate servers, all of which are part of the same distributed Cassandra instance. The same logic is applied for queries, directing them to the correct server.

Finally, we rely on the *Boost* libraries [Kar05] throughout DCDB to implement our RESTful APIs and thread pools, as well as for logging, parsing of configuration files and other features. On top of those discussed here, DCDB has a series of other minor dependencies: these, however, are mostly dependencies of Pusher or Wintermute plugins necessary to achieve certain monitoring or ODA capabilities, and are not required to compile the core DCDB components.

3.3.2. Available Plugins

We currently provide many different Pusher plugins, supporting in-band resource performance counters (*Perfevents* [Wea13], *Model-Specific Register* (MSR) [Bau+16] and *Nvidia Management Library* (NVML) [17]), server-side sensors and metrics (*ProcFS* [18] and *SysFS*), I/O metrics (*General Parallel File System* (GPFS) [SH02] and Intel Omni-Path, or *OPA* [19]), out-of-band sensors of IT components (*Intelligent Platform Management Interface* (IPMI) [Int13] and *Simple Network Management Protocol* (SNMP) [Cas+90]), RESTful APIs, and building management systems (*BACnet* [Ame10]). System administrators deploying DCDB are encouraged to extend Pushers according to their own needs by implementing plugins for new data sources. To simplify their implementation DCDB provides a series of *generator* scripts, which create all files required for a new plugin and fill them with code skeletons to connect to the plugin interface. Comment blocks point to all locations where custom code has to be provided, greatly reducing the effort required to implement a new plugin.

Similarly, Wintermute provides a series of plugins to perform most common ODA tasks, such as aggregation of different sensors, smoothing over time and computation of performance indicators (e.g., *Cycles Per Instruction* (CPI)). Furthermore, Wintermute provides a series of machine learning-oriented plugins to perform online regression, classification or clustering of sensors. Finally, a series of *sink* plugins are also available: these have the task of writing sensor data to specific data sinks without producing any output, and are meant as the final stage of a Wintermute ODAC pipeline. More details about available plugins can be found in Appendix A.

3.3.3. Interfaces

DCDB provides several interfaces to access the monitoring data. Users and administrators of an HPC system can do this with the support of a dynamic library (*libDCDB*), as well as with command line tools built on top of it and via RESTful APIs. Furthermore, data can be easily visualized and analyzed using the *Grafana* tool [20].

libDCDB. Access to Storage Backends is performed via a well-defined API that is independent from the underlying database implementation, hiding unnecessary low-level details to users and simplifying the task of switching to a different database implementation. Currently, the API has only been implemented in a C++ library, libDCDB, but other bindings could easily be implemented as well. Additionally, the C++ library can also be used in Python scripts and hence covers a wide range of use cases. libDCDB allows to perform a large variety of tasks associated with the storage of monitoring data: these include the insertion and retrieval of sensor data, the management of metadata and user job information, down to schema-related configuration operations.

Virtual Sensors. LibDCDB supports the definition of *virtual sensors*, which supply a layer of abstraction over raw sensor data and can be used to provide derived or converted metrics. A virtual sensor is generated according to user-specified arithmetic expressions of arbitrary length, whose operands may either be sensors or virtual sensors themselves. This can be used, for instance, to aggregate data from several sources in order to gain insight about the status of a system as a whole (e.g., aggregating the power sensors of individual compute nodes in an HPC system), or to calculate key performance indicators such as the PUE. Virtual sensors can be used like normal sensors and are evaluated lazily: they are only computed upon a query and only for the queried period of time. As queries to virtual sensors may potentially be expensive (in terms of computation as well as I/O), results of previous queries are always written back to a Storage Backend so that they can be re-used later. The units of the underlying physical sensors are converted automatically and we account for different sampling intervals by linear interpolation. While Wintermute is equally capable of performing aggregation between sensors, virtual sensors are a preferable alternative when their data is rarely required and lazy evaluation is more efficient.

Command Line Tools. DCDB offers a series of command line tools that leverage libDCDB for access to Storage Backends. The *DCDBconfig* tool allows administrators to perform basic database management tasks (e.g., deleting old data) as well as configuring and fetching the properties of sensors such as units and scaling factors or defining virtual sensors. The *DCDBquery* tool then allows users to obtain sensor data for a specified time period in CSV format or perform basic analysis tasks on the data such as integrals or derivatives. Additionally, a series of secondary tools offers utility features, like a *CSVimport* tool to import CSV data into Storage Backends.

RESTful APIs. Pushers and Collect Agents further support data retrieval through RESTful APIs. In a Pusher this interface can be used to retrieve the current configuration (e.g., of plugins or sensors) and allows for starting and stopping individual plugins. This can be useful, for example, to avoid conflicts with user software accessing the same data source, or to enable additional data sources for individual applications. Additionally, plugins can be reloaded at runtime after a configuration change, allowing for seamless re-configuration without interrupting the Pusher. Further, the RESTful API also provides access to the sensor caches in Pushers and Collect Agents, which can be used by other processes (either locally or remotely) to easily read any sensor from user space. The RESTful APIs allow control of Wintermute plugins as well: on top of the reload, start, stop and cache access actions mentioned earlier, they can also be used to trigger on-demand computation or other plugin-specific actions (see Section 3.3.5).

3.3.4. Visualization of Data

DCDB leverages the open-source Grafana tool for the visualization of monitoring data. Grafana is a nice fit for DCDB, and monitoring in general, since it provides a comprehensive set of visualization options (e.g., graphs, heatmaps, histograms and tables) and it allows users to define alerts and receive notifications via multiple channels, which is a desirable feature for system administrators. Moreover, it is designed following an extensible architecture, allowing to integrate arbitrary data sources via plugins, and finally it has a strong user and development community. However, although Grafana supports several database implementations, it does not provide any plugin for Apache Cassandra. We therefore develop our own libDCDB-based plugin, which profits from current and future features offered by DCDB on top of offering standard Apache Cassandra integration.

To the best of our knowledge, all existing open-source Grafana plugins for different database implementations miss the possibility to exploit the hierarchy of available metrics to allow, for example, to only select metrics at a specific level or build hierarchical queries. This functionality is useful in HPC or data center environments, where system administrators could make use of browsing the different hierarchical levels of a system (e.g., a rack, a chassis, or a server) and query data from sensors available at a certain level. This becomes particularly beneficial if the monitored environment comprises a very large number of sensors, which can potentially be in the order of millions for large-scale

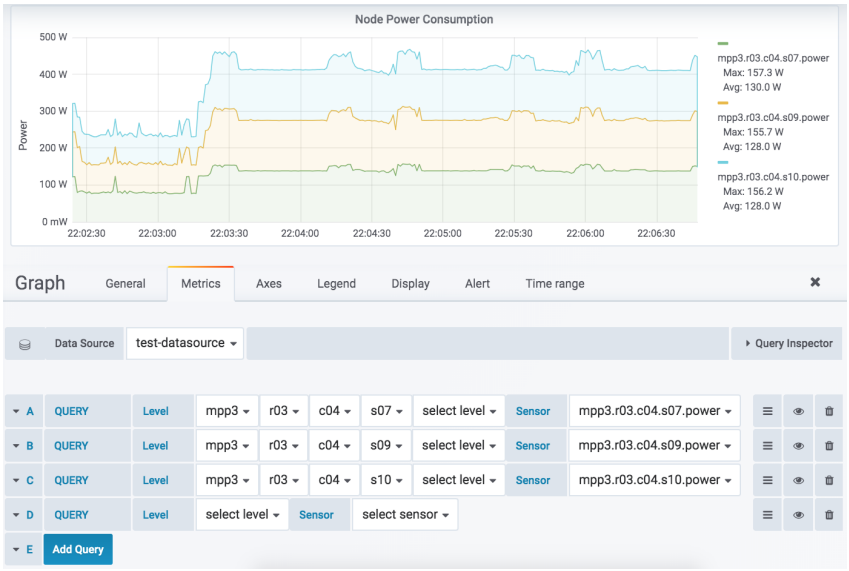


Figure 3.4.: A Grafana dashboard using the DCDB data source plugin [Net+19b].

HPC systems. As DCDB incorporates this kind of hierarchical sensor scheme in all of its components (see Section 3.4), our data source plugin also exposes it in Grafana. Figure 3.4 illustrates a visualization example of this feature, specifically showing the power consumption of three different compute nodes on one of our production systems at LRZ: users can query sensors at a specific hierarchical level by navigating through them with the support of multiple drop-down menus, which are shown for the three queries (i.e., A, B and C) associated with the sensors. The visualization of data may further benefit from convenient Grafana built-in features such as stacking of time-series data or formatting of axes and legends (e.g., for displaying running averages or other statistics). In order to keep the plugin's implementation simple, we encapsulated all sensor hierarchy and data retrieval-related logic in a small, ad-hoc server with which the data source plugin communicates using a RESTful API.

3.3.5. Workflow of Wintermute

Here we discuss the workflow and key features that are brought about by the integration of Wintermute in DCDB. The options that we describe here are backed by examples and extensive documentation on the DCDB GitLab repository [13], and accommodate all of the requirements laid out in Section 2.3.3 by providing effective ways to implement a lightweight and reliable ODA infrastructure.

Operator Location. As described in Section 3.2.4, Wintermute is included in Pushers and Collect Agents: as such, operators can be instantiated in both locations by loading the appropriate plugins. In a Collect Agent, access to the entire system's sensor space is available (out-of-band). If possible, data is retrieved from the local sensor cache or

otherwise queried from the Storage Backend, to which the outputs of operators are also written. This location is optimal for system or infrastructure-level analysis and control. In a Pusher, on the other hand, operators have only access to locally sampled sensors and their sensor cache data (in-band). This location is optimal for runtime models requiring data liveness, low latency and, when distributing models across a large number of Pushers, horizontal scalability. For example, a regression operator used to predict power consumption for CPU frequency tuning [Oze+19] can be deployed in a Pusher so as to leverage in-memory processing for minimal latency.

In both scenarios, the query engine gives higher priority to data in the local sensor caches, which is faster to retrieve compared to querying the Storage Backend. Moreover, queries can be performed in two modes, affecting how the caches are accessed: in the first, *relative* time-stamps are supplied as an offset against the most recent reading, and the cache view to be returned can be computed in $O(1)$ time. In the second, *absolute* time-stamps are used, resulting in a binary search with $O(log(N))$ time complexity.

Operational Modes. In our Wintermute implementation, operators can be configured to work in two different ways depending on their requirements. In *online* mode, an operator is invoked at regular time intervals, producing time series-like sensor data as its output and thus resulting in continuous analysis. This is ideal for applications such as fault detection or runtime tuning. In *on-demand* mode, on the other hand, an operator's capabilities must be explicitly invoked via the RESTful API (or, in general, any remote interface) by querying a specific block. Output data is propagated only as a response to the RESTful request. This mode is ideal for supporting scheduling systems, which can be triggered at arbitrary times, and is in general a very effective way to limit overhead and data volume. For example, a resource manager could contact an on-demand fault detection operator at scheduling time [Tun+18] to determine the current status of each idle compute node and thus optimize allocation decisions.

Block Management. When using the online mode, the blocks of a single operator can be arranged with respect to the underlying model: as *sequential*, all blocks share the same operator, and are processed sequentially at each computation interval to avoid race conditions; as *parallel*, one distinct operator is created for each block, allowing to parallelize computation and improve scalability. For example, an application fingerprinting operator [Ate+18] in a Collect Agent with one block per compute node of an HPC cluster, could make use of the parallel option for scalable ODA performance.

Analysis Pipelines. As the output data produced by online operators shares the same format and is identical to all other sensor data in DCDB, operators can use the output of other operators as input. This, in turn, allows developers to create an *operator pipeline*, in which the multiple stages of a complex analysis are divided among several operators. This can be used to split computational load between multiple locations (e.g., Pusher and Collect Agent) or to achieve complex analyses with few, general-purpose plugins.

Furthermore, this method allows us to implement control loops in any system, via a *sink operator* (see Section 3.3.2) at the end of a pipeline that uses processed data to tune knobs. For example, an operator which estimates the optimal inlet cooling water temperature for an HPC cluster based on sensor data [Con+15] could feed its predictions to a second operator, devoted to issuing SNMP requests to tune the cooling system. Careful planning of operator pipelines can reduce redundancy in processing of sensor data, by allowing multiple operators to consume the data produced by a single one.

3.4. The Block System

With more and more data sources to tap into, navigating the space of available sensors in a monitored HPC or data center environment and configuring ODA models (particularly ODAC models) at scale becomes difficult and error-prone [Gim+17]. This calls for a structured sensor specification system with the tasks of simplifying the navigation of large sensor spaces with millions of entries, as well as allowing to derive the hierarchical relationships existing between sensors. Finally, such a system must provide template-like constructs to simplify sensor specifications for ODA models. Here, we introduce the set of abstractions we designed and subsequently implemented in Wintermute to address these issues, with the final goal of simplifying the specification of complex ODA models in large-scale environments, in a generic way.

3.4.1. The Sensor Tree

In Wintermute we model the sensor space as a hierarchical *sensor tree*, an example of which is depicted in Figure 3.5. In the case of DCDB, the keys used to identify sensors are MQTT topics - hence, we will use this format in the rest of the section for our examples. Topics are forward slash-separated strings similar to file system paths, expressing the physical or logical placement of a sensor in an HPC system, room or data center. For example, */rack4/chassis2/server3/power* is a valid sensor topic.

The last segment of a topic is the name of the sensor itself, and the preceding path elements express its placement. This representation can be exploited to construct a tree, in which each internal node is a system component (e.g., a compute node or a rack) and each leaf is a sensor. The constructed tree then supplies a comprehensive view of the monitored environment's structure, as well as a natural way to correlate hierarchically-related sensors (e.g., the sensors of a compute node and those of the rack it belongs to). The structure of the sensor tree is analogous to a file system: components of an HPC system or facility represented by internal tree nodes can be seen as *directories* whereas the sensors themselves corresponding to leaves are akin to *files*. This approach has been already employed in efforts such as *Perftrack* [Kna+07], proving its effectiveness. Here, we extend it for the purpose of ODA model configuration.

The effectiveness of this representation depends on the level of detail expressed by the hierarchy of topics, and the responsibility for devising such a hierarchy lies on system

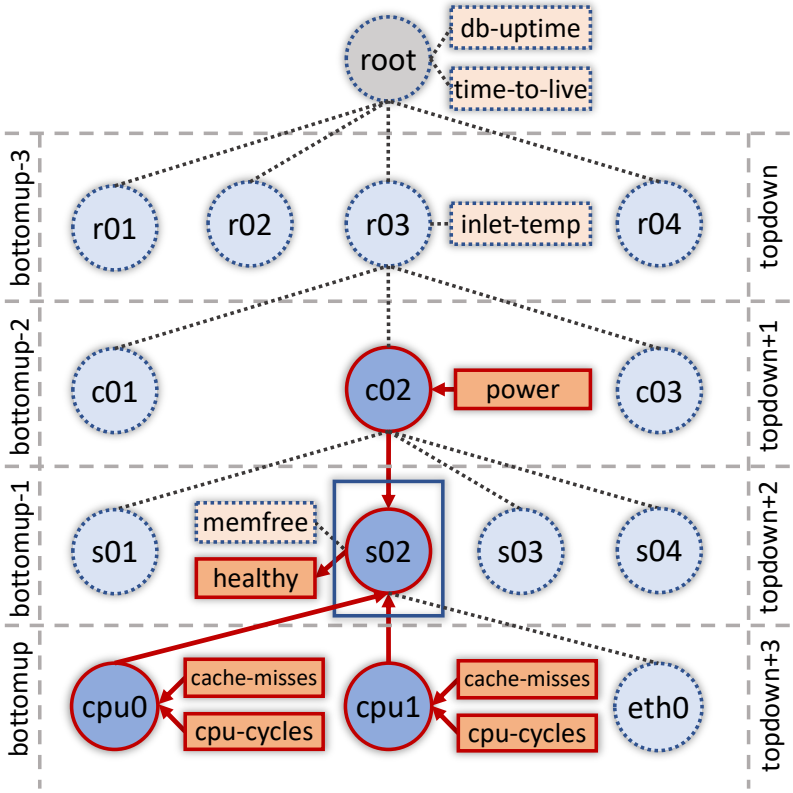


Figure 3.5.: An HPC system's sensor tree and a Wintermute block. Circles and rectangles respectively represent internal tree nodes and sensors. Red, non-dashed lines highlight nodes and sensors belonging to the block, while the others are collapsed for convenience [Net+20].

administrators and designers. Some data centers might also employ monitoring systems with naming conventions different from the one we discuss here. This is taken into account in our implementation of Wintermute, which is not dependent on a specific naming scheme, but also supports the definition of arbitrary hierarchy schemes supplied as lists of regular expressions, each identifying a separate level.

3.4.2. Blocks and Block Templates

In Wintermute, *blocks* are data structures that act as atomic containers on which analysis computations are performed. A block represents a node in the sensor tree, from which it takes its *name*. Then, a block references a set of *input* and *output* sensor topics: the output sensors are used to deliver analysis results, and are leaves of the node the block represents. Input sensors, which provide the data for the analysis, can either be leaves of that same node, or belong to any other node in the sensor tree connected by an ascending or descending path to it. Figure 3.5 shows a generic example for a block, named *s02*, a compute node in an HPC system. In this example, the block has the output

Listing 1 An example of a Wintermute block and the associated sensor expressions.

```
input:
    <topdown+1>power
    <bottomup, filter cpu>cpu−cycles
    <bottomup, filter cpu>cache−misses
output:
    <bottomup−1>healthy
```

sensor *healthy*, and a series of input sensors: the *cycles* and *cache misses* counters of the CPUs in the compute node, plus the *power* sensor of the chassis it belongs to. With its input and output sensors, a block forms a *sub-tree* in the sensor tree.

While blocks can be defined by specifying actual sensor topics as inputs and outputs, they may also be defined in a generic way via *templates*: here, sensors are referenced via their position in the sensor tree, expressing only their last topic segment and omitting the components to which they belong (preceding topic path), which are replaced by a tree level (*vertical* navigation) and a filter (*horizontal* navigation). In particular, vertical navigation allows to identify different levels in a sensor tree and the relationships between them (e.g., between CPUs and compute nodes in Figure 3.5), while horizontal navigation is needed to select specific nodes within a certain tree level (e.g., CPUs or Ethernet interfaces in Figure 3.5). The set of sensor topics matched by a sensor expression is its *domain* in the tree. Further, a particular block can be instantiated from a template by specifying a node in the sensor tree (i.e., its name), thus creating a binding: each sensor expression is then replaced with a sensor topic from its domain that is hierarchically-related to the block's node. Since multiple topics may satisfy this, one expression can produce multiple actual topics. Conversely, if no topic satisfies it, the block cannot be built.

Recalling the similarity between the sensor tree and a file system, describing sensors through sensor expressions can be interpreted as defining files using *relative paths*: these paths can match multiple points in the file system, and they are fully resolved in function of the *current working directory*, whose analogous in this case is the name of the block. The main difference between the two is that the tree level of sensors in sensor expressions is defined with an absolute level, whereas for relative file system paths it is defined as a relative offset with respect to the current working directory.

3.4.3. Template Instantiation

The example block shown in Figure 3.5 can be built from a generic template using the sensor expressions depicted in Listing 1. In sensor expressions, the *topdown* and *bottomup* keywords drive the *vertical* navigation and indicate the highest and lowest level in the tree, respectively; the root node of the sensor tree is excluded from this representation,

and other levels can be reached through relative offsets. The *filter* keyword defines the *horizontal* navigation and is used to filter the set of topics that the expression matches, within its tree level. In this example, the block's name is set to */r03/c02/s02/*, which identifies an HPC compute node. Once this is set, the rest of the block is resolved: the *power* expression is resolved as */r03/c02/power*, since it specifies that the sensor should be one level below the highest tree level, at *c02*. Conversely, the *cpu-cycles* and *cache-misses* expressions are on the lowest level, with two nodes (*cpu0* and *cpu1*) belonging to their domains. As such, sensors from both of them are added to the block. As the *healthy* output sensor expression lies at the same level as *s02*, it is simply resolved as */r03/c02/s02/healthy*.

3.4.4. Configuring Blocks in Wintermute

Templates are used in Wintermute's plugin configurators to instantiate the blocks operators work on, as explained in Section 3.2.3. In detail, block generation works in the following steps, starting from a block template defined in a configuration file:

1. Based on the current sensor tree, the set of topics matching the expressions of the output sensors (their domain) is computed. The expressions of all output sensors must be equal to prevent ambiguity.

2. For each retrieved sensor topic in the domain, we extract its location (preceding topic path) and create a corresponding block using this as name.

3. For each instantiated block, its set of input and output sensors is resolved according to the domains of the respective expressions.

On top of block-level outputs, users may also define a set of operator-level outputs that can, for example, store the average error of a model applied to a set of blocks. Recalling the example of template in Section 3.4.3, applying the configuration algorithm described above will result in as many blocks as compute nodes in the HPC system (e.g., */r03/c02/s[01-04]/*). This demonstrates how the block system enables instantiation of thousands of independent ODA models in large-scale systems, each with its own set of sensors, by using only a small configuration file. Moreover, configurations are independent from the location to which a model is deployed (e.g., Pusher or Collect Agent), as the blocks are resolved automatically from the available sensor tree. It should be noted, however, that a block template is not guaranteed to be portable across systems or facilities with different sensor hierarchies. For example, if we wanted to port the template in Section 3.4.3 to a system whose hierarchy has an additional level close to the root of the tree, with block names such as */i01/r03/c02/s02/*, the expression <*topdown+1*>*power* must become <*topdown+2*>*power*. Additional details and examples regarding the Wintermute configuration process can be found in Appendix B.

3.5. Experimental Evaluation

In this section we discuss the performance of DCDB and Wintermute from different viewpoints. We evaluate the performance of Pushers in both production and test configurations, followed by an analysis of the Collect Agents. Finally, we look at the performance of Wintermute. By characterizing all aspects of DCDB's performance, from overhead to scalability and resource footprint for all of its components, we are able to demonstrate its suitability for large-scale HPC and data center installations.

3.5.1. HPC Systems and Monitoring Configurations

A variety of HPC systems with different scales are active at LRZ. Their composition and the specific architectures are summarized in Table 3.1 - for the sake of brevity, we omit the systems that are solely housed at LRZ and are not available for public use, as well as some secondary systems that are not relevant in the context of this dissertation. The *CooLMUC-2* and *CooLMUC-3* HPC systems belong to the *Linux-Cluster* [21], a class of systems designed for open use by universities and research institutions in Bavaria. In detail, CooLMUC-2 is a medium-scale HPC system with 812 compute nodes, each equipped with 2 Intel Haswell CPUs and a Mellanox Infiniband [22] interconnect; CooLMUC-3, on the other hand, is a small-scale system with 148 nodes, each equipped with a many-core Intel Knights Landing CPU, 16GB of *High-Bandwidth Memory* (HBM) and an Intel Omni-Path [19] interconnect.

SuperMUC-NG [6] is the flagship HPC system at LRZ, reaching a peak performance of roughly 27PFlop/s. It has been among the 10 most capable HPC systems in the world until June 2020, as per the Top500 list [23]. The system comprises a total of 6,480 nodes, each equipped with 2 Intel Skylake CPUs, distributed over 8 islands each containing 792 nodes. It employs a fat-tree network topology using the Intel Omni-Path interconnect technology. All systems employ warm-water cooling, while SuperMUC-NG also uses *adsorption chilling* (i.e., using the heat in waste warm water to produce new cold water). These technologies are very effective in Munich's temperate weather, improving both energy efficiency and waste heat re-use [Wil+17]. All of the HPC systems use *SUSE Linux Enterprise Server* (SLES) as operating system and the *Simple Linux Utility for Resource Management* (SLURM) [YJG03] to schedule and allocate user jobs. Similarly, a large-scale GPFS distributed file system provides the main storage for all systems. While SuperMUC-NG's and CooLMUC-2's CPUs provide strong single-thread performance, CooLMUC-3's many-core CPU is comparatively weak in this regard.

On the three systems at our disposal we test a variety of production DCDB configurations, which are also depicted in Table 3.1. These leverage the same DCDB Pusher plugins: ProcFS collects data from the *meminfo*, *vmstat* and *procstat* files, whereas we use SysFS to sample various temperature and energy sensors. We use Perfevents to sample performance counters on CPU cores, and finally OPA to measure metrics related to Omni-Path. Here, we only deploy plugins that perform in-band measurements with a sampling interval of 1s; out-of-band measurements would be performed on separate

Table 3.1.: The architectures of the main HPC systems at LRZ and the associated DCDB configurations as of February 2021.

HPC System	SuperMUC-NG	CooLMUC-2	CooLMUC-3
Nodes	6,480 Skylake	812 Haswell	148 KNL
CPU	Intel Xeon Platinum 8174 2 cpus x 24 cores x 2 threads	Intel Xeon E5-2697 v3 2 cpus x 14 cores x 2 threads	Intel Xeon Phi 7210-F 64 cores x 4 threads
Memory	96GB	64GB	96GB 16GB HBM
Interconnect	Intel Omni-Path	Mellanox Infiniband	Intel Omni-Path
Plugins	Perfevents, ProcFS, SysFS, OPA	Perfevents, ProcFS, SysFS	Perfevents, ProcFS, SysFS, OPA
Sensors	2,477	750	3,176
Overhead	1.77%	0.69%	4.14%

machines and hence do not incur overhead on running applications. Additionally, in some experiments we only deploy a special *Tester* plugin, which can generate an arbitrary number of sensors with negligible overhead. This allows us to isolate the overhead of the various monitoring backends (e.g., Perfevents) from that of the Pusher, which is mostly communication-related. We use a dedicated server to host our Collect Agent and Storage Backend, equipped with two Intel E5-2650 v2 CPUs, 64GBs of RAM and a 240GB Viking Tech SSD drive.

3.5.2. Experimental Methodology

Here we present our experimental methodology, from the reference applications that we use to the performance metrics that we evaluate throughout our experiments.

Reference Applications

In order to estimate how DCDB impacts running applications, we perform tests by executing instances of well-known benchmarks on multiple architectures. First, we present a series of tests replicating a production environment (Section 3.5.3), where we characterize the impact of DCDB on real applications sensitive to network and memory bandwidth, as well as CPU contention. To this end, we employ a selection of *Message Passing Interface* (MPI) proxy applications from the CORAL-2 suite [24], which cover a large portion of the the HPC applications' spectrum - hence, results obtained with these can be considered representative of real workloads. The applications are as follows:

- **Kripke**: a structured deterministic transport using wavefront algorithms, which stresses both memory and network latency and bandwidth [KBB15].

- **AMG**: a parallel algebraic multigrid solver for linear systems. It is highly memory and network-bound, relying on many small messages [Yan+02].
- **LAMMPS**: a classical molecular dynamics code, which evenly stresses most components in an HPC system [Pli95].
- **Quicksilver**: a Monte Carlo transport benchmark, with significant branching and stressing memory latency [Ric+17].

Due to the large spectrum of performance behaviors they cover, the applications above allow us to fully characterize the performance of DCDB on our systems. In a subset of tests and case studies presented in Sections 3.6 and 3.7, on the other hand, we use the following two additional CORAL-2 applications:

- **Nekbone**: a mini-app derived from the Nek5000 Navier-Stokes CFD solver, implementing a compute-intensive conjugate gradient solver [Fis+08].
- **PENNANT**: a mini-app for hydrodynamics on unstructured meshes, making use of highly irregular memory access patterns [Fer15].

Later on, we characterize the impact of DCDB on computational resources and its scalability with various test configurations (Section 3.5.3), stressing the communication and sampling subsystems. Here, we focus on using the shared-memory version of the *High-Performance Linpack* (HPL) benchmark [DLP03], supplied with the Intel *Math Kernel Library* (MKL) [25]. Being a compute-bound application, tests performed against HPL give us insights on the behavior of DCDB in a worst-case scenario. All benchmarks are configured to instantiate one MPI process (where possible) per node, and use as many OpenMP threads as physical CPU cores that are available. Pushers are configured to use 2 sampling threads and a sensor cache size of 2 minutes.

Evaluation Metrics

Each experiment involving application runs was repeated 10 times to ensure statistical significance and, to account for outliers and performance fluctuations, we use median run times. We then use the following metrics to evaluate the performance of DCDB:

- **Overhead** is defined as the fraction of time an application spends in excess due to interference from Pushers, compared to running without DCDB active. We obtain it by comparing the *reference* execution time (T_r) of an application against the one observed when a Pusher is active (T_p), and we compute it as $O = (T_p - T_r)/T_r$. The overhead helps quantify the impact of a component on system performance, and by using reference applications we obtain scalable and reproducible experiments. Our evaluation process does not include system throughput among the selected metrics as it would significantly depend on the underlying workload, and as such would require a large-scale dedicated environment. Nevertheless, based on our

experience on production systems, we do expect low overhead to directly translate into low throughput change.

- **CPU Load** is defined as the percentage of active per-core CPU time spent by a process against its total run time, as measured by the Linux *ps* command; this metric characterizes a Pusher's and a Collect Agent's performance when used in a out-of-band context, with no overhead concerns.
- **Memory Usage** of a process is quantified by *ps*, in terms of total allocated virtual memory. It helps characterize the impact of different monitoring configurations in Pushers and Collect Agents.

3.5.3. Pusher Performance

First, we present the performance of Pushers in terms of computational overhead against different types of applications running on an HPC system.

Overhead in a Production Configuration

To assess a Pusher's performance in typical production environments, we leverage the actual configurations that we use on our systems, described in Table 3.1. We measured the overhead against the CORAL-2 MPI proxy applications with different node counts on our SuperMUC-NG system, using a weak scaling approach. We present our results in Figure 3.6. The experiment was performed twice: once with the Pusher configuration presented in Table 3.1 (labeled *production*), and once using a configuration with the same number of sensors, produced with the Tester plugin (labeled *test*). It can be seen that the overhead for LAMMPS, Quicksilver and Kripke is low and never goes above 3%. Moreover, when scaling the number of nodes, the overhead increase is minimal. The AMG benchmark represents an exception, showing a linear increase with respect to the node count, and peaking at 9% with 1,024 nodes: this application is notorious for using many small MPI messages and fine-granularity synchronization. Consequently, it is extremely sensitive to network interference. This is also confirmed by the experiments with the Tester plugin: LAMMPS, Quicksilver and Kripke are affected to a very limited extent by the Pushers' network interference, whereas in AMG this causes most of the total overhead. Moreover, we observed the best performance for AMG when Pushers were configured to send sensor data to a Collect Agent in regular bursts twice per minute, reducing network interference. The remaining applications, on the other hand, perform better when the Pushers' data is sent out continuously in a non-bursty manner (as shown here). This type of interference can be avoided by using a separate network interface (e.g., for management) to transmit data.

We present overhead results against single-node HPL runs for all three HPC systems in Table 3.1. HPL is designed to measure the peak floating-point performance of computing systems by solving a system of linear equations: it is hence purely compute-bound and extremely sensible to interference by other processes [DLP03]. Moreover, HPL is the

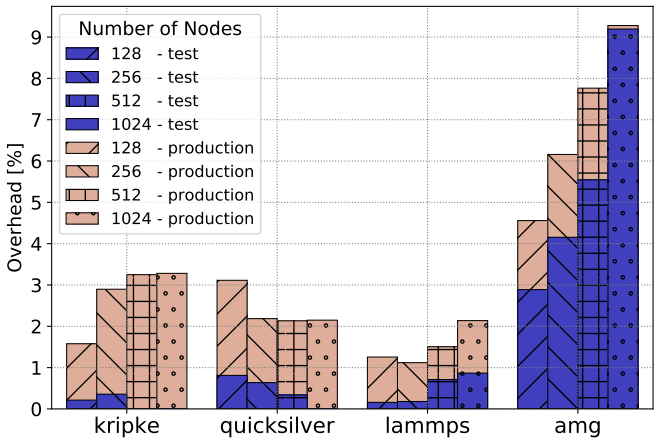


Figure 3.6.: Overhead of DCDB Pusher against four CORAL-2 MPI benchmarks on SuperMUC-NG, using production and test configurations [Net+19b].

reference benchmark used to rank HPC systems in the Top500 lists [1]. These features make HPL the ideal testbed to evaluate DCDB's worst-case overhead in a generic fashion. In most configurations the overhead is low despite the large number of sensors being pushed each second. The worst performer is CooLMUC-3: this was expected due to the weak single-thread performance of its Knights Landing CPU and the much larger number of collected sensors than in the SuperMUC-NG and CooLMUC-2 configurations, which is due to the large number of *Simultaneous Multithreading* (SMT) cores on this architecture. Average memory usage ranges between 25MB (CooLMUC-2) and 72MB (CooLMUC-3), whereas average per-core CPU load ranges between 1% (CooLMUC-2) and 9% (CooLMUC-3).

Overhead in a Test Configuration

In this second part we estimate the scalability of the Pusher's core, once again using the Tester plugin. We analyze a total of 25 configurations, which differ in terms of sampling intervals and number of sensors. Figure 3.7 depicts the computational overhead against single-node HPL runs, for each of the three analyzed HPC systems. In the plot, a value of 0 denotes no overhead, meaning that the median run time when using a Pusher in the experiment was equal or less than the reference median run time. In all cases the overhead is low and it is below 1% in all configurations with 1,000 sensors or less, which are typical for production environments. Even when pushing 100,000 sensor readings per second (10,000 sensors sampled every 100ms) overhead remains acceptable for all platforms. The SuperMUC-NG node architecture, in particular, is unaffected by the various Pusher configurations and shows consistent overhead values, likely due to its Skylake CPU. The slightly older CooLMUC-2 and CooLMUC-3 show clearer gradients with increasing overhead in the most intensive configurations, with the latter of the two exhibiting the worse results due to the weak single-thread performance of its CPU.

(a) SuperMUC-NG. (b) CooLMUC-2. (c) CooLMUC-3.

Figure 3.7.: Heatmaps of DCDB Pusher's overhead at various sampling intervals and sensor counts on three HPC systems, against HPL [Net+19b].

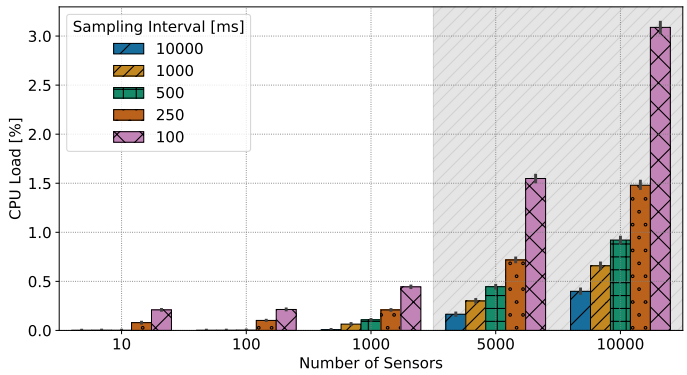
DCDB is thus usable in scenarios that involve large numbers of sensors as well as high sampling rates.

In Figure 3.8 we show results in terms of average per-core CPU load and memory usage for each configuration. We show results only for the SuperMUC-NG nodes, since all node types scale similarly. Memory usage is dependent on both the sampling interval and number of sensors, as these will result in different sensor cache sizes. In the most intensive configuration with 100,000 sensor readings per second, memory usage averages at 350MB, and is well below 50MB for typical production configurations with 1,000 sensors or less. It can be further reduced by tuning the size of sensor caches. CPU load peaks at 3% in the most intensive configuration, proving that there is ample room for much more intensive and fine-granularity configurations, especially in out-of-band environments where computational overhead is not a concern.

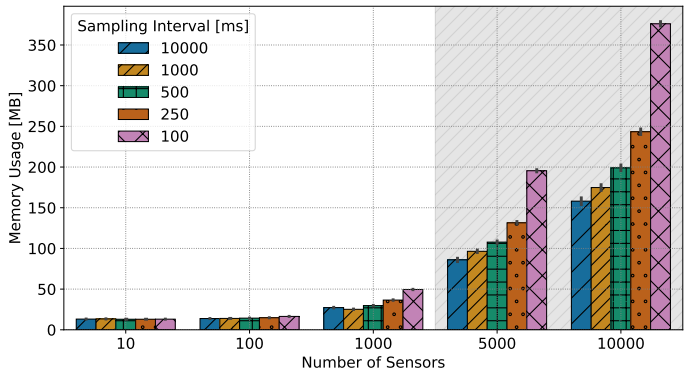
Performance Scaling Modeling

Being able to forecast the impact of a certain monitoring configuration on system performance, before the deployment stage, can reduce maintenance efforts and is hence desirable. In light of this, we now discuss a generic model to infer the performance of the Pusher in terms of per-core CPU load on each architecture at our disposal, as a function of the sensor rate (i.e., the ratio between the number of sensors and the sampling interval). We use the performance data obtained in the experiments in Section 3.5.3 on our three HPC systems. Figure 3.9 shows the observed average per-core CPU load across configurations as well as fitted curves resulting from linear regression. Since we cover a broad range of sensor rates, the X-axis is shown in logarithmic scale.

Results show that varying levels of performance are achieved across the reference architectures. SuperMUC-NG's Skylake architecture, in particular, shows the best scaling curve with 3% peak CPU load, whereas CooLMUC-3's Knights Landing once again shows the worst results, with 8% peak CPU load. In all architectures, however, CPU load is below 1% for configurations with a sensor rate of 1,000 or less. Most importantly,



(a) Average per-core CPU load.



(b) Average memory usage.

Figure 3.8.: Average DCDB Pusher CPU load and memory usage on a SuperMUC-NG compute node. The white area highlights typical configurations for production HPC and data center environments [Net+19b].

Pushers follow a distinctly linear scaling curve on all architectures. This implies that system administrators can infer the average CPU load of the Pusher on a certain CPU architecture by means of linear interpolation, with the following equation:

$$L_p(s) = L_p(a) + (s - a)\frac{L_p(b) - L_p(a)}{b - a} \tag{3.1}$$

In Equation 3.1, L_p is the average CPU load, while s is the target sensor rate, and a and b are two reference sensor rates for which the average CPU load was measured. For the models fitted in Figure 3.9, the median relative error with respect to the original data points is roughly 33%: since we are interested in obtaining forecasts about the CPU load's rough magnitude as opposed to an accurate prediction, this kind of error does not represent a concern.

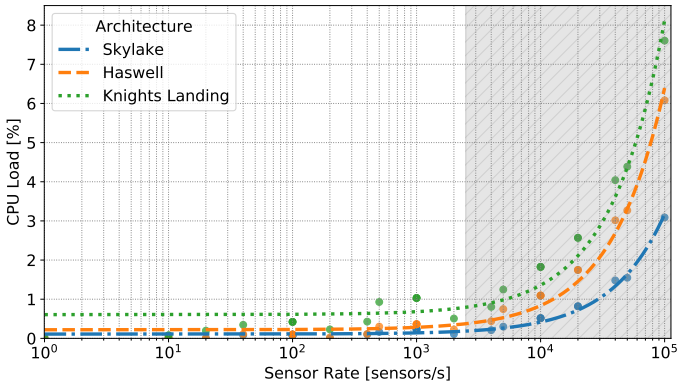


Figure 3.9.: CPU load scaling on three different architectures under different sensor rates. The white area highlights typical configurations for production HPC and data center environments [Net+19b].

3.5.4. Collect Agent Performance

In this section we analyze the performance and scalability of our Collect Agent component, to prove its effectiveness in supporting a large-scale monitoring infrastructure. In order to evaluate its scalability we focus on the CPU load metric, as defined in Section 3.5.2, and we use a dedicated node as described in Section 3.5.1. We do not analyze the performance of the Cassandra Storage Backend to which Collect Agents write, as it is a separate and widely documented component. Similar to Section 3.5.3, we perform tests by running Pushers with the Tester plugin under different configurations. Here, we used a sampling interval of 1s and experimented with different numbers of Pushers, executed from separate nodes, each sampling a certain number of sensors.

We show the results of our tests in Figure 3.10. In the configurations that use 1,000 sensors or less, we reach saturation of a single CPU core only with 50 concurrent hosts. In the most intensive configurations, multiple CPU cores are fully used, but even in the worst-case scenario we observe an average CPU load of 900%, which translates into 9 fully-loaded cores: this corresponds to a Cassandra insert rate of 500,000 sensor readings per second (10,000 sensors sampled each second by 50 concurrent Pushers), which is equivalent to that of a production configuration in a medium-scale system.

