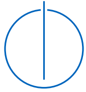
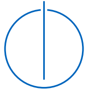


Department of Informatics
Technical University of Munich

Master's Thesis in Informatics

Development and Evaluation of a Generic Framework for Sensor Data Aquisition, Aggregation and Propagation in HPC Systems

Micha Müller



Department of Informatics
Technical University of Munich

TUM

Master's Thesis in Informatics

Development and Evaluation of a Generic Framework for Sensor Data Aquisition, Aggregation and Propagation in HPC Systems

Entwicklung und Evaluierung eines generischen Frameworks zur Erfassung, Aggregation und Weiterleitung von Sensordaten in HPC-Systemen

Author: Micha Müller
Supervisor: Prof. Dr. Martin Schulz
Advisor: Dr. Michael Ott
Submission Date: 15.11.2019

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.11.2019

____
MICHA MÜLLER

Acknowledgements

The author would like to express his gratitude towards a number of people that significantly contributed to this thesis by their involvement.
First, the author would like to thank his thesis supervisor Prof. Dr. Martin Schulz for his feedback throughout the thesis work that was always appreciated.
Second, special thanks to the thesis advisor Dr. Michael Ott, who not only provided valuable feedback, but also supplied technical assistance and clever ideas whenever problems arose.
Third, the author wants to say a big thank you to the whole *DCDB* team, namely Alessio Netti, Carla Guillen, Daniele Tafani, and Michael Ott. Apart from providing an always uncomplicated and constructive work environment, they also offered immediate support regarding all issues.
Fourth, the author would like to thank all proofreaders that contributed to make this thesis as perfect as possible: Karsten Emrich, Hanna Müller, Iris Müller, and Thomas Schilling.
Fifth, the author would like to acknowledge the financial funding of his parents for his fancy living.
Lastly, the author is grateful for the always pleasant atmosphere before and throughout the thesis that emanated from all the people involved.

Abstract

Current HPC systems get ever more powerful and complex. To efficiently detect component failure and allow for energy and performance optimization despite these trends, an adequate monitoring solution is required. To provide such a tool that provides holistic monitoring from facility to application level is the goal of the *DCDB* project. *DCDB* employs a modular architecture with a *Pusher* component that is responsible of acquiring monitoring data in the first place.
This thesis describes a newly developed *Pusher* component for *DCDB*. The *Pusher* implementation is designed as framework that provides the infrastructure for data acquiring plugins. *Pusher*'s design goals, functionality and implementation are presented in detail. Further on, the internal architecture of *Pusher* plugins will be presented as well as the following concrete plugin implementations: *BACnet*, *IPMI*, *OPA*, *Perf Events*, *REST*, *SNMP*, and *SysFS*.
Additionally, this thesis introduces the *Caliper* plugin. It allows to gain introspection into a user's application by employing the *Caliper* toolbox provided by the Lawrence Livermore National Laboratory (LLNL).
Both, the *Pusher* component in general and the *Caliper* plugin in particular are evaluated. It is shown, that *Pusher*'s runtime overhead collecting general in-band data does usually not exceed 5%. Overhead of *Pusher* collecting *Caliper* data, however, can significantly surpass 5% in certain cases. The runtime results for *Caliper* have to be interpreted carefully, though, as they appear not to be fully stable in all cases. Additional memory usage of an application induced by *Caliper*'s integration is determined to be ca. 500 MiB. *Pusher*'s resource usage is shown to be sufficiently low when gathering *Caliper* data. In an usual configuration, *Pusher* does not exceed 1% CPU and 160 MiB of memory usage.

Contents

Acknowledgements iv

Abstract v

List of Figures viii

List of Tables ix

List of Listings x

Nomenclature xi

1 Introduction 1
- 1.1 Problem Statement . . . 1
- 1.2 Contribution . . . 2

2 Background: DCDB 3
- 2.1 Design Principles . . . 3
- 2.2 Components . . . 4
- 2.3 Shortcomings . . . 6

3 Related Work 7

4 DCDB Pusher 9
- 4.1 Design Goals . . . 9
- 4.2 Functionality . . . 10
- 4.3 Implementation . . . 11

5 Pusher Plugins 16
- 5.1 Functionality . . . 16
- 5.2 Implementation . . . 16
- 5.3 Plugin Details . . . 24
  - 5.3.1 IPMI, SNMP, SysFS . . . 24
  - 5.3.2 BACnet . . . 24
  - 5.3.3 OPA . . . 25
  - 5.3.4 REST . . . 25
  - 5.3.5 PerfEvent . . . 25

6 Caliper: A Hybrid Plugin 26
- 6.1 The Caliper Framework . . . 26
- 6.2 Use Cases . . . 27
- 6.3 Caliper-Pusher Communication . . . 29
- 6.4 Caliper Service . . . 29
- 6.5 Pusher Plugin . . . 31