3.5.5. Query Engine Performance

While the previous sections validate DCDB's performance and scalability from a monitoring standpoint, we now evaluate the overhead of Wintermute itself, focusing on its query engine component. Similarly to Section 3.5.3, we run the HPL benchmark in single-node configurations on a CooLMUC-3 compute node while a Pusher is running - in this case, a Wintermute *Querytest* plugin is also active. This instantiates a set of operators in online mode, which simply perform a certain number of queries over the

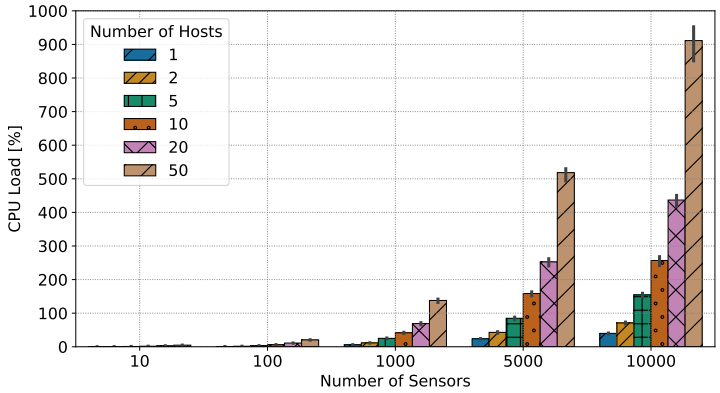
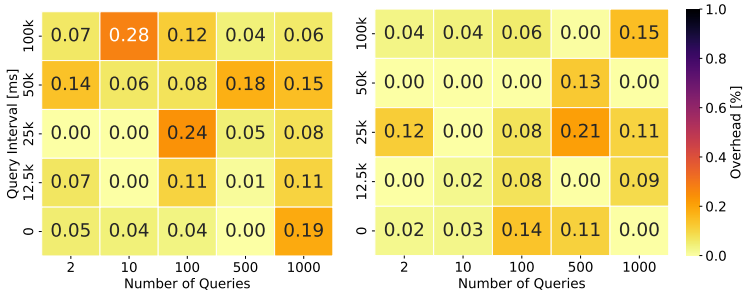


Figure 3.10.: Average per-core CPU load of the Collect Agent under different amounts of connected hosts and incoming sensors [Net+19b].

input sensors of their blocks. The input monitoring data is provided once again by the Tester plugin, producing a total of 1,000 monotonic sensors with negligible overhead and providing a reliable baseline for evaluating Wintermute. All plugins use a sampling interval of 1s and a cache size of 3 minutes.

Figure 3.11 presents the results of our performance evaluation. The two heatmaps depict overhead values when varying the number of queries performed at each analysis interval, as well as the temporal range of each query, using the query engine in absolute and relative mode respectively. Overhead is below 0.5% in all cases, with absolute mode performing slightly worse than relative and showing higher peak overhead values: this is expected, as absolute mode employs binary search and has a higher computational complexity. Further, no clear trend can be observed when increasing the amount of queried sensor data, showing that the query engine has good scalability and minimal impact on overhead. The heatmaps are considerably noisy, likely indicating that OS noise and application variability have a larger impact on observed overhead than Wintermute. Average per-core CPU load of the Pusher is mostly uniform and peaks at 1.2%. Likewise, memory usage never exceeds 25MB.

The resource footprint of Wintermute might be different when taking into account instantiated models and the characteristics of a production deployment. As a practical example, we measure the computational overhead of the Wintermute case studies carried out in Section 3.7 on CooLMUC-3. First, we consider the case study in Section 3.7.1, in which Wintermute implements a machine learning power consumption regression model at a fine granularity: the additional overhead of performing regression on top of standard monitoring is below 0.1% against HPL and thus negligible, showing once again the light resource footprint of Wintermute. Similar results were obtained for the case study in Section 3.7.2, in which Wintermute computes derived performance metrics (e.g., CPI) in-band. Here, we found the additional overhead of Wintermute to be always lower than 0.5% for both HPL and the considered CORAL-2 applications, executed on 32 nodes. In this case, the overhead increases mainly due to network interference



(a) Overhead in absolute mode. (b) Overhead in relative mode.

Figure 3.11.: Heatmaps of Wintermute's query engine's overhead at various time ranges and sensor amounts, against HPL on CooLMUC-3. With a query interval value of 0 only the most recent value of each sensor is retrieved [Net+20].

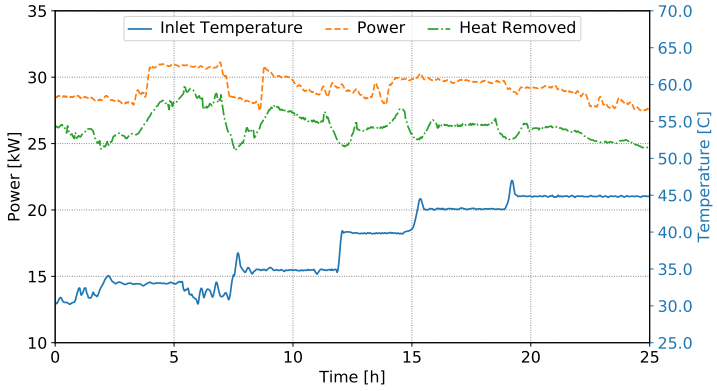
associated with the high number of sensors, as performance metrics are computed on a per-CPU core basis. On the other hand, the case study in Section 3.7.3 was executed out-of-band, and therefore overhead measurements are not applicable to it. We also omit quantitative comparisons with other tools: as discussed in Chapter 2, the insular and diverse nature of existing ODA solutions renders tool comparability difficult, which is hence useful only from a qualitative standpoint.

3.6. DCDB Case Studies

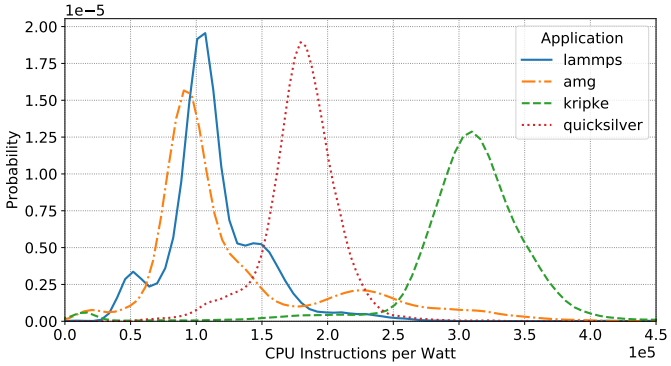
In the following we present two real-world monitoring case studies using DCDB, illustrating both facility and application-level collection of metrics on the CooLMUC-3 system (see Section 3.5.1). In the first case study, we monitor and correlate infrastructure data from different sources by analyzing the cooling system's ability to remove heat, whereas in the second we focus on showing the usefulness of high-frequency monitoring data collected in compute nodes to characterize the power consumption of applications.

3.6.1. Efficiency of Heat Removal

One of the requirements in the procurement of the CooLMUC-3 HPC system was to achieve high energy efficiency through the employment of direct warm-water cooling. Megware [26], the chosen system integrator, provided a 100% liquid-cooled solution that not only liquid-cools the compute nodes, but also all other components, including power supplies and network switches. Hence, the entire system does not require any fans in the compute racks and therefore can be thermally insulated. This reduces the heat emission to the compute room to close to zero. To help study its efficiency, the system is broadly instrumented and provides a wide range of infrastructure sensors and measuring devices, such as power sensors and flow meters. We monitor these sensors



(a) Efficiency of Heat Removal.



(b) Application Characterization.

Figure 3.12.: Plots depicting two distinct use cases of DCDB, each leveraging sensor data at a different level of the CooLMUC-3 HPC system [Net+19b].

and devices in DCDB to evaluate the efficiency of the system's water cooling solution by calculating the ratio between the heat removed via warm water and the system's total electrical power consumption.

Figure 3.12a depicts in detail the behavior of the monitored metrics for our case study, specifically the total power consumption of the system, the total heat removed from the system by the liquid-cooling circuit, and its inlet water temperature. All data has been collected out-of-band by running a dedicated Pusher on a management server and by leveraging the Pusher's REST and SNMP plugins, with a sampling interval of 10s. As it may be expected, the instrumentation employs sensors only at the node or rack levels, which do not supply a unified picture of the entire system's status; hence, we defined virtual sensors in DCDB (as described in Section 3.3.3) in order to aggregate this data and gain a more comprehensive view. Using DCDB, we were able to easily record all relevant sensors and to calculate the average ratio between the total heat removed and the power drawn, which turned out to be approximately 90%, showing very high efficiency of our system's water cooling solution. We further observe that,

for rising inlet water temperatures, the gap between power and heat removed does not increase, suggesting that the insulation of the entire racks is effective in reducing the emission of heat radiation to ambient air. Finally, it can be seen that the inlet water temperature is not stable when set to 30C and fluctuates consistently: this can be simply attributed to the fact that a warm-water cooling system is designed to run at higher inlet temperatures, starting from 35C.

3.6.2. Application Characterization

As we described in Section 1.2, monitoring data can be used to implement a feedback control loop inside of a system, so as to make informed and adaptive management decisions and thus realize an ODAC process. One such use case involves using monitoring data in compute nodes to characterize the relationship between the throughput and power consumption of running applications, which can subsequently be used by ODAC processes to change parameters such as the CPU frequency at runtime to improve overall energy efficiency. In this scenario, for example, when applying monitoring data to drive *Dynamic Voltage and Frequency Scaling* (DVFS) [HM07], both the frequency of sampling and of control needs to be high (i.e., greater than 1Hz) so as to react quickly to the frequent changes that occur in application behavior [Mit14].

Here, we present a characterization of the four applications from the CORAL-2 suite presented in Section 3.5.1, which are Kripke, AMG, LAMMPS and Quicksilver, from a pure monitoring perspective. We execute several runs of the applications on a single node in our CooLMUC-3 system, while using DCDB with a 100ms sampling interval. The application, node and DCDB configurations are as described in Section 3.5.1. In particular, we try to gain insight into the characteristics of each single application by analyzing the ratio between the number of per-core *retired instructions* and the node's power consumption at each time point. The former metrics are collected via the Perfevents plugin, while the latter with SysFS. In Figure 3.12b we show the fitted *Probability Density Function* (PDF) of the resulting time series for each application. We can see that each application shows a distinct behavior: Kripke and Quicksilver exhibit very high mean values, which can imply high computational efficiency or large I/O activity, while applications such as LAMMPS or AMG show lower values. Moreover, the distributions of the two latter applications show multiple trends, indicating a dynamic behavior that changes over time. Obviously, these variations vastly depend on the code profile of the underlying application. In this context, the use of a fine-grained monitoring tool, such as DCDB, contributes to distinguishing different application patterns and to supporting the implementation of effective ODAC models, leading to better system efficiency (e.g., by selecting optimal CPU frequencies).

3.7. Wintermute Case Studies

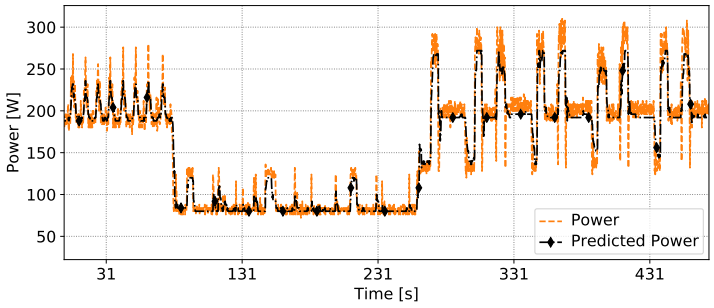
In this section we present several case studies showing the capabilities of Wintermute. These represent a mix of typical ODAV and ODAC techniques in the literature, so as to highlight its flexibility and effectiveness: without a framework such as Wintermute, carrying out these case studies would require the development of a substantial amount of dedicated, non-reusable code to retrieve sensor data, control the analysis and to expose its output. All experiments described in this section were carried out on the CooLMUC-3 system (see Section 3.5.1).

3.7.1. Power Consumption Prediction

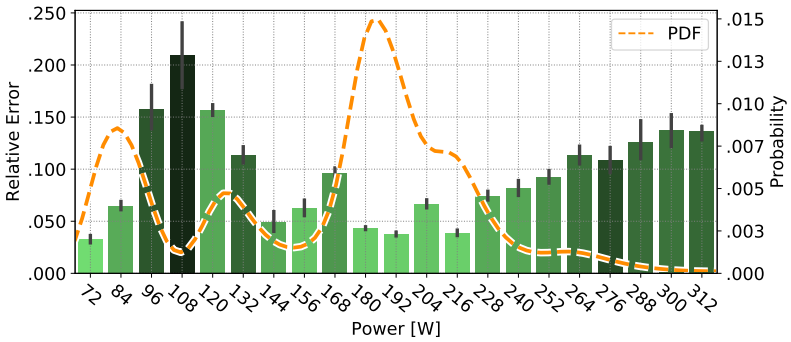
The first study shows the use of Wintermute for predicting the power consumption of a compute node (precisely, overall node power measured at the power supply) in CooLMUC-3, which can be used to steer online control decisions in the power and runtime systems. As introduced in Section 3.6.2, an example of this is DVFS CPU frequency tuning, which can be exploited in an automated way to save significant amounts of energy without sacrificing application performance. In this scenario data is collected in-band, at a fine time scale, and is immediately re-used for control purposes. The model is an online implementation of the one proposed by Ozer et al. [Oze+19] and represents a typical example of ODAC-oriented models.

Configuration

In a Pusher we instantiate a single operator from the Wintermute *Regressor* plugin, which implements a generic random forest-based online regression model. Its input data consists of a set of performance metrics and sensors, and both sampling and regression operate at a 250ms interval. The plugin, which is based on the *OpenCV* library [27], works in the following way: at each computation interval, for each input sensor of a certain block, a series of statistical features are computed from its recent readings - these are the 25th and 75th percentiles, mean, standard deviation, sum of changes and latest value. These features are then combined to form a feature vector, which is fed into the random forest model to perform regression and output a sensor prediction for the next 250ms. The training of the model, which is shared by all blocks of an operator, is performed automatically: feature vectors are accumulated in memory until a certain training set size is reached, alongside the responses from the sensor to be predicted. In this case, the responses come from the power sensor, with the model set to predict its value in the next 250ms. With the Pusher running, we execute the Kripke, AMG, Nekbone and LAMMPS proxy HPC applications from the CORAL-2 suite with as many threads as physical cores, while the regression operator builds its training set. Here the operator has only one block, corresponding to the compute node itself, and the training set size is set to 30,000. Once training of the model is complete, we evaluate the regression with new DCDB data.



(a) Time series of the real and predicted power.



(b) Relative error of the predicted power.

Figure 3.13.: Performance of our power consumption prediction model in terms of time-series behavior and relative error. Average relative error is 6.2% [Net+20].

Results

Figure 3.13 summarizes the results of the model. It shows a small excerpt from the time series of the real and predicted power sensors: we see that the time series of the predicted power consumption follows the measured time series closely, capturing status changes and periodic behaviors before they occur. However, the predicted time series fails to capture some short spikes or oscillations in power consumption, and presents itself like a *smoothed* version of the measured one. These events are difficult to predict, as they are usually related to the CPU's power management policy, which may exhibit short-term spikes for throughput improvement (e.g., *Turbo* mode on Intel CPUs) or may be related to electrical or sensor noise. The phenomena described above are even more apparent in Figure 3.13b, which shows the average relative prediction error for each measured power band, together with the fitted PDF of the latter. It can be seen that prediction is worse for higher power values; as it can be observed from the PDF, these values represent a minimal portion of the distribution, and have negligible impact on the overall error. Moreover, this imbalance in the distribution translates directly to an imbalance in the training set of the model, which does not have enough data to capture

this type of behavior. Similarly, some rare low-power states are not predicted well by the model. However, in the regions of the distribution where most of the data concentrates, error is always close to 5%, demonstrating the model's effectiveness.

We obtained comparable average relative error values when sampling and predicting power consumption at a time interval of 125ms (10.4%) and 500ms (6.7%). In the work by Ozer et al. [Oze+19], the offline validation of this approach shows comparable results to the ones presented here, proving its generality. While specialized techniques such as *PRACTISE* [Xue+15] are able to yield more accurate prediction (i.e., with a relative error below 1%), this example shows that very good results can be obtained with general-purpose plugins, and with little effort. Moreover, a significant improvement of the results can be obtained by training and validating a model offline using historical, balanced data, which can then be deployed online using Wintermute.

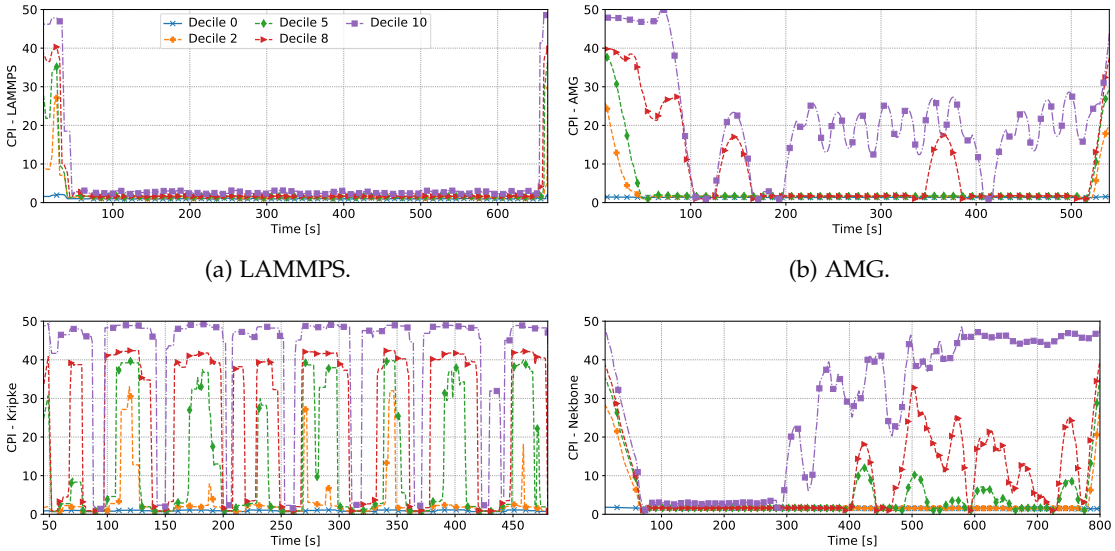
3.7.2. Analysis of Job Behavior

In the second case study, we use Wintermute to produce aggregated performance metrics on a per-job basis. We combine two different plugins, showing how pipelines can be used in Wintermute to perform complex analyses and split computational load. The plugins discussed here represent a re-implementation of the Persyst framework [GHB14]: their purpose is to enable online visualization of job performance data for HPC users, allowing them to quickly adapt configurations and spot issues. This case study thus belongs to the ODAV category. Because of its online and user-oriented nature, this approach differs from others like the *roofline model* [WWP09], which are more suited for offline analysis.

Configuration

We deploy two distinct operator plugins, implementing a pipeline as described in Section 3.3.5. The first *Perfmetrics* plugin, instantiated in the Pushers, takes as input CPU and node-level data and computes as output a series of derived performance metrics, such as the CPI, *Floating Point Operations Per Second* (FLOPS) or *vectorization ratio*, which are useful to evaluate application performance. A second *Persyst SQL* job operator plugin is instantiated in the main Collect Agent: at each computing interval, it queries the set of running jobs on the HPC system, and for each of them it instantiates a block according to its configuration. In this case, blocks have as input one of the Perfmetrics derived metrics from all compute nodes on which the job is running. From these, the operator computes a series of job-level statistical indicators as output; these are the deciles of the metric's distribution, as well as its average value. In the Pushers and Collect Agent, sampling and computation are performed at 1s intervals. The Persyst plugin supports multiple databases for storage of data, but in this context we leverage the default Cassandra Storage Backend.

We executed four jobs, each on 32 CooLMUC-3 nodes and running the Kripke, AMG, Nekbone and LAMMPS applications. The job runs were repeated multiple times



(a) LAMMPS.

(b) AMG.

(c) Kripke.

(d) Nekbone.

Figure 3.14.: Deciles of the aggregated per-core CPI values in time, for 4 jobs running different CORAL-2 proxy applications on CooLMUC-3 [Net+20].

and under different node configurations to ensure consistency. Here we focus on the CPI metric: thus, we configured the Perfmetrics plugin to have an operator with one block per CPU core, each producing as output its CPI value. Then, we use a Persyst operator which outputs the deciles of the job-level CPI distribution at each time point, as computed by aggregating the corresponding input values for each job. Since the latter are computed per-core, each decile is aggregated from 2,048 samples at a time. This allows us to gain an overall understanding of the behavior of the applications running on the HPC system, whereas the full extent of available metrics allows to characterize their performance profile and bottlenecks.

Results

Figure 3.14 shows the results of our analysis: for each job, we show the time series for deciles 0, 2, 5, 8 and 10 of the aggregated per-core CPI values while running the corresponding CORAL-2 codes; deciles 0, 5 and 10 correspond to the minimum, median and maximum CPI values respectively. It can be seen that the applications exhibit distinctly different behaviors depending on the underlying computational workload, supporting what was observed in Section 3.6: LAMMPS shows low CPI values averaging at 1.6, with minimum spread in the distribution, which is due to its mostly compute-bound nature. A similar behavior can be observed with AMG, with low CPI values up to decile 5: however, deciles 8 and 10 show spikes up to CPI values of 30. As AMG is a network and memory-bound application based on fine-grained synchronization, this

behavior could be caused by network latency affecting I/O, as well as load imbalance.

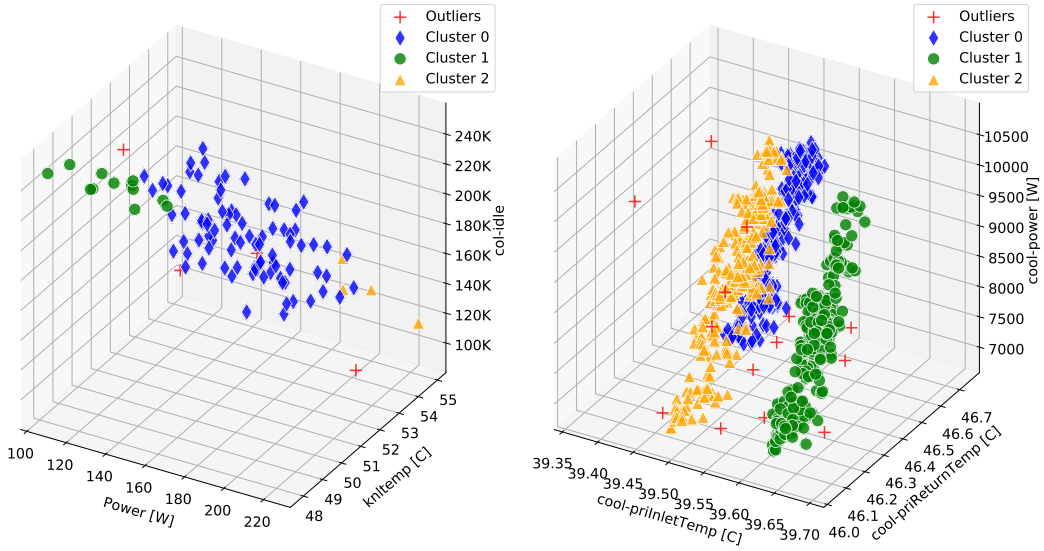
Kripke has a very distinctive profile: it is possible to separate each single iteration thanks to the increase and decrease in CPI values across all deciles. Similarly to AMG, Kripke is also a network and memory-bound application, and is thus characterized by relatively high CPI values. Finally, Nekbone shows the most interesting behavior: low CPI values can be observed in the first part of the application run, which is expected as the conjugate gradient solver implemented in Nekbone is compute-bound. In the second part of the run, however, the spread across deciles increases dramatically, with at least 20% of the CPUs exhibiting higher CPI values. Our hypothesis is that, as Nekbone performs a batch of tests on increasing problem sizes, the application becomes memory-limited as soon as it grows past the 16GB of HBM available in CooLMUC-3 nodes. This shows how visualization of performance metrics can be used to spot bottlenecks in HPC applications.

3.7.3. Identification of Performance Anomalies

For the final case study, we conduct a long-term analysis on coarse-grained monitoring data from all compute nodes in CooLMUC-3. By applying unsupervised learning techniques, we characterize the performance of the entire HPC system and highlight variance between nodes, as well as identify outliers and anomalies: unlike the previous case studies, this can be both used to automatically raise alerts to system administrators (ODAV) or to improve resource allocation decisions (ODAC).

Configuration

We use a single *Clustering* plugin, that implements *Bayesian Gaussian Mixture Models* (BGMM) [Rob+98], in the main Collect Agent. This plugin is configured to have one operator with as many blocks as compute nodes, each having as input a node's power, temperature and CPU idle time sensors, and as output a label of the cluster to which it belongs. More precisely, at every computation interval the operator computes two-week averages for the input sensors of each block. Then, each block is treated as a data point in a three-dimensional space, and clustering is applied to them. Sampling in Pushers is performed every 10s and clustering every hour. We adopt BGMMs because, unlike ordinary *Gaussian Mixture Models* (GMM), they are able to determine the optimal number of clusters from data. This is useful in an online, continuous scenario, where the diverse states of an HPC system can be captured without manual tuning of the model's parameters. The number of input sensors to the clustering algorithm (and thus the number of dimensions) can be changed at will in the plugin's configuration, as well as the length of the averages' aggregation window. Since the job run time limit is set to two days on CooLMUC-3, we choose a value of two weeks to extract the performance profile of each node without knowledge of running jobs.



(a) Compute nodes.

(b) Racks.

Figure 3.15.: Results of clustering applied at different levels of CooLMUC-3: each point is either a compute node (a) or a rack (b) at a certain point in time [Oze+20].

Results - Compute Node Analysis

Figure 3.15a shows the result of the clustering process for a set of points belonging to the same time window. The points in the scatter plot correspond to compute nodes in CooLMUC-3, whose coordinates are their two-week average power (*Power*), temperature (*knltemp*) and CPU idle time (*col-idle*). First, it can be observed that the three metrics are strongly correlated and that they describe a clear linear trend: this is expected, as a compute node will consume less power if idling, and its temperature will be lower as well. Most nodes concentrate in Cluster 1 towards the center of the plot with relatively little spread, especially on the temperature and CPU idle time axes.

Despite the two-week aggregation window we adopted, some stark differences in node behavior can still be observed: compute nodes belonging to Cluster 0 have a higher CPU idle time, showing low power and temperature values accordingly. Conversely, nodes in Cluster 2 were under heavier load compared to other nodes, peaking at 200W of average power consumption for a single node. While this behavior could simply be due to specific sequences of applications running on the nodes, it is more likely the symptom of a job scheduling policy that does not account for workload balance between nodes. A few points were classified as outliers when their probability was lower than a certain threshold (0.001 in our case) in the PDFs of all fitted Gaussian components, and the behavior of the corresponding nodes diverges significantly from the rest of the system. One node in particular shows a concerning trend, consuming roughly 20% more power than other nodes with similar CPU idle time.

Results - Rack Analysis

We conducted a variant of the experiment presented above in an offline context, this time considering per-rack data. Since the offline implementation of our model is functionally identical to the Wintermute one, the results that we present here can be reproduced in the latter. For this analysis, we use sensors associated to the warm-water cooling unit for each of the 3 racks in CooLMUC-3, each containing in turn roughly 50 nodes. The sampling interval of the sensors is 10s and we use two weeks of data, from June 28th to July 11th 2019. The results of the experiment are depicted in Figure 3.15b: here each point represents a rack in CooLMUC-3, with its hourly averages of the water's primary inlet temperature (*cool-priInletTemp*), its primary return temperature (*cool-priReturnTemp*) and finally the amount of heat removed (*cool-power*) in Watts. Each point thus represents the status of a rack at a specific point in time during the two weeks being analyzed. Additional details about the specific meaning of these metrics, the difference between primary and secondary temperatures, and the operation of the cooling units can be found in Section 5.3.2.

Unlike in our compute node-level analysis, only the return temperature and the heat removed metrics appear to be strongly correlated: this is expected, as the greater the temperature difference between the primary inlet and return water is, the greater is the amount of heat removed from the system. As the primary inlet temperature is enforced externally at the building infrastructure level, this metric does not show any correlation with the others and shows little variance across racks. Furthermore, each rack in the system is clearly separated and modeled by a distinct Gaussian distribution: our approach thus supplies a statistical description of each rack's cooling performance, which simplifies system maintenance as well as anomaly detection. It can also be seen that Cluster 0 (i.e., Rack 2) shows a consistently higher return temperature, and that some outliers are present. Three points in particular, which show deviation with respect to the inlet water temperature, come from the same time frame. This hints at the presence of an anomaly in the cooling system at that time, likely caused by environmental factors. As shown, this type of analysis is very effective at supplying a comprehensive view of an HPC system's behavior, and can be useful to system administrators and researchers alike. Similarly, this can also be used to improve scheduling policies by considering recent node behavior.

3.8. DCDB and Wintermute as Production Tools

In this chapter we discussed the DCDB and Wintermute frameworks for HPC monitoring and ODA, demonstrating their suitability for production environments with an extensive set of performance experiments. Based on this foundation, in Chapter 5 we will discuss their long-term production use, implementing a variety of ODA use cases. HPC systems, with their rigid operational and performance requirements, represent a small domain compared to cloud data centers: in the latter, systems normally operate in a loosely-coupled and malleable manner, with applications that are not as compute-intensive and

performance-dependent as HPC ones [Net+18b]. For these reasons, we expect DCDB and Wintermute to be suitable for cloud environments as well.

Own publication acknowledgement. The architectures of DCDB and Wintermute, as well as the associated experiments, were originally discussed in [Net+19b] (for DCDB) and in [Net+20] (for Wintermute). The clustering case study in Section 3.7.3 was extracted from a broader discussion in [Oze+20]. Lastly, the power consumption prediction case study in Section 3.7.1 is based on conceptual prior work in [Oze+19].

4. The Correlation-wise Smoothing Method

In this chapter we present the *Correlation-wise Smoothing* (CS) method to process and extract actionable knowledge from monitoring data. In Section 4.1 we provide a formal problem statement and describe the baseline methods that will be used for comparison. In Section 4.2 we then describe the CS algorithm, while in Section 4.3 we present HPC-ODA, a collection of HPC monitoring datasets that we released publicly. In Section 4.4 we discuss the experimental results obtained when applying the CS method to HPC-ODA in a series of use cases. We conclude the chapter in Section 4.5.

4.1. Problem Statement

As introduced in Section 1.2.2, converting raw monitoring data to a compact, processed representation is a step required by most ODA models, in particular those that belong to the ODAC category. We refer to this process as *monitoring data processing* and, in the context of our dissertation, we limit ourselves to sensor monitoring data which can be represented as multi-dimensional time-series data - therefore, we do not consider unstructured data sources such as log streams. Furthermore, we refer to the processed monitoring data as a *signature*, since it should supply a compact representation that is descriptive of a system's behavior. However, as explained in Section 2.3.2, this process is not trivial: a production HPC or data center environment is a mine field of constraints that severely limit the complexity of the algorithms that can be used, implying that most common data mining techniques (e.g., PCA or clustering) are not applicable to the ODA domain. For this reason, we investigate a novel solution that is able to extract meaningful knowledge from monitoring data, while being as lightweight and generic as possible. We now provide a series of formal definitions and present three baselines methods in the ODA domain.

4.1.1. Formal Definitions

The problem at hand consists of computing signatures from multi-dimensional time-series data originating from a sensor monitoring system, which can be used as feature sets for machine learning and data mining models (ODAC) or simply be visualized (ODAV). This data can be represented as a *sensor matrix*, in which each row corresponds to a sensor (e.g., power or temperature) in the system and each column to a time-stamp - each matrix element is thus a numerical reading for a sensor at a specific time-stamp. The sensor matrix S has n rows, as many as sensors, and t columns, as many as individual time-stamps, with indexes starting from 1, and can be visualized as a *heatmap* as shown

in Figure 4.1. For the sake of simplicity, we assume that the sensors in S are time-aligned and have the same sampling rate: this is not necessarily true for real datasets, and an interpolation step may be required to align the data.

Seen formally, a *signature method* is a function $Sig()$ that takes as input a sub-matrix S^w of the sensor matrix S and that gives as output a column vector s^w. The sub-matrix S^w has n rows and w^l columns: this last parameter is the time *aggregation window* in samples, and in order for the signature method to be useful, the relation $l << n \cdot w^l$ must be satisfied, where l is the final length of the signature. Depending on the method, l can either be an arbitrary parameter or a function of n and w^l. We also define w^s, which is the *step* between successive aggregation windows. In other words, the signature method $Sig()$ performs aggregation of the sensor data in S^w, returning a compressed representation that is more compact than the data it was computed from. This can be interpreted as a set of coefficients describing the status of a software or hardware component at a given point in time.

4.1.2. Baseline Methods

We have identified three baseline signature methods for sensor monitoring data in the literature. These have been applied to many ODA use cases and are suitable for use in production data centers. Any modifications applied to the original algorithms are pointed out explicitly, and were done solely to improve performance with the high-dimensionality datasets at our disposal. While we could select more sophisticated data mining algorithms as baselines, this would be unrealistic as they would never be usable, due to factors such as overhead and latency. The methods are as follows:

- **Tuncer** [Tun+18]: for each sensor row in S^w, a series of statistical indicators is computed from its w^l samples. These are the mean, standard deviation, minimum, maximum, 5th, 25th, 50th, 75th and 95th percentiles, sum of changes and absolute sum of changes. We modified the original method to use the sum of changes and absolute sum of changes instead of the skewness and kurtosis, as the former led to better performance. The signature's size is $l = n \cdot 11$.

- **Bodik** [Bod+10]: similarly to the Tuncer method, a series of statistical indicators is computed for each row in S^w. In this case only the minimum, maximum and 5th, 25th, 35th, 50th, 65th, 75th and 95th percentiles are used, in order to characterize the sensor's value distribution. The signature's size is $l = n \cdot 9$.

- **Lan** [LZL09]: each sensor row in S^w is sub-sampled to a fixed size w^r (smaller than w^l) using a mean filter and is then concatenated to the final signature. The original method simply *flattened* the sub-matrix S^w to the vector s^w and employed an additional PCA step for outlier detection. We omit the latter and introduce a sub-sampling step for performance reasons. The signature's size is $l = n \cdot w^r$.

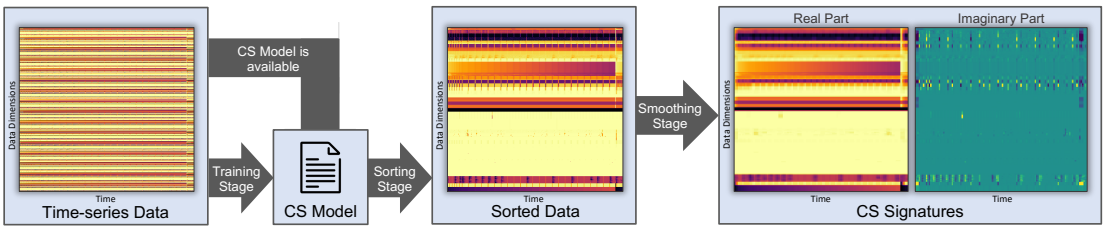


Figure 4.1.: The three stages of the CS algorithm. On the left we show a heatmap of the raw sensor data from 16 compute nodes during a run of AMG, from the HPC-ODA Application segment. In the center, the same data is depicted after applying the sorting stage of the CS algorithm. On the right side we show the final signature heatmaps using 160 blocks. Darker colors represent higher values and each column is a separate signature [Net+21b].

4.2. The CS Algorithm

The methods described in Section 4.1.2 do not exploit the relationships between sensors to improve the signatures' quality, which is especially relevant in the case of sensor monitoring data: HPC systems, in particular, are traditionally used to execute large-scale parallel, homogeneous codes, leading to strong correlations between the sensors in the associated compute nodes. Even without this assumption, sensors in system components tend to be highly correlated due to their performance relationships, and sometimes the vast amount of available metrics simply supplies redundant information that can be pruned [Gim+17]. Under these assumptions, we propose the Correlation-wise Smoothing (CS) method: it is able to perform aggregation *across* sensors, in turn allowing users to enforce arbitrary signature sizes [Net+21b]. The CS method is designed for lightweight online operation with up to thousands of dimensions and its algorithm consists of three stages, which we describe in the following. Figure 4.1 shows an overview of the algorithm's flow, starting from the raw monitoring data (left) and finishing with the corresponding CS signatures (right).

4.2.1. Training Stage

In this stage the CS algorithm learns the relationships between data dimensions, using a sensor matrix S that contains historical data characterizing the behavior of the system component that is the object of the analysis (e.g., a compute node or CPU core). Our intuition is that the rows in S can be *permutated*, such that sensors which are correlated to one another will be grouped together, improving both visualizability and compressibility. To this end we use the following metrics:

$$\rho_{S_i,S_j} = \frac{cov(S_i, S_j)}{\sigma_{S_i}\sigma_{S_j}} + 1 \qquad \rho_{S_i} = \frac{1}{n-1}\sum_{j \neq i} \rho_{S_i,S_j} \tag{4.1}$$

In Equation 4.1, ρ_{S_i,S_j} is the *Pearson correlation coefficient* between rows i and j of S,

Listing 2 Correlation-wise Smoothing - Training Stage.

Input: the coefficients ρ_{S_i,S_j} and ρ_{S_i} for $0 < i, j \leq n$
Output: a permutation vector p
1: **vector of int** $p \leftarrow \varnothing$
2: **set of int** $d \leftarrow (1, 2, ..., n)$
3: next $\leftarrow argmax(\rho_{S_k}$ for $k \in d)$
4: d.remove(next)
5: p.append(next)
6: **while** d is not empty **do**
7: next $\leftarrow argmax(\rho_{S_k,S_{next}} \cdot \rho_{S_k}$ for $k \in d)$
8: d.remove(next)
9: p.append(next)
10: **end while**
11: **return** p

which is computed as the ratio between their covariance $cov(S_i, S_j)$ and the product of the respective standard deviations σ_{S_i} and σ_{S_j}. Coefficients are shifted by 1 to ensure that they are non-negative and in the interval $[0, 2]$. Then, the *global* correlation coefficient ρ_{S_i} is defined as the sum of all correlation coefficients with respect to row i: this metric quantifies the relevance of a certain dimension in the entire dataset, and how well it can describe the system's status. Using these two metrics we compute a row permutation vector p as described in Listing 2: starting from the row with the maximal ρ_{S_i}, rows are selected iteratively as those maximizing the product between ρ_{S_i} and the correlation coefficient with the latest row added to p.

This type of ordering is generic and interpretable: the sensors at the beginning of p are those that meaningfully describe the system's status and that have an overall positive correlation with other sensors. Sensors at the middle of p have little correlation with other sensors and are akin to noise. Finally, sensors at the end of p are again descriptive of the system's status, but are negatively correlated with those at the beginning. On top of the permutation vector p, the training stage also computes the lower and upper bounds of each row to enable min-max normalization of the data: these data structures compose a *CS model*, which can be stored and re-used for the subsequent stages of the algorithm. The computational complexity of this stage is dominated by the correlation matrix computation and amounts to $O(n^2 t)$.

4.2.2. Sorting Stage

The sorting and smoothing stages are performed any time a signature must be computed from a sub-matrix S^w. In the sorting stage, min-max normalization is applied to the sensor data, ensuring that all sensor values fall in the interval $[0, 1]$. Then, the permutation vector p is used to sort the rows of S^w, as depicted in Figure 4.1: simply re-arranging the ~800 rows in the heatmap (i.e., ~50 sensors for 16 compute nodes) brings clear visual patterns to the surface, vastly improving the quality of visualization. This is due to the fact that highly-correlated sensors follow the same trends over time,

which are highlighted by grouping them together. Moreover, the processed heatmaps can be treated as images, opening up the possibility to leverage a whole branch of image processing techniques for data center monitoring. By default the training stage of the CS algorithm is performed only once, reusing the CS model for all new sensor data, thus greatly reducing its footprint. However, the algorithm can be configured to repeat the training stage whenever required: this can be useful when dealing with data from multiple components, whose relationships and correlations change over time depending on how they are used. For example, this applies to the compute nodes of a large-scale HPC system, which may be allocated to different user jobs over time, altering their relationships. The computational complexity of this stage is dominated by the min-max normalization, amounting to $O(w^l n)$.

4.2.3. Smoothing Stage

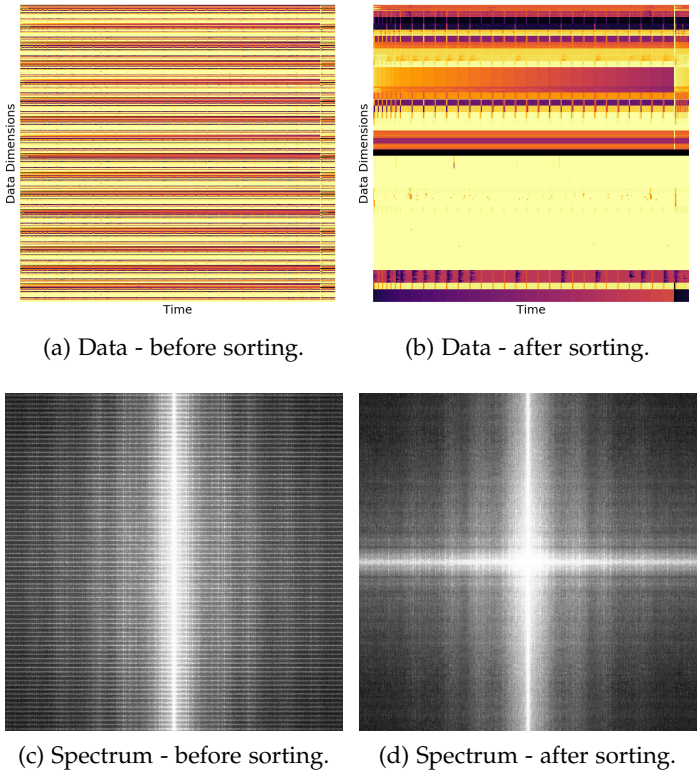
The last stage performs a *smoothing* operation over the sorted S^w sub-matrix and creates the final signature, which is composed of an arbitrary number l of complex-valued elements (or *blocks*), each aggregating a partially overlapping range of sensors:

$$b_i = 1 + \lfloor (i-1) \cdot n/l \rfloor \qquad e_i = \lceil i \cdot n/l \rceil \tag{4.2}$$

In Equation 4.2, b_i and e_i express the indexes of the first and last sensors aggregated in each block i in the range $[1, l]$. This blocking scheme has two properties: first, each block refers to a specific set of sensors, with their ordering being an importance indicator. Second, when the modulo $n\%l$ is not zero, a corresponding amount of blocks will be extended by 1 sensor; these larger blocks are distributed uniformly over the signature space due to the periodicity of the modulo operation. Finally, as the set of raw sensors belonging to a block is clearly defined, root cause analysis is simplified. The values of each block are then computed as follows:

$$Re(s_i^w) = \frac{1}{w^l(e_i - b_i + 1)} \sum_{j=b_i}^{e_i} \sum_{k=1}^{w^l} S_{j,k}^w \qquad Im(s_i^w) = \frac{1}{w^l(e_i - b_i + 1)} \sum_{j=b_i}^{e_i} \sum_{k=1}^{w^l} S_{j,k}^{w'} \tag{4.3}$$

In Equation 4.3, $S^{w'}$ is the matrix of row-wise first-order derivatives computed from S using backward finite differences and corresponding to the time window of S^w. The real part of each block s_i^w thus contains the average value of the sensors in block i, in the time range described by S^w; the imaginary part, instead, contains the corresponding average first-order derivatives. The right side of Figure 4.1 shows two heatmaps containing the real and imaginary parts of the signatures computed from the data on the flowchart's left side: each column represents a separate signature whose real part describes the *static* properties of the system, while the imaginary part describes its *dynamic* properties. Moreover, the signatures can be scaled at will using traditional image processing algorithms, ensuring comparability as well as portability, while still allowing users to choose the resolution (defined by the l, w^l and w^s parameters) that they deem



(a) Data - before sorting. (b) Data - after sorting.

(c) Spectrum - before sorting. (d) Spectrum - after sorting.

Figure 4.2.: Effects of the CS method's sorting stage in the frequency domain.

appropriate. Further, more aggressive compression is possible: as the central signature coefficients represent the least insightful sensors in the system, they can be potentially eliminated with minimal loss of information. Due to its normalization step, the CS algorithm cannot be used with monotonic time series (e.g., energy), but this was not found to be an issue for sensor monitoring data, since it can also be transformed using backward finite differences. The complexity of the smoothing stage amounts to $O(w^l n)$.

4.2.4. Frequency Domain Interpretation

From an intuitive standpoint, the CS method's effectiveness derives from the fact that, by laying each time series on a two-dimensional space and sorting them according to their correlations, we obtain a "continuous" picture that is easy to visualize and is fit for a smoothing process. A more formal explanation can be found by analyzing the effects of the CS method's sorting stage in the frequency domain: in Figure 4.2 we show once again the monitoring data heatmaps introduced in Figure 4.1, referring to a run of the AMG benchmark on 16 SuperMUC-NG nodes; the heatmap in Figure 4.2a shows the raw, unsorted data, while the one in Figure 4.2b shows the data after the CS method's sorting stage, with no smoothing applied. Alongside the heatmaps, we also show the respective

centered spectra computed via a two-dimensional *Fast Fourier Transform* (FFT) [All77]. It can be seen that the spectrum of the data before sorting (Figure 4.2c) is noisy and spread over the entirety of the frequency space. Moreover, some frequency *aliasing* can be observed, likely due to the fact that the 16 compute nodes from which the data is collected behave similarly, thus injecting a form of periodicity on the sensor (i.e., vertical) axis. On the other hand, the spectrum of the data after the CS method's sorting stage (Figure 4.2d) is much cleaner, does not show any aliasing, and most of the energy is concentrated towards the center.

Our intuition is the following: the CS method's sorting stage does not alter the total *spectral power* (i.e., the sum of amplitude values in the spectrum) of the raw monitoring data, and hence also not its information content, but instead it re-distributes it over different parts of the frequency space, in particular lower-frequency ones. This stems from the fact that, statistically, grouping together sensors that are highly correlated to one another will produce regions with low-frequency content in the resulting heatmap. This is highly beneficial for smoothing and compression, since these two processes can be interpreted as applying a low-pass *Finite Impulse Response* (FIR) filter [LP83], which only captures frequency content up to its cut-off frequency: applying such a filter to data treated with the CS method's sorting stage will result in a higher amount of the total spectral power being captured, and thus better fidelity to the original data.

4.3. The HPC-ODA Dataset Collection

Sensor data and logs from production data center and HPC environments are typically considered sensitive both due to privacy and operational concerns, making publicly available datasets rare in this area. To support the work in this dissertation we therefore introduce HPC-ODA, a comprehensive collection of datasets captured on several HPC installations, and make it openly available to the research community to aid ODA research. HPC-ODA is representative of HPC centers and their typical workloads, and contains a large number of sensors covering the operations of several systems.

4.3.1. Overview of the Dataset Collection

HPC-ODA is composed of five *segments*, which are self-contained datasets, each acquired separately on a particular HPC system and containing monitoring sensor data associated with different components (e.g., CPU cores, compute nodes or cooling systems) at a well-defined time granularity. The structure of the datasets is simple and summarized in Table 4.1: each sensor's time series is stored in a separate CSV file, with each entry being a time-stamp/value pair. Each of the segments has an associated ODA use case, usually of the ODAC type, which is representative of the state-of-the-art of data center operations as discussed in Section 2.3.3. The monitoring frameworks used to acquire data are DCDB [Net+19b] and LDMS [Age+14]. However, the data captured in HPC-ODA is generic and does not depend on the framework being used.

Table 4.1.: An overview of the features for each segment in the HPC-ODA dataset collection, together with the parameters of the associated ODA use cases.

Segment	HPC System	Nodes	Sensors	Data Points	Length	Sampling Interval	Feature Sets	w^l	w^s
Fault	ETH Testbed	1	128	∼1,000k	16d	1s	∼130k	60s	10s
Application	SuperMUC-NG	16	52	∼1,000k	1d	1s	∼250k	30s	5s
Power	CooLMUC-3	1	47	∼300k	8h	100ms	∼60k	1s	500ms
Infrastructure	CooLMUC-3	148	31	∼450k	16d	10s	∼75k	300s	60s
Cross-arch	Multiple	3	(52,46,39)	∼300k	1d	1s	∼140k	30s	2s

The rationale behind the HPC-ODA segment structure is to provide several vertical slices of the monitoring data typically available in a large-scale data center environment, with their different granularities and time scales. While having a production dataset from an entire HPC system - from the infrastructure down to the CPU level - at a fine time scale would be ideal, this is often not feasible due to the privacy concerns of the data (e.g., user names or allocation times), as well as the sheer amount of storage space required. To avoid such problems, HPC-ODA provides a compact and sanitized monitoring snapshot that can be used as a reference to evaluate ODA approaches. Moreover, we use HPC applications that are representative of production workloads, such as those from the CORAL-2 [24] suite introduced in Section 3.5.2: more details about application configurations and the data acquisition process can be found in the documentation of HPC-ODA, which is openly available [28] for use by the research community, as well as in Appendix C. Since its original release, HPC-ODA has been updated to include an additional segment, which contains monitoring data from the compute nodes of the *DEEP-EST* HPC system under different thermal conditions - this is discussed in Section 5.3.1. We now describe the five segments composing HPC-ODA, as well as the ODA use cases that we selected for them.

4.3.2. Fault Segment

This segment was collected from a single compute node in a testbed HPC system at ETH Zurich, while it was subjected to fault injection. It contains compute node-level data, as well as the labels for six single-node applications and eight injected faults: each fault has two possible settings and reproduces various software or hardware issues (e.g., CPU cache contention or memory allocation errors). This segment was extracted from a larger public dataset [Net+19a] and was not originally acquired in the context of this dissertation. Details regarding the applications being used and the injected faults can be found in the documentation of the original dataset [29].

The LDMS framework was used to acquire the data: specifically, we used the *Procstat*, *Meminfo* and *Vmstat* plugins to collect generic information about memory and CPU usage from the *proc* file system, together with the *Procinterrupts* plugin to collect interrupt-related information, *Procdiskstats* to collect hard drive-related metrics, *Procsensors* to collect various temperature measures and finally Perfevents for CPU performance counters. We use this segment to perform fault *classification*, that is, we attempt to detect

each of the eight possible faults running on the system, together with healthy operation. This can be used to optimize management decisions upon anomalous states in HPC components [Tun+18].

4.3.3. Application Segment

This segment was collected from 16 compute nodes in the SuperMUC-NG [6] HPC system at LRZ, while running one of six multi-node MPI applications, each under three different input configurations. The applications that we use are Kripke, AMG, PENNANT, LAMMPS and Quicksilver from the CORAL-2 suite [24], plus the MPI version of the HPL benchmark. These supply a diverse set of codes that cover the spectrum of most HPC production applications.

Data is collected with DCDB, it is at the level of each compute node and is paired with the labels of running applications. DCDB is configured to use the ProcFS plugin to collect memory and CPU usage data, SysFS to collect various power and temperature sensors, Perfevents to collect CPU performance counters and finally OPA to collect network-related metrics. This segment is used to perform application *classification*, where we try to distinguish the six possible applications running on each compute node, as well as idle operation. This allows to improve performance for running applications as well as spot rogue codes and purge them from the system [Ate+18].

4.3.4. Power Segment

This segment was collected from a single compute node in the CooLMUC-3 [21] HPC system at LRZ, while running several single-node OpenMP applications each under two different input configurations. These are Kripke, AMG, Nekbone, LAMMPS and HPL, providing diverse power consumption profiles due to the communication or compute-bound nature of the applications.

Data is collected once again using DCDB; in this case, the segment contains both node-level and CPU core-level data at a very fine time scale. Similarly to the Application segment, the ProcFS, SysFS, Perfevents and OPA plugins were employed. We use this segment to perform *regression* of power consumption, measured at the outlet: in particular, we try to predict the average power consumption in the next three samples, corresponding to $\sim$300ms. This knowledge enables system tuning (e.g., via changes in CPU frequency) to optimize performance based on predicted workloads [Oze+19].

4.3.5. Infrastructure Segment

This segment was collected from the infrastructure of the CooLMUC-3 [21] HPC system at LRZ and contains data from its power distribution and warm-water cooling systems. Due to privacy concerns, we have no knowledge of the specific applications executed by users on the system - instead, we only have a global overview of the components in CooLMUC-3 and their usage at a coarse granularity.

The data, which was collected using DCDB on a management server, is at the rack level, with some sensors being at the chassis level. We used the SNMP plugin to collect metrics from the warm-water cooling units, and the REST plugin to collect metrics from the power distribution infrastructure, similarly to the case study in Section 3.6.1. The segment is used to perform *regression* of the amount of heat removed by the cooling system: specifically, we predict the average heat removed from each rack over the next 30 samples, or ∼300s. This allows to optimize infrastructure-level knobs (e.g., inlet cooling water temperature) in function of environmental or workload changes [Con+15].

4.3.6. Cross-architecture Segment

This is a variant of the Application segment, containing data associated with the same six applications each under three input configurations. In this case, however, we run single-node OpenMP configurations of the applications on three compute nodes separately, each with a different hardware architecture. The compute nodes belong to the SuperMUC-NG and CooLMUC-3 systems, as well as to a testbed system at LRZ using Intel Skylake, Knights Landing and AMD Rome CPUs respectively.

We use once again DCDB to collect sensor data, which is at the compute node level and paired with the application labels. In this segment, the monitoring configurations are different for each architecture: the SuperMUC-NG and CooLMUC-3 nodes both use the ProcFS, SysFS, Perfevents and OPA plugins with 52 and 46 sensors respectively, while the testbed node only uses the ProcFS and Perfevents plugins with 39 sensors. The segment is used to perform application *classification* across different architectures.

4.4. Experimental Evaluation

In this section we present our experimental evaluation of the CS method applied to the HPC-ODA dataset collection, analyzing its performance from several standpoints and in comparison with the three baseline methods we have chosen. Each of the experiments targets some of the requirements stated in Section 2.3.2, in order to demonstrate the suitability of the CS method for production ODA use cases.

4.4.1. Experimental Methodology

We implement all of the ODA use cases introduced in Section 4.3 as classification and regression machine learning tasks. For each selected signature method, we process each segment in HPC-ODA using specific w^l and w^s values as specified in Table 4.1. The order of the feature sets (i.e., the signatures) in each resulting dataset is then shuffled, and we apply 5-fold cross-validation to them using a stratified K-fold strategy: the input to a model is always a single signature vector, and never a signature heatmap. As machine learning model we employ a random forest (with 50 estimators and using the Gini impurity to evaluate the quality of splits), due to its proven effectiveness in ODA use cases [Tun+18; Ate+18]. In a subset of our tests, we also employ a multi-layer

perceptron (with 2 hidden layers each having 100 neurons and using the rectified linear unit as activation function). The models are trained as classifiers or regressors depending on the specific problem at hand, and we use two metrics to evaluate machine learning performance: for classification problems we use the *F1-score*, which is computed as the harmonic mean between the *precision* and *recall* metrics. For regression problems we instead use the *Normalized Root Mean Square Error* (NRMSE): in order to show both metrics in a comparable *higher-is-better* fashion, referred to as *ML score*, we use the formula $NRMSE_c = 1 - NRMSE$ to complement the NRMSE. Experiments are repeated 5 times and we show average results.

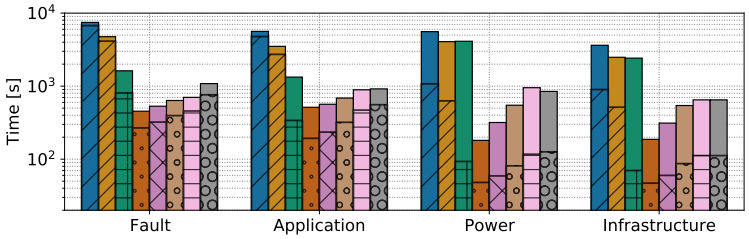
All experiments were carried out in a Python 3.6 environment, using the *scikit-learn* 0.20.3 and *numpy* 1.16.2 packages. The baseline methods described in Section 4.1.2, as well as the CS method, were implemented in Python leveraging native numpy functions to optimize performance. All of our implementations, together with the scripts needed to reproduce all of the results obtained against HPC-ODA in this section, are included in the dataset distribution as a self-contained Python framework [28]. Finally, the machine on which the tests were conducted is equipped with two 14-core Intel Haswell Xeon E5-2697 v3 CPUs, 64GB of RAM and runs Linux SLES15 SP1.

4.4.2. The Jensen-Shannon Divergence

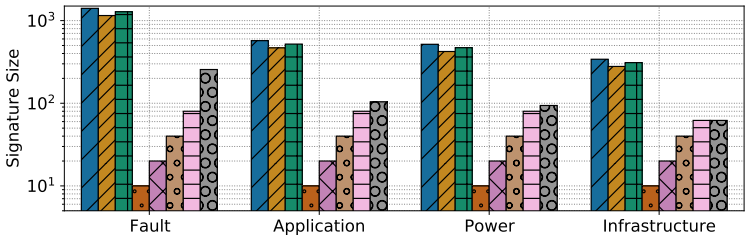
In some experiments we also analyze the fidelity of the compression generated by the CS method, and how it changes with different signature sizes. Several metrics are available in the literature, such as the *Kullback-Leibler* (KL) [KL51] or the *Jensen-Shannon* (JS) *divergence* [DLP97]: however, these are heavily affected by the curse of dimensionality, since they rely on the joint probability distributions of the datasets' dimensions. A generalized form of the JS divergence exists [AP07] and can be applied to an arbitrary number of dimensions, but it operates by comparing them against one another and not in separate sets. To circumvent this issue we leverage the property that data treated with the CS method's sorting stage can be visualized in image-like form, as shown in Figure 4.1: as such, dimensions in the original data can be directly mapped to those of the CS signatures and there is no need to use n-dimensional probability distributions in the computation. Instead, we collapse the probability distributions to a two-dimensional space, where the vertical axis captures the dimensions and the horizontal axis their values. This allows us to use the following JS divergence equation:

$$JS(P_d||P_s) = H\left(\frac{P_d + P_s}{2}\right) - \frac{H(P_d) + H(P_s)}{2} \tag{4.4}$$

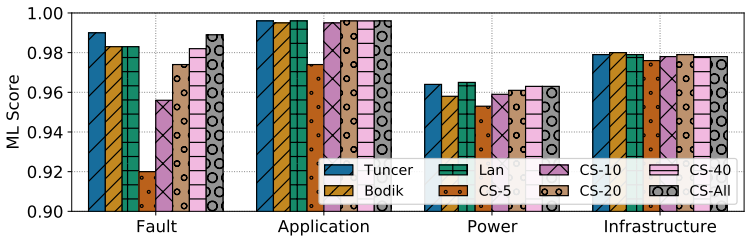
In Equation 4.4, P_d and P_s represent the two-dimensional probability distributions of the original uncompressed data (treated with the CS method's sorting stage) and that of the CS signatures respectively, while $H()$ is the Shannon entropy function. In practical terms, $P_d(v, y)$ or $P_s(v, y)$ signifies the *probability of value v on dimension y*, which is computed from the marginal probability distribution of dimension y and divided by n, to ensure that P_d and P_s are in turn probability density functions. Furthermore,



(a) Dataset generation (bottom bar) and cross-validation time (top bar).



(b) Size of the signatures.



(c) Machine learning score.

Figure 4.3.: Average testing times (a), resulting signature size (b) and machine learning scores (c) in terms of F1-score (Fault and Application) and 1-NRMSE (Power and Infrastructure) for each method [Net+21b].

we apply nearest-neighbor interpolation to the CS data along its vertical axis to ensure that it has the same number of dimensions as the original. This process is performed twice, once for the real components of the CS signatures and once for the imaginary ones, against the uncompressed first-order derivatives of the original data, and we then compute an average JS divergence value.

4.4.3. Machine Learning Performance

Here we show the results of 5-fold cross-validation applied to the first 4 segments of the HPC-ODA collection using the baseline methods in Section 4.1.2 as well as the CS method with 5, 10, 20, 40 or *all* (as many as the sensors in the segment) blocks; in this experiment we use random forests as models. Results are shown in Figure 4.3: in Figure 4.3a we show the testing time for each signature method on each segment, separated between the dataset generation and cross-validation phases, while in Figure 4.3b we show the corresponding signature sizes and in Figure 4.3c the ML scores. Among the baseline

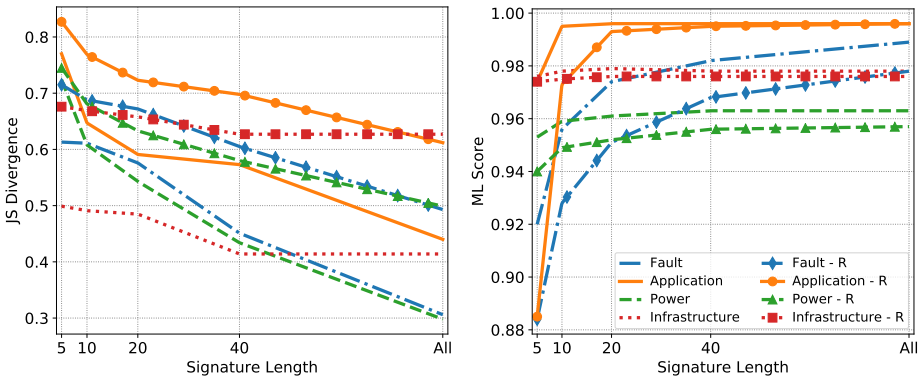
methods, the Tuncer one tends to achieve the best accuracy, but it is consistently the worst performer in terms of testing time due to its statistical computations. The Bodik method shows a similar behavior, while the Lan method is a lightweight alternative to the two, with slightly better testing times.

We see that the CS method achieves the same ML scores as the baseline methods: for some segments this is only achieved using a large amount of blocks (e.g., Fault), whereas for other segments even using only 5 blocks suffices (e.g., Infrastructure). In all cases, however, these results are obtained with signature sizes that are up to one order of magnitude smaller than those associated with the baseline methods. This is reflected in the testing times, where both the dataset generation times and the cross-validation times (especially for regression tasks) are up to one order of magnitude smaller when using the CS method. The implications of these results are not trivial: system administrators can evaluate the performance of ODA models using the CS method with varying numbers of blocks, and then freely choose the configuration that yields the best compromise in function of overhead on the system and runtime constraints. Moreover, they can also train models using low-resolution signatures and then feed down-scaled high-resolution signatures to them (or do the opposite), allowing administrators to compute a single CS signature per HPC component that can then be scaled and fed into different ODA models according to their specific needs. The near-optimal ML scores of the CS method are comparable with those of the baseline methods and satisfy the *Performance* requirement.

4.4.4. Quality of Compression

Here we focus on characterizing the quality of the CS method's compression, again with respect to the first four segments of HPC-ODA. In Figure 4.4 we show the JS divergence of the signature sets processed with the CS method against the original HPC-ODA data, as well as the ML scores in function of l. We also show results obtained when using only the real components of the CS signatures, thus omitting the information regarding first-order derivatives. The information in the two plots is complementary: in all cases, an increase in l (or similarly a decrease in w^l, which we do not show here for space reasons) corresponds to both higher ML scores and lower JS divergence, suggesting a correlation between the two metrics. This implies that the CS method's compression scales in a stable and predictable way, with more information content being captured at higher block counts. Furthermore, not all HPC-ODA segments are affected in the same way by changes of l: the Power and Fault segments are affected to a greater degree, and their ML scores improve significantly together with l. The Infrastructure segment sees only minor effects, with both metrics remaining mostly stable in function of l, while in the Application segment the decrease in JS divergence between an l of 5 and 20 translates faithfully in a proportional increase in ML scores; after this point, model performance for this specific problem reaches its peak and any further reduction in JS divergence does not have any beneficial effect.

Removing the imaginary components of the CS signatures produces interesting results: all segments see a noticeable impact on JS divergence (~0.2 increase), whereas ML scores



(a) Jensen-Shannon divergence.

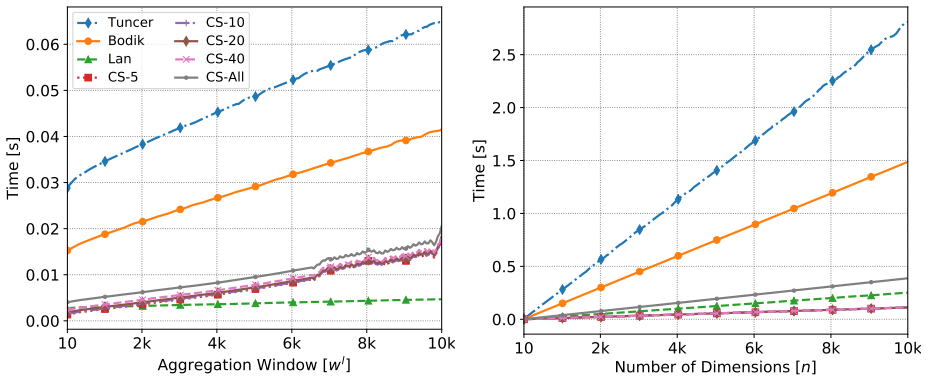
(b) Machine learning score.

Figure 4.4.: Jensen-Shannon divergence and machine learning score values for all use cases at different CS signature lengths, as well as when using only the real components (labeled as -*R*) [Net+21b].

change based on the features and requirements of the specific problem at hand. The Power and Fault segments, in particular, see a significant degradation in the ML score, whereas the Application segment is affected only for values of l lower than 20. The Infrastructure segment, on the other hand, exhibits the same ML scores in both scenarios. In general, since the imaginary components capture trend-related information and allow to distinguish states that have the same static (i.e., average) information, their relevance depends on the nature of the data, its granularity as well as the machine learning problem in question. It should also be noted that in all cases both the JS divergence and ML scores remain monotonic in function of l, even when removing the imaginary components. With its clear compression properties and its remarkable machine learning performance even at very low values of l, the CS method satisfies the *Compression* and *Manipulability* requirements. In Appendix D we discuss additional experiments conducted with our implementation of the JS divergence.

4.4.5. Scalability and Computational Complexity

We now characterize the CS method's scalability, in terms of the time required to compute a signature, and compare it to its theoretical computational complexity. To this end we generated a series of random S^w matrices with increasing aggregation window lengths (w^l) and numbers of dimensions (n). We then computed signatures for each matrix size and each method, repeating the process 20 times and picking median times; we use the median in place of the average to minimize the fluctuations associated with measuring the timing of a short computation, such as that of the CS method. We also exclude the CS method's training stage from this evaluation: in most use cases (e.g., in-band ODA in compute nodes) training is performed only once and potentially offline, with a fixed CS model over time. On the other hand, most use cases in which training



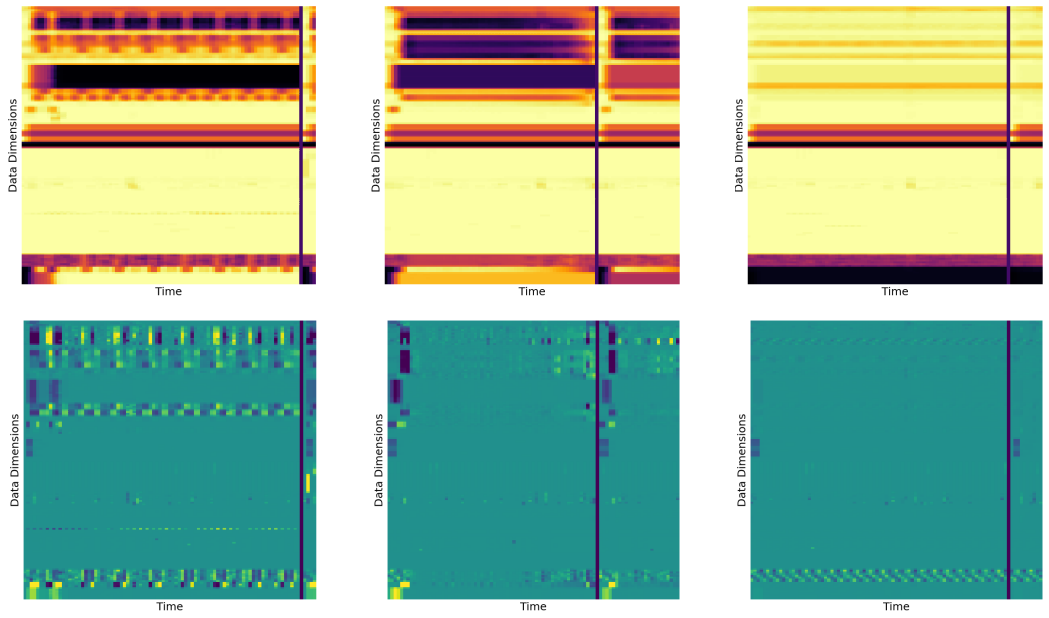
(a) Aggregation window.

(b) Number of dimensions.

Figure 4.5.: Time to compute a signature for all methods, in function of the aggregation window w^l (a) or the number of dimensions n (b). When varying w^l, n is fixed to 100 and vice-versa [Net+21b].

at every time step is necessary (e.g., system-wide ODA) are out-of-band, and hence overhead and complexity are not a concern. Figure 4.5 shows the observed time to compute a signature for each method in function of w^l and n.

It can be seen that the CS method scales linearly both in function of w^l and n, as expected from its $O(w^l n)$ complexity. The baseline methods exhibit similar behavior, scaling linearly in function of n as they process each dimension's data independently. On the other hand, the Tuncer and Bodik methods show slightly non-linear behavior in function of w^l, which is due to the computation of percentiles on each dimension, with a complexity of $O(w^l log(w^l))$. Even though all methods exhibit similar scaling, it can be seen that the CS method performs significantly better than Tuncer and Bodik at high values of w^l and n, with Lan being the only exception due to its minimal nature. In particular, the CS method requires one order of magnitude less time to compute a signature from 10k data dimensions compared to Tuncer and Bodik. Furthermore, it can be seen that the number of blocks chosen for the CS method has minimal impact on its footprint. Some differences can still be seen, similarly to what was observed in Figure 4.3a when processing the HPC-ODA segments: this is a side effect of our Python implementation. There, a larger l results in a larger number of calls to native functions and hence worse performance, but we expect it to be negligible in a compiled implementation. Similar considerations can be had for the fluctuations at high values of w^l in Figure 4.5a, which are likely due to the interaction between the Python runtime and the system's memory. Thanks to its scalability compared to the baseline methods, the CS method satisfies the *Footprint* and *Scalability* requirements.



(a) Kripke. (b) HPL. (c) Quicksilver.

Figure 4.6.: Real (top) and imaginary (bottom) components of the signature heatmaps for three applications using the CS method with 160 blocks. Solid vertical lines indicate the end of a run [Net+21b].

4.4.6. Fitness for Visualization

Having demonstrated the effectiveness of the CS method in various machine learning scenarios, we now demonstrate how it produces signatures that can be easily visualized and interpreted. In Figure 4.6 we show three signature heatmaps in the same format as those in Figure 4.1: these correspond to Kripke, HPL and Quicksilver runs extracted from the HPC-ODA Application segment and were computed with data from all available compute nodes, for a total of ~800 sensors. We use 160 blocks, and show both the real and imaginary parts. The applications were executed using one MPI process per node and as many OpenMP threads as physical cores. Solid vertical lines separate different application runs.

The three applications can be clearly distinguished and exhibit interpretable performance patterns. For example, Kripke shows a very clear iterative behavior which can be observed in both the real and imaginary components. HPL, on the other hand, exhibits a much more constant load on the compute nodes, with a very pronounced initialization phase. Quicksilver, finally, shows very light load on computational resources with small values across all blocks, but a very unique behavior emerges: at the bottom of the imaginary components we can observe a clear periodic pattern, which is consistent across runs. Upon further inspection, we found that the corresponding blocks aggregate data

associated to CPU cycles, and as such this pattern indicates oscillating CPU frequencies on all compute nodes, induced by Quicksilver's code mix. The AMG application shown on the right side of Figure 4.1 exhibits yet a different behavior: in the upper half of the real components a gradient in the block values can be observed, which is associated with increasing memory usage over the run. Furthermore, similarly to Kripke, also AMG shows a distinct iterative behavior.

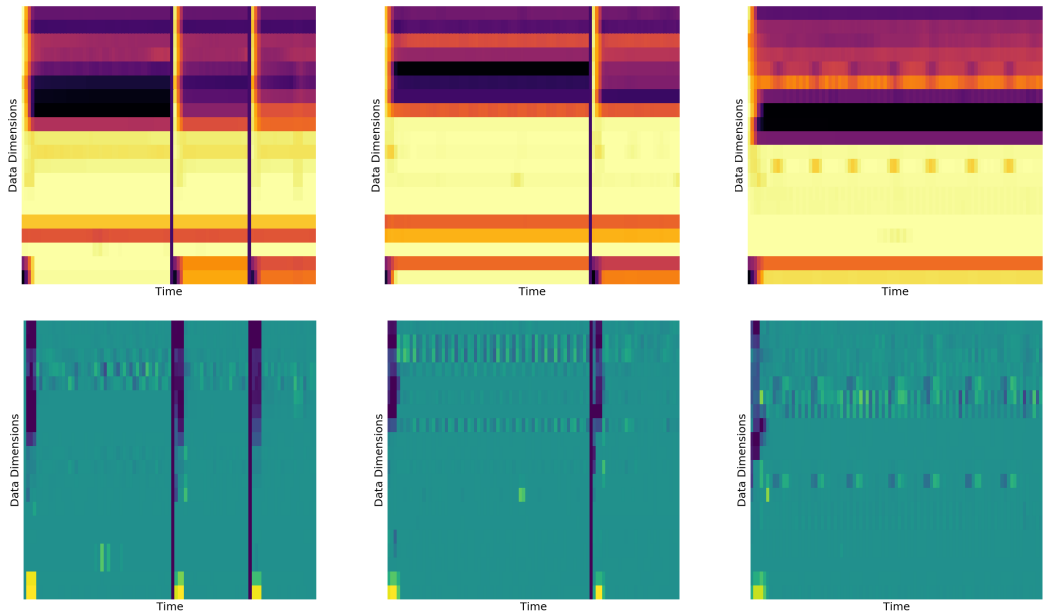
The structure of the CS signatures is consistent in all figures and matches our description in Section 4.2: the upper section includes blocks that are descriptive of system behavior, while the middle part groups those carrying little information content, with metrics being almost constant or unaffected by the system's status. Finally, the blocks in the bottom part are again descriptive of system behavior, but their values are inversely proportional to those of the blocks at the top. Like in a periodic signal, CS signatures are able to highlight periodic behaviors only where their period $p > 2 \cdot w^l$, in accordance with the sampling rate of the original data. Unlike the baseline methods we have chosen, the CS method produces interpretable data representations which can be used as a guide for root cause analysis, and satisfies the *Visualizability* requirement.

4.4.7. Portability across Architectures

Here we demonstrate the generality of the signatures computed with the CS method. To this end, we performed a test using the Cross-architecture segment of HPC-ODA, replicating the classification task of the Application segment. Here, the testbed, SuperMUC-NG and CooLMUC-3 nodes used to acquire data have different architectures and amounts of sensors: the testbed node is equipped with two 64-core AMD Epyc Rome 7742 CPUs and has 39 sensors, while the SuperMUC-NG node uses two 24-core Intel Skylake Xeon Platinum 8174 CPUs and has 52 sensors. Finally, the CooLMUC-3 node uses a 64-core Intel Xeon Phi 7210-F many-core CPU and has 46 sensors. Using a single node per hardware architecture allows us to focus on their peculiarities and the impact they have on monitoring and machine learning models, as opposed to network and application behavior in a distributed context; as a consequence, the underlying applications in the segment are executed in shared-memory OpenMP fashion. We process this heterogeneous data as in the following:

1. The CS method is applied to the data of each of the 3 nodes independently, generating 20-block signatures.
2. We merge the three resulting datasets into a single one and perform shuffling.
3. We perform 5-fold cross-validation, classifying running applications with no knowledge of the original architecture.

This experiment led to an average F1-score of 0.995 using a random forest model, a result that is equivalent to the one presented for the Application segment with no significant degradation in performance. When using a multi-layer perceptron classifier,



(a) Testbed node. (b) SuperMUC-NG node. (c) CooLMUC-3 node.

Figure 4.7.: Real (top) and imaginary (bottom) components of the signature heatmaps for the LAMMPS application on three different compute node types, using the CS method with 20 blocks. Solid vertical lines indicate the end of a run.

the F1-score amounts to 0.992. This confirms that the CS method produces generic signatures and enables portability of models across systems, as well as robustness against changes in the sets of available sensors. It should be noted that this experiment cannot be reproduced at all using the baseline methods in Section 4.1.2, since they produce signatures with different lengths depending on the specific compute node, that cannot be merged into a single dataset. Similarly, due to this segment's heterogeneity, we do not show scaling results for it as done in Section 4.4.4. The CS method allowed us to train a single model to recognize applications running under different architectures and monitoring configurations, satisfying the *Portability* requirement.

In Figure 4.7 we show the signature heatmaps associated with the LAMMPS application, executed under the same configuration on each of the 3 compute nodes. The format of the signatures changes slightly across the architectures, and there are obvious execution time differences due to the varying performance of the nodes. However, the same performance patterns can be recognized in all cases, both in terms of periodicity in the imaginary parts and distribution of the metrics in the real ones. This is confirmed upon further inspection of the sensors' ordering after applying the CS method: we found that, even though the data of each compute node was treated separately, the order was very consistent and similar.

4.5. Applying the CS Method to Generic Data

In this chapter we proposed the CS method for processing multi-dimensional time-series data deriving from sensor monitoring of large-scale HPC systems. We demonstrated its properties in terms of scalability, compression, portability, suitability for visualization as well as machine learning performance through a series of experiments carried out with the HPC-ODA dataset collection. Additionally, in Section 5.3 we will discuss a production ODA deployment leveraging the CS method for the tuning of an HPC system's cooling infrastructure. While the scope of our studies falls within the data center domain, the nature of the sensor data we use (e.g., power and temperature measurements) as well as the associated use cases are not dissimilar from what is found in the wider industrial landscape: as a practical example, Karlstetter et al. demonstrated the use of fine-grained sensor data to detect anomalies within gas turbines, as well as for visualization purposes [Kar+19]. Given the above, the CS method has significant potential for a wide range of applications in the industrial domain, extending as far as generic multi-dimensional time-series data: demonstrating its practical suitability for these scenarios will be the object of future work.

Own publication acknowledgement. The CS method, as well as the first release of the HPC-ODA dataset collection [28], were originally discussed in [Net+21b]. The Fault segment of HPC-ODA, instead, was extracted from a larger public dataset [29] which was acquired in the context of prior work by the author, not associated with this dissertation [Net+18a; Net+19a].

5. Production Operational Data Analytics Experiences

In this chapter we demonstrate how we used DCDB, Wintermute and the CS method for two different ODAV and ODAC deployments, carried out on different production HPC systems, and we share the associated insights we gained from them. In Section 5.1 we discuss the DCDB and Wintermute configuration used on the SuperMUC-NG HPC system at LRZ, while in Section 5.2 we provide an overview of our experience on the prototype HPC system realized in the scope of the DEEP-EST project. In Section 5.3 we then present the experimental insights associated with proactive cooling control on this machine and in Section 5.4 we conclude the chapter.

5.1. DCDB on SuperMUC-NG

Here we outline our experience deploying DCDB and Wintermute on the SuperMUC-NG HPC system at LRZ, starting from the motivation and proceeding with the system's configuration and a usage example. In this use case, the frameworks are configured to perform online visualization of user job data.

5.1.1. Motivation

As introduced in Section 3.5.1, SuperMUC-NG is the flagship HPC system at LRZ with more than 6,000 compute nodes and 27PFlop/s of peak performance. Effective and efficient usage of such a large-scale system, by users and administrators alike, requires deep understanding of application behavior, which is a challenging task due to the complexity, scale and duration of the runs that are usually performed on a system of this kind. Monitoring simplifies this process considerably, as we have shown in Section 3.7.2: the previous iterations of the SuperMUC line employed the Persyst framework [GHB14], which is an end-to-end solution to collect metrics of interest from compute nodes, store them in a database and finally visualize them on a per-job basis in a web frontend. Persyst relies on computing quantiles from the distributions of certain derived performance metrics, which can effectively highlight performance bottlenecks and inefficiencies in large-scale user jobs.

Monitoring on a large-scale system entails a series of technical challenges and requirements, which have been discussed in Section 2.3.1: these include, for example, *Scalability*, *Footprint*, *Extensibility* and *Holism*. To overcome these challenges, it was decided to decouple the Persyst web frontend from the backend component performing

the collection and aggregation of monitoring data on the system itself. The DCDB and Wintermute frameworks were selected instead for this ODAV use case: these are tasked with performing all of the necessary monitoring and data aggregation in a scalable and transparent way, feeding the processed data into a dedicated database and subsequently enabling visualization via the Persyst frontend.

5.1.2. DCDB Configuration

The placement of DCDB components on the system is summarized in Figure 5.1. There is one DCDB Pusher daemon running in each of the SuperMUC-NG compute nodes and collecting in-band monitoring data with a series of plugins, whose configuration is very similar to the one discussed in Section 3.5.1. In particular, we use the following Pusher plugins, each sampling data every 10s:

- **Perfevents**: it samples a variety of CPU performance counters, characterizing the computational, cache and floating point behavior of applications.
- **MSR**: it samples a small subset of CPU performance counters (i.e., CPU instructions and cycles) with a low-level method that results in very little overhead.
- **SysFS**: it samples a variety of metrics from the SysFS virtual file system, including CPU and RAM energy and temperature, as well as performance counters from the Intel Omni-Path network interface.
- **ProcFS**: it samples a set of metrics from the ProcFS virtual file system, such as used RAM or the percentage of idle and user CPU activity.
- **GPFSMon**: it samples several metrics associated with I/O activity on the GPFS distributed file system.

The SuperMUC-NG system is divided into 8 different islands (corresponding to partitions in the fat tree network topology the system uses), each containing 792 compute nodes: each Pusher sends its monitoring data via MQTT to a Collect Agent residing in a management server within the same island. The data is transferred using a dedicated Ethernet network interface for telemetry, so as not to interfere with the communication of user applications occurring on the default Intel Omni-Path interface. The Collect Agents finally insert the data in a distributed Cassandra instance on two dedicated nodes, from which it can then be queried. A *MariaDB* [30] database instance runs on the same nodes as the Cassandra database: this is used for accounting of user activity, as well as to store all Persyst data that is used specifically for visualization. SuperMUC-NG also provides 144 *fat* compute nodes, which are equipped with 768GB of RAM as opposed to the 96GB of the other nodes: these are monitored by dedicated Pusher daemons with an associated Collect Agent, and are treated as a separate island in SuperMUC-NG.

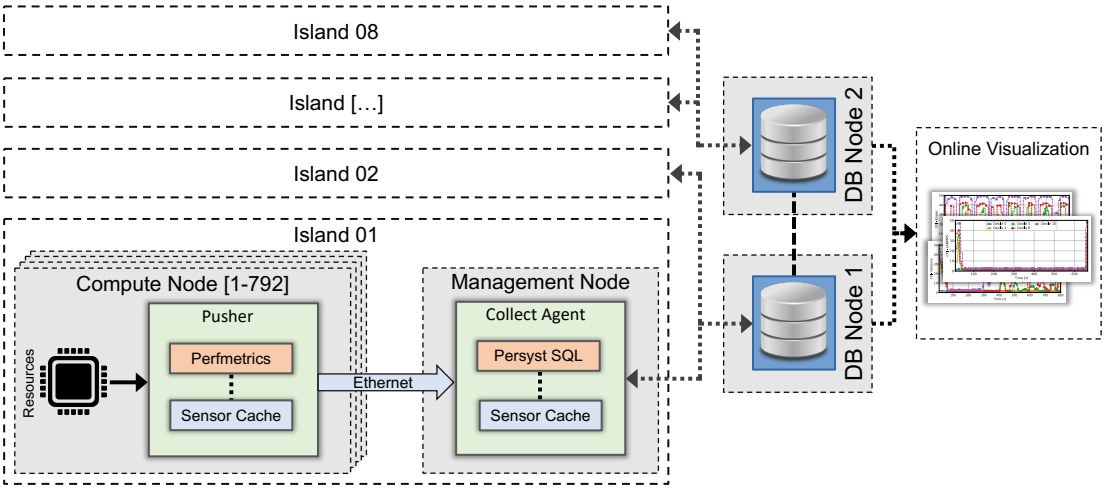


Figure 5.1.: An overview of the DCDB and Wintermute configuration as deployed on SuperMUC-NG. For the sake of simplicity, we omit the system's fat island.

5.1.3. Wintermute Configuration

Like for DCDB, the Wintermute configuration on SuperMUC-NG is very similar to the one presented in Section 3.7.2, employing two different operator plugins in order to realize a pipeline and in turn distribute load. It is represented in Figure 5.1 together with the DCDB configuration: we use the Perfmetrics plugin in the Pusher daemons and the Persyst SQL plugin in Collect Agents - both operate at an interval of 120s, which was determined to be sufficient to characterize job behavior while resulting in very light overhead and storage requirements. In fact, all of the job data computed via Wintermute must be kept for the entire life time of the system and is never deleted. The information about jobs running on the system is exposed via a dedicated tool, which pulls data from SLURM and makes it available in Cassandra.

The Perfmetrics plugin in the Pushers is configured to compute 27 derived per-core or per-node performance metrics, which are handled by 7 different operators. The operators are invoked every 120s, but they only pick the most recent value for each input sensor, associated to the latest 10s: the effectiveness of this type of statistical sampling was proven in the context of the original Persyst framework [GHB14]. The final set of metrics is able to fully characterize the performance of a user application and includes indicators about CPU performance (e.g., FLOPS or CPI), memory activity (e.g., memory bandwidth or used amount), energy efficiency (e.g., energy of the CPU or RAM) and finally network usage (e.g., data rates or file system operations). The Persyst SQL plugin in the Collect Agents, on the other hand, is configured to aggregate each of the 27 derived metrics from the compute nodes of jobs running on SuperMUC-NG, every 120s: for each of them, it computes the associated deciles, average and a *severity measure*, which is an efficiency indicator against a pre-defined threshold. The final per-job data is then pushed to one of the Storage Backend types specifically supported by the Persyst

SQL plugin. Here, the data is pushed to the available MariaDB instance for long-term storage, as opposed to the default Cassandra database. The data coming from the Perfmetrics operators in the Pushers, instead, is stored in the latter as normal. We now discuss the challenges associated with the large scale and integration requirements of this ODAV deployment, as well as the solutions we adopted.

In-memory Processing. The DCDB and Wintermute configuration described here results in a total of 14.5 millions of sensors, excluding the per-job aggregated data. However, the sensors associated with the raw monitoring metrics sampled in the Pushers (e.g., CPU performance counters), which amount to roughly 6.8 millions of the total, are configured such that they are never sent to Collect Agents. Instead, they are simply kept in the local sensor cache, so as to be leveraged by the Perfmetrics operators - this is only possible due to the tight integration between Wintermute and DCDB, which allows for direct in-memory processing of data. The derived performance metrics computed by the latter every 120s are then sent out to Collect Agents without restrictions: the reasoning behind this type of configuration is that the raw sensors sampled by the Pushers have no use on the Persyst web frontend, which instead relies on more informative metrics. As such, the burden incurred by the transmission and storage of the former was deemed unnecessary, greatly simplifying the monitoring infrastructure. This results in roughly 60,000 inserts per second into the Cassandra database as opposed to the 700,000 of the naive configuration, with a total of 7.7 millions of public sensors that can be queried, 1,213 per compute node. Ignoring the per-job aggregated data that is stored in the MariaDB instance, each sensor's data has a time-to-live of 30 days, further reducing storage requirements down to 2.5TB for Cassandra. An interval of 120s, on the other hand, results in a monthly average of 10GB of per-job data added to MariaDB, which is sustainable for years to come.

Workload Distribution. Processing the FLOPS metric for a job running on 1,024 nodes requires fetching data for 49,152 distinct sensors, one for each CPU core associated with the job. This would result in several millions of database queries every 2 minutes to process all 27 metrics: as such, we use a specific strategy to distribute work among Collect Agents. In particular, each of them only aggregates data for jobs which have the majority of their compute nodes in the same island as the Collect Agent itself. As a tie breaker, if a single majority island is undefined for a given job, the Collect Agent with the lowest island identifier will seize it. This strategy has a double advantage - first, it is the most natural way to distribute work on the SuperMUC-NG system, as all islands have the same size and hence a similar amount of jobs. Secondly, this allows to optimize access to sensor data: in fact, since most compute nodes for a given job always belong to the same island as the Collect Agent, the respective Perfmetrics sensor data will be readily available in the local sensor cache, reducing load to the Storage Backend. Sensors that are not available in the local cache (e.g., for jobs spanning multiple islands) are queried transparently from the Cassandra database.

Integration with Legacy Tools. As the legacy web frontend for the Persyst framework supported only MariaDB as Storage Backend and no development was planned on this component, integration had to be done on the DCDB side. The plugin-based and generic nature of Wintermute aided us in this purpose: in fact, we integrated all appropriate logic to manage MariaDB connections and perform insert operations within the Persyst SQL plugin itself - in general, a *sink* Wintermute plugin is one that is able to write to alternative Storage Backends. This does not impair the plugin's ability to query sensor data from (as well as publish to) the default Cassandra database. This approach offered a clean and transparent upgrade path - no code changes were required to the Persyst, DCDB or Wintermute cores, obtaining in turn an open-source plugin that might be useful for future use cases as well.

5.1.4. Usage Example

Here we provide a brief overview of how the per-job aggregated data stored in the MariaDB database through our ODA pipeline can be visualized by HPC users on the Persyst web frontend. An extensive analysis of the performance metrics exposed to users and their meaning was the object of previous work [GHB14]. On the SuperMUC Phase 1 and Phase 2 HPC systems at LRZ, users of the Persyst framework were able to characterize the performance of application codes accurately, gaining insight over details ranging from CPU vectorization characteristics to inter-node imbalance and allowing in turn to identify numerous code optimization opportunities [Bra+20]. Users are required to log into the frontend using the same credentials as on the SuperMUC-NG system, leading to an initial web page containing a list of jobs submitted by the authenticated user whose data can be visualized - jobs do not necessarily have to be finished for their data to be visualized, and they can be analyzed as they run, online.

Selecting a job leads to an overview page, as shown at the top of Figure 5.2: here, a global view of the job's performance is presented, showing the medians of all available performance metrics over time in a heatmap-like style. Each pixel of the heatmap shows the median of a certain performance metric over a specific 120s time window, as aggregated by the Wintermute Persyst SQL plugin from all compute nodes associated with the job. Placing the cursor over any of the data points highlights the full distribution (shown in terms of deciles) of the selected performance metric in the given time range, allowing to spot performance anomalies and imbalances with ease. Clicking on any of the metric names, finally, leads to a new page (shown at the bottom of Figure 5.2) which allows users to visualize the deciles and average of the selected metric over time for a more detailed analysis. In this type of visualization, multiple metrics at a time can be shown for comparison purposes. In this example, the deciles and average of the CPI metric are visualized over time, highlighting a clear periodic behavior that is likely associated with the job's underlying HPC application cycling between compute and communication-intensive phases. On top of using the web frontend, users can also query all available data via the DCDB command-line tools.

Figure 5.2.: Two screenshots the Persyst web frontend's interface, showing the job overview (top) and the metric visualization screens (bottom) [Net+21c].

5.2. DCDB on the DEEP-EST Prototype

Here we discuss our ODA experiences carried out on the prototype system built in the context of the DEEP-EST project. We start with an overview of the system and the rationale behind its design, proceeding with the DCDB and Wintermute configuration deployed on it, which is tailored for proactive control of the cooling infrastructure.

5.2.1. Motivation

DEEP-EST is the third iteration in the DEEP project series [12] funded by the European Union's research programmes: these focus on developing novel exascale-ready HPC system architectures based on the *Modular Supercomputer Architecture* (MSA) concept,

Table 5.1.: The architectures of the modules composing the DEEP-EST cluster [31].

Module	CM	ESB	DAM
Nodes	50	75	16
CPU	Intel Xeon Gold 6146 2 cpus x 12 cores x 2 threads	Intel Xeon Silver 4215 8 cores x 2 threads	Intel Xeon Platinum 8260M 2 cpus x 24 cores x 2 threads
Accelerators	None	Nvidia V100 32GB GPU	Nvidia V100 32GB GPU Intel STRATIX 10 FPGA
Memory	192GB	48GB	384GB 2TB non-volatile memory
Storage	400GB NVMe SSD	512GB SSD	240GB SSD (OS) 2 Intel Optane 1.5TB SSDs
Interconnect	Mellanox Infiniband	Extoll	Extoll

which are then evaluated in the form of small-scale prototypes [SEL19]. Compared to traditional homogeneous HPC systems, the MSA design decouples the components that limit scalability (e.g., CPU-based compute nodes) from those that have a higher scaling potential (e.g., GPU-based nodes with weak CPUs), from both the computational and energy efficiency standpoints. This results in highly heterogeneous HPC systems, which are separated in *modules*, each with different compute node and interconnect architectures, as well as different scale and functionality [Erl+19]. The HPC prototype built in the context of DEEP-EST is hosted at the *Juelich Supercomputing Centre* (JSC), which has also the role of project coordinator.

LRZ is one of the partners of the DEEP project series and its role is to supply monitoring for the developed prototypes, as well as insightful data analytics - these two tasks are carried out in DEEP-EST with the DCDB and Wintermute frameworks respectively. The DEEP-EST prototype's heterogeneous nature is a good fit for DCDB's holistic design, with Wintermute being used for multiple purposes: the chosen ODA functionality is modeled around the MSA design, with the aim of supporting and highlighting its advantages. As such, Wintermute implements a proactive ODAC control loop for the inlet warm-water cooling temperature of the individual prototype racks. The racks operate in different conditions due to the heterogeneity of the underlying compute node architectures, motivating the use of a granular control strategy of this kind. On top of this, Wintermute provides job-level and module-level aggregation of metrics (similarly to Section 5.1) as well as sub-sampling of sensors for long-term storage, both of which are ODAV use cases.

5.2.2. System Overview

The DEEP-EST prototype is a small-scale heterogeneous HPC cluster divided in 3 modules for a total of 141 compute nodes, whose architecture is summarized in Table 5.1. The first module, named *Cluster Module* (CM), is a traditional CPU-based HPC design

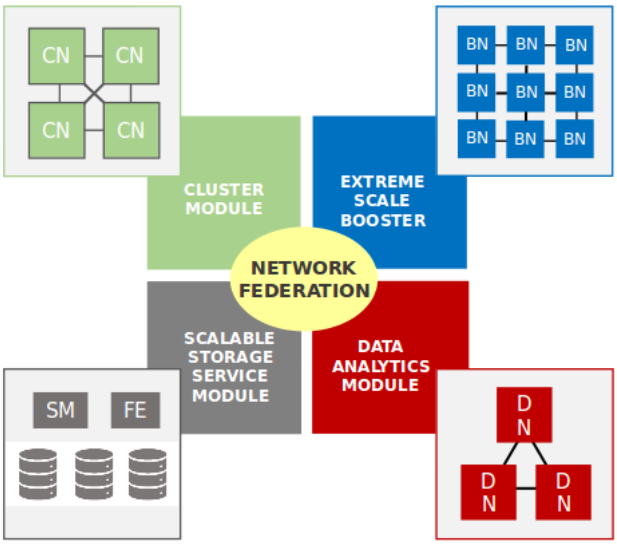


Figure 5.3.: A high-level overview of the DEEP-EST prototype's architecture [31].

comprising 50 nodes within a single rack. These are equipped with two 12-core, high-performance CPUs, 192GB of RAM and a Mellanox Infiniband interconnect. The second module is the *Extreme-Scale Booster* (ESB), which is composed of 75 nodes hosted in 3 separate racks: these follow an unconventional architecture, using a weak 8-core CPU supported by a GPU accelerator, coupled with 48GB of RAM and an Extoll interconnect [Frö15]. Finally, the *Data Analytics Module* (DAM) comprises 16 compute nodes housed in a separate rack. These employ two 24-core CPUs, coupled by both a GPU and an FPGA accelerator. DAM nodes are then equipped with 384GB of RAM, 2TB of non-volatile memory, and an Extoll interconnect. While the CM and ESB modules are warm-water cooled, the DAM is air-cooled. All compute nodes employ the *CentOS* 7 operating system, while the distributed file system is supported by multiple *BeeGFS* [Hei14] and GPFS instances.

The modules of the DEEP-EST prototype are meant to be used in different ways: the CM is designed for traditional HPC workloads (or parts thereof) that are not well-suited for GPU or FPGA acceleration and that require strong CPU performance. Conversely, the ESB provides a highly efficient solution for workloads that have GPU acceleration potential. The DAM, finally, provides a middle ground between the CM and ESB, with nodes that provide accelerators and strong CPU performance at the same time: these nodes are designed for data analytics workloads (e.g., training of machine learning models), which are notoriously dependent on strong single-node performance and ample memory availability. Beyond using each of the modules separately, users of the prototype are encouraged to adapt their applications to leverage the prototype's modular nature, by decoupling CPU-intensive and accelerator-friendly code regions and

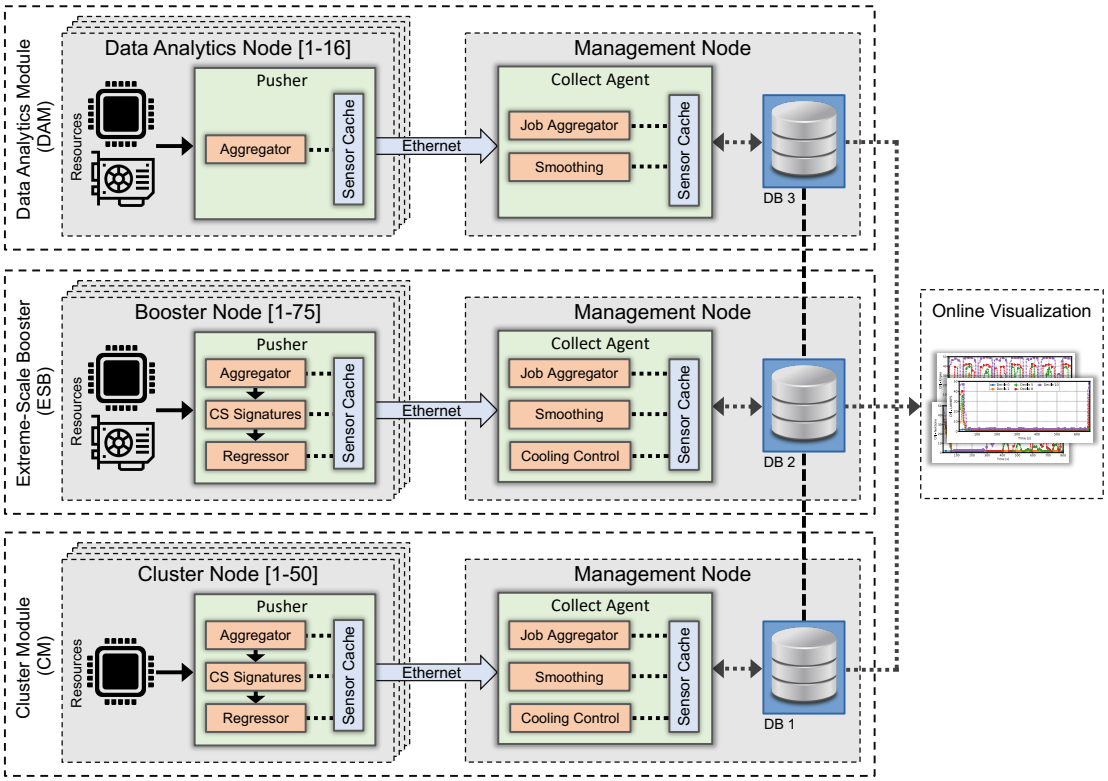


Figure 5.4.: An overview of the DCDB and Wintermute setup as deployed on the DEEP-EST modular HPC system.

having them run on different compute node types (e.g., CM and ESB) at the same time. This is the final aim of the MSA design, as opposed to using a single homogeneous system where performance at scale is often constrained by bottlenecks, which render high energy efficiency harder to obtain.

The DEEP-EST prototype includes a series of additional components to orchestrate the modules' operation, as shown in Figure 5.3. Due to the heterogeneity of the network interconnects used in the various modules, a series of gateway compute nodes are present to enable communication across them - this results in the prototype's modules forming a *network federation*. Additional components (under *scalable storage service module* in Figure 5.3) ensure access to the available distributed file systems from all modules. Finally, the prototype as a whole is managed as a single SLURM cluster, and specific options are supplied to configure jobs running over different modules.

5.2.3. DCDB Configuration

The placement of DCDB software components over the DEEP-EST prototype is summarized in Figure 5.4. According to the heterogeneity of CM, ESB and DAM nodes, the Pusher daemons running in these have different configurations. In general, we use the

following plugins on all modules, with a sampling interval of 10s:

- **Perfevents**: it samples a variety of performance counters on available CPUs, characterizing the computational, cache and floating point behavior of applications.
- **SysFS**: it samples a variety of metrics from the SysFS virtual file system, including node, CPU and RAM energy and temperature, as well as GPU energy (if available) and performance counters from the Infiniband or Extoll interconnect.
- **ProcFS**: it samples a set of metrics from the ProcFS virtual file system, such as used RAM or the percentage of idle and user activity on available CPUs.

In addition to the plugins described above, the NVML plugin is used on ESB and DAM nodes, sampling performance metrics such as utilization, clock and energy from available GPUs. The Pusher daemons transmit their data via MQTT to Collect Agents running on dedicated management nodes, one for each module of the prototype. These nodes host both the Collect Agent daemons and the distributed Cassandra database. Similarly to Section 5.1, communication occurs via dedicated Ethernet interfaces that are used for management, so as not to impact user applications. Additional daemons (not shown in Figure 5.4) run on the management nodes so as to provide access to a Grafana web frontend. A Pusher to collect data from the warm-water cooling and power infrastructure via the SNMP and REST plugins is also present.

5.2.4. Wintermute Configuration

Wintermute plugins are used for a wide variety of purposes on the DEEP-EST prototype, with several pipelines in place spanning compute and management nodes. First, in all of the CM, ESB and DAM Pushers there is an *Aggregator* plugin running: it has the role of aggregating per-CPU core performance counters to the compute node and socket (in the CM and DAM) levels. This results in 51 operators in the CM and DAM, and 15 in the ESB, with as many associated output sensors being computed every 10s from the most recent per-CPU readings.

The CM and ESB Pushers have a more complex configuration compared to the DAM ones, as they implement a pipeline to predict CPU and GPU temperatures in order to steer control decisions for the associated warm-water cooling system. This is supported by two operator plugins, both running with a sampling interval of 60s: first, a *CS Signatures* plugin is tasked with computing signatures with the CS method (see Chapter 4) using as input a set of sensors queried over the last minute of operation - these are both raw sensors and aggregated ones produced by the Aggregator plugin. The operators are configured to produce signatures with 20 complex blocks, and hence 40 sensors. These sensors are leveraged by operators of the Regressor plugin (as described in Section 3.7.1), which take them as input and produce a prediction of the maximum CPU or GPU temperature in the upcoming minute of activity. Regressor operators are configured to not produce any additional features from the input sensors, but use the

raw readings as feature vector components, since this stage is already managed by the CS Signatures plugin. Both in the CM and ESB, the CS Signatures and Regressor plugins employ two operators: one for each of the two CPU sockets in the first, and two for the available CPU and GPU in the second. In both cases, the CS and OpenCV random forest models used to produce predictions were trained offline using archived data, and are loaded from files alongside the DCDB configuration.

Three additional operator plugins are configured to run in each Collect Agent: a *Smoothing* plugin queries recent data from all sensors in the system periodically and computes sub-sampled 5 and 60-minute averages from them. Then, a Job Aggregator plugin computes job-level aggregates from raw sensor data, with 16 operators computing the average, median and sum corresponding to certain input metrics, similarly to what was described in Section 5.1. In order to distribute load among Collect Agents, the Smoothing and Job Aggregator plugins respectively consider only the sensors belonging to the same module as the daemon, and only the jobs that have the majority of their compute nodes in said module. The Job Aggregator plugin is aware of jobs spanning multiple modules, and computes module-level aggregates in addition to the job-level ones. The last Wintermute plugin in the Collect Agent is the *Cooling Control* plugin which, unlike the former two, runs only within the CM and ESB management nodes. Each operator of this plugin is tasked with determining a new setting for the secondary inlet water temperature of a given rack every minute: as such, one operator is associated with the CM rack and three are used for the ESB racks. When invoked, the operators fetch the predicted CPU and GPU temperature readings produced by the Regressor plugin for nodes in a given rack and, depending on how many of these exceed a fixed threshold, an increase or decrease in inlet temperature is determined. The new setting is then applied to the rack's cooling unit via the SNMP protocol. To ensure operational safety, an *Health Checker* operator plugin (not shown in Figure 5.4) runs in the CM and ESB Collect Agents, verifying at 60s intervals that the temperature of each CPU or GPU in the associated compute nodes never reaches the thermal throttling point: if this happens, system administrators are notified via e-mails.

The monitoring and ODA setup described above results in a total of 66,846 sensors, whose data has a time-to-live of 1 year, for a projected 3.5TB of Cassandra storage. It should be noted that, since the associated sensors are only used within CM and ESB nodes for prediction purposes, CS signatures are configured to be kept in local memory and are never transmitted to Collect Agents. Moreover, the total amount of sensors does not include the job-level and sub-sampled aggregates, which are instead expected to be kept for the entire life time of the DEEP-EST prototype. Pressure on the Cassandra database amounts to roughly 7,000 inserts per second. We now highlight the challenges of this ODAC deployment, focusing on the complexity of using a machine learning-based control pipeline in a heterogeneous HPC system.

From In-band Data to Out-of-band Control. Orchestrating a system's operation with data and knobs at multiple levels is one of the main challenges of ODAC. Wintermute's

Listing 3 An excerpt of the Cooling Control plugin's configuration for the DEEP-EST CM rack, showing the block system's template constructs used for sensor specification.

```
controller c1 {
        default def1

        input {
                sensor "<topdown 3, filter cn/s[0-9]{2}/socket>temp-pred" {
                        hotThreshold 73000
                        critThreshold 93000
                }
        }
}
```

holistic nature, with its generic interfaces for querying and publishing sensor data, allows us to overcome this problem: we are thus able to connect compute node-level data processed via in-band means (through the Regressor plugin), to further out-of-band processing in the Collect Agents, which ultimately translates into new settings for SNMP system knobs and into a complete feedback loop. Furthermore, we are able to manage multiple ODA use cases (e.g., cooling control and job data aggregation) within the same framework with ease, demonstrating Wintermute's suitability for complex ODAV and ODAC deployments.

Deployment of Machine Learning Models. To simplify the training and deployment of machine learning models, we leverage the CS Signatures and Regressor plugins' ability to load model data from files: in particular, we trained the CS model (describing the sensors' permutation vector, as well as their lower and upper bounds) and the Regressor model (describing an OpenCV random forest) offline using archived data, which was processed via a Python framework - the final models are saved as files, which can be loaded by the C++ Wintermute plugins. In case further adjustments are required, online model training can be triggered via the Wintermute RESTful API.

System Heterogeneity. Configuring ODAC models on a modular system such as the DEEP-EST prototype is unsustainable at scale: in our case, the Cooling Control operators for each CM and ESB rack require predicted temperatures from a specific set of components (i.e., CPUs or GPUs) within certain compute nodes. However, we are able to leverage the abstraction capabilities of Wintermute's block system, which allows to use compact template-like expressions to pick specific sets of nodes and components with ease. An excerpt of the Cooling Control configuration for the operator associated with the CM rack is shown in Listing 3.

Table 5.2.: An overview of the reference datasets collected on the DEEP-EST prototype for the purpose of training temperature prediction models.

Dataset Type	Prototype Module	Nodes	Sensors	Data Points	Length	Sampling Interval
Experiment	CM - CPU	16	35	∼200k	20h	10s
Experiment	ESB - CPU	16	32	∼100k	20h	10s
Experiment	ESB - GPU	16	27	∼100k	20h	10s
Production	ESB - CPU	16	32	∼40k	7h	10s
Production	ESB - GPU	16	27	∼40k	7h	10s

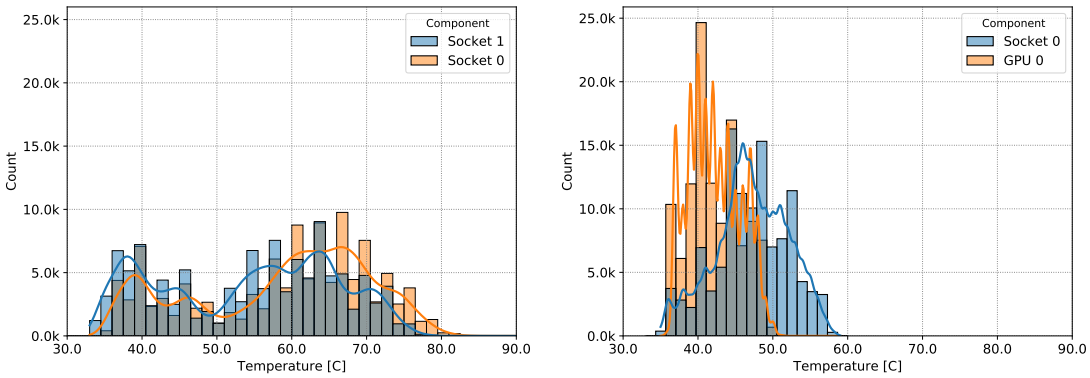
5.3. Cooling Control on the DEEP-EST Prototype

In the following we describe the machine learning models used to support the Wintermute cooling control pipeline on the DEEP-EST prototype. We start by describing the reference dataset acquired for this task and proceed with data exploration. We then discuss the machine learning results associated with CPU and GPU temperature prediction, and conclude with the overall operational cooling control results.

5.3.1. Reference Dataset

In order to gain an overview of the compute nodes' behavior and train machine learning models for CPU and GPU temperature prediction, we acquired several datasets on the DEEP-EST prototype, leveraging the already active production DCDB installation. The features of the datasets are summarized in Table 5.2: similarly to what was done for the Application segment of the HPC-ODA dataset collection (see Section 4.3), for each dataset we executed several parallel applications under three possible configurations on 16 compute nodes of the chosen prototype module, in random order. On the CM, the applications are the CPU-only versions of Kripke, AMG, LAMMPS, Quicksilver, Nekbone and PENNANT from the CORAL-2 suite [24], plus the HPL benchmark. On the ESB, on the other hand, we use the CPU-only versions of Kripke, AMG and HPL, plus the GPU-accelerated versions of LAMMPS, Quicksilver and HPL itself. The datasets cover roughly 20 hours of operation each, which is the maximum allowed duration for user jobs on the prototype; for most applications we use one MPI process per node with as many OpenMP threads as physical CPU cores, with the exception of the GPU-accelerated LAMMPS and Quicksilver benchmarks, where we use one OpenMP thread and as many MPI processes as physical cores per node. Additional details about the applications' configuration can be found in Appendix C.

On top of running the chosen set of HPC applications, we also applied multiple settings for the secondary inlet water temperature of each rack's cooling unit throughout our experiments: in particular, we used 35C, 37.5C, 40C, 42.5C and 45C as settings, each applied for roughly 4 hours. This way, we are able to characterize the behavior of compute nodes under different cooling conditions. The experiments described so far (labeled as *Experiment* in Table 5.2) were engineered to produce diverse operating conditions in the compute nodes' CPUs and GPUs: while this was deemed sufficient for



(a) CM.

(b) ESB.

Figure 5.5.: Temperature histograms for CPUs on the CM, as well as CPUs and GPUs on the ESB. Continuous lines represent fitted PDFs.

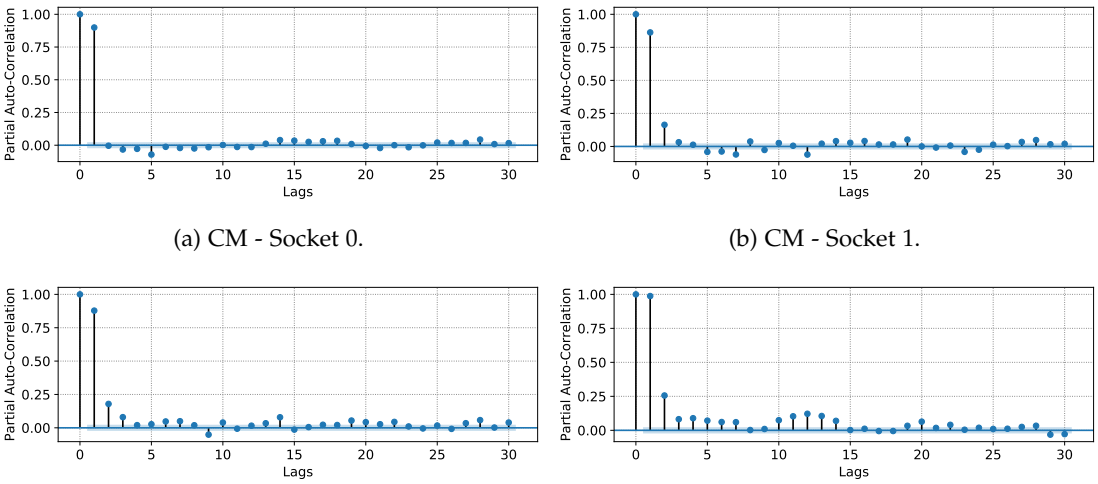
the CPU-only CM module, we chose to integrate additional data in the ESB datasets (labeled as *Production* in Table 5.2) in order to capture the full spectrum of GPU-accelerated applications commonly used on the DEEP-EST prototype. These supplementary datasets are associated with user jobs running parallel machine learning training workloads based on the *PyTorch* [32] library.

5.3.2. Data Exploration

We now discuss our exploratory analysis of the data collected on the DEEP-EST prototype, highlighting aspects that are descriptive of compute node behavior, and that will be useful for training of temperature prediction models.

Distribution of Temperature

Here we analyze the temperature distributions for the CPUs and GPUs of the CM and ESB. In Figure 5.5, in particular, we show the CPU temperature histograms for the dual-socket CM nodes, as well as those associated with the CPU and GPU available in ESB nodes - each histogram is computed by combining data from all compute nodes of the same module in the dataset. We also show the fitted PDFs derived from each histogram. The first clear observation is that CM and ESB compute nodes exhibit very different thermal properties: while the temperatures of CPUs and GPUs on the ESB peak at roughly 60C and 50C respectively, those of the CPUs on the CM reach 80C. Given that the cooling system being used is the same for both rack types, this difference is to be attributed to the fact that the 12-core CM CPUs have a *Thermal Design Point* (TDP) of 165W, as opposed to the 85W of the 8-core ESB CPUs, and hence produce more heat. The ESB GPUs, on the other hand, exhibit the best thermal results despite having the highest TDP rating, at 250W.



(a) CM - Socket 0.

(b) CM - Socket 1.

(c) ESB - Socket 0.

(d) ESB - GPU 0.

Figure 5.6.: Partial auto-correlation of temperature for CPUs on the CM *dp-cn02* node, as well as for the CPU and GPU on the ESB *dp-esb28* node.

It can also be seen that the two CPUs in CM nodes exhibit different thermal behaviors despite being identical: their two distributions have very similar shapes, but they appear to be shifted on the temperature axis. This is due to the fact that the cooling circuit inside CM compute nodes is not *parallel*, but rather *serial*, leading the second CPU in the circuit (identified as Socket 0 in Figure 5.5) to receive warmer water than the first, and hence operate at higher temperatures. In general, the diverse thermal behaviors of CM and ESB compute nodes provide a strong motivation for the implementation of a rack-level cooling control system, thus allowing to exploit the high thermal efficiency of certain architectures (e.g., GPUs on the ESB) and use inlet water at a higher temperature, leading to better energy efficiency.

Partial Auto-correlation of Temperature

The partial auto-correlation of a time series can be interpreted as its correlation with a version of itself shifted by a certain number of samples (or *lags*), and it is an effective way to understand its behavior and gain insight over the potential extent of prediction [Box+15]. For this reason, we now look at partial auto-correlation for the temperature time series of CPUs and GPUs on the CM and ESB: in Figure 5.6, in particular, we show the partial auto-correlation at different lags for the time series of each of the two CPUs in CM nodes, as well as for CPU and GPU temperature in ESB nodes. The reference time series are taken from one of the 16 nodes in the respective datasets.

In most cases, CPU temperature exhibits positive auto-correlation up to 3 or 4 lags, while the ESB GPU's temperature retains a positive auto-correlation up to 7 lags, which is likely due to its very efficient thermal performance. The only exception lies in one

of the CPUs in CM nodes (i.e., Socket 0), which shows a positive auto-correlation for a very short time, equivalent to 2 time lags: as described earlier, this is due to the fact that this CPU comes second in the compute node's cooling circuit, and hence receives water at varying temperatures depending on the behavior of the previous CPU. This variation, which does not depend on the CPU itself, can be interpreted as noise and reduces the CPU temperature's predictability. These results lead us to believe that prediction of temperature would be effective only up to a few time lags in the future, and not exceeding 5 or 6: longer prediction would lead to over-fitting of the chosen model on the training dataset, and to poor performance in production operation. Given the sampling interval of 10s we use on the DEEP-EST prototype, a prediction of 6 time lags would correspond to 60s in the future, which we deem sufficient for the purpose of predictive cooling control.

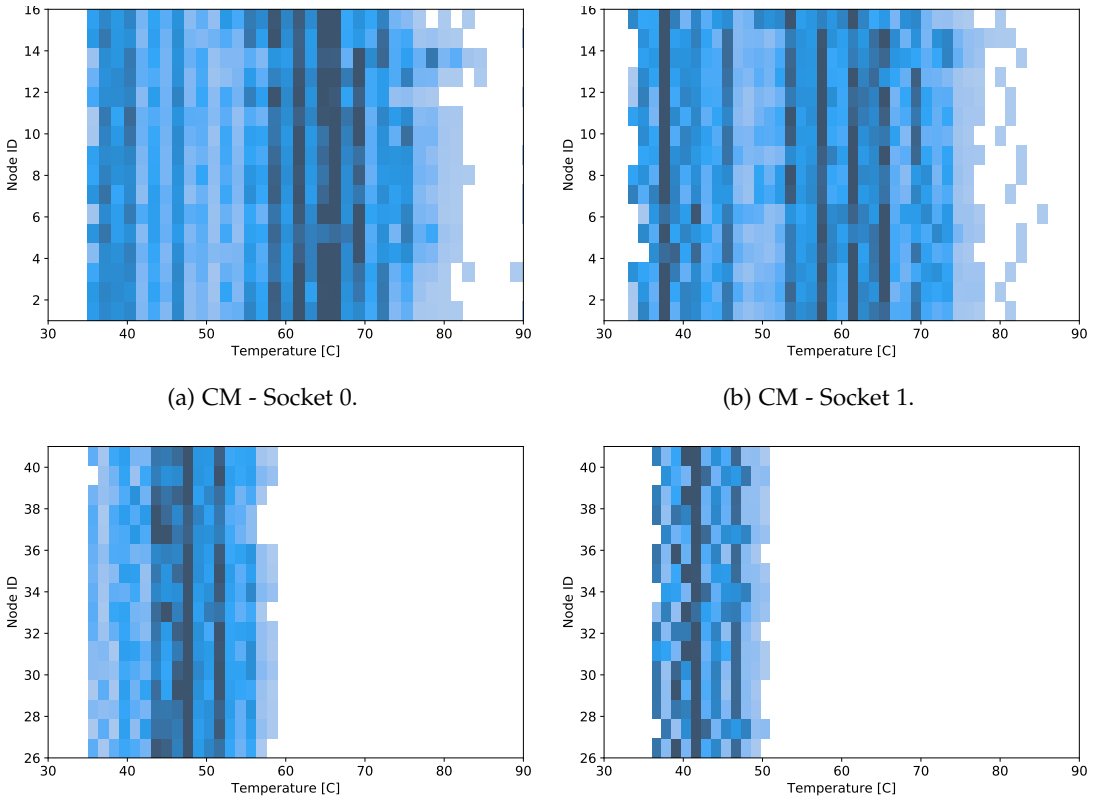
Variation of Temperature across Nodes

Here we briefly discuss the variation of temperature across the nodes used in our reference dataset. Specifically, in Figure 5.7 we show the histograms of the temperature time series for each node in the CM and ESB separately, for either the two CPUs on the former or the CPU and GPU on the latter. Darker colors indicate a higher number of occurrences. In general, results are compatible with what was observed with the combined distributions earlier; moreover, CPU and GPU temperatures follow very similar distributions across compute nodes, and appear to be mostly aligned.

A certain extent of variation is obviously present, and is to be attributed to ordinary component manufacturing variation [Ina+15]. In normal circumstances, this leads to energy consumption differences (usually below 10%) across chips, which has a direct impact on their thermal efficiency and operational temperatures. The most striking differences are related to the temperature maxima in CM CPUs, but these are not deemed relevant, as they represent a minimal part of the sample. The similarity in temperature distributions across nodes suggests that it will be possible to use a single model for temperature prediction across all CPUs or GPUs of a certain type, without the need to train separate models for each of the compute nodes. This will in turn greatly reduce the effort to maintain the overall cooling control pipeline.

Correlation between GPU Temperature and Energy Consumption

Throughout the acquisition process of our reference dataset, we observed a peculiar behavior in the ESB GPUs: a direct correlation between the card's temperature and the energy it consumes, for the same workload, is present. This effect is exemplified in Figure 5.8, in which we show a scatter plot of the GPUs' energy consumption across all ESB nodes over time, with colors representing the corresponding temperature, while running the same configuration of the GPU-accelerated HPL benchmark under 35C and 45C as inlet water temperatures. It can be seen that the two runs exhibit the same performance patterns, with three distinctive blocks corresponding to the three



(a) CM - Socket 0.

(b) CM - Socket 1.

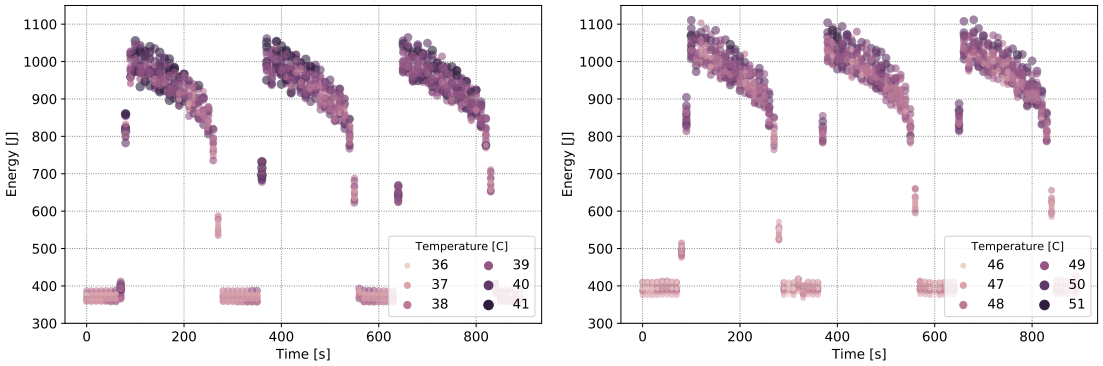
(c) ESB - Socket 0.

(d) ESB - GPU 0.

Figure 5.7.: Temperature histograms for all CPUs in CM nodes, as well as for CPUs and GPUs in ESB nodes. Darker colors represent higher values.

iterations of the benchmark; moreover, the duration of two runs differs by less than 1%. Temperatures between the two runs are shifted by 10C, according to the base inlet water temperature, but they show the same delta of 5C between idle operation and full load. However, running HPL using 45C as inlet water temperature results in a 5% higher energy consumption, with no differences in computational performance.

The effect described above is not unheard of and can be observed in CPUs as well, where it can be usually attributed to a chip's micro-architecture and internal power manager [DeV+14]. However, the manufacturing processes of most new-generation data center CPUs result in consistent energy consumption at different temperature settings: in fact, we were not able to observe any energy consumption variation in the CM and ESB CPUs when using different inlet temperatures. Nonetheless, it is important to keep in mind this behavior for the specific GPU type at hand on ESB nodes, and consider that an inlet water temperature increase might simultaneously result in less energy being consumed to operate the cooling system, and more energy being consumed to drive the GPUs themselves. In Figure 5.8 it can also be observed that peak energy



(a) 35C. (b) 45C.

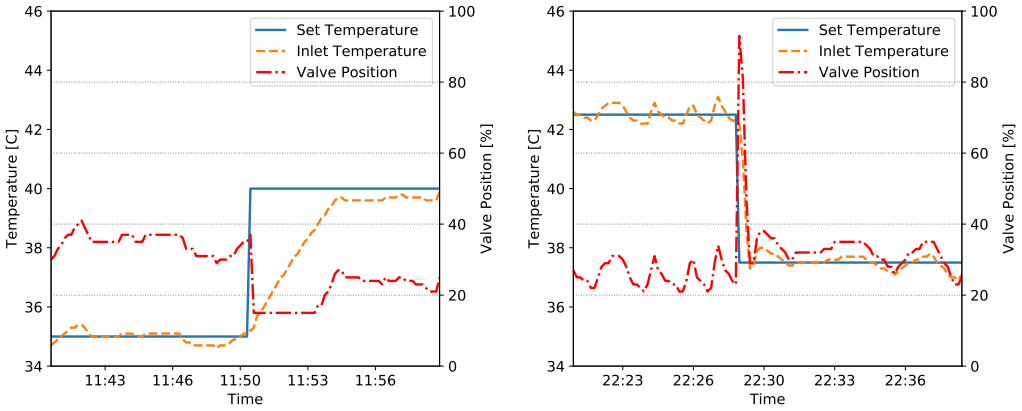
Figure 5.8.: GPU energy consumption and temperature over time for 16 ESB compute nodes running HPL, with two different inlet water temperature settings. The runs last roughly 850s, with less than 1% difference.

consumption in a 10s window amounts to 1,100J, which corresponds to roughly 110W, a value that is much lower than the GPUs' TDP rating of 250W. This is due to the fact that the public GPU-accelerated HPL version provided by Nvidia was designed for older architectures and, as of February 2021, Nvidia does not provide an updated version of HPL to the general public. However, we observed the same performance behavior described here when analyzing the production applications in our dataset (*Production* dataset in Table 5.2), which reach up to TDP power consumption.

Rack Cooling Unit Behavior Analysis

We conclude our data exploration by analyzing the behavior of the cooling units present in each of the CM and ESB racks, since our cooling control algorithm will be designed to operate at this level. The rack cooling units used in the DEEP-EST prototype assume the role of heat exchangers between compute nodes and the building infrastructure for cooling: they receive cold water from the latter (*inlet*), and they exhaust warm water coming from the former (*outlet*). Since the temperatures of inlet and outlet water depend on environmental conditions as well as system load, the cooling unit ensures that water coming from the building infrastructure (*primary inlet*) is brought to a consistent temperature before being fed to the compute nodes (*secondary inlet*), according to a *set temperature* value; this is obtained by mixing the inlet and outlet water flows via a series of valves, which open and close dynamically in order to reach a certain temperature. The cooling unit's set temperature will be the target knob for our cooling control algorithm: increasing this value results in the unit's primary inlet water valve to close down which, at scale, reduces the amount of cold water that needs to be produced by the building infrastructure and thus the energy being consumed.

In Figure 5.9 we provide an example of a cooling unit's behavior when either increasing



(a) Temperature Increase.

(b) Temperature Decrease.

Figure 5.9.: Behavior of the CM and ESB rack cooling unit's valve when applying a new inlet water temperature setting.

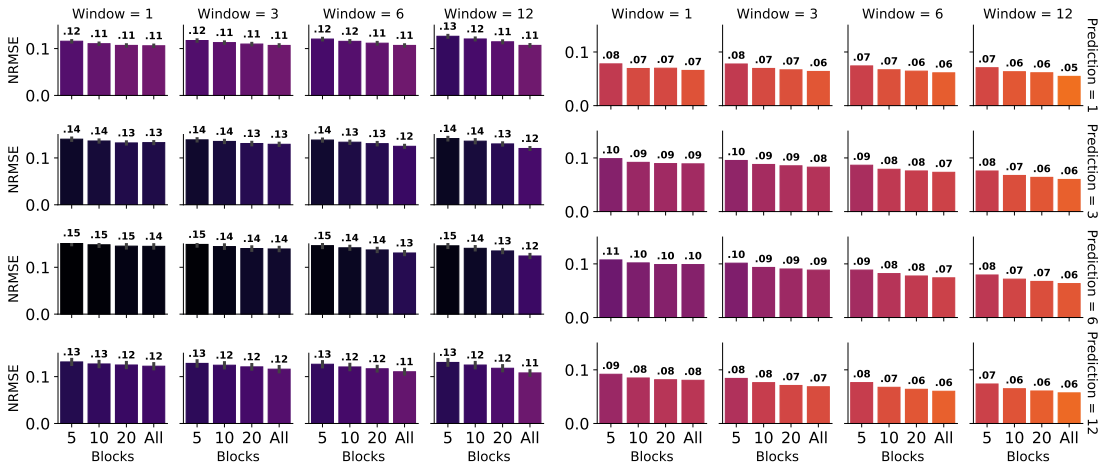
or decreasing the set temperature: upon increasing it, it can be seen that the primary inlet water valve closes, thus restricting the flow of cold water. Then, the secondary inlet water temperature increases gradually until the target value is reached, at which point the valve opens again to maintain equilibrium. The speed at which the secondary inlet water temperature increases depends on the amount of heat being produced by the compute nodes. When decreasing the set temperature, on the other hand, the cooling unit reacts much more quickly: here, the primary inlet water valve opens up almost completely, causing a steep and quick decrease in the secondary inlet water temperature. This behavior is ideal for the implementation of a proactive cooling control loop, as the system is guaranteed to react quickly when a decrease in the secondary inlet temperature is needed (e.g., when a CPU or GPU is overheating). In turn, this implies that prediction of CPU and GPU temperatures should only cover a few samples and does not need to extend far into the future.

5.3.3. CPU and GPU Temperature Prediction

Here we describe the machine learning models used in our pipeline for CPU and GPU temperature prediction on the CM and ESB modules of the DEEP-EST prototype, presenting our experimental methodology and insights.

Experimental Methodology

We employ a similar experimental methodology as in Section 4.4.1, processing the datasets described in Section 5.3.1 using the CS method introduced in Chapter 4 and training corresponding machine learning models for CPU or GPU temperature prediction under a variety of settings. In this case, the models are trained to predict the



(a) Per-CPU model.

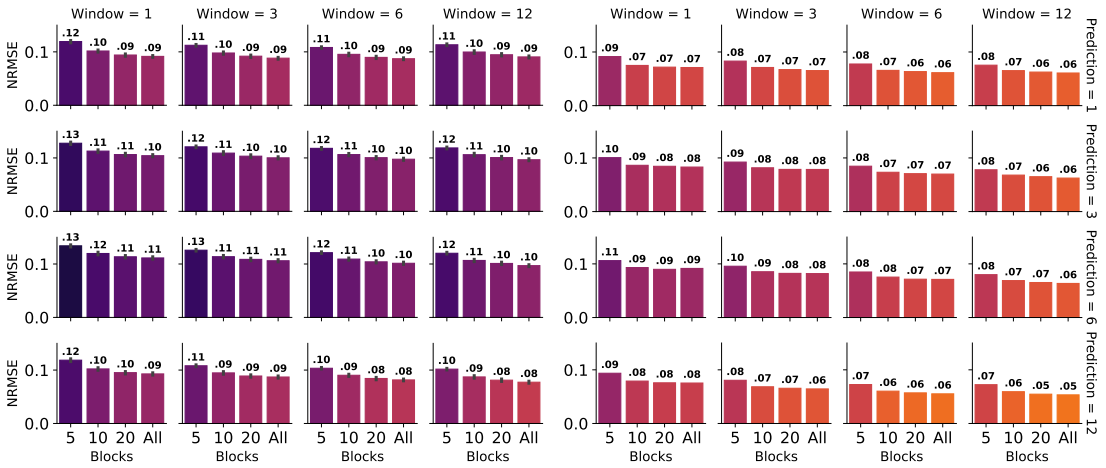
(b) Global CPU model.

Figure 5.10.: Bar plots showing the average temperature prediction NRMSE on CM CPUs with different aggregation, prediction and CS block values, as well as when using separate or global models.

maximum temperature in the upcoming time window: for the purpose of cooling control, this is more appropriate compared to predicting the average or a single sample, as it allows to estimate a temperature upper bound and hence prevent overheating of components more effectively. The Experiment and Production datasets in Table 5.2 are merged together in order to get a single dataset per target component of the DEEP-EST prototype: hence, we use one dataset for the CM CPUs, one for the ESB CPUs and one for the corresponding GPUs. As machine learning model we use once again a random forest (with 50 estimators and using the Gini impurity to evaluate the quality of splits), which is evaluated via 5-fold cross-validation under a stratified K-fold strategy. As error metric we use the NRMSE. Experiments in this section are carried out using the model implementations of the Python scikit-learn 0.20.3 library: as the Wintermute Regressor plugin uses the OpenCV library for its random forest model, we verified that results obtained with the former with a given model type can be replicated under the latter. The final CS and random forest models, trained offline in the Python environment, can then be saved to separate files and loaded by the corresponding Wintermute plugins for online production use.

Prediction Results

We now discuss the results of temperature prediction for CM CPUs, ESB CPUs and ESB GPUs in terms of average NRMSE. We conducted a series of experiments by varying the following parameters for model training:



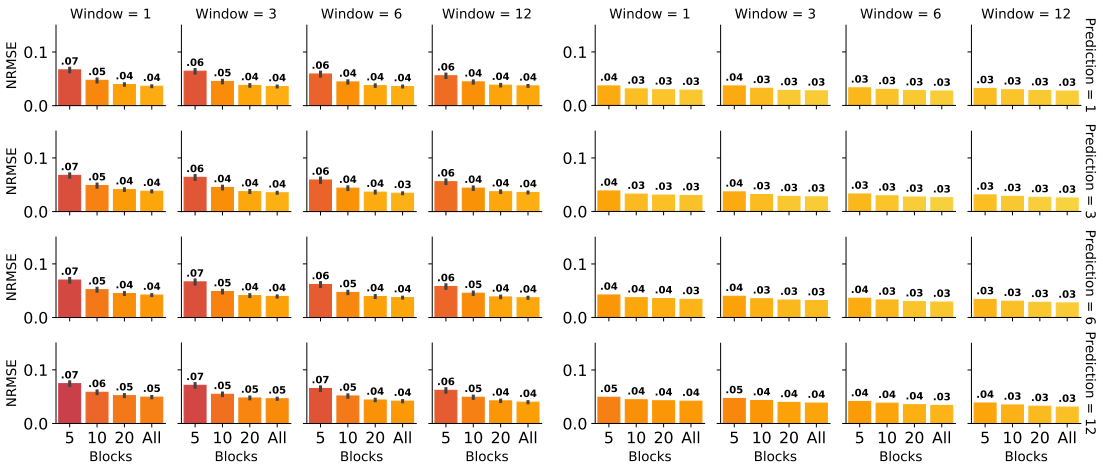
(a) Per-CPU model.

(b) Global CPU model.

Figure 5.11.: Bar plots showing the average temperature prediction NRMSE on ESB CPUs with different aggregation, prediction and CS block values, as well as when using separate or global models.

- **Model Type**: using distinct models for each CPU or GPU, or a single global model trained with data combined from all CPUs or GPUs of the same type.
- **Window**: the aggregation window (corresponding to the w^l parameter in Section 4.1) used to build CS signatures.
- **Prediction**: length of the time window (in samples) over which prediction of the maximum temperature is performed.
- **Blocks**: the amount of blocks used to build CS signatures.

The results of our study are summarized in Figures 5.10, 5.11 and 5.12 for the CM CPUs, ESB CPUs and ESB GPUs respectively. First, it can be seen that temperature prediction results are overall worse for CM CPUs, while ESB CPUs fare slightly better. Temperature prediction for ESB GPUs, instead, performs much better than either CPU types. The wide temperature range of CM CPUs is likely the reason behind this performance gap, since it complicates temperature prediction as opposed to the more predictable behavior of ESB CPUs and, in particular, GPUs. The presence of two sockets on CM nodes and the serial nature of the underlying cooling circuit is an additional complexity factor. In all cases, a counter-intuitive result is the fact that using a single global model for all components of a given type yields significantly better results than using distinct models for each CPU or GPU unit: one would expect, in fact, that tailoring models according to the performance characteristics of each separate chip would improve the accuracy of prediction. Our hypothesis is that the natural performance variation of



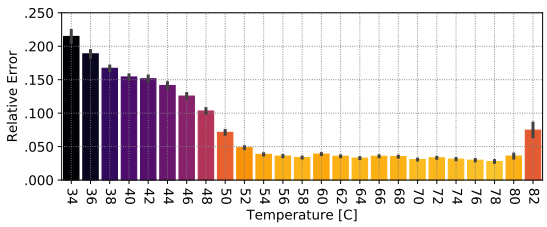
(a) Per-GPU model.

(b) Global GPU model.

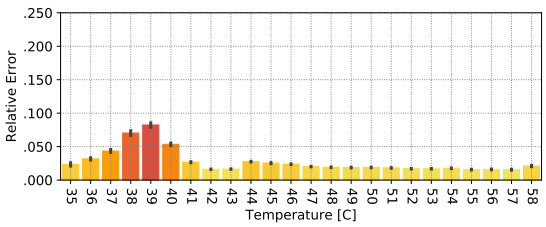
Figure 5.12.: Bar plots showing the average temperature prediction NRMSE on ESB GPUs with different aggregation, prediction and CS block values, as well as when using separate or global models.

CPUs and GPUs introduces a healthy amount of noise in the global training set, which in turn makes the regressions model's training process more accurate and robust against over-fitting issues.

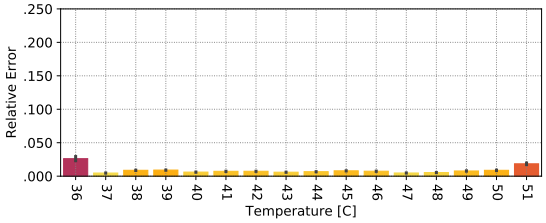
The impact of the remaining model parameters on prediction is as expected: increasing the extent of temperature prediction (*Prediction* parameter) leads to a slight NRMSE degradation, while using more historical data (*Window* parameter) to build the CS signatures leads to more robust feature sets and to an improvement of the average NRMSE. Moreover, in accordance to what was observed in Section 4.4 under a variety of regression problems, increasing the size of CS signatures (*Blocks* parameter) leads to better performance, at the expense of a more complex model. Switching from 5 to 10 blocks yields the largest gain in terms of NRMSE, whereas performance gains for larger block counts are more modest. Given the results obtained here, we decided to configure our final pipeline by using a global prediction model per component type, with an aggregation window of 6 samples (past minute), predicting the maximum temperature in the next 6 samples (upcoming minute) and employing 20 CS blocks: while results show that it would be possible to predict temperatures in the upcoming 12 samples with minimal performance degradation, the resulting model would over-fit on the training dataset and generalize poorly, due to the relatively short auto-correlation of the temperature time series we observed earlier. In Appendix 4.4 we discuss additional JS divergence results obtained with the DEEP-EST dataset.



(a) CM CPU.



(b) ESB CPU.



(c) ESB GPU.

Figure 5.13.: Bar plots of the average relative error values for different CPU or GPU temperature bands, for each of the three final trained models.

Relative Error Distribution

Here we discuss in detail the performance of the chosen temperature prediction model (i.e., a random forest predicting the maximum temperature in the next minute with 20 CS blocks, using data from the past minute) in terms of error distribution for the CM and ESB CPUs, as well as for the ESB GPUs. In Figure 5.13, for each of the three models, we show the average relative error in function of the respective temperature band. Confirming what was observed in the previous section, the CM CPU prediction model shows comparatively worse results with respect to the ESB CPU model and, in particular, the ESB GPU one. However, all of the three models exhibit similar overall behavior, with larger error at the edges of the temperature range, and a lower one for temperature bands in-between. The reason behind this behavior is twofold: first, as observed in Section 5.3.2, most of the temperature samples for all components are concentrated towards the middle of their range (e.g., between 40C and 70C for CM CPUs), which is thus over-represented in the model training set, leading to worse results for temperature samples at the edge.

The second reason for the observed error distributions is that our prediction model is naturally biased, since it is trained to predict the maximum temperature in the upcoming time frame rather than a single value or an average - for this reason, it will tend to over-estimate temperatures, especially with respect to very low ones. This, however, is not considered to be a problem for our cooling control algorithm: in this case we are not interested in the accuracy of the model itself, but rather in its ability to predict

whether a given component will cross certain temperature thresholds, in order to prevent overheating. Given the reasonably low error at medium and high temperature values, we are confident that our model will be able to predict dangerous thermal states in CPUs and GPUs effectively.

5.3.4. Cooling Control Strategy

In this section we describe the cooling control algorithm implemented for the DEEP-EST prototype. We first describe its parameters and overall flow, and then provide an overview of its configuration on the system.

Parameters of the Algorithm

Our algorithm is tasked with determining new set temperature values for warm-water rack cooling units based on predicted compute node temperatures, with the aim of keeping water temperature as high as possible and reduce the cooling infrastructure's energy consumption. The algorithm operates based on a series of configuration parameters, which are the following:

- T_{min}: Minimum temperature setting for a cooling unit r. Usually this is limited by the primary inlet temperature, as the unit does not provide any form of active cooling.
- T_{max}: Maximum temperature setting for a cooling unit r. This should be established based on safety concerns for the operation of components in the rack.
- T^i_{hot}: Temperature threshold for a component i belonging to rack r (e.g., a CM CPU). If such a component's temperature crosses this threshold, it is considered *hot* and hence in need of cooling.
- T^i_{crit}: Emergency threshold for a component i. If such a component's temperature crosses this threshold its operation is considered unsafe (i.e., prone to throttling or safety shutdown) and it must be cooled down urgently.
- P_{th}: Maximum fraction of components i in rack r which are allowed to be running hot at a given time.
- W: Length of the time window used to analyze the temperature of a component and establish its thermal state.

Flow of the Algorithm

In Figure 5.14 we summarize the flow of the entire cooling control pipeline, including the compute node-side CPU and GPU temperature prediction, while Listing 4 describes the logic used to determine the new set temperature within a rack. This is based on a previous work [Jia+19] and employs few simple steps: at each time-stamp N at which

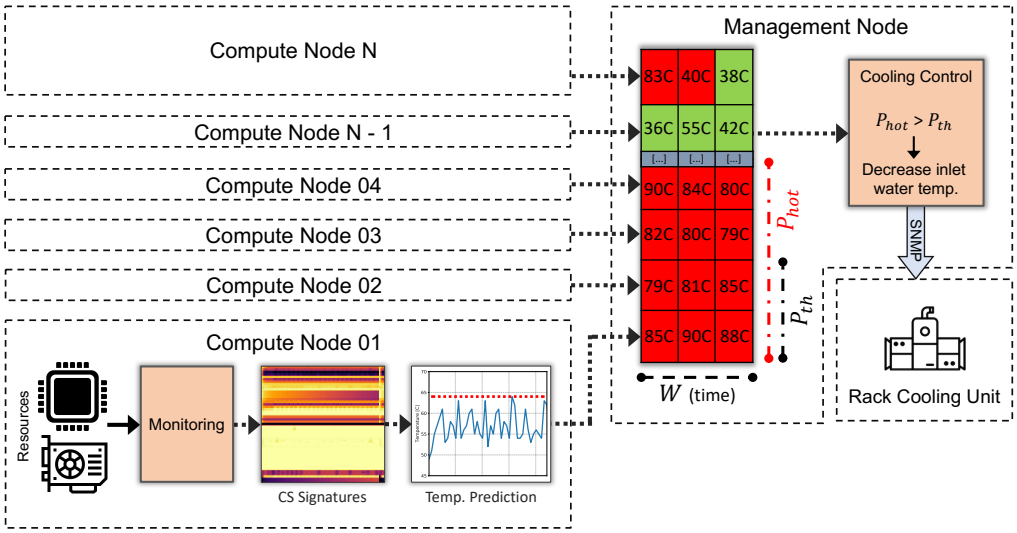


Figure 5.14.: A diagram representing the overall flow of the cooling control algorithm implemented for the DEEP-EST prototype.

the set temperature is to be updated, the algorithm fetches temperature readings for all components i in rack r, in the time window defined by $[N - W, N]$. If all returned readings for a given component i exceed its T^i_{hot} value, then it is considered as hot. At the same time, if at least one of the readings exceeds T^i_{crit}, the algorithm enters an emergency state and lowers the set temperature to the minimum T_{min} value in order to prevent overheating, returning thereafter. If this emergency condition does not occur, the algorithm proceeds by computing the fraction P_{hot} of components in r that are running hot. This value is compared to the threshold P_{th} and is used to compute an offset value for the current set temperature T: if P_{hot} is higher than P_{th} the new set temperature will be lower than the previous one, lowering in turn P_{hot}; conversely, it will be higher if P_{hot} does not reach P_{th}.

The updated set temperature T is then transmitted via the SNMP protocol to the rack cooling unit, where it is applied: this is done according to two possible strategies, named *Continuous* and *Stepped*. The Continuous strategy transmits the new T value to the cooling unit any time the control pipeline is invoked; the Stepped strategy, on the other hand, divides the $[T_{min}, T_{max}]$ range in a series of bins (whose number is configurable) and transmits the updated value of T only if it crosses the boundary between two bins. This measure was implemented for cases in which fine-grained tuning of the set temperature is not desirable, as it leads to continuous activity of the water valves inside the rack cooling unit and to a potential decrease of their life span. In general, the algorithm described here tries to keep the rack's set temperature as high as possible, as long as P_{hot} is lower than P_{th}: in practice, the set temperature will oscillate around a value for which P_{hot} and P_{th} are very close to each other, ensuring both operational safety and higher energy efficiency for cooling. The algorithm is not effective

Listing 4 Flow of the set temperature control algorithm.

Input: current set temperature T for rack r, a time-stamp N
Output: updated set temperature T for rack r

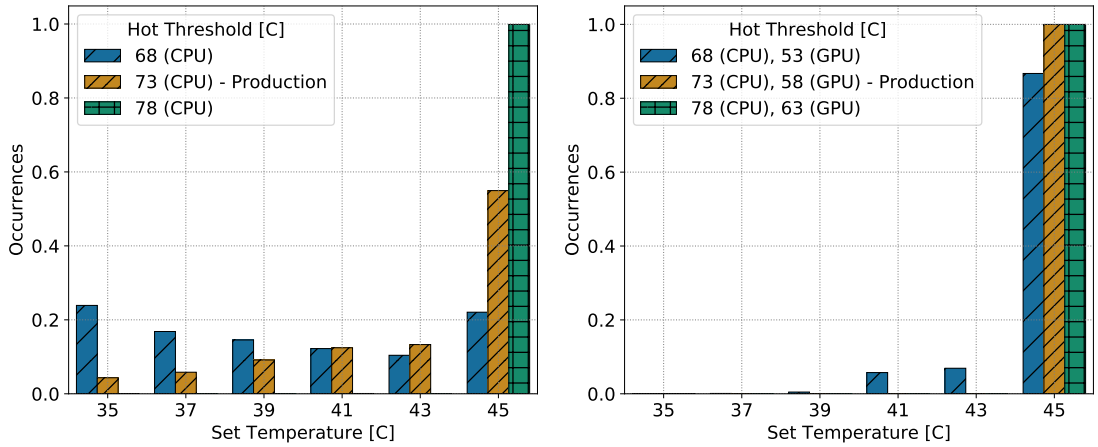
float $P_{hot} \leftarrow 0$
vector of int $buffer \leftarrow \varnothing$
for $i \in r$ **do**
  $buffer \leftarrow$ query i's temperature in time range $[N - W, N]$
  if not $\forall\ v \in buffer < T^i_{crit}$ **then**
    return T_{min}
  else if $\forall\ v \in buffer \geq T^i_{hot}$ **then**
    $P_{hot} \leftarrow P_{hot} + 1$
  end if
end for
$P_{hot} \leftarrow P_{hot}/|r|$
$T \leftarrow T + (T_{max} - T_{min}) \cdot (P_{th} - P_{hot})$
return T

against compute-intensive workloads, which are very likely to push the temperature of components to their limit, but it can exploit the presence of memory-bound and network-bound workloads, which tend to produce less heat. Similarly, the algorithm can take advantage of idle nodes, for which cooling is not critical.

Configuration for the DEEP-EST Prototype

As discussed in Section 5.2.4, there are two instances of the Wintermute Cooling Control plugin running in the management nodes of the CM and ESB DEEP-EST modules: the first uses a single operator to set the inlet temperature of the CM rack, with each component i being one of the two CPUs in each compute node. The second instance uses instead three operators, each associated to one of the ESB racks. Here, each component i is either a CPU or a GPU in an ESB compute node. For both the CM and ESB, the Cooling Control operators leverage predicted CPU or GPU temperatures supplied by the Wintermute Regressor plugin running in compute nodes; while they could also leverage current temperature readings instead of predicted ones, using the latter allows to have a *proactive* control approach instead of a *reactive* one.

In all operators we employ a T_{min} of 35C and a T_{max} of 45C, both of which are safe for the operation of warm-water cooled systems [Con+15], using the Stepped control strategy with 6 bins. Then, we use a P_{th} of 0.2 and a W of 0, meaning that we only pick the most recent temperature prediction to establish the status of a CPU or GPU. The main difference in the configuration of the operators lies in the T^i_{crit} and T^i_{hot} parameters: for the CM and ESB CPUs, these are respectively equal to 93C and 73C. The first value sits 5C below the temperature at which CPU thermal throttling via frequency reduction first occurs [33], while the second value is 20C further below it. In a similar way, ESB GPUs employ 78C and 58C respectively for the T^i_{crit} and T^i_{hot} parameters, with the first value being 5C below the temperature at which the frequency of Nvidia GPUs is throttled by 50% [34]. These settings ensure that no performance degradation occurs



(a) CM. (b) ESB.

Figure 5.15.: Histograms of set temperature values for the CM and ESB cooling units when using our control algorithm with different T^i_{hot} values [Net+21c].

in any scenario, while providing enough headroom to experiment with inlet water temperatures higher than the 45C used here. The same T^i_{crit} settings described here for the Cooling Control plugin are also used to trigger its Health Checker counterpart.

5.3.5. Operational Results

Here we finally discuss the operational results obtained with our cooling control pipeline on the DEEP-EST prototype. The control pipeline is configured as discussed earlier and we experiment with different T^i_{hot} values: on top of using the default settings (i.e., 73C for CM and ESB CPUs, 58C for ESB GPUs) we also perform experiments with settings 5C above or below them. Each experiment was performed by observing the behavior of the CM or ESB racks for a period of 20h, under certain T^i_{hot} values, combining production user jobs and the applications described in Section 5.3.1 for a realistic workload. We target only one of the three ESB racks, as we assume the other two will exhibit identical behavior under normal operation.

General Overview

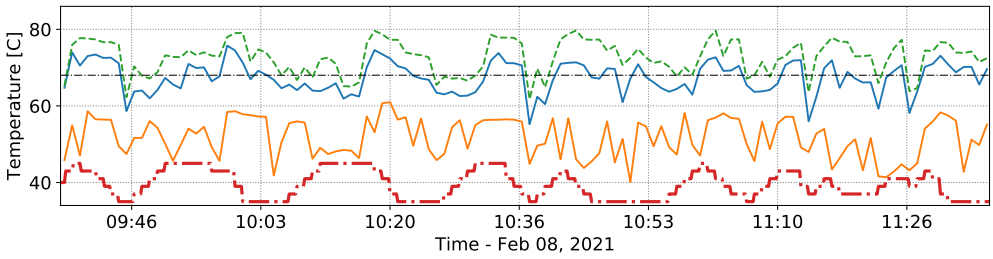
We start with a general overview of the results. In Figure 5.15 we show the set temperature histograms throughout our experiments for both the CM and ESB modules; due to the high amount of heat that can be produced by the associated CPUs, the cooling control algorithm's behavior changes significantly when using different T^i_{hot} values on the CM: in the 78C case, due to the CPUs rarely reaching this temperature in production operation, the algorithm never chooses set temperatures below 45C. Choosing 73C as T^i_{hot} yields slightly more dynamic results, with the algorithm choosing 45C as set tem-

perature most of the time, but occasionally descending to lower values when demanding workloads are being executed; this particular setting seems to represent the sweet spot for CM CPUs, and it was therefore chosen for our production configuration. Finally, using 68C as T^i_{hot} results in the cooling control algorithm being very sensible to CPU behavior, and all possible set temperature values are used uniformly. On the other hand, results obtained on the ESB (in this case, rack 2 of the module) reflect the high thermal efficiency of its CPUs and GPUs: both when using 78C or 73C as T^i_{hot} for the CPUs, and 63C or 58C for the GPUs, the cooling control algorithm rarely chooses set temperature values below 45C. Only when using extremely low T^i_{hot} thresholds (i.e., 68C for the CPUs and 53C for the GPUs) the algorithm is observed to descend to lower values. This suggests that it would be possible to run the ESB at much higher T_{max} values, such as 50C or 55C. Moreover, no component on the CM or ESB ever reached its T^i_{crit} threshold in our experiments. Due to its high sensitivity to cooling control parameters in terms of thermal behavior, in the following we focus on the CM module to conduct an in-depth analysis of the pipeline's impact on system operation.

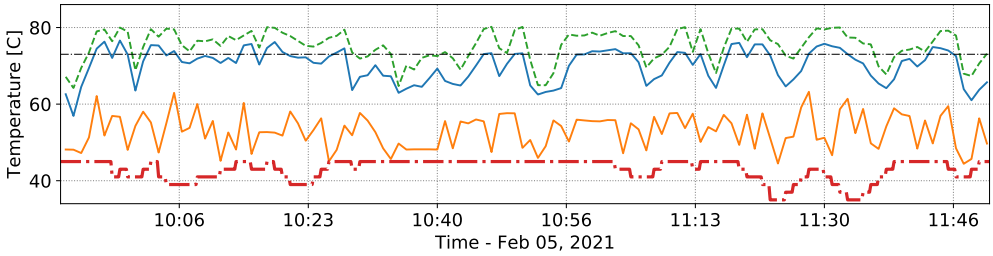
Cluster Module Analysis - CPU Temperatures

After providing a general overview of the results, we now focus on the CM due to its more dynamic thermal behavior as opposed to the ESB. Figure 5.16 shows the interaction between the cooling control pipeline and the underlying compute node hardware - in particular, we show the cooling unit's set temperature, as applied by the control algorithm, as well as the minimum, maximum and 8th decile of CPU temperatures in the rack, as predicted by our machine learning model. The 8th decile of predicted CPU temperatures is directly connected to the P_{th} parameter, which is set to 0.2: if the former is higher than T^i_{hot}, it implies that at least P_{th} CPUs are expected to run in a hot state, and hence the set temperature must be lowered. The opposite applies when the 8th decile is lower than T^i_{hot}. We present several time-series snapshots, corresponding to experiments using 68C, 73C or 78C as T^i_{hot} for the CM CPUs.

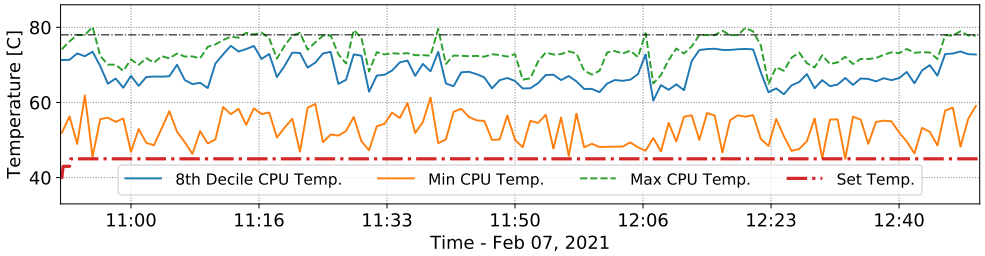
It can be seen that all metrics interact in a very organic way: peaks in the 8th decile and maximum of predicted CPU temperatures are accompanied by decreases in set temperature, which in turn leads to a gradual CPU temperature decrease; the higher is the CPU temperature peak with respect to T^i_{hot}, the quicker is the decrease in set temperature. Since 68C is a relatively low T^i_{hot} threshold, set temperature is unstable and tends to oscillate between 35C and 45C. Using 73C as T^i_{hot} results in a much more stable behavior, with set temperature decreasing only when very high CPU temperatures are reached. Finally, using 78C as T^i_{hot} leads to set temperature being fixed at 45C, suggesting that this setting is too high for our system. In all experiments, however, the difference between the 8th decile and maximum of predicted CPU temperatures remains roughly constant over time and amounts to less than 10C, which is within the boundaries of normal manufacturing variation [Ina+15]. Moreover, maximum CPU temperatures never reach the T^i_{crit} threshold of 98C, which makes the idea of using T_{max} temperatures above 45C promising. We observed these behaviors in the original CPU temperature data as



(a) 68C.



(b) 73C - Production.



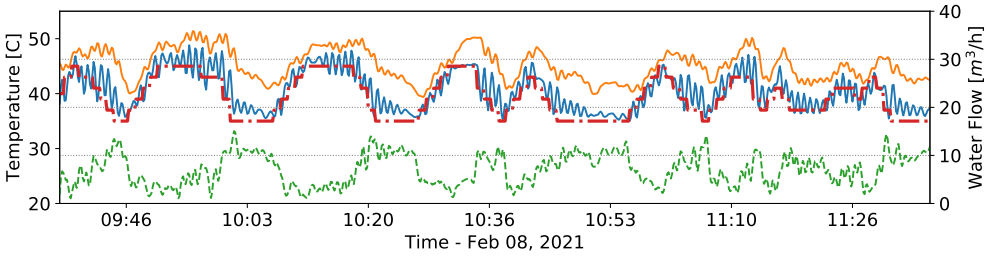
(c) 78C.

Figure 5.16.: Interaction between the CM's set and CPU temperatures under different T^i_{hot} values, which are indicated by dashed horizontal lines [Net+21c].

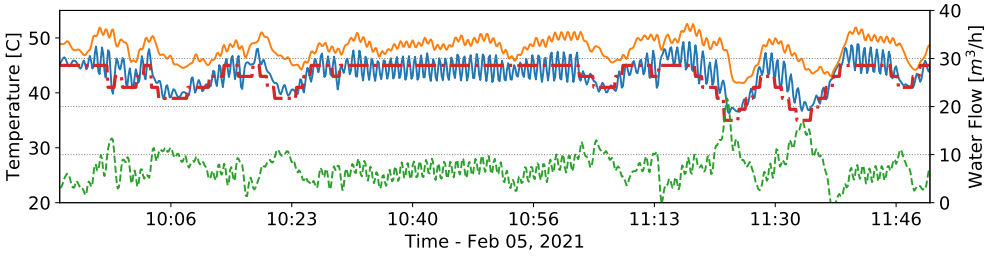
well, confirming that the predictions produced by our model represent it accurately. An exception to this is the approximation of CPU idle states: it can be seen in Figure 5.16 that the time series of minimum CPU temperatures is noisy, with values that are much higher than the base set temperature. This artifact is to be expected from our model's architecture, as it was trained to predict the maximum potential CPU temperature in the upcoming minute, confirming what was already observed in Section 5.3.3.

Cluster Module Analysis - Cooling Unit

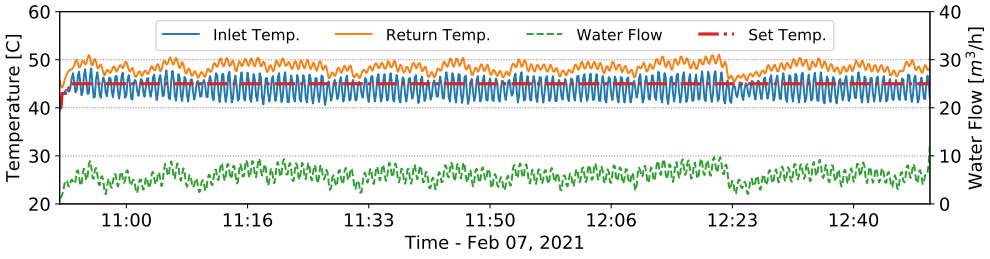
In Figure 5.17 we show the impact of the cooling control pipeline on the rack cooling unit itself, under the same experiments and in the same time frames as in Figure 5.16. In particular, we show the rack cooling unit's set temperature, as well as the secondary



(a) 68C.



(b) 73C - Production.



(c) 78C.

Figure 5.17.: Impact of the CM cooling unit's set temperature on primary inlet water flow, as well as secondary inlet and return temperatures under different T^i_{hot} values when using our control algorithm [Net+21c].

inlet and return temperatures and the flow rate of primary inlet water (in cubic meters per hour) - this quantifies the amount of cold water consumed per unit of time.

Once again, it can be seen that set temperature has a direct impact on all other cooling metrics: decreasing it leads to a proportional decrease in secondary inlet and return water temperatures, as well as to a temporary increase in water flow. The opposite applies whenever the set temperature is increased. The impact of using different T^i_{hot} values is also similar to what was observed when analyzing CPU temperatures: at 68C as T^i_{hot} threshold both the inlet and return temperatures oscillate considerably, with differences of up to 10C in relatively short time spans. Using 73C or 78C as threshold leads to a much smaller range of variation over time, in most cases lower than 5C. In general, secondary inlet temperatures are not stable, and oscillate around

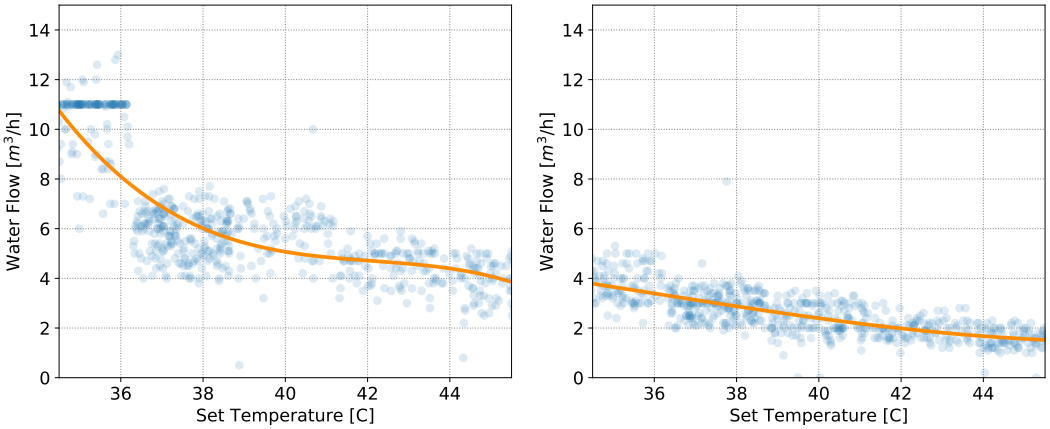
the set temperature while the rack cooling unit attempts to reach equilibrium. Still, the temperature difference between secondary inlet and return water lies between 5C and 10C, which is expected from this type of system [Con+15; Wil+17]. Water flow is also a relevant metric for production operation: it can be seen that steep changes in set temperature translate to drastic variations in this metric, with differences of up to 10 cubic meters per hour when switching from 35C to 45C in a short time. Adopting a T^i_{hot} setting (e.g., 68C) that leads to unstable set temperatures is thus undesirable, as estimating the demand for cold water under normal operation would be difficult.

5.3.6. Generalization of Results

Here we extrapolate general conclusions from the results obtained previously. As a first step, we state that our results can be generalized to T_{max} settings that are different from the 45C used on the DEEP-EST prototype: if we assume that the inlet water's heat transfer properties remain the same at different temperatures, applying our control algorithm with a certain T_{max} and T^i_{hot} is equivalent to using it with arbitrary $T_{max} + x$ and $T^i_{hot} + x$ settings. In our case, for example, we expect to obtain the same results observed with a T_{max} of 45C and a T^i_{hot} of 73C on the CM module, if we increase both settings to 50C and 78C respectively. This is useful in cases where evaluating the cooling system's behavior at very high inlet temperatures might be dangerous, for example because of excessive water pressure building up within pipes.

In general, the choice of appropriate T^i_{hot} settings is based on operational constraints and desired system behavior: in some cases, for example, the wide return temperature swings associated with very low T^i_{hot} values may be undesirable (e.g., as observed in Figure 5.16 when using 68C as threshold on the CM). Following a similar rationale, some data centers may prefer to keep return temperatures as high as possible at all times in order to drive adsorption chilling devices [Wil+17]. Finally, sites residing in temperate areas may be able to rely on *free cooling* (i.e., cooling that relies on cold atmospheric conditions and that does not use any active devices), making an energy efficiency-oriented cooling strategy less attractive: for these same reasons, extrapolating general conclusions about the energy consumption of a data center's cooling infrastructure is difficult, as its operation greatly depends on environmental, seasonal and operational factors which are hard to decouple from one another.

Despite the concerns expressed above, we can still estimate the amount of cold water saved thanks to our approach: in Figure 5.18 we show the fitted inlet water flow rate in function of the set temperature for the CM rack and one ESB rack. The original data refers to the dataset discussed in Section 5.3.1; a small amount of noise was added to set temperature values to ease the fitting process. It can be seen that, in both the CM and ESB racks, there is a 50% (or more) flow reduction when switching from 35C to 45C as set temperature, with the CM exhibiting higher flow rates than the ESB rack due to the greater quantity of heat it produces. Based on the histograms presented in Figure 5.15, the CM and ESB racks show respectively 42.8C and 45C as average set temperatures when using our cooling control pipeline with the associated production configuration,



(a) CM.

(b) ESB.

Figure 5.18.: Fitted flow rate, associated with primary inlet water, in function of set temperature values for the CM and ESB racks of the DEEP-EST prototype. Scatter plots depict the original data used for fitting.

leading to 50% less cold water being used compared to the 35C setting. As these are per-rack results, they can be scaled to an entire HPC system or data center, and it is foreseeable that a 50% reduction in cold water produced at the infrastructure level will lead to significant energy savings.

5.4. Applicability to Other Use Cases

In this chapter we discussed two different production deployments for our end-to-end ODA platform, covering both ODAV (job data visualization on SuperMUC-NG) and ODAC (cooling control on the DEEP-EST prototype). While the deployments refer to specific HPC systems and operational requirements, their designs can easily accommodate a much wider variety of use cases. For example, converting our SuperMUC-NG ODAV deployment to an ODAC one (e.g., for performance data-based CPU frequency tuning) would require minimal changes to the ODA infrastructure, involving only the set of DCDB and Wintermute plugins to be deployed to implement the desired functionality. This was one of the main design drivers behind DCDB, Wintermute and the CS method: enabling developers to deploy generic ODA infrastructures with ease and focus on the use cases of their own data centers.

Own publication acknowledgement. The DCDB and Wintermute deployments on the SuperMUC-NG and DEEP-EST systems (including the extension of the HPC-ODA dataset collection [28]) were described in [Net+21c], using a more compact format compared to this dissertation for space reasons.

6. The Deployment Process

In light of our experiences in Chapter 5, here we discuss the process that is behind bringing a monitoring and ODA infrastructure into production from a software engineering perspective. We start in Section 6.1, where we propose a formal description of this process. After this, we discuss the complexity factors that we identified during our experiences, from both a technical standpoint (Section 6.2) and a human one (Section 6.3), and propose action items to mitigate them in Section 6.4.

6.1. Software Engineering Perspective

The process of deploying a monitoring and ODA infrastructure in a production data center environment can be framed within a software engineering perspective and can be divided in steps. Among available models for software engineering processes, we found simple ones such as *Waterfall* [PWB09] or *Agile* [Bec+01] to be the most suitable for monitoring and ODA: here, the software development cycle is iterative and divided in steps, corresponding to the main phases of design, development, testing and deployment. With these two reference models in mind, we formulated the ODA process as a series of iterative steps which are summarized in Figure 6.1. In this context, however, we focus on the peculiar characteristics of the HPC and data center fields and the associated constraints that drive the development process as a whole.

It should be noted that the development reality of most supercomputing centers is inherently agile, with small teams performing development and research activities according to operational requirements. For this reason, more sophisticated software engineering models, such as *Scrum* [SB02], cannot easily be adapted to data center operations, as they are designed for coordinating large software teams working on very complex products. The steps to a successful production monitoring and ODA infrastructure, according to our experience, can be formulated as follows:

1. **Requirements**: planning of the expected monitoring and ODA functionality is performed. This consists in defining the sensors of interest from a high-level perspective (e.g., energy consumption of compute nodes), as well as the scope of ODA functionality (e.g., CPU frequency tuning with certain energy saving targets). If performed at the time of a system's procurement, the hardware resources associated to monitoring and ODA can be defined in this phase.

2. **Specification**: here, the high-level requirements are translated into actual monitoring data sources and ODA models. Therefore, this phase consists in the investiga-

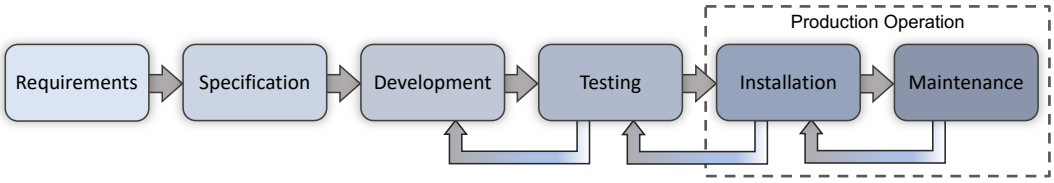


Figure 6.1.: The software engineering cycle of monitoring and ODA for data centers.

tion of available data sources on the system (e.g., in-band SysFS or out-of-band IPMI sensors for compute node energy consumption), as well as the most suitable model types for the target ODA functionality. After this initial investigation, the placement of software components over the system can be defined.

3. **Development**: given the requirements, any missing functionality in the chosen monitoring and ODA framework is developed and integrated into it. Usually, this consists in the development of additional plugins to support data sources using certain protocols or models of a certain type, but can also include the development of secondary integration tools. For example, a tool may be required to expose the resource manager's user job information to the monitoring infrastructure. At the end of this stage, configuration of all software components is performed according to their planned functionality and placement.

4. **Testing**: the entire monitoring and ODA infrastructure is tested on the target system, usually in the context of a maintenance window and not during production operation. Beyond functional correctness and scalability, other parameters of interest at this stage are, for example, overhead against typical workloads on the system or usage of resources (e.g., CPU or network).

5. **Installation**: the monitoring and ODA components, as well as the associated configurations, are packaged into a form suitable for installation in the target environment (e.g., *RPM* packages). These are then delivered to the appropriate machines using automated tools. This phase is usually not instantaneous and is carried out over the course of several days, especially if performed during production operation.

6. **Maintenance**: long-term correct operation of the monitoring and ODA infrastructure is ensured by making use of alerting and ticketing systems. Further development and configuration activities may be required to accommodate new changes to the system throughout its life time.

6.2. Technical Complexity Factors

We now focus on the technical factors that contribute to the complexity of the software engineering process described above. Some of these have been discussed in recent

literature [Bra+16; Ahl+18] - here we focus on our experiences (described in Chapter 5) carrying out production monitoring and ODA deployments. We also supply general guidelines to mitigate the problems we encountered.

6.2.1. Software Integration

In an ideal production environment, a holistic monitoring framework such as DCDB would be the only provider of sensor data, with most ODA functionality being condensed within a framework such as Wintermute. However, this is rarely possible in practice, especially if monitoring and ODA are added to a system during the course of its life: in these cases, it is very likely that other management frameworks that are essential for the system's operation are already accessing the data sources that are to be monitored or, additionally, are already tuning the knobs that are to be affected by ODA. These software components usually cannot be altered in function of monitoring, requiring instead non-trivial orchestration and coordination mechanisms to ensure safe concurrent access to knobs or data sources.

Our experience in this aspect stems from the production deployment of DCDB and Wintermute on SuperMUC-NG described in Section 5.1, which was added after the system had entered production operation: here, a CPU performance counter access conflict arose between DCDB, for monitoring, and the EAR framework, for online tuning of CPU frequencies. In particular, the former uses the Linux Perfevents interface [Wea13] directly to sample the counters on a system-wide basis, whereas the latter uses the *Performance Application Programming Interface* (PAPI) [Muc+99] to track them on a per-process basis. Like DCDB, also PAPI uses Perfevents as a backend to access the performance counter information. Even though concurrent sampling of performance counters is theoretically supported, this usage of Perfevents silently resulted in both frameworks not being able to sample any performance counters, without raising any error; the source of the problem was eventually identified in a bug associated with the scheduling policies for CPU performance counters in the Linux kernel. It is expected that this behavior, which was present in the SLES 12 operating system used on SuperMUC-NG compute nodes as of September 2020 (when the issue was originally observed), will be addressed with newer releases of the Linux kernel. In general, establishing the role of monitoring at the procurement stage can help mitigate this issue, by clearly defining the responsibilities of all software components involved and thus preventing conflicts associated with usage of resources.

6.2.2. Hardware Limitations

Monitoring and ODA require a significant amount of hardware resources: this usually translates to a set of machines that are able to handle the large influx of data coming from an entire system, as well as to a scalable storage solution to support a distributed database. These requirements become more demanding for large-scale machines - for example, in our SuperMUC-NG deployment, described in Section 5.1, we use eight

different machines to handle the sensor data coming from the DCDB Pushers (via Collect Agents), in addition to two servers that host a distributed Cassandra instance. Even in the case of the DEEP-EST deployment described in Section 5.2, we use three different management machines in order to accommodate the peculiar configuration of this small-scale cluster. ODA configurations that rely on compute-intensive models will further require dedicated computational resources in order to operate effectively. Beyond the availability of computational resources, storage is the main factor that must be kept in mind for a successful deployment: insufficient storage space or bandwidth can severely cripple the effectiveness of monitoring, forcing a very short time-to-live for the sensor data, long sampling intervals, or simply resulting in a lower amount of available sensors and data sources.

As described in Section 5.1, in our SuperMUC-NG deployment we disable the transmission and storage of raw sensors (sampled every 10s), keeping only the derived performance metrics (sampled every 120s). If we wanted to store the former alongside the latter, the Cassandra database would have to manage an additional 680,000 inserts per second, resulting from the 6.8 million raw sensors available in compute nodes. While Cassandra is able to deal with such insert rates [Net+19b], further reducing the sampling interval to 5s or 1s would result in several millions of inserts per second, which is not manageable by our configuration. Correspondingly, more than 1TB of data would be produced for each day of operation, even with a sampling interval of 10s [Rao+20]: storage for long time spans and at a fine granularity is hence a problem that cannot be solved at the server level, but requires a Petabyte-grade distributed file system. Similarly to what we discussed regarding software integration, defining monitoring at the procurement level can be essential to solve these issues. There, adequate hardware resources can be allocated for the monitoring and ODA infrastructure, eliminating the need to rely on shared management machines and limited storage capabilities.

6.2.3. Intrinsic Complexity

From many perspectives, monitoring and ODA are intrinsically complex tasks. As discussed in Section 5.1, a large-scale HPC system may supply up to millions of distinct sensors that can be sampled - ensuring that all of them are functioning properly and produce sensible readings is a daunting task that has already been an object of discussion in the literature [Gim+17]. Furthermore, ensuring consistency of data requires specific domain knowledge about sensors and the meaning of the data sampled by them. CPU performance counters, in particular, are heavily affected by the features of the hardware being monitored as well as by the applications running on it, requiring deep understanding of both in order to interpret them. Any issues encountered at this level carry on to ODA models, impairing their functionality. From our experience, we were able to identify three levels of consistency for monitoring data:

- **Numerical**: a sensor should produce readings that are sensible from a numerical standpoint. Problems of this kind may be related, for example, to a mismatch

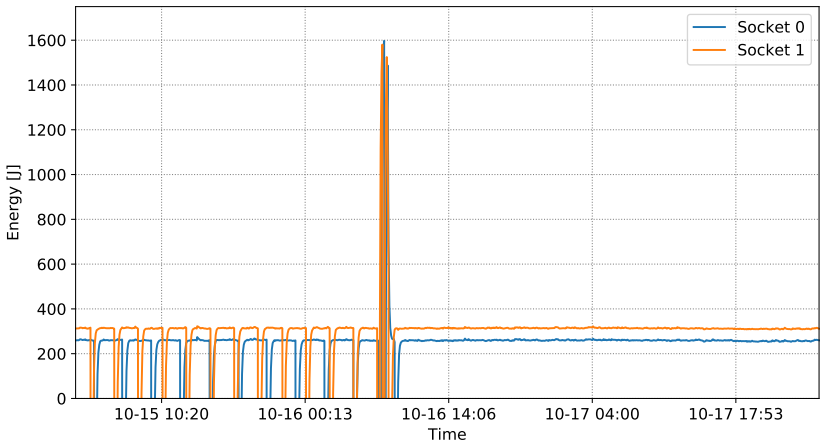


Figure 6.2.: The effect of incorrect upper bounds on the values of RAPL energy counters, for the two CPUs of a CM compute node in the DEEP-EST prototype.

between numerical types during the sampling process or to a misconfiguration of the sensor itself.

- **Spatial**: the same sensor should produce consistent readings across components, for example compute nodes. Issues of this kind may be associated with differences among components (e.g., OS, firmware or hardware variation), as well as with other administration aspects (e.g., permission management).

- **Temporal**: a sensor should produce readings that are consistent over time. Issues of this type may arise upon sudden changes to the affected components (e.g., OS updates) as well as upon faults and other anomalous behaviors.

We encountered several consistency issues throughout our production experiences, in particular in the scope of the DEEP-EST deployment discussed in Section 5.2: for example, we discovered early in the process that most of the CPU energy counters in compute nodes exposed by the Intel *Running Average Power Limit* (RAPL) interface occasionally produced unrealistic readings (i.e., numerical inconsistency), with negative or very large values. Eventually, the source of the problem was found in the upper bounds of the monotonic energy counters, which did not correspond to the width of the underlying register, but rather complied to a custom value reported in an undocumented SysFS file alongside the energy counter itself. Figure 6.2 shows the energy consumption of the two CPUs in a DEEP-EST CM compute node between October 15th and 17th 2020: as it can be seen on the left side of the plot, the untreated overflows in the energy counters produced peculiar comb-like patterns in the data.

On a similar note, soon after deploying DCDB on the system we discovered that some compute nodes were occasionally not able to sample the CPU's temperature (i.e., spatial inconsistency): the culprit was the path of the corresponding SysFS sensor, which was

not fixed, but instead changed randomly at each machine reboot, depending on the loading order of kernel modules. This is a known issue related to the *HWmon* [35] kernel module. Finally, after several weeks of normal operation, we discovered that compute nodes in the system were progressively silently stopping to sample CPU performance counters (i.e., temporal inconsistency): some users were making use of certain PAPI-based profiling frameworks in their jobs, which interfered with DCDB and resulted in the same issue we observed in our SuperMUC-NG deployment, as described previously in Section 6.2.1.

The issues described above are intrinsic to monitoring and hence are very difficult to eliminate completely. Still, they can be mitigated effectively to a certain extent: making use of a properly configured alerting system can automate the task of verifying the monitoring and ODA infrastructure's correct operation across components and over time, while active use of a ticketing or versioning system to report and track issues allows for spotting undetected problems quickly, as they appear.

6.3. Human Complexity Factors

Beyond the technical complexity of enabling monitoring and ODA in a production data center environment, there is a series of human factors whose impact on daily operations should not be underestimated. These pertain the interactions between researchers and administrators, and in general among the people involved in the daily operation of a data center, including users and high-level managers, among others.

6.3.1. Coordination with System Operators

The developers of monitoring and ODA frameworks, especially in the HPC field, are very often researchers that are not directly responsible for operating HPC systems. For this reason, tight coordination between these two groups of people is required in order to attain a successful deployment. This dynamic is complicated by the fact that researchers and system administrators usually possess different sets of competences: while the former are usually well-versed in research fields such as data mining, or in any case in fields associated with the analysis of monitoring data, it is the latter that possess the practical expertise that is required for the deployment of software packages. This includes, for example, knowledge and control of installation and logging tools, management of permissions over system features and access to infrastructure machines. Bridging the gap between researchers and administrators is thus essential. A second issue is the fact that the expertise of most ODA researchers is very specific: satisfactory ODA results are often the product of years of work in highly specific domains, spent training and maintaining models. As it stands, pervasive use of ODA would thus require impractical amounts of highly specialized manpower.

While the impact of this aspect is heavily dependent on the personal and team dynamics of the people involved, there are several strategies to mitigate it: first, as

mentioned earlier, monitoring should be established with a commitment at the system's procurement stage, which would in turn ensure both support from management as well as the necessary authority to roll out significant changes or features to a system in production operation. Secondly, interaction between researchers and administrators should be mediated by an internal communication system documenting any changes. This could consist, for example, of a versioning (e.g., *Git*) or ticketing system: on top of being an effective way to organize work and responsibilities among different people, such an approach is also effective at establishing trust and accountability among the different parties, which in turn simplifies technical work. In the case of the DEEP-EST deployment described in Section 5.2, we made extensive use of the ticketing system hosted by JSC: as DEEP-EST is a project involving a consortium of parties to carry out different tasks, this helped establishing responsibilities and, in turn, achieving a painless deployment. Finally, enabling the sharing of code, models and competences across institutions is essential to further simplify deployment and maintenance efforts for complex ODA pipelines.

6.3.2. Value Added to the System

The importance of monitoring in the data center and HPC fields is by now established, and it is thus a natural process in the operations of most data centers. System administrators are usually particularly interested in coarse-grained operational data (e.g., CPU or memory utilization sampled every minute) and stepping up to holistic fine-grained monitoring is normally acceptable, especially if it does not introduce significant overhead into the system or other adverse effects. However, as discussed in Chapter 2, ODA is not as established as monitoring and therefore the deployment of complex models (especially of the ODAC type) can encounter significant resistance from administrators and managers alike. The reasons behind this are twofold: first, even when ODA models are not simply perceived as a hindrance, their value-add to a system's operation is often not understood clearly. Secondly, as discussed in Section 2.3.2, ODA has a significant cost in terms of man power, as models need to be constantly monitored and kept up to date. This is made worse by the fact that the impact on a system of certain ODA techniques, such as fault detection, is hard to quantify and is usually clear only after years of continuous operation.

Given the issues above, the adoption of ODA techniques in a data center should be always proposed with an accompanying quantitative analysis: this should forecast the impact of a certain ODA model on the target system under realistic conditions, using clear metrics (e.g., TCO or PUE) that are understandable by managers and administrators. If the benefit of adopting a certain ODA technique cannot be proven, it is likely simply not ready for production operation. As an example, we refer to the work by Bourassa et al. [Bou+19] describing the ODA measures adopted over the years at NERSC: this center's systems are equipped with automatic PUE calculation at 15-minute intervals, simplifying impact estimation for changes on system operation. Based on this, the impact of all ODA measures described in the article (which are mostly of the ODAV

type) is summarized with a clear PUE reduction score, accompanied by an estimation of yearly energy savings based on average costs - in this case, a 0.054 total PUE reduction is reported, with roughly USD 175,000 of associated energy savings. This kind of quantitative analysis provides a clear-cut, understandable motivation for the adoption of ODA in production environments.

6.3.3. Impact on User Operation

While monitoring is usually transparent to users, ODA has a perceivable impact on their use of an HPC system, cloud platform or data center facility. Some techniques are clearly perceived as beneficial to user activity (e.g., fault detection or scheduling and allocation ODAC techniques), since they result in an improvement of many user-oriented metrics such as job waiting times or application performance. However, techniques that mainly benefit data center operators (e.g., runtime tuning or infrastructure management for energy efficiency purposes) may be perceived by users as detrimental to performance, and thus to be avoided if possible. Sometimes, ODA features can be simply hidden from users (e.g., for infrastructure management), thus eradicating the problem; in other cases, however, a certain degree of control over the ODA dynamics must be granted to users to ensure correct and fair operation of a system. Without precise restrictions, the resulting knobs can be potentially abused.

As a practical example, we refer to the SuperMUC-NG HPC system at LRZ: as described in Section 2.1.3, here the EAR framework optimizes CPU frequencies within compute nodes in order to save energy, usually resulting in a small performance penalty. Due to its nature, EAR's operation is perceived as beneficial by system operators, but as detrimental to performance by users. In fact, in the current system setup, a SLURM option to momentarily disable EAR in user jobs is supplied to users: even though this is meant as an extraordinary measure to simplify benchmark runs, it was observed that a non-negligible fraction of user jobs employs it. Interestingly, disabling EAR does not result in compute nodes using the maximum allowed CPU frequency of 2.7GHz, but rather the base frequency of 2.3GHz, meaning that users were unconsciously renouncing a potential performance boost. Given this data point, we conclude that user-facing ODA techniques should always be accompanied by incentive mechanisms to boost their adoption: this could be, for example, the adoption of *energy billing*, where the cost of using an HPC or data center system is quantified by the amount of energy consumed by users, rather than the CPU core hours, thus incentivizing energy consumption optimization. Clear and transparent documentation describing the impact of user-facing ODA techniques and of the associated knobs is also helpful.

6.4. Action Items

The complexity factors highlighted above, coupled with the associated mitigation strategies, can be condensed into a series of *action items* that the ODA research community

should undertake in order to further simplify the adoption of ODA in data centers, as well as to make it maintainable in the long term.

Portability and Generality. Making pervasive ODA attainable passes through the use of open-source and generic tools by the community, coupled with the sharing of competences across institutions. Creating public ODA models that are portable to multiple system architectures is also important: for example, a data center with years of expertise in failure prediction could create and update machine learning models that can be leveraged by other centers, reducing maintenance efforts. DCDB, Wintermute and the CS method were designed with this scenario in mind.

Commitment to ODA. Committing to ODA at the procurement stage of a system is valuable: it allows for defining the appropriate hardware resources, as well as integration efforts with other tools, the associated responsibilities and personnel requirements. In the currently too rare cases where ODA reaches production, this happens at a late stage in a system's life - hence, better management and policing guidelines would provide the ODA community with means to shape future data centers in a more meaningful way.

Maintainability of Models. ODA models are often susceptible to the changes in operational conditions that occur throughout a system's life: this is especially true for supervised machine learning models, such as the CPU and GPU temperature prediction model proposed in Section 5.3.3, whose effectiveness depends on the training methodology. In order to improve maintainability, it would be beneficial to employ techniques (e.g., reinforcement learning) that can adapt to changing user workloads, component aging and other operational factors.

Self-Monitoring Mechanisms. Ensuring the consistency of monitoring data fed into ODA models is paramount for their correct operation, but this is an excessive burden for system administrators. Hence, automated mechanisms for self-monitoring of ODA pipelines are desirable: these could be as simple as the Health Checker mechanism discussed in Section 5.2.4, or use more complex, statistical approaches to detect deviations and gaps in the monitoring data.

Own publication acknowledgement. Some of the complexity factors discussed in Sections 6.2 and 6.3, as well as the action items in Section 6.4, were presented in [Net+21c], in the broader context of the SuperMUC-NG and DEEP-EST ODA deployments.

7. Conclusions and Future Work

In this chapter we draw the conclusions of our work and outline the steps planned for future research. In Section 7.1 we summarize the work carried out throughout this dissertation, as well as the associated lessons learned. In Section 7.2 we then discuss its main shortcomings, from both the technical and conceptual standpoints - this provides a basis for future work, which is finally outlined in Section 7.3.

7.1. Conclusions

The employment of Operational Data Analytics in a pervasive and highly-integrated way is paramount to ensure the sustainability of exascale HPC platforms, as well as of future data centers. However, the gap between ODA research and its usage in production environments is still wide, resulting in techniques of this kind being used sparingly. The main goal of this dissertation is to bridge this gap, and enable the use of complex, data mining-based ODA techniques in production environments with ease. We tackle this objective from three different sides - monitoring, monitoring data processing and ODA itself - with the aim of creating a standardized platform for ODA models across institutions. The monitoring aspect is addressed via the scalable DCDB framework (Chapter 3); monitoring data processing is instead tackled via the generic data representations of the CS method (Chapter 4), while the encompassing ODA aspect is dealt with via the holistic Wintermute framework (Chapter 3).

On top of being scalable and effective from a practical standpoint, the platform resulting from our work simplifies the implementation and re-use of ODA techniques thanks to its reliance on generic frameworks (Wintermute), data representations (CS method) and collection of data (DCDB). This provides a common ground to stimulate collaboration between data centers as well as sharing of models, thus encouraging the spreading of domain-specific competences beyond the boundaries of single institutions and driving HPC forward as a community. By applying our approach to two distinct production HPC systems (Chapter 5), we demonstrate that it is effective both for large-scale data center installations (i.e., the SuperMUC-NG HPC system) and for heterogeneous, complex systems (i.e., the DEEP-EST prototype); suitability for both ODAV and ODAC use cases is also demonstrated. By sharing the analysis of requirements that led to the designs proposed in this work (Chapter 2) on top of the experiences deriving from them (Chapter 6), we provide a solid foundation for future ODA research and development.

Although our experiences are still limited, we found our standardized approach to be very effective at improving re-use of ODA codes: we were able to leverage many

DCDB and Wintermute plugins over a multitude of projects, making us confident in the reliability of the underlying codes and in their suitability for production environments. This aspect is traditionally the bane of ODA, whose data processing pipelines usually come with large amounts of ad-hoc, hard to maintain integration code. On the other hand, thanks to the compression capabilities of the CS method, we were able deploy compact machine learning models on production HPC systems with no performance impact concerns: this was particularly beneficial in the case of our temperature prediction models discussed in Chapter 5, where the CS method allowed us to employ a single trained model for all CPUs (and one for all GPUs) in the DEEP-EST prototype, drastically reducing maintenance efforts. Combined together, we hope that our approach will finally allow system administrators to tap into the vast well of research ODA techniques in a simple, streamlined manner.

7.2. Shortcomings

Several technical aspects of our work can still be improved. For example, frameworks like DCDB and Wintermute operate well when ODA control decisions can be taken from inside the frameworks themselves - in this case, the entire monitoring, analysis and control logic is contained within the frameworks and is hence easy to port to other systems or apply to other use cases. However, sometimes this is not possible: in the case of scheduling and allocation, monitoring and ODA models can only *influence* control decisions, which are ultimately still applied by the resource manager. In practice, this means that in these cases there is no clear way to avoid developing glue code to integrate the ODA and resource management components, rendering the solutions proposed in our work slightly less attractive. On a similar note, there may be ODA use cases for which processing of monitoring data via the CS method is not suitable, requiring specific and non-portable methods: while we never encountered an ODA use case with such a requirement throughout our experiences, this is certainly possible in practice. Finally, while the Wintermute framework is designed to be generic, enabling its complete feature set still requires considerable integration work - even though this was easily attainable with DCDB, integration may not be as straightforward for other monitoring systems.

Some other aspects are critical points of ODA as a field and are inherent to it. We explored most of these in Chapter 6: in general, one of the main issues behind the low adoption of ODA in production environments is the gap in competences between system administrators and researchers. As the former are usually not well-versed in data mining and similar disciplines, it is often difficult for them to maintain, control and sometimes even understand complex ODA models. Specularly, researchers are often not fully aware of the complexity factors and constraints behind the operation of a data center. This leads to an overall skepticism around the use and effectiveness of ODA, which can only be addressed via clear and tight communication between the research and administration groups of a data center. Moreover, the effort for a standardized ODA platform in this work does not take into account the many factors that often

push into the opposite direction: it is very often the case, in fact, that institutions and vendors are motivated to push their own software solutions (which may not necessarily be open-source) rather than generic ones, for a variety of reasons that can be, among others, economical or political. These themes, however, do not pertain our research activities and are beyond the scope and purpose of this dissertation.

7.3. Future Work

The action items proposed in Section 6.4 target unresolved issues in the ODA field, ranging from the generality of the models used for most techniques, to the need for a more prominent role of ODA at system procurement, as well as for self-monitoring mechanisms to simplify day-to-day operations. These items, which are directed towards the ODA community as a whole, are meant to ensure that ODA is not only usable on production HPC systems, but also maintainable in the long term. On top of this foundation, we aim to explore as future work many contributions that could not be exhausted in this dissertation. Regarding Wintermute, one of the most important points is the integration with monitoring frameworks used in other data centers, beyond DCDB: this would ensure its generality in a final way and it would also propel forward our standardization efforts in the community. Framing Wintermute within the context of wider standardization efforts, such as *PowerStack* [36], is also a possible development. In general, like with any complex software project, much work is still needed to fully streamline the DCDB and Wintermute code bases. Aside from the development aspects, we plan to leverage DCDB and Wintermute to their full extent in upcoming projects, and to identify additional production ODA use cases for their employment; while in this work we were indeed able to tackle both the ODAC and ODAV aspects, the effectiveness of Wintermute in scheduling and allocation use cases is still not fully demonstrated. On top of finding new use cases, we expect to draw significant long-term conclusions from the Wintermute-based production ODA infrastructures that were described in this work, particularly regarding their impact on quality of service, energy efficiency and other operational metrics relevant to users and administrators alike.

The CS method is arguably the aspect of our research that is richest with possibilities for future work, due to its generic nature as a time-series data processing technique. Like Wintermute, we plan to employ the CS method in many other production ODA use cases in addition to the ones presented here, in order to evaluate its effectiveness in practice; among these, applying the method to the data coming from an entire HPC system, with up to thousands of dimensions, is the most enticing. Secondly, we plan to apply the algorithm to time-series datasets that do not necessarily belong to the data center domain, including for example environmental sensor datasets or sound signals. Finally, we expect to keep exploring and characterizing the properties of the CS method: in this work we proposed empirical demonstrations for most of its scaling and semantic properties, but it would be likely possible to demonstrate some of them from a theoretical standpoint as well.

Appendices

A. Plugins for DCDB and Wintermute

In the following we supply a complete list of the plugins available for DCDB and Wintermute as of September 2021, together with a short description for each of them.

A.1. List of DCDB Plugins

DCDB supplies the following Pusher plugins for monitoring:

- **BACnet**: collects data from devices that support the BACnet protocol for building automation and control [Ame10].
- **Caliper**: collects event data from applications that have been instrumented with the Caliper framework [Boe+16].
- **GPFSMon**: collects data associated with usage of a GPFS file system [SH02].
- **IPMI**: collects out-of-band data from devices that support the IPMI protocol [Int13].
- **MSR**: collects a small set of pre-defined CPU performance counters via a lightweight interface [Bau+16].
- **NVML**: collects sensors associated with usage of NVidia GPUs [17].
- **OPA**: collects sensors associated with usage of Intel Omni-Path interconnects [19]. This plugin is now deprecated.
- **REST**: collects out-of-band sensor data using a RESTful API interface.
- **Perfevents**: collects a wide variety of CPU performance counters using the Linux Perfevents interface [Wea13].
- **ProcFS**: collects sensors from files in the Linux ProcFS virtual file system [18].
- **SNMP**: collects data from devices that support the SNMP protocol for management and control [Cas+90].
- **SysFS**: collects sensor data from files in the Linux SysFS virtual file system.
- **Tester**: a plugin that generates an arbitrary number of synthetic sensors.

A.2. List of Wintermute Plugins

In the following are the currently available Wintermute operator plugins:

- **Aggregator**: computes statistical indicators (e.g., average or percentiles) from an arbitrary set of sensors, over a certain time range.
- **Job Aggregator**: performs aggregation of sensors for each job running on an HPC system. It supports the same functionality as the Aggregator plugin.
- **Perfmetrics**: computes a series of performance indicators (e.g., CPI or FLOPS) from specific raw sensors.
- **Persyst SQL**: this is a variation of the Job Aggregator plugin, that outputs all job-level metrics to an arbitrary SQL MariaDB instance.
- **Smoothing**: performs smoothing and sub-sampling of sensors over time.
- **Clustering**: supplies an unsupervised GMM clustering model for arbitrary sensor data, based on the OpenCV implementation.
- **Regressor**: supplies a machine learning regression model for arbitrary sensor data based on random forests, using the OpenCV implementation.
- **Classifier**: a variation of the Regressor plugin that can be used for classification.
- **CS Signatures**: a plugin that computes CS signatures from the data of an arbitrary set of sensors, as described in Chapter 4.
- **Cooling Control**: a plugin that controls the inlet water temperature of an HPC system or rack based on compute node temperatures, using the SNMP protocol.
- **Health Checker**: a plugin that checks for a variety of threshold-based conditions on sensor data and raises alarms by calling arbitrary scripts.
- **File Sink**: a plugin that writes arbitrary sensor data to specific files, without producing any sensor output.
- **Querytest**: a plugin that performs an arbitrary number of sensor queries without producing any sensor output.

B. Configuring DCDB and Wintermute

In this appendix we provide additional details regarding the configuration process of DCDB and Wintermute, highlighting its generic and intuitive nature.

B.1. Overview of the Configuration Process

All of the DCDB software components (i.e., Pusher, Collect Agent and Wintermute) rely on the same uniform configuration process. It employs a series of text configuration files that allow users to select data sources and plugins in a modular way. In particular, two levels of configuration files are available:

1. **Global Configuration File**: this type of file is used to configure a DCDB software component as a whole. It can include generic configuration parameters (e.g., verbosity, listening ports, RESTful API settings), as well as the list of Wintermute and monitoring plugins (the latter for the Pusher only) that must be loaded, and the paths to their configuration files. In order to load a certain configuration, the considered DCDB software component must be pointed to a file of this type.

2. **Plugin Configuration Files**: one file of this kind must be present for each Wintermute or monitoring plugin that is to be loaded in the current configuration. This type of file describes the data sources or operators that have to be instantiated for each plugin, together with their specific parameters. Configuration files of this kind support the creation of templates in order to reduce redundancy and streamline the configuration process.

All configuration files in DCDB and Wintermute leverage the *INFO* property tree format [37] supported by the Boost libraries [Kar05]: it is a simple format for describing hierarchical trees of values, which can be easily parsed and converted to popular formats such as *JSON* or *XML*. An example of a Pusher global configuration file using this format can be seen in Listing 5, instantiating a set of monitoring plugins (*plugins*) and Wintermute plugins (*operatorPlugins*). Collect Agent configuration files, while exposing different parameters, follow the same general format. On the other hand, Wintermute leverages the configuration file of the daemon it is integrated into.

Listing 5 An example of a global configuration file for the DCDB Pusher, using the Boost INFO format.

```
global {
    mqttBroker localhost:1883
    threads 8
    verbosity 3
    cacheInterval 120
    statisticsInterval 600
}

plugins {
    plugin sysfs
    plugin perfevent
    plugin procfs
}

operatorPlugins {
    operatorPlugin aggregator
    operatorPlugin regressor
}
```

B.2. Configuring DCDB Monitoring Plugins

Here we briefly describe the configuration process for Pusher monitoring plugins. An example of a configuration file for the ProcFS plugin, which collects metrics from various files in the homonymous virtual file system, can be seen in Listing 6. In general, any sensor or sensor group templates (the latter using the *template_file* keyword in ProcFS) should appear at the beginning of the configuration file, so that they can be referenced in the rest of the configuration. Then, sensor groups (here with the *file* keyword) usually represent the outermost level of the hierarchy, with sensors (here with the *metric* keyword) being instantiated inside them and exposing their own configuration parameters. In the case of plugins that support sensor entities, these can be instantiated as an additional hierarchical level that encapsulates sensor groups. In the ProcFS example shown here, a sensor group is associated to a single file that is to be parsed - this can point to, for example, the */proc/meminfo* file. The sensors, instead, refer to specific metrics within the file and are identified via the *type* parameter.

B.3. Configuring Wintermute Plugins

We now focus on the configuration process for Wintermute operator plugins: in Listing 7 we show a simplified version of the configuration for the Regressor plugin that was

Listing 6 An example of a DCDB configuration block for the ProcFS plugin.

```
template_file def1 {
    interval 10000
    minValues 3
}

file procstat {
    default def1
    path /proc/stat
    type procstat
    cpus 0-47

    metric col_user {
        type col_user
        mqttsuffix /col-user
        perCpu on
        delta on
    }

    metric intr {
        type intr
        mqttsuffix /intr
        perCpu off
    }

    metric ctxt {
        type ctxt
        mqttsuffix /ctxt
        perCpu off
    }
}
```

originally used in Section 3.7.1, for the purpose of compute node power consumption prediction. The overall configuration structure is very similar to that of Pusher monitoring plugins, including the definition of templates. In the case of Wintermute plugins, operators represent the outermost hierarchical level (as opposed to sensor groups), while the input and output sensors of their blocks are defined within them - there is no equivalent to the concept of sensor entities.

The most relevant part of the configuration is represented by the sensors within each operator: these are grouped in separate *input* and *output* lists, which respectively define the input and output sensors of each block in the operator, as described in Section 3.4. Optionally, a *globalOutput* list can be used to define operator-level output sensors in plugins that support them. It can be seen that all sensors are defined using the template

Listing 7 An example of a Wintermute configuration block for the Regressor plugin, using the template constructs of the block system.

```
regressor reg1 {
    interval 250
    streaming true
    window 3000
    trainingSamples 30000

    input {
        sensor <bottomup 1>knltemp
        sensor <bottomup 1>active
        sensor <bottomup 1>frequency
        sensor <bottomup 1>intr
        sensor <bottomup 1>col-idle
        sensor <bottomup 1>col-user
        sensor <bottomup 1>col-system
        sensor <bottomup>cpu-cycles
        sensor <bottomup>instructions
        sensor <bottomup>branch-instructions
        sensor <bottomup>cache-misses

        sensor <bottomup 1>power {
            target true
        }
    }

    output {
        sensor <bottomup 1>power-pred {
            mqttsuffix /power-pred
        }
    }
}
```

syntax introduced in Section 3.4.2, rendering Wintermute configurations generic and usable in different Pusher and Collect Agent instances. Among the input sensors, it can be seen that only the *power* sensor has the *target* parameter enabled: this instructs the operator to use the readings of this sensor as responses during training of the random forest model. As such, the *power-pred* output sensor will store predicted power consumption values in the inference phase.

C. Additional Information on HPC-ODA

Here we provide additional details regarding the HPC-ODA dataset collection [28] which was originally introduced in Section 4.3, its format and the configuration of the underlying applications in the data acquisition process.

C.1. Format of the Dataset Collection

Starting from the *root* directory, HPC-ODA exposes the following directory structure:

1. One directory for each segment: these are named *anomaly_injection*, *application_classification*, *power_prediction*, *infrastructure_management* and *cross_architecture*.
2. A *sensors* and *responses* directory for each segment, containing respectively its sensor data, as well as the responses for its reference ODA problem.
3. A series of CSV files each containing data belonging to a different sensor. Response files are always named as *responses.csv*, while ordinary sensor files are named according to the original sensor (e.g., *node1.instructions.csv*).
4. Optionally, one or more sub-directories containing sensor data or responses related to sub-components in the segment (e.g., CPU core-level sensors in a compute node-level segment).

Each CSV file in HPC-ODA stores the time series of a certain sensor and has a very simple format: it is a two-column file, where each line contains a $time-stamp, value$ pair. The time-stamps are expressed in 64 bits (i.e., nanoseconds), whereas sensor values are always integers. In the case of label files (e.g., for the Fault or Application segments), these values are expressed as strings - conversely, response files that refer to ordinary sensors are still in integer format. Monotonic sensors (e.g., CPU performance counters) were converted to their first-order derivatives using backward finite differences; this was not performed for metrics that are intrinsically monotonic, such as energy consumption. The sensor files are named to reflect the name of the corresponding data source, and can be easily identified. Moreover, all sensor and response files are time-aligned within a certain segment and contain the same number of readings. Sometimes, the response files correspond to a processed version of a sensor (e.g., after smoothing): if this is the case, a raw version of the response file is always supplied as *responses_raw.csv*. In the case that multiple alternative versions of the response files are available (e.g., fault or application labels in the Fault segment), these are also supplied.

A small Python framework is supplied with the HPC-ODA collection, providing scripts and classes to automatically parse each of the 5 segments and evaluate the machine learning performance of the respective ODA use cases. The segments can be processed with the baseline signature methods introduced in Section 4.1.2 as well as with the CS method, and several information loss metrics are supplied to evaluate the quality of the compression; moreover, scripts are provided to reproduce all of the experiments presented in Section 4.4. The framework is modular and users can easily integrate new analysis algorithms by implementing the appropriate interfaces. Its only dependencies consist of the numpy, scikit-learn, *scipy* and *seaborn* (optional for visualization) packages, and the suggested execution environment is Python 3.6. Before running the example script *launcher.py*, the directories of the HPC-ODA segments must be moved inside the *hpc-signatures-o-matic/datasets* directory.

C.2. Additional Information on the Segments

Here we provide additional information regarding the acquisition process for each of the segments in HPC-ODA, which may be useful for reproducibility purposes.

Fault Segment

The data in this segment was acquired in the context of a previous work [Net+19a] and is publicly available as the *Antarex HPC Fault Dataset* [29]. In HPC-ODA, we only leverage the *CPU-MEM* and *HDD* parts (corresponding to blocks I and III in the original dataset) that employ single-thread applications. The authors executed benchmark applications and at the same time injected faults in the system at specific times via dedicated programs, so as to trigger anomalies in the behavior of the applications: this process was orchestrated by the *FINJ* framework [Net+18a], also developed by the authors, which is able to inject faults at specific times and for specific durations following a certain statistical workload. Moreover, the fault programs that were used to reproduce anomalous conditions are available at the Github repository associated with the original work, and each fault program can operate in a high or low-intensity mode, thus doubling the number of possible fault conditions.

The benchmarks that were used to load the compute node while acquiring the dataset are *DGEMM* and *HPCC* [38], as well as HPL [DLP03] and *STREAM* [39]. In addition, the *Bonnie++* [40] and *IOZone* [41] benchmarks were used to stress the compute node's hard drive. The available fault programs then reproduce issues ranging from resource contention (*memeater*, *leak*, *ddot* and *dial*), to misconfiguration (*cpufreq* and *copy*) and down to hardware issues (*ioerr* and *pagefail*). Both the benchmarks and the faults were executed as single-thread programs on CPU core 0 of the compute node being used. The latter is equipped with two Intel Xeon E5-2630 v3 CPUs, 128GB of RAM, a Seagate ST1000NM0055-1V4 1TB hard drive and runs the CentOS 7.3 operating system.

Due to the single-thread nature of this segment, we include in HPC-ODA all compute

Table C.1.: The main command-line and configuration parameters used for each of the applications in the HPC-ODA Application segment. Parameters that remain unaltered in smaller configurations are omitted.

Application	Configuration
Kripke	–groups 64 –niter 8 –gset 1 –quad 128 –dset 128 –legendre 4 –zones 256,128,128 –procs 4,2,2 **Small**: –zones 128,128,128 **Smaller**: –niter 10 –zones 128,128,64
AMG	-problem 2 -n 160 160 160 -P 4 2 2; *time_steps*=30 **Small**: -n 120 120 120 **Smaller**: -n 100 100 100
PENNANT	"test/leblancx4/leblancx4.pnt"; *tstop=10.0*, *meshparams*=640 5760 1.0 9.0 **Small**: *meshparams*=480 4320 1.0 9.0 **Smaller**: *tstop=12.0*, *meshparams*=320 2880 1.0 9.0
LAMMPS	-v x 32 -v y 32 -v z 16 -in in.reaxc.hns **Small**: -v x 32 -v y 16 -v z 16 **Smaller**: -v x 16 -v y 16 -v z 16
Quicksilver	-i "Coral2_P1.inp" -X 256 -Y 128 -Z 128 -x 256 -y 128 -z 128 -I 4 -J 2 -K 2 –nParticles 2621440 **Small**: -X 128 -Y 128 -Z 128 -x 128 -y 128 -z 128 –nParticles 1310720 **Smaller**: -X 128 -Y 128 -Z 64 -x 128 -y 128 -z 64
HPL	*Ns*=258816, *NBs*=384, *Ps*=4, *Qs*=4 **Small**: *Ns*=192000 **Smaller**: *Ns*=173568

node-level sensors sampled with the LDMS framework, plus the CPU performance counters associated with CPU core 0. The main response file of this segment contains the labels of the fault programs (at most one at a time) being injected in the compute node. However, an alternative response file is available, named *responses_applications.csv*, containing instead the labels of the benchmark applications being executed.

Application Segment

This segment contains sensor data collected from 16 distinct compute nodes in the SuperMUC-NG HPC system, while running certain parallel applications: these are Kripke, AMG, PENNANT, LAMMPS, Quicksilver and HPL. In this segment we use the standard, open-source version of the HPL benchmark [42]. The three configurations that we use for each of the applications differ in terms of problem size (e.g., grid size or number of particles), but the underlying code structure is never altered; the specific application parameters are shown in Table C.1. In this segment, runs were performed using one MPI process per compute node and 48 OpenMP threads for each of them, as many as the available CPU cores.

The sensor data for each compute node is stored in a separate sub-directory in the segment, and the response files follow the same scheme: one response file for each of the 16 compute nodes is supplied, containing the labels of the applications running at each point in time. The segment can also be modified to use one single response file

Table C.2.: The main command-line and configuration parameters used for each of the applications in the HPC-ODA Power segment. Parameters that remain unaltered in smaller configurations are omitted.

Application	Configuration
Kripke	–groups 64 –niter 7 –gset 1 –quad 128 –dset 128 –legendre 4 –zones 64,64,64 –procs 1,1,1 **Small**: –niter 10 –zones 64,64,32
AMG	-problem 1 -n 192 384 384 -P 1 1 1 **Small**: -n 192 192 384
Nekbone	*iel0*,*ielN*,*ielD*=256 512 1 **Small**: *iel0*,*ielN*,*ielD*=256 448 4
LAMMPS	-v x 8 -v y 16 -v z 16 -in in.reaxc.hns **Small**: -v x 8 -v y 8 -v z 16
IHPL	*problem sizes*=20000 22000 25000 26000, *times to run a test*=2 2 2 1 **Small**: *problem sizes*=1000 2000 5000 10000 15000 18000 20000, *times to run a test*=4 2 2 2 2 2 2

and to combine the sensors coming from the single compute nodes in a larger dataset. In this case application classification can still be performed, since the sensor data from all compute nodes is time-aligned and the applications were executed in parallel on all nodes at the same time.

Power Segment

This segment contains fine-granularity sensor data from a compute node in the CooLMUC-3 HPC system at LRZ. Its use case is tied to prediction of power consumption, which can be used to steer runtime management decisions such as CPU frequency assignment. While acquiring the data, we executed the Kripke, AMG, Nekbone, LAMMPS and HPL applications under two possible different configurations, in order to load the compute node and obtain realistic data. In this case, the configurations are summarized in Table C.2: it should be noted that, in this segment, we do not use the standard version of HPL benchmark, but rather the shared-memory one supplied with the Intel MKL library and optimized for the respective architectures [25]. As such, this version of the benchmark uses slightly different configuration parameters compared to what can be observed in the Application and Cross-architecture segments. For all application runs, we use a number of OpenMP threads equal to 64, as the available CPU cores, and a single MPI process.

This segment contains a total of 47 metrics at the compute node level. In addition, we also supply a series of performance counters and activity indicators for each of the 64 CPU cores in the compute node. These are contained in separate sub-directories, one for each core, and can be used for additional analyses. The response file of the

segment contains the power measurements of the compute node, each averaged with the subsequent 2 samples: the resulting regression task predicts the average power consumption in the next 300ms. The raw power measurements are supplied as a *responses_raw.csv* file. In all cases, the measurements are captured by a sensor at the power outlet, and describe the entire node's activity. A third *responses_applications.csv* file contains the labels of the applications being executed at each point in time.

Infrastructure Segment

This segment contains coarse-granularity sensor data from the entire CooLMUC-3 HPC system. It is approximately 16 days long, and it covers the time frame between December 14th and 31st, 2019. Available sensors capture the state of the warm-water cooling system, as well as power measurements at different levels: in particular, all cooling-related metrics are at the rack level (since cooling units operate on a per-rack basis), while power and energy measurements can be found at both the rack and chassis levels. In the segment, each rack is represented by a separate sub-directory, whereas the name of a sensor's chassis, if any, is encoded in its file name. On a similar note, the segment has one response file for each rack, stored in a separate sub-directory - these contain each cooling unit's removed heat measurements, averaged with the subsequent 29 samples. The resulting regression problem predicts the average amount of heat removed in each rack in the next 5 minutes. We also supply the raw removed heat measurements, with files named *responses_raw.csv*.

Cross-architecture Segment

This segment contains sensor data collected from 3 distinct compute nodes, respectively belonging to the SuperMUC-NG and CooLMUC-3 HPC systems, as well as to a testbed system at LRZ. The associated ODA use case and the underlying applications are the same as in the Application segment: in this case, however, application classification is performed across the different architectures. Moreover, application configurations were adapted for single-node use, as shown in Table C.3. Like for the Application Classification segment, all applications were executed with multiple configurations in order to minimize bias when performing classification. The runs were done using shared-memory parallelism, with only one MPI process and as many threads as the available physical CPU cores.

The SuperMUC-NG and CooLMUC-3 HPC systems were already introduced in the Application and Power segments, and the corresponding sensor data has the same format. The testbed node is instead equipped with two 64-core AMD Epyc Rome 7742 CPUs and 512GB of RAM - due to its experimental nature, this node provides a limited amount of sensors. The sensor data for each compute node type is stored in a separate sub-directory in the segment, and the response files follow the same scheme. In this case, no CPU core-level data is stored, and all information is at the compute node level. The segment has one response file for each compute node type, containing the application

Table C.3.: The main command-line and configuration parameters used for each of the applications in the HPC-ODA Cross-architecture segment. Parameters that remain unaltered in smaller configurations are omitted.

Application	Configuration
Kripke	–groups 64 –niter 14 –gset 1 –quad 128 –dset 128 –legendre 4 –zones 64,64,64 –procs 1,1,1 **Small**: –niter 20 –zones 64,64,32 **Smaller**: –zones 64,32,32
AMG	-problem 2 -n 160 160 160 -P 1 1 1; *time_steps*=60 **Small**: -n 120 120 120 **Smaller**: -n 100 100 100
PENNANT	"test/leblancx4/leblancx4.pnt"; *tstop=10.0*, *meshparams*=320 2880 1.0 9.0 **Small**: *meshparams*=240 2160 1.0 9.0 **Smaller**: *tstop=12.0*, *meshparams*=160 1440 1.0 9.0
LAMMPS	-v x 16 -v y 16 -v z 16 -in in.reaxc.hns **Small**: -v x 8 -v y 16 -v z 16 **Smaller**: -v x 8 -v y 8 -v z 16
Quicksilver	-i "Coral2_P1.inp" -X 128 -Y 128 -Z 64 -x 128 -y 128 -z 64 -I 1 -J 1 -K 1 –nParticles 163840 **Small**: -X 128 -Y 64 -Z 64 -x 128 -y 64 -z 64 –nParticles 81920 **Smaller**: -X 64 -Y 64 -Z 64 -x 64 -y 64 -z 64
HPL	*N*=4, *Ns*=57600 57600 57600 57600, *NBs*=384, *Ps*=1, *Qs*=1 **Small**: *Ns*=53760 53760 53760 53760 **Smaller**: *Ns*=49920 49920 49920 49920

labels at each point in time. Unlike in the Application segment, the data of the three node types can be combined only if some of the sensors are dropped - hence, all compute node types will have the same monitoring structure - or if the data is pre-processed to a uniform representation. The latter approach is pursued in Section 4.4.7 by using the CS method, in the context of application classification.

DEEP-EST Dataset

Here we provide additional details on the monitoring dataset collected on the DEEP-EST prototype, introduced in Section 5.3.1. The experimental part of the dataset, containing the application runs described in the following, is publicly available as part of the HPC-ODA collection; the part of the dataset associated with production jobs on the prototype, instead, is not available for confidentiality reasons.

Cluster Module

The first part of the dataset is associated with the dual-socket CM partition of the DEEP-EST prototype: here, we executed the Kripke, AMG, PENNANT, LAMMPS, Quicksilver, HPL and Nekbone applications on 16 compute nodes according to the configurations in Table C.4. Once again, we use three different configurations for the applications, with one MPI process per compute node and as many OpenMP threads as physical

Table C.4.: The main command-line and configuration parameters used for the applications in the DEEP-EST dataset associated with the CM. Parameters that remain unaltered in smaller configurations are omitted.

Application	Configuration
Kripke	–groups 64 –niter 8 –gset 1 –quad 128 –dset 128 –legendre 4 –zones 256,256,128 –procs 4,2,2 **Small**: –zones 256,128,128 **Smaller**: –zones 128,128,128 **ID**: 1
AMG	-problem 2 -n 160 160 160 -P 4 2 2; *time_steps*=30 **Small**: -n 140 140 140 **Smaller**: -n 120 120 120 **ID**: 2
LAMMPS	-v x 64 -v y 32 -v z 32 -in in.reaxc.hns **Small**: -v x 32 -v y 32 -v z 32 **Smaller**: -v x 32 -v y 32 -v z 16 **ID**: 3
Quicksilver	-i "Coral2_P1.inp" -X 512 -Y 256 -Z 256 -x 512 -y 256 -z 256 -I 4 -J 2 -K 2 –nParticles 2621440 **Small**: -X 256 -Y 256 -Z 256 -x 256 -y 256 -z 256 **Smaller**: -X 256 -Y 128 -Z 128 -x 256 -y 128 -z 128 **ID**: 4
HPL	*N*=1, *Ns*=283392, *NBs*=384, *Ps*=4, *Qs*=4 **Small**: *N*=2, *Ns*=192000 192000 **Smaller**: *Ns*=173568 173568 **ID**: 5
PENNANT	"test/leblancx4/leblancx4.pnt"; *tstop=11.5*, *meshparams*=640 5760 1.0 9.0 **Small**: *meshparams*=480 4320 1.0 9.0 **Smaller**: *tstop=14.0*, *meshparams*=320 2880 1.0 9.0 **ID**: 6
Nekbone	*iel0*,*ielN*,*ielD*=64 512 1 **Small**: *iel0*,*ielN*,*ielD*=64 512 2 **Smaller**: *iel0*,*ielN*,*ielD*=64 448 1 **ID**: 7

cores. Unlike in the other segments of the HPC-ODA collection, here the order of the applications is not sequential, but it was rather randomized at runtime.

The available monitoring data for each of the 16 compute nodes reflects the production monitoring setup described in Section 5.2.3 and it is structured as in the following: one sub-directory for each of the compute nodes' CPU sockets is present, for a total of 32 different sub-directories. Each of them contains socket-specific metrics (e.g., CPU instructions), as well as several compute node-level metrics (e.g., amount of used memory); correspondingly, 32 response files are present, each containing a CPU socket's temperature time series. This is used for the purpose of maximum temperature prediction, as discussed in Section 5.3.3. In addition to this data, a separate sub-directory of the dataset contains a series of metrics associated with the cooling unit of the CM rack: this is useful to correlate the behavior and performance of compute nodes with the different inlet water temperature settings employed throughout the experiment.

A series of alternative responses files, named *responses_applications.csv*, contain the labels of the applications being executed: these include a numerical identifier of the application, as described in Table C.4, plus a second numerical identifier indicating the

Table C.5.: The main command-line and configuration parameters used for the CPU and GPU applications in the DEEP-EST dataset associated with the ESB. Parameters that remain unaltered in smaller configurations are omitted.

Application	Configuration
Kripke	–groups 64 –niter 10 –gset 1 –quad 128 –dset 128 –legendre 4 –zones 256,128,128 –procs 4,2,2 **Small**: –zones 128,128,128 **Smaller**: –zones 128,128,64 **ID**: 1
AMG	-problem 2 -n 140 140 140 -P 4 2 2; *time_steps*=30 **Small**: -n 120 120 120 **Smaller**: -n 100 100 100 **ID**: 2
GPU LAMMPS	-v x 32 -v y 32 -v z 32 -in in.reaxc.hns **Small**: -v x 32 -v y 32 -v z 16 **Smaller**: -v x 32 -v y 16 -v z 16 **ID**: 3
GPU Quicksilver	-i "Coral2_P1.inp" -X 512 -Y 256 -Z 256 -x 512 -y 256 -z 256 -I 8 -J 4 -K 4 –nParticles 2621440 **Small**: -X 256 -Y 256 -Z 256 -x 256 -y 256 -z 256 **Smaller**: -X 256 -Y 256 -Z 128 -x 256 -y 256 -z 128 **ID**: 4
GPU HPL	*N*=3, *Ns*=192000 192000 192000, *NBs*=768, *Ps*=4, *Qs*=4 **Small**: *Ns*=172800 172800 172800 **Smaller**: *Ns*=141696 141696 141696 **ID**: 5
HPL	*N*=1, *Ns*=141696, *NBs*=384, *Ps*=4, *Qs*=4 **Small**: *N*=2, *Ns*=96000 96000 **Smaller**: *Ns*=86784 86784 **ID**: 6

specific configuration being used. This can be either 1, 2 or 3, corresponding respectively to the *normal*, *small* and *smaller* configurations. The two identifiers are separated by an underscore, and 0 indicates the idle state.

Extreme-Scale Booster

We now discuss the part of the DEEP-EST dataset associated with the ESB prototype module. Similarly to its CM counterpart, we executed the Kripke, AMG and HPL applications, plus the GPU-accelerated versions of HPL itself, LAMMPS and Quicksilver on 16 compute nodes. Once again, each application could be executed under three possible configurations, and their order throughout the dataset was randomized. Kripke, AMG and HPL (both CPU and GPU versions) use one MPI process per node with as many OpenMP threads as physical CPU cores, while LAMMPS and Quicksilver use a single OpenMP thread and as many MPI processes as CPU cores.

This dataset is divided in two different sections, both containing 16 sub-directories (one per compute node) and as many response files: the first is associated to the CPUs of the ESB nodes, while the second is strictly GPU-related and contains metrics collected via the NVML DCDB plugin. Both sections of the dataset include component-specific metrics (e.g., CPU instructions or GPU utilization) as well as some compute node-level

metrics (e.g., amount of used memory). The corresponding response files contain the temperature time series for a certain CPU or GPU, which are once again used for prediction as outlined in Section 5.3.3. Like for the CM, alternative response files (named *responses_applications.csv*) containing the labels of applications are available, using the naming scheme previously described, and additional metrics for the corresponding ESB rack's cooling unit are also present - all compute nodes used in the experiment belong, in fact, to the same ESB rack.

D. Experiments with the JS Divergence

In this appendix we describe in detail our implementation of the JS divergence described in Section 4.4.1 in the context of the CS method, and discuss further experiments on the DEEP-EST dataset introduced in Section 5.3.1.

D.1. Details on the Implementation

There are several possible ways to implement the JS divergence metric described in Section 4.4.1 to compare two datasets - in the following we describe the algorithm used throughout our experiments, referring to the uncompressed time-series data as *original* dataset, and to the data produced via the CS method from it as *signatures* dataset. Like the standard JS divergence, our custom implementation yields values in the $[0, 1]$ range, where 0 implies that the original and signatures datasets are identical, and 1 means that they are completely disjointed. Moreover, we assume that the signatures dataset is complex and already normalized. The algorithm comprises the following steps:

1. We perform min-max normalization of the original dataset. Like for the CS method, each time series composing the dataset is normalized separately.

2. We construct the imaginary part of the normalized original dataset, which is composed of the first-order derivatives computed via backward differences for each time series.

3. We approximate the two-dimensional probability distributions of the original and signatures datasets: this is done by computing the histogram of each time series. We repeat this process twice, once for the real parts (in the $[0, 1]$ range) and once for the imaginary parts (in the $[-1, 1]$ range). We use 0.01 as bin width.

4. We apply nearest-neighbor interpolation to the histogram matrices of the signatures dataset, in order to obtain the same number of rows (i.e., data dimensions) of the original dataset.

5. The histogram matrices are normalized against the total number of elements in the respective datasets (number of dimensions multiplied by number of samples) such that the sum of their elements equals 1.

6. We compute the JS divergence by comparing the flattened histogram matrices of the original and signatures datasets and by using a logarithm function with base 2

to compute the Shannon entropy. Once again, the process is applied to the real and imaginary histogram matrices separately.

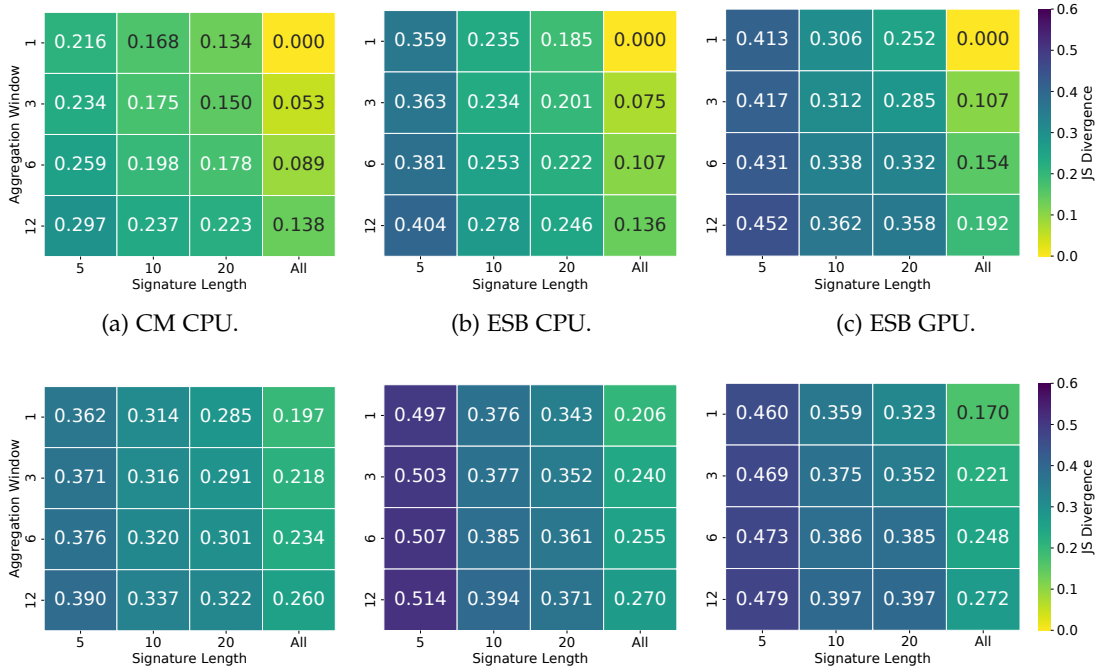
7. The final JS divergence between the two datasets is an average of the real and imaginary parts' divergences.

D.2. Corner Cases

The implementation described above works well in most use cases, but some parameters may require tuning for specific types of datasets. The first of these parameters is the bin width used to compute the histogram of each time series, which by default is 0.01: it is an effective way to control the *resolution* of the JS divergence, with smaller values resulting in a higher sensitivity to minute differences between the original and signatures datasets. However, bin width cannot be decreased arbitrarily, as very small values lead to noisy histograms and hence unstable behavior of the JS divergence. Datasets requiring very small bin width values are those, for example, that include time series that are almost constant, where first-order derivatives are extremely small and close to 0 - for cases such as these we implemented a special *re-normalization* mode, in which the imaginary parts of the original and signatures datasets are normalized again after being computed, ensuring that all first-order derivatives fall into the $[-1, 1]$ range and are thus treated at the appropriate resolution. This approach circumvents the need for extremely low bin width values, and it was employed to compute the results in Section 4.4.4.

A distinct alternative to the computation of histograms is the use of *Kernel Density Estimation* (KDE) methods to approximate the PDF of each time series more effectively. However, this possibility was ultimately discarded due to the high computational cost of KDE methods, which hinders scalability. A pre-requisite for the reliability of the JS divergence measure is, in fact, a very high number of samples (e.g., one million) in the original and signatures datasets. Moreover, KDE methods often require tuning of specific parameters in order to perform well when applied to certain datasets.

In Section 4.4.4, in addition to the basic JS divergence results when comparing the original and signatures datasets associated with the HPC-ODA segments, we also present results when using only the real components of CS signatures: in this case, we assume that the signatures' imaginary parts are always equal to 0 and thus the corresponding PDFs will always have a value of 1 (i.e., a certain outcome) at 0. Intuitively, this means that discarding the CS signatures' imaginary parts may be effective for datasets that are mostly constant over time, where the information about first-order derivatives is redundant. Missing imaginary data in CS signatures could be interpreted in different ways, for example by assuming that the distributions of the respective time series are unknown and hence should be considered as uniform. However, the interpretation proposed here follows most closely the mathematical formulation of the CS method.



(a) CM CPU. (b) ESB CPU. (c) ESB GPU.

(d) CM CPU - Real. (e) ESB CPU - Real. (f) ESB GPU - Real.

Figure D.1.: JS divergence heatmaps when applying the CS method to the DEEP-EST dataset, under different aggregation windows and signature lengths, while using either the full signatures or only their real components.

D.3. Results on the DEEP-EST Dataset

In order to complement the JS divergence results obtained with the HPC-ODA dataset collection in Section 4.4.4, here we present additional experiments performed on the DEEP-EST dataset discussed in Section 5.3.1. Results are shown in Figure D.1 in the form of heatmaps, where each point corresponds to a specific combination of l (i.e., the number of CS blocks) and w^l (i.e., the aggregation window) - we show separate heatmaps for the three parts of the DEEP-EST dataset, which are associated with CM and ESB CPUs, as well as with ESB GPUs. Once again, we show results by either using the full CS signatures or by discarding their imaginary parts, and we do not employ the re-normalization approach described previously.

It can be observed in all heatmaps how the JS divergence transitions smoothly from its maximum value, obtained with an l of 5 and a w^l of 12, down to its minimum value, corresponding to a w^l of 1 and to an l equal to the number of dimensions in the dataset. This is particularly evident when employing full CS signatures, as the minimum JS divergence value is 0, meaning that the original and signatures datasets are identical. Moreover, decreases in JS divergence translate to proportional accuracy improvements for the CPU and GPU temperature prediction models discussed in Section 5.3.3. Confirming

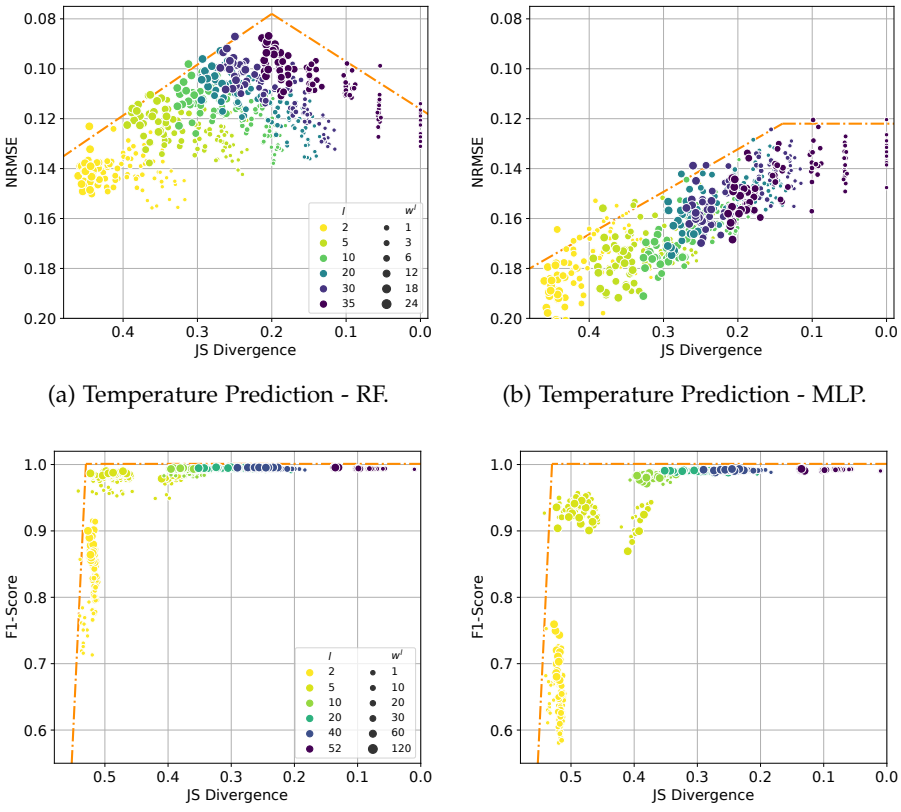
what was observed in Section 4.4.4, removing the imaginary parts of CS signatures results in a noticeable increase in JS divergence. Moreover, it can be noticed how in all experiments doubling l has a larger impact on JS divergence compared to halving w^l, even though both variations have the same effect on the compression ratio against the original data. This means that, for datasets similar to the ones presented here, increasing w^l is likely to bring benefits in terms of machine learning performance without degrading similarity to the original data.

D.4. Correlating ML Performance with the JS Divergence

We now attempt to correlate the JS divergence with ML performance in a more explicit manner with respect to the results discussed in Section 4.4.4. In Figure D.2, in particular, we show scatter plots of the JS divergence and associated ML performance (i.e., NRMSE or F1-score) for the DEEP-EST temperature prediction use case (see Section 5.3.3), as well as for application classification in HPC-ODA (see Section 4.4). We use a variety of w^l and l settings for producing CS signatures and we divide each dataset in small batches of roughly 10,000 feature vectors each, which are evaluated independently - in the case of the DEEP-EST dataset, we only consider the CM-related partition. Additionally, we experiment both with random forests (RF) and multi-layer perceptrons (MLP) as models, using the configurations described in Section 4.4.1. A quick observation of Figure D.2 may lead to wondering why is it necessary to use the JS divergence as opposed to a simpler metric such as the *compression factor*, which is defined as $w^l \cdot n/l$. The explanation can be found in Figure D.1: due to the different impact of w^l and l on information fidelity, datasets processed with the CS method under the same compression factor may still exhibit different JS divergences, proving that simple metrics do not capture the CS method's behavior completely.

The point formations shown in Figure D.2 appear to follow clear scaling patterns according to the specific machine learning problem and model being used - we highlight the envelope of each scatter plot with two lines, forming a *double roofline* shape. A clear example of this is the random forest-based temperature prediction use case in Figure D.2a: starting from the original uncompressed data, with a JS divergence value of 0, the NRMSE improves gradually as the w^l parameter is increased, capturing more and more time-related information at the expense of JS divergence. Performance peaks at 0.08 as NRMSE with a JS divergence of 0.2, and starts degrading from that point onward as the l parameter is decreased. In other words, the right-hand slope describes performance degradation due to an insufficient amount of time-related information, while the left-hand slope is associated with excessive compression. Using a multi-layer perceptron, as in Figure D.2b, yields different results: while the left-hand slope is similar to Figure D.2a, the right-hand one seems to be mostly horizontal. This implies the absence of correlation between ML performance and JS divergence, with the former thus being limited by the model itself or by the quality of data.

Results for application classification, shown in Figures D.2c and d, are similar regard-



(a) Temperature Prediction - RF.

(b) Temperature Prediction - MLP.

(c) Application Classification - RF.

(d) Application Classification - MLP.

Figure D.2.: JS divergence and ML performance scatter plots for DEEP-EST temperature prediction and HPC-ODA application classification, while using CS signatures under different w^l and l settings.

less of the model being used: here, the right-hand slope is horizontal and occupies most of the JS divergence space, meaning that applications are easily recognizable even at relatively high compression factors. The left-hand slope instead occupies a very small region and, while very steep, it is associated with extreme compression settings for the CS method. In all scenarios, it can be seen how ML performance and JS divergence can be effectively coupled together to understand the behavior of models when applied to specific machine learning problems.

E. List of Own Publications

Works Associated with the Dissertation

Conference Articles

- Alessio Netti, Daniele Tafani, Michael Ott and Martin Schulz. "Correlation-wise smoothing: Lightweight knowledge extraction for HPC monitoring data." In *Proceedings of the 35th International Parallel and Distributed Processing Symposium*, pp. 2-12. IEEE, 2021. **Role**: main contributor to all activities.

- Alessio Netti, Micha Müller, Carla Guillen, Michael Ott, Daniele Tafani, Gence Ozer, and Martin Schulz. "DCDB Wintermute: Enabling online and holistic operational data analytics on HPC systems." In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 101-112. ACM, 2020. **Role**: main contributor to all activities; other authors contributed to development activities.

- Alessio Netti, Micha Müller, Axel Auweter, Carla Guillen, Michael Ott, Daniele Tafani, and Martin Schulz. "From facility to application sensor data: modular, continuous and holistic monitoring with DCDB." In *Proceedings of the 32nd International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-27. ACM, 2019. **Role**: main contributor to all activities; other authors contributed to development activities.

Workshop Articles

- Alessio Netti, Woong Shin, Michael Ott, Torsten Wilde and Natalie Bates. "A Conceptual Framework for HPC Operational Data Analytics." In *Proceedings of the Energy Efficient HPC State of the Practice Workshop*. IEEE, 2021. **Role**: main contributor in a collaborative work.

- Gence Ozer, Alessio Netti, Daniele Tafani, and Martin Schulz. "Characterizing HPC Performance Variation with Monitoring and Unsupervised Learning." In *Proceedings of the 1st International Workshop on Monitoring and Operational Data Analytics*. Springer, Cham, 2020. **Role**: secondary contributor to experimentation and paper writing.

- Gence Ozer, Sarthak Garg, Neda Davoudi, Gabrielle Poerwawinata, Matthias Maiterth, Alessio Netti, and Daniele Tafani. "Towards a predictive energy model

for HPC runtime systems using supervised learning." In *Proceedings of the European Conference on Parallel Processing Workshops*, pp. 626-638. Springer, Cham, 2019. **Role**: secondary contributor to experimentation and paper writing.

Preprint Articles

- Alessio Netti, Michael Ott, Carla Guillen, Daniele Tafani and Martin Schulz. "Operational Data Analytics in Practice: Experiences from Design to Deployment in Production HPC Environments." In *arXiv preprint arXiv:2106.14423*. 2021. **Role**: main contributor to all activities; other authors contributed to development and deployment activities.

Prior Works

Conference Articles

- Alessio Netti, Zeynep Kiziltan, Ozalp Babaoglu, Alina Sirbu, Andrea Bartolini, and Andrea Borghesi. "Online fault classification in HPC systems through machine learning." In *Proceedings of the European Conference on Parallel Processing*, pp. 3-16. Springer, Cham, 2019. **Role**: main contributor to all activities.

- Alessio Netti, Cristian Galleguillos, Zeynep Kiziltan, Alina Sirbu, and Ozalp Babaoglu. "Heterogeneity-aware resource allocation in HPC systems." In *Proceedings of the International Conference on High Performance Computing*, pp. 3-21. Springer, Cham, 2018. **Role**: main contributor to all activities.

- Cristian Galleguillos, Zeynep Kiziltan, and Alessio Netti. "AccaSim: an HPC simulator for workload management." In *Proceedings of the Latin American High Performance Computing Conference*, pp. 169-184. Springer, Cham, 2017. **Role**: secondary contributor to development, experimentation and paper writing.

- Alessio Netti, Gianluca Diodati, Francesco Camastra and Vincenzo Quaranta. "FPGA Implementation of a Real-Time Filter and Sum Beamformer for Acoustic Antenna". In *Proceedings of the 44th International Congress and Exposition on Noise Control Engineering*, vol. 250, no. 3, pp. 3458-3469. Institute of Noise Control Engineering, 2015. **Role**: main contributor to all activities.

Journal Articles

- Alessio Netti, Zeynep Kiziltan, Ozalp Babaoglu, Alina Sirbu, Andrea Bartolini, and Andrea Borghesi. "A machine learning approach to online fault classification in HPC systems." In *Future Generation Computer Systems* 110 (2020): 1009-1022. **Role**: main contributor to all activities.

- Cristian Galleguillos, Zeynep Kiziltan, Alessio Netti, and Ricardo Soto. "AccaSim: a customizable workload management simulator for job dispatching research in HPC systems." In *Cluster Computing* 23, no. 1 (2020): 107-122. **Role**: secondary contributor to paper writing.

Workshop Articles

- Alessio Netti, Zeynep Kiziltan, Ozalp Babaoglu, Alina Sirbu, Andrea Bartolini, and Andrea Borghesi. "FINJ: A fault injection tool for HPC systems." In *Proceedings of the European Conference on Parallel Processing Workshops*, pp. 800-812. Springer, Cham, 2018. **Role**: main contributor to all activities.

Glossary

Application	Software program solving a certain scientific problem, that can run in parallel on HPC systems.
Collect Agent	Software component in DCDB acting as a data broker and aggregator.
CS	Correlation-wise Smoothing: data mining technique to compress raw monitoring data into signatures.
DCDB	Data Center Data Base: open-source monitoring framework designed for large-scale HPC systems.
Energy Efficiency	The ability of a system to deliver optimal performance at the minimum energy expense.
Exascale	Qualifies an HPC system which is able to deliver a billion billion floating point operations per second.
HPC	High-Performance Computing: the use of distributed systems to solve scientific problems.
HPC System	Large-scale, distributed system aiming at achieving high performance in scientific problems.
Interconnect	Network interface designed for high bandwidth and low latency in HPC environments.
Monitoring	The act of sampling, collecting and storing sensor data from a large-scale data center environment.
Monitoring Data Processing	The act of processing raw monitoring data into signatures, for subsequent use by ODA models.
Node	Individual server part of a distributed HPC system and either compute, infrastructure or management.
ODA	Operational Data Analytics: techniques aiming at improving the operation of a data center via the analysis of monitoring and log data.

ODAC	Operational Data Analytics for Control: ODA techniques that implement a feedback loop into the monitored system, by proactively tuning knobs.
ODAV	Operational Data Analytics for Visualization: ODA techniques that support administration tasks and provide visualization of data.
Overhead	The performance degradation associated with running a certain software component together with a reference one.
Production Environment	A distributed system supplying a continuous, critical service to users with certain QoL constraints.
Pusher	Software component in DCDB devoted to collecting monitoring sensor data.
Resiliency	The ability of a system to prevent or mask fault and failure events, avoiding catastrophic losses.
Resource Manager	Software component tasked with scheduling (i.e., assigning a start time) and allocation (i.e., assigning a set of nodes) of user jobs.
Scalability	The ability of a software component to retain good performance when used in a large-scale context.
Sensor	Atomic software or hardware data entity that can be sampled to obtain a certain metric.
Signature	Processed monitoring data representation, describing a software or hardware component's status.
Storage Backend	Distributed database providing permanent storage for DCDB sensor data.
System Knobs Control	The act of tuning certain system knobs (e.g., fan speeds) in order to optimize performance.
Wintermute	Open-source framework to enable online and holistic ODA in HPC environments.

Acronyms

ANL	Argonne National Laboratory.
BGMM	Bayesian Gaussian Mixture Model.
BMC	Board Management Controller.
CapEx	Capital Expenses.
CM	Cluster Module.
COP	Coefficient of Performance.
CPI	Cycles Per Instruction.
CS	Correlation-wise Smoothing.
DAM	Data Analytics Module.
DCDB	Data Center Data Base.
DVFS	Dynamic Voltage Frequency Scaling.
EAR	Energy-Aware Runtime.
EDP	Energy Delay Product.
EEHPCWG	Energy Efficient HPC Working Group.
EER	Energy Efficiency Ratio.
ESB	Extreme-Scale Booster.
ETS	Energy-To-Solution.
FFT	Fast Fourier Transform.
FIR	Finite Impulse Response.
FLOPS	Floating Point Operations Per Second.
GEOPM	Global Extensible Open Power Manager.
GMM	Gaussian Mixture Model.
HBM	High-Bandwidth Memory.
HPC	High-Performance Computing.
HPL	High-Performance Linpack.
ICA	Independent Component Analysis.
IoT	Internet of Things.

IPMI	Intelligent Platform Management Interface.
ITUE	IT-power Usage Effectiveness.
JS	Jensen-Shannon.
JSC	Juelich Supercomputing Centre.
KDE	Kernel Density Estimation.
KL	Kullback-Leibler.
LDMS	Lightweight Distributed Metric Service.
LRZ	Leibniz Supercomputing Centre.
MDS	Multidimensional Scaling.
MKL	Math Kernel Library.
MPI	Message Passing Interface.
MQTT	Message Queuing Telemetry Transport.
MRNet	Multicast Reduction Network.
MRPC	Most Relevant Principal Components.
MSA	Modular Supercomputer Architecture.
MSR	Model-Specific Register.
NCSA	National Center for Supercomputing Applications.
NERSC	National Energy Research Scientific Computing.
NRMSE	Normalized Root Mean Square Error.
NTP	Network Time Protocol.
NVML	Nvidia Management Library.
ODA	Operational Data Analytics.
ODAC	Operational Data Analytics for Control.
ODAV	Operational Data Analytics for Visualization.
OMNI	Operations Monitoring Notification Infrastructure.
OpEx	Operating Expenses.
ORNL	Oak Ridge National Laboratory.
PAPI	Performance Application Programming Interface.
PCA	Principal Component Analysis.
PDF	Probability Density Function.
PDU	Power Distribution Unit.
PUE	Power Usage Effectiveness.
QoS	Quality of Service.

RAPL	Running Average Power Limit.
RAS	Reliability, Availability and Serviceability.
SAX	Symbolic Aggregate Approximation.
SLES	SUSE Linux Enterprise Server.
SLURM	Simple Linux Utility for Resource Management.
SNMP	Simple Network Management Protocol.
SVD	Singular Value Decomposition.
TCO	Total Cost of Ownership.
TDP	Thermal Design Point.
TUE	Total-power Usage Effectiveness.
VQ	Vector Quantization.

Index

List of Figures

List of Tables

List of Listings

Bibliography

[Abd+18] G. Abdulla, A. M. Bailey, J. Weaver, and R. Bockmon. *Forecasting Extreme Site Power Fluctuations Using Fast Fourier Transformation*. Lawrence Livermore National Laboratory. 2018. URL: `https://eehpcwg.llnl.gov/assets/sc18_workship_forecasting_adbulla.pdf`.

[Adh+10] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, et al. "HPCToolkit: tools for performance analysis of optimized parallel programs." In: *Concurrency and Computation: Practice & Experience* 22.6 (2010), pp. 685–701.

[Age+14] A. Agelastos, B. Allan, J. Brandt, P. Cassella, et al. "The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications." In: *Proc. of SC 2014*. IEEE. 2014, pp. 154–165.

[Ahl+18] V. Ahlgren, S. Andersson, J. Brandt, N. Cardo, et al. "Large-Scale System Monitoring Experiences and Recommendations." In: *Proc. of CLUSTER 2018*. IEEE. 2018, pp. 532–542.

[All77] J. Allen. "Short term spectral analysis, synthesis, and modification by discrete Fourier transform." In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 25.3 (1977), pp. 235–238.

[Ame10] American Society of Heating, Refrigerating and Air-Conditioning Engineers (ASHRAE). *BACnet—A Data Communication Protocol for Building Automation and Control Networks*. Standard. Atlanta, GA, Jan. 2010.

[AP07] J. A. Aslam and V. Pavlu. "Query hardness estimation using Jensen-Shannon divergence among multiple scoring functions." In: *Proc. of ECIR 2007*. Springer. 2007, pp. 198–209.

[Ash+10] S. Ashby, P. Beckman, J. Chen, P. Colella, et al. "The opportunities and challenges of exascale computing." In: *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee* (2010), pp. 1–77.

[Ate+18] E. Ates, O. Tuncer, A. Turk, V. J. Leung, et al. "Taxonomist: Application Detection through Rich Monitoring Data." In: *Proc. of Euro-par 2018*. Springer. 2018, pp. 92–105.

[Auw+14] A. Auweter, A. Bode, M. Brehm, L. Brochard, et al. "A case study of energy aware scheduling on SuperMUC." In: *Proc. of ISC 2014*. Springer. 2014, pp. 394–409.

[AW10] H. Abdi and L. J. Williams. "Principal component analysis." In: *Wiley interdisciplinary reviews: computational statistics* 2.4 (2010), pp. 433–459.

[Bar08] W. Barth. *Nagios: System And Network Monitoring*. 2nd ed. Open Source Press GmbH, 2008. ISBN: 1-59327-179-4.

[Bau+16] G. H. Bauer, J. Brandt, A. Gentile, A. Kot, et al. "Dynamic model specific register (MSR) data collection as a system service." In: *Cray Users Group (CUG)* (2016).

[Bau+19] E. Bautista, M. Romanus, T. Davis, C. Whitney, et al. "Collecting, Monitoring, and Analyzing Facility and Systems Data at the National Energy Research Scientific Computing Center." In: *Proc. of the ICPP 2019 Workshops*. ACM. 2019, p. 10.

[Bec+01] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, et al. "Manifesto for agile software development." In: (2001).

[Ben+17] F. Beneventi, A. Bartolini, C. Cavazzoni, and L. Benini. "Continuous learning of HPC infrastructure models using big data analytics and in-memory processing tools." In: *Proc. of DATE 2017*. IEEE. 2017, pp. 1038–1043.

[BF07] C. Bash and G. Forman. "Cool Job Allocation: Measuring the Power Savings of Placing Jobs at Cooling-Efficient Locations in the Data Center." In: *Proc. of USENIX 2007*. Vol. 138. 2007, p. 140.

[BNS19] L. Bortot, W. Nardelli, and P. Seto. "Data Centers are a software development challenge." In: *Proc. of ICPP 2019*. 2019.

[Bod+10] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, et al. "Fingerprinting the datacenter: automated classification of performance crises." In: *Proc. of EuroSys 2010*. ACM. 2010, pp. 111–124.

[Boe+16] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, et al. "Caliper: performance introspection for HPC software stacks." In: *Proc. of SC 2016*. IEEE. 2016, p. 47.

[Bor+16] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, et al. "Predictive modeling for job power consumption in HPC systems." In: *Proc. of HiPC 2016*. Springer. 2016, pp. 181–199.

[Bor+19] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, et al. "A semisupervised autoencoder-based approach for anomaly detection in high performance computing systems." In: *Engineering Applications of Artificial Intelligence* 85 (2019), pp. 634–644.

[Bou+19] N. Bourassa, W. Johnson, J. Broughton, D. M. Carter, et al. "Operational Data Analytics: Optimizing the National Energy Research Scientific Computing Center Cooling Systems." In: *Proc. of the ICPP 2019 Workshops*. ACM, 2019, 5:1–5:7.

[Box+15] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[BPG10] S. Benedict, V. Petkov, and M. Gerndt. "Periscope: An online-based distributed performance analysis tool." In: *Tools for High Performance Computing 2009*. Springer, 2010, pp. 1–16.

[Bra+16] J. Brandt, A. Gentile, M. Showerman, J. Enos, et al. "Large-scale Persistent Numerical Data Source Monitoring System Experiences." In: *Proc. of the IPDPS Workshops 2016*. IEEE. 2016, pp. 1711–1720.

[Bra+20] D. Brayford, C. Bernau, W. Hesse, and C. Guillen. "Analyzing Performance Properties Collected by the PerSyst Scalable HPC Monitoring Tool." In: *arXiv preprint arXiv:2009.06061* (2020).

[Bri+16] T. Bridi, A. Bartolini, M. Lombardi, M. Milano, et al. "A constraint programming scheduler for heterogeneous high-performance computing machines." In: *IEEE Transactions on Parallel and Distributed Systems* 27.10 (2016), pp. 2781–2794.

[BS18] O. Babaoglu and A. Sirbu. "Cognified Distributed Computing." In: *Proc. of ICDCS 2018*. IEEE. 2018, pp. 1180–1191.

[Cap+14] F. Cappello, A. Geist, W. Gropp, S. Kale, et al. "Toward exascale resilience: 2014 update." In: *Supercomputing frontiers and innovations* 1.1 (2014), pp. 5–28.

[Car12] D. Carasso. *Exploring Splunk - Search processing Language (SPL) Primer and Cookbook*. 1st ed. CITO Research, 2012. ISBN: 978-0-9825506-7-0.

[Cas+90] J. Case, M. Fedor, M. L. Schoffstall, and D. James. *A Simple Network Management Protocol (SNMP)*. RFC 1157. RFC Editor, May 1990, pp. 1–36.

[CB19] J. Corbalan and L. Brochard. *EAR: Energy management framework for supercomputers*. 2019.

[CC08] M. A. Cox and T. F. Cox. "Multidimensional scaling." In: *Handbook of data visualization*. Springer, 2008, pp. 315–347.

[CF99] K.-P. Chan and A. W.-C. Fu. "Efficient time series matching by wavelets." In: *Proc. of ICDE 1999*. IEEE. 1999, pp. 126–133.

[Coh+05] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, et al. "Capturing, indexing, clustering, and retrieving system history." In: *ACM SIGOPS Operating Systems Review*. Vol. 39. 5. ACM. 2005, pp. 105–118.

[Con+15] C. Conficoni, A. Bartolini, A. Tilli, G. Tecchiolli, et al. "Energy-aware cooling for hot-water cooled supercomputers." In: *Proc. of DATE 2015*. IEEE. 2015, pp. 1353–1358.

[Cui+17] Y. Cui, C. Ingalz, T. Gao, and A. Heydari. "Total cost of ownership model for data center technology evaluation." In: *Proc. of ITherm 2017*. IEEE. 2017, pp. 936–942.

[DeV+14] K. DeVogeleer, G. Memmi, P. Jouvelot, and F. Coelho. "Modeling the temperature bias of power consumption for nanometer-scale cpus in application processors." In: *Proc. of SAMOS 2014*. IEEE. 2014, pp. 172–180.

[DH21] W. Di and L. Hong-Liang. "Microprocessor Architecture and Design in Post Exascale Computing Era." In: *Proc. of ICSP 2021*. IEEE. 2021, pp. 20–32.

[DLP03] J. J. Dongarra, P. Luszczek, and A. Petitet. "The LINPACK benchmark: past, present and future." In: *Concurrency and Computation: practice and experience* 15.9 (2003), pp. 803–820.

[DLP97] I. Dagan, L. Lee, and F. Pereira. "Similarity-based methods for word sense disambiguation." In: *Proc. of EACL 1997*. ACL. 1997, pp. 56–63.

[DSB13] O. DeMasi, T. Samak, and D. H. Bailey. "Identifying HPC codes via performance logs and machine learning." In: *Proc. of CLHS Workshop 2013*. ACM. 2013, pp. 23–30.

[Eas+17] J. Eastep, S. Sylvester, C. Cantalupo, B. Geltz, et al. "Global extensible open power manager: A vehicle for HPC community collaboration on co-designed energy management solutions." In: *Proc. of ISC 2017*. Springer. 2017, pp. 394–412.

[Eit+19] J. Eitzinger, T. Gruber, A. Afzal, T. Zeiser, et al. "ClusterCockpit—A web application for job-specific performance monitoring." In: *Proc. of CLUSTER 2019*. IEEE. 2019, pp. 1–7.

[Eja+10] J. Ejarque, A. Micsik, R. Sirvent, P. Pallinger, et al. "Semantic resource allocation with historical data based predictions." In: *Proc. of CLOUD 2010*. IARIA, 2010.

[Eme+15] J. Emeras, S. Varrette, M. Guzek, and P. Bouvry. "Evalix: Classification and Prediction of Job Resource Consumption on HPC Platforms." In: *Proc. of JSSPP 2015*. Springer. 2015, pp. 102–122.

[Erl+19] E. Erlingsson, G. Cavallaro, H. Neukirchen, and M. Riedel. "Scalable Workflows for Remote Sensing Data Processing with the Deep-Est Modular Supercomputing Architecture." In: *Proc. of IGARSS 2019*. IEEE. 2019, pp. 5905–5908.

[Esm+12] B. Esmael, A. Arnaout, R. K. Fruhwirth, and G. Thonhauser. "Multivariate time series classification by combining trend-based and value-based approximations." In: *Proc. of ICCSA 2012*. Springer. 2012, pp. 392–403.

[Eva+14] T. Evans, W. L. Barth, J. C. Browne, R. L. DeLeon, et al. "Comprehensive Resource Use Monitoring for HPCSystems with TACC Stats." In: *Proc. of the HUST Workshop 2014*. IEEE, 2014, pp. 13–21.

[FBB08] K. B. Ferreira, P. Bridges, and R. Brightwell. "Characterizing application sensitivity to OS interference using kernel-level noise injection." In: *Proc. of SC 2008*. IEEE. 2008, p. 19.

[Fei01] D. G. Feitelson. "Metrics for parallel job scheduling and their convergence." In: *Proc. of JSSPP 2001*. Springer. 2001, pp. 188–205.

[Fer15] C. R. Ferenbaugh. "PENNANT: an unstructured mesh mini-app for advanced architecture research." In: *Concurrency and Computation: Practice and Experience* 27.17 (2015), pp. 4555–4572.

[Fis+08] P. Fischer, J. Kruse, J. Mullen, H. Tufo, et al. "Nek5000: Open source spectral element CFD solver." In: *Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, IL* (2008).

[Frö15] H. Fröning. "EXTOLL and Data Movements in Heterogeneous Computing Environments." In: *Sustained Simulation Performance 2014*. Springer, 2015, pp. 127–139.

[Gal+15] S. M. Gallo, J. P. White, R. L. DeLeon, T. R. Furlani, et al. "Analysis of XDMoD/SUPReMM Data Using Machine Learning Techniques." In: *Proc. of CLUSTER 2015*. IEEE. 2015, pp. 642–649.

[Gal+17] C. Galleguillos, A. Sirbu, Z. Kiziltan, O. Babaoglu, et al. "Data-driven job dispatching in HPC systems." In: *Proc. of MOD 2017*. Springer. 2017, pp. 449–461.

[GC15] A. Gainaru and F. Cappello. "Errors and faults." In: *Fault-Tolerance Techniques for High-Performance Computing*. Springer, 2015, pp. 89–144.

[Geo+15] Y. Georgiou, D. Glesser, K. Rzadca, and D. Trystram. "A scheduler-level incentive mechanism for energy efficiency in HPC." In: *Proc. of CCGrid 2015*. IEEE. 2015, pp. 617–626.

[GF13] Q. Guan and S. Fu. "Adaptive anomaly identification by exploring metric subspace in cloud computing infrastructures." In: *Proc. of SRDS 2013*. IEEE. 2013, pp. 205–214.

[GHB14] C. Guillen, W. Hesse, and M. Brehm. "The PerSyst Monitoring Tool - A Transport System for Performance Data Using Quantiles." In: *Proc. of the Euro-Par 2014 Workshops*. Springer, 2014, pp. 363–374.

[Gim+17] A. Giménez, T. Gamblin, A. Bhatele, C. Wood, et al. "ScrubJay: deriving knowledge from the disarray of HPC performance data." In: *Proc. of SC 2017*. ACM. 2017, p. 35.

[GPG15] R. E. Grant, K. T. Pedretti, and A. Gentile. "Overtime: A tool for analyzing performance variation due to network interference." In: *Proc. of the Exascale MPI Workshop 2015*. ACM. 2015, p. 4.

[Gra84] R. Gray. "Vector quantization." In: *IEEE Assp Magazine* 1.2 (1984), pp. 4–29.

[GRC09] F. Guim, I. Rodero, and J. Corbalan. "The resource usage aware backfilling." In: *Proc. of JSSPP 2009*. Springer. 2009, pp. 59–79.

[Gri+18] D. Griebler, D. De Sensi, A. Vogel, M. Danelutto, et al. "Service Level Objectives via C++11 Attributes." In: *Proc. of REPARA Workshop 2018*. Springer. 2018.

[Hei14] J. Heichler. *An introduction to BeeGFS*. 2014.

[HM07] S. Herbert and D. Marculescu. "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors." In: *Proc. of ISLPED 2007*. IEEE. 2007, pp. 38–43.

[HN18] X. S. Hu and M. Niemier. "Cross-layer efforts for energy-efficient computing: towards peta operations per second per watt." In: *Frontiers of Information Technology & Electronic Engineering* 19.10 (2018), pp. 1209–1223.

[HO00] A. Hyvärinen and E. Oja. "Independent component analysis: algorithms and applications." In: *Neural networks* 13.4-5 (2000), pp. 411–430.

[HPE18] Y. Hui, B. H. Park, and C. Engelmann. "A comprehensive informative metric for analyzing HPC system status using the LogSCAN platform." In: *Proc. of the FTXS Workshop 2018*. IEEE. 2018, pp. 29–38.

[IHH18] C. Imes, S. Hofmeyr, and H. Hoffmann. "Energy-efficient Application Resource Scheduling using Machine Learning Classifiers." In: *Proc. of ICPP 2018*. ACM. 2018, p. 45.

[Ils+19] T. Ilsche, D. Hackenberg, R. Schöne, M. Höpfner, et al. "MetricQ: A Scalable Infrastructure for Processing High-Resolution Time Series Data." In: *Proc. of DAAC 2019*. IEEE. 2019, pp. 7–12.

[Ina+15] Y. Inadomi, T. Patki, K. Inoue, M. Aoyagi, et al. "Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing." In: *Proc. of SC 2015*. IEEE. 2015, pp. 1–12.

[Int13] Intel Corporation, Hewlett-Packard Company, Dell Computer Corporation, NEC Corporation. *Intelligent Platform Management Interface Specification v2.0 rev. 1.1*. Industry Specification. Oct. 2013.

[Iwa16] H. Iwai. "End of the scaling theory and Moore's law." In: *Proc. of the IWJT Workshop 2016*. IEEE. 2016, pp. 1–4.

[Iza+18] R. Izadpanah, N. Naksinehaboon, J. Brandt, A. Gentile, et al. "Integrating Low-latency Analysis into HPC System Monitoring." In: *Proc. of ICPP 2018*. ACM. 2018, p. 5.

[JDD12] W. M. Jones, J. T. Daly, and N. DeBardeleben. "Application monitoring and checkpointing in HPC: looking towards exascale systems." In: *Proc. of ACM-SE 2012*. ACM. 2012, pp. 262–267.

[Jha+18] S. Jha, J. Brandt, A. Gentile, Z. Kalbarczyk, et al. "Characterizing Supercomputer Traffic Networks Through Link-Level Analysis." In: *Proc. of CLUSTER 2018*. IEEE. 2018, pp. 562–570.

[Jia+19] W. Jiang, Z. Jia, S. Feng, F. Liu, et al. "Fine-grained Warm Water Cooling for Improving Datacenter Economy." In: *Proc. of ISCA 2019*. ACM. 2019, pp. 474–486.

[Kan+01] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, et al. "Workload management with loadleveler." In: *IBM Redbooks* 2.2 (2001), p. 58.

[Kar+19] R. Karlstetter, R. Widhopf-Fenk, J. Hermann, D. Rouwenhorst, et al. "Turning dynamic sensor measurements from gas turbines into insights: a big data approach." In: *Proc. of Turbo Expo 2019*. Vol. 58677. American Society of Mechanical Engineers. 2019, V006T05A021.

[Kar05] B. Karlsson. *Beyond the C++ standard library: an introduction to boost*. Pearson Education, 2005.

[KBB15] A. J. Kunen, T. S. Bailey, and P. N. Brown. *KRIPKE-a massively parallel transport mini-app*. Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2015.

[Kjæ+16] M. B. Kjærgaard, K. Arendt, A. Clausen, A. Johansen, et al. "Demand response in commercial buildings with an assessable impact on occupant comfort." In: *Proc. of SmartGridComm 2016*. IEEE. 2016, pp. 447–452.

[KL51] S. Kullback and R. A. Leibler. "On information and sufficiency." In: *The annals of mathematical statistics* 22.1 (1951), pp. 79–86.

[Kna+07] R. L. Knapp, K. Mohror, A. Amauba, K. L. Karavanic, et al. "PerfTrack: Scalable application performance diagnosis for linux clusters." In: *Proc. of LCI 2007*. Citeseer. 2007, pp. 15–17.

[Lag+13] I. Laguna, S. Mitra, F. A. Arshad, N. Theera-Ampornpunt, et al. "Automatic problem localization via multi-dimensional metric profiling." In: *Proc. of SRDS 2013*. IEEE. 2013, pp. 121–132.

[LBB18] A. Libri, A. Bartolini, and L. Benini. "Dwarf in a Giant: Enabling Scalable, High-Resolution HPC Energy Monitoring for Real-Time Profiling and Analytics." In: *ArXiv e-prints* (2018).

[Lep+20] K. Lepenioti, A. Bousdekis, D. Apostolou, and G. Mentzas. "Prescriptive analytics: Literature review and research challenges." In: *International Journal of Information Management* 50 (2020), pp. 57–70.

[Li+09] J. Li, X. Ma, K. Singh, M. Schulz, et al. "Machine learning based online performance prediction for runtime parallelization and task scheduling." In: *Proc. of ISPASS 2009*. IEEE. 2009, pp. 89–100.

[Lia+18] X.-k. Liao, K. Lu, C.-q. Yang, J.-w. Li, et al. "Moving from exascale to zettascale computing: challenges and techniques." In: *Frontiers of Information Technology & Electronic Engineering* 19.10 (2018), pp. 1236–1244.

[Lig17] R. A. Light. "Mosquitto: server and client implementation of the MQTT protocol." In: *The Journal of Open Source Software* 2.13 (2017), p. 265.

[Lin+07] J. Lin, E. Keogh, L. Wei, and S. Lonardi. "Experiencing SAX: a novel symbolic representation of time series." In: *Data Mining and knowledge discovery* 15.2 (2007), pp. 107–144.

[Loc10] D. Locke. "Mq telemetry transport (mqtt) v3. 1 protocol specification." In: *IBM developerWorks Technical Library* (2010), p. 15.

[LP83] Y. Lim and S. Parker. "FIR filter design over a discrete powers-of-two coefficient space." In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 31.3 (1983), pp. 583–591.

[LWP16] X. Lin, Y. Wang, and M. Pedram. "A Reinforcement Learning-Based Power Management Framework for Green Computing Data Centers." In: *Proc. of IC2E 2016*. IEEE, 2016, pp. 135–138.

[LZL09] Z. Lan, Z. Zheng, and Y. Li. "Toward automated anomaly identification in large-scale systems." In: *IEEE Transactions on Parallel and Distributed Systems* 21.2 (2009), pp. 174–187.

[LZL10] Z. Lan, Z. Zheng, and Y. Li. "Toward automated anomaly identification in large-scale systems." In: *IEEE Transactions on Parallel and Distributed Systems* 21.2 (2010), pp. 174–187.

[MCC04] M. L. Massie, B. N. Chun, and D. E. Culler. "The ganglia distributed monitoring system: design, implementation, and experience." In: *Parallel Computing* 30.7 (2004), pp. 817–840.

[McK+16] R. McKenna, S. Herbein, A. Moody, T. Gamblin, et al. "Machine learning predictions of runtime and IO traffic on high-end clusters." In: *Proc. of CLUSTER 2016*. IEEE. 2016, pp. 255–258.

[MF10] A. Matsunaga and J. A. Fortes. "On the use of machine learning to predict the time and resources consumed by applications." In: *Proc. of CCGrid 2010*. IEEE. 2010, pp. 495–504.

[Mic+12] R. Miceli, G. Civario, A. Sikora, E. César, et al. "Autotune: A plugin-driven approach to the automatic tuning of parallel applications." In: *International Workshop on Applied Parallel Computing*. Springer. 2012, pp. 328–342.

[Mil91] D. L. Mills. "Internet time synchronization: the network time protocol." In: *IEEE Transactions on communications* 39.10 (1991), pp. 1482–1493.

[Mit14] S. Mittal. "Power management techniques for data centers: A survey." In: *arXiv preprint arXiv:1404.6681* (2014).

[MR05] O. Maimon and L. Rokach. "Data mining and knowledge discovery handbook." In: (2005).

[Muc+99] P. J. Mucci, S. Browne, C. Deane, and G. Ho. "PAPI: A portable interface to hardware performance counters." In: *Proc. of HPCMP Users Group Conference*. Vol. 710. 1999.

[Net+18a] A. Netti, Z. Kiziltan, O. Babaoglu, A. Sirbu, et al. "FINJ: A fault injection tool for HPC systems." In: *Proc. of the Resilience Workshop 2018*. Springer. 2018, pp. 800–812.

[Net+18b] M. A. Netto, R. N. Calheiros, E. R. Rodrigues, R. L. Cunha, et al. "HPC cloud for scientific and business applications: taxonomy, vision, and research challenges." In: *ACM Computing Surveys (CSUR)* 51.1 (2018), pp. 1–29.

[Net+19a] A. Netti, Z. Kiziltan, O. Babaoglu, A. Sirbu, et al. "A machine learning approach to online fault classification in HPC systems." In: *Future Generation Computer Systems* (2019).

[Net+19b] A. Netti, M. Mueller, A. Auweter, C. Guillen, et al. "From Facility to Application Sensor Data: Modular, Continuous and Holistic Monitoring with DCDB." In: *Proc. of SC 2019*. ACM, 2019.

[Net+20] A. Netti, M. Mueller, C. Guillen, M. Ott, et al. "DCDB Wintermute: Enabling Online and Holistic Operational Data Analytics on HPC Systems." In: *Proc. of HPDC 2020*. ACM, 2020.

[Net+21a] A. Netti, W. Shin, M. Ott, T. Wilde, and N. Bates. "A Conceptual Framework for HPCOperational Data Analytics." In: *Proc. of the EEHPC SOP Workshop 2021* (2021).

[Net+21b] A. Netti, D. Tafani, M. Ott, and M. Schulz. "Correlation-wise Smoothing: Lightweight Knowledge Extraction for HPC Monitoring Data." In: *Proc. of IPDPS 2021* (2021).

[Net+21c] A. Netti, D. Tafani, M. Ott, and M. Schulz. "Operational Data Analytics in Practice: Experiences from Design to Deployment in Production HPC Environments." In: *arXiv preprint arXiv:2106.14423* (2021).

[NS18] M. Naghshnejad and M. Singhal. "Adaptive Online Runtime Prediction to Improve HPC Applications Latency in Cloud." In: *Proc. of CLOUD 2018*. IEEE. 2018, pp. 762–769.

[Olu16] R. Olups. *Zabbix Network Monitoring*. 2nd ed. Packt Publishing, 2016. ISBN: 9781782161288.

[Ott+20] M. Ott, W. Shin, N. Bourassa, T. Wilde, et al. "Global Experiences with HPC Operational Data Measurement, Collection and Analysis." In: *Proc. of CLUSTER 2020*. IEEE. 2020, pp. 499–508.

[Oze+19] G. Ozer, S. Garg, N. Davoudi, G. Poerwawinata, et al. "Towards a Predictive Energy Model for HPC Runtime Systems Using Supervised Learning." In: *Proc. of PMACS Workshop 2019*. Springer. 2019.

[Oze+20] G. Ozer, A. Netti, D. Tafani, and M. Schulz. "Characterizing HPC Performance Variation with Monitoring and Unsupervised Learning." In: *Proc. of MODA Workshop 2019*. Springer. 2020.

[Pat+06] C. D. Patel, R. K. Sharma, C. E. Bash, and M. H. Beitelmal. "Energy flow in the information technology stack: Introducing the coefficient of performance of the ensemble." In: *Proc. of IMECE 2006*. Vol. 47861. 2006, pp. 233–241.

[Pat+13] M. K. Patterson, S. W. Poole, C.-H. Hsu, D. Maxwell, et al. "TUE, a new energy-efficiency metric applied at ORNL's Jaguar." In: *Proc. of ISC 2013*. Springer. 2013, pp. 372–382.

[PC19] L. Perennou and R. Chiky. "Applying Supervised Machine Learning to Predict Virtual Machine Runtime for a Non-hyperscale Cloud Provider." In: *Proc. of ICCCI 2019*. Springer. 2019, pp. 676–687.

[Pli95] S. Plimpton. "Fast parallel algorithms for short-range molecular dynamics." In: *Journal of computational physics* 117.1 (1995), pp. 1–19.

[PWB09] K. Petersen, C. Wohlin, and D. Baca. "The waterfall model in large-scale development." In: *Proc. of PROFES 2009*. Springer. 2009, pp. 386–400.

[RAM03] P. C. Roth, D. C. Arnold, and B. P. Miller. "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools." In: *Proc. of SC 2003*. ACM, 2003, p. 21.

[Rao+20] A. Raoofy, R. Karlstetter, D. Yang, C. Trinitis, et al. "Time Series Mining at Petascale Performance." In: *Proc. of ISC 2020*. Springer. 2020, pp. 104–123.

[Reu+18] A. Reuther, C. Byun, W. Arcand, D. Bestor, et al. "Scalable system scheduling for HPC and big data." In: *Journal of Parallel and Distributed Computing* 111 (2018), pp. 76–92.

[Ric+17] D. F. Richards, R. C. Bleile, P. S. Brantley, S. A. Dawson, et al. "Quicksilver: a proxy app for the Monte Carlo transport code mercury." In: *Proc. of CLUSTER 2017*. IEEE. 2017, pp. 866–873.

[Rob+98] S. J. Roberts, D. Husmeier, I. Rezek, and W. Penny. "Bayesian approaches to Gaussian mixture modeling." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20.11 (1998), pp. 1133–1142.

[Sar+14] O. Sarood, A. Langer, A. Gupta, and L. Kale. "Maximizing throughput of overprovisioned hpc data centers under a strict power budget." In: *Proc. of SC 2014*. IEEE. 2014, pp. 807–818.

[SB02] K. Schwaber and M. Beedle. *Agile software development with Scrum*. Vol. 1. Prentice Hall Upper Saddle River, 2002.

[SB16a] A. Sirbu and O. Babaoglu. "Power consumption modeling and prediction in a hybrid CPU-GPU-MIC supercomputer." In: *Proc. of Euro-Par 2016*. Springer. 2016, pp. 117–130.

[SB16b] A. Sirbu and O. Babaoglu. "Towards operator-less data centers through data-driven, predictive, proactive autonomics." In: *Cluster Computing* 19.2 (2016), pp. 865–878.

[SEL19] E. Suarez, N. Eicker, and T. Lippert. *Modular Supercomputing Architecture: from Idea to Production*. Tech. rep. Jülich Supercomputing Center, 2019.

[SH02] F. B. Schmuck and R. L. Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters." In: *Proc. of FAST*. Vol. 2. 19. 2002.

[Sho+17] H. Shoukourian, T. Wilde, D. Labrenz, and A. Bode. "Using machine learning for data center cooling infrastructure efficiency prediction." In: *Proc. of the IPDPS 2014 Workshops*. IEEE. 2017, pp. 954–963.

[SK20] H. Shoukourian and D. Kranzlmüller. "Forecasting power-efficiency related key performance indicators for modern data centers using LSTMs." In: *Future Generation Computer Systems* (2020).

[SM06] S. S. Shende and A. D. Malony. "The TAU parallel performance system." In: *The International Journal of High Performance Computing Applications* 20.2 (2006), pp. 287–311.

[SR19] L. Stanisic and K. Reuter. "MPCDF HPC Performance Monitoring System: Enabling Insight via Job-Specific Analysis." In: *Proc. of Euro-Par 2019*. Springer. 2019, pp. 613–625.

[SS16] R. Soltanpoor and T. Sellis. "Prescriptive analytics for big data." In: *Proc. of ADC 2016*. Springer. 2016, pp. 245–256.

[Ste+19a] R. Stevens, J. Ramprakash, P. Messina, M. Papka, et al. *Aurora: Argonne's Next-Generation Exascale Supercomputer*. Tech. rep. Argonne National Laboratory (ANL), Argonne, IL (United States), 2019.

[Ste+19b] G. L. Stewart, G. A. Koenig, J. Liu, A. Clausen, et al. "Grid accommodation of dynamic HPC demand." In: *Proc. of the ICPP 2019 Workshops*. 2019, pp. 1–4.

[SV18] D. Shaykhislamov and V. Voevodin. "An approach for dynamic detection of inefficient supercomputer applications." In: *Procedia Computer Science* 136 (2018), pp. 35–43.

[ŢCH02] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth. "Active Harmony: Towards Automated Performance Tuning." In: *Proc. of SC 2002*. IEEE, 2002, pp. 1–11.

[THW10] J. Treibig, G. Hager, and G. Wellein. "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments." In: *Proc. of the ICPP 2010 Workshops*. IEEE. 2010, pp. 207–216.

[Tun+18] O. Tuncer, E. Ates, Y. Zhang, A. Turk, et al. "Online diagnosis of performance variation in HPC systems using machine learning." In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (2018), pp. 883–896.

[VAN08] A. Verma, P. Ahuja, and A. Neogi. "Power-aware dynamic placement of HPC applications." In: *Proc. of ICS 2008*. ACM. 2008, pp. 175–184.

[Vaz+17] S. S. Vazhkudai, R. Miller, D. Tiwari, C. Zimmer, et al. "GUIDE: a scalable information directory service to collect, federate, and analyze logns for operational insights into a leadership HPC facility." In: *Proc. of SC 2017*. 2017, pp. 1–12.

[Vet+18] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, et al. "Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity." In: (Dec. 2018).

[Vil+14] O. Villa, D. R. Johnson, M. Oconnor, E. Bolotin, et al. "Scaling the power wall: a path to exascale." In: *Proc. of SC 2014*. IEEE. 2014, pp. 830–841.

[Wan+17] Z. Wang, Z. Tian, J. Xu, R. K. V. Maeda, et al. "Modular Reinforcement Learning for Self-Adaptive Energy Efficiency Optimization in Multicore System." In: *Proc. of ASP-DAC 2017*. IEEE. 2017, pp. 684–689.

[WAS14] T. Wilde, A. Auweter, and H. Shoukourian. "The 4 Pillar Framework for energy efficient HPC data centers." In: *Computer Science-Research and Development* 29.3 (2014), pp. 241–251.

[Wea13] V. M. Weaver. "Linux perf_event features and overhead." In: *Proc. of the FastPath Workshop 2013*. Vol. 13. 2013.

[Whi11] A. White. "Exascale challenges: Applications, technologies, and co-design." In: *From Petascale to Exascale: R&D Challenges for HPC Simulation Environments ASC Exascale Workshop*. 2011.

[Wil+15] T. Wilde, T. Clees, H. Schwichtenberg, H. Shoukourian, et al. "Towards energy-efficient data center infrastructure-a holistic approach based on software for modeling, simulation, and (re) configuration of the energy network." In: *EnviroInfo/ICT4S (2)*. 2015, pp. 245–246.

[Wil+17] T. Wilde, M. Ott, A. Auweter, I. Meijer, et al. "CooLMUC-2: A supercomputing cluster with heat recovery for adsorption cooling." In: *Proc. of SEMI-THERM 2017*. IEEE. 2017, pp. 115–121.

[WS16] J. M. Wing and D. Stanzione. *Progress in computational thinking, and expanding the HPC community*. 2016.

[WT12] G. Wang and J. Tang. "The nosql principles and basic application of cassandra model." In: *Proc. of CSSS 2012*. IEEE. 2012, pp. 1332–1335.

[WWP09] S. Williams, A. Waterman, and D. Patterson. "Roofline: an insightful visual performance model for multicore architectures." In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[Wya+18] M. R. Wyatt II, S. Herbein, T. Gamblin, A. Moody, et al. "PRIONN: Predicting Runtime and IO using Neural Networks." In: *Proc. of ICPP 2018*. ACM. 2018, p. 46.

[Xue+15] J. Xue, F. Yan, R. Birke, L. Y. Chen, et al. "PRACTISE: Robust prediction of data center time series." In: *Proc. of CNSM 2015*. IEEE. 2015, pp. 126–134.

[Yan+02] U. M. Yang et al. "BoomerAMG: a parallel algebraic multigrid solver and preconditioner." In: *Applied Numerical Mathematics* 41.1 (2002), pp. 155–177.

[YJG03] A. B. Yoo, M. A. Jette, and M. Grondona. "Slurm: Simple linux utility for resource management." In: *Proc. of JSSPP 2003*. Springer. 2003, pp. 44–60.

[YKK17] C.-C. M. Yeh, N. Kavantzas, and E. Keogh. "Matrix profile VI: Meaningful multidimensional motif discovery." In: *Proc. of ICDM 2017*. IEEE. 2017, pp. 565–574.

[YM13] J. Yuventi and R. Mehdizadeh. "A critical analysis of power usage effectiveness and its use in communicating data center energy consumption." In: *Energy and Buildings* 64 (2013), pp. 90–94.

[ZF13] Y. Zeldes and D. G. Feitelson. "On-line fair allocations based on bottlenecks and global priorities." In: *Proc. of ICPE 2013*. 2013, pp. 229–240.

[Zha+12] H. Zhang, H. You, B. Hadri, and M. Fahey. "HPC usage behavior analysis and performance estimation with machine learning techniques." In: *Proc. of PDPTA 2012*. 2012, p. 1.

[Zhu+19] Y. Zhu, W. Zhang, Y. Chen, and H. Gao. "A novel approach to workload prediction using attention-based LSTM encoder-decoder network in cloud environment." In: *EURASIP Journal on Wireless Communications and Networking* 2019.1 (2019), p. 274.

[ZMW14] W. Zheng, K. Ma, and X. Wang. "Exploiting thermal energy storage to reduce data center capital and operating expenses." In: *Proc. of HPCA 2014*. IEEE. 2014, pp. 132–141.

Sitography

[1] *Top500 Lists*. URL: https://www.top500.org.

[2] *Frontier HPC System*. URL: https://www.olcf.ornl.gov/frontier.

[3] *Fugaku HPC System on Top500*. URL: https://www.top500.org/system/179807.

[4] *Gartner IT Glossary*. URL: https://www.gartner.com/en/glossary.

[5] *SuperMUC HPC System*. URL: https://doku.lrz.de/display/PUBLIC/Decommissioned+SuperMUC.

[6] *SuperMUC-NG HPC System*. URL: https://doku.lrz.de/display/PUBLIC/SuperMUC-NG.

[7] *Elastic Stack*. URL: https://www.elastic.co/products/.

[8] *Icinga*. URL: https://www.icinga.com.

[9] *Apache Spark*. URL: https://spark.apache.org.

[10] *Zenoss*. URL: https://www.zenoss.com.

[11] *Mont-Blanc Projects*. URL: https://www.montblanc-project.eu/project.

[12] *DEEP Projects*. URL: https://www.deep-projects.eu.

[13] *DCDB Repository*. URL: https://dcdb.it.

[14] *Influx DB*. URL: https://www.influxdata.com.

[15] *Kairos DB*. URL: https://kairosdb.github.io.

[16] *Open TSDB*. URL: http://opentsdb.net.

[17] *NVML Library*. URL: https://developer.nvidia.com/nvidia-management-library-nvml.

[18] *ProcFS*. URL: http://man7.org/linux/man-pages/man5/proc.5.html.

[19] *Intel Omni-Path*. URL: https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-driving-exascale-computing.html.

[20] *Grafana*. URL: https://grafana.com.

[21] *Linux-Cluster at LRZ*. URL: https://doku.lrz.de/display/PUBLIC/Linux+Cluster.

[22] *Mellanox Infiniband*. URL: https://www.nvidia.com/en-us/networking/products/infiniband.

[23] *SuperMUC-NG HPC System on Top500*. URL: https://www.top500.org/system/179566.

[24] *CORAL-2 Benchmarks*. URL: https://asc.llnl.gov/coral-2-benchmarks.

[25] *Intel MKL Benchmarks Suite*. URL: https://software.intel.com/en-us/articles/intel-mkl-benchmarks-suite.

[26] *Megware*. URL: https://www.megware.com.

[27] *OpenCV*. URL: https://opencv.org.

[28] *HPC-ODA Dataset Collection*. URL: https://zenodo.org/record/4671477.

[29] *Antarex HPC Fault Dataset*. URL: https://zenodo.org/record/2553224.

[30] *MariaDB Server*. URL: https://mariadb.org.

[31] *DEEP-EST System User Guide*. URL: https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide.

[32] *PyTorch Library*. URL: https://pytorch.org.

[33] *Intel Xeon Thermal Guide*. URL: https://www.intel.com/content/dam/www/public/us/en/documents/guides/xeon-scalable-thermal-guide.pdf.

[34] *Nvidia Tesla Product Brief*. URL: https://images.nvidia.com/content/tesla/pdf/Tesla-V100-PCIe-Product-Brief.pdf.

[35] *HWmon Kernel Module*. URL: https://www.kernel.org/doc/Documentation/hwmon/sysfs-interface.

[36] *HPC PowerStack*. URL: https://powerstack.caps.in.tum.de.

[37] *Boost INFO Format*. URL: https://www.boost.org/doc/libs/1_65_1/doc/html/property_tree.html.

[38] *HPCC Benchmark*. URL: https://icl.utk.edu/hpcc.

[39] *STREAM Benchmark*. URL: https://www.cs.virginia.edu/stream.

[40] *Bonnie++ Benchmark*. URL: https://linux.die.net/man/8/bonnie++.

[41] *IOZone Benchmark*. URL: http://www.iozone.org.

[42] *HPL Benchmark*. URL: https://www.netlib.org/benchmark/hpl.