7 Evaluation **33**
7.1 Setup 33
7.2 Pusher Framework Overhead 35
7.3 Caliper Overhead 36
7.3.1 Benchmarking Problems 36
7.3.2 Sampler 36
7.3.3 Sampler Frequency 41
7.3.4 Sampler with Events 41
7.3.5 Conclusion 45

8 Summary and Outlook **46**

A Software Dependencies **47**

B Additional Evaluation Information **48**

Bibliography **50**

List of Figures

2.1 A deployment example of *DCDB* visualizing its different components and their hierarchical structure. 5

4.1 UML class diagram depicting the components of the *Pusher* framework and their relations. Classes with blue background are global to the *DCDB* project. For the sake of simplicity, implementation details are left out. 11

5.1 UML class diagram depicting all involved plugin components and their relations. Classes with blue background are global to the *DCDB* project. Classes with dark gray background are optional. For the sake of simplicity, implementation details are left out. 19
5.2 Overview of all components of the *Pusher* framework and its plugins highlighting the data flow. This figure also includes an introspection into *Collect Agent* and *Storage Backend*. 23

6.1 A histogram of the most meaningful function samples sorted by binaries, as acquired with the *Caliper* plugin for a single-node HPL run on an AVX-512 Skylake system. 28
6.2 Visualization of a fictional use case where CPU monitoring data is enriched with *Event* annotations. 28

7.1 *Pusher*'s overhead on different benchmarks and system sizes on the SNG system. 35
7.2 Runtime overhead induced by *Pusher* and *Caliper* using the *Sampler* configuration on the SNG system. 37
7.3 Runtime overhead induced by *Pusher* and *Caliper* using the *Sampler* configuration on the CM2 system. 38
7.4 Additional memory usage of benchmarks induced by the *Caliper* integration using the *Sampler* configuration on the SNG system. 39
7.5 *Pusher*'s average CPU usage using the *Sampler* configuration on the SNG system. 40
7.6 *Pusher*'s average memory usage using the *Sampler* configuration on the SNG system. 40
7.7 Runtime overhead induced by *Pusher* and *Caliper*, as well as additional memory usage of benchmarks caused by *Caliper*'s integration. Both measured with different *Sampler* frequencies on the SNG system. 42
7.8 *Pusher*'s average CPU and memory usage with different *Sampler* frequencies on the SNG system. 42
7.9 Runtime overhead induced by *Pusher* and *Caliper* using the *Sampler* with *Events* configuration on the SNG system. 43
7.10 *Pusher*'s average CPU usage using the *Sampler* with *Events* configuration on the SNG system. 44
7.11 *Pusher*'s average memory usage using the *Sampler* with *Events* configuration on the SNG system. 44

List of Tables

2.1 Overview of required data sources that should be realized in plugins. 6

4.1 Overview of all RestAPI endpoints supported by *Pusher*. 14

5.1 Overview of all plugins and their features. 18

7.1 Overview of the system hardware used for the evaluation. 34
7.2 Overview of the system hardware used for the *Collect Agents*. 34

A.1 External software dependencies *Pusher* and its plugins rely on. 47

B.1 Compiler versions used for the evaluation. 48

List of Listings

4.1 Excerpt from a configuration file for *Pusher*. 10
4.2 Pseudocode of `MQTTPusher`'s primary `push` method. 13
4.3 Pseudocode of `RESTHttpsServer`'s functionality that will be invoked whenever a request is received. 14

5.1 Excerpt from a configuration file for a generic plugin that uses the optional *Entity* component. 17
5.2 Pseudocode of a *SensorGroup*'s data reading functionality that is executed during runtime. Code in comments shows an alternative use case with an *Entity* involved. 22

B.1 Configuration for *Pusher*'s *Caliper* plugin as used in Section 7.3. 48
B.2 *Pusher*'s configuration file. 49
B.3 Runtime configuration of the Caliper toolbox as used in Section 7.3. 49
B.4 Runtime configuration of the Caliper toolbox as used for the HPL run without *Pusher* in Section 7.3.2. 49

Nomenclature

Abbreviation	Description
ASLR	Address space layout randomization
BACnet	Building Automation and Control Networks
BMC	Baseboard Management Controller
CM2	CoolMUC-2
DCDB	Data Center DataBase
GPFS	General Parallel File-System
HPC	High Performance Computing
HPL	High-Performance Linpack
IPC	Inter-Process Communication
IPMI	Intelligent Platform Management Interface
LLNL	Lawrence Livermore National Laboratory
MKL	Math Kernel Library
MQTT	Message Queuing Telemetry Transport
MSR	Model-specific register
NTP	Network Time Protocol
OPA	Intel Omni-Path
PC	Program counter
PID	Process Identifier
ProcFS	`/proc` file system
Regex	Regular expression
REST	RESTful API plugin
RestAPI	RESTful API
RSS	Residual Set Size
SNG	SuperMUC-NG
SNMP	Simple Network Management Protocol
SysFS	`/sysfs` file system

1 Introduction

The current trends of machine learning and artificial intelligence as well as the general interest of researchers in running ever larger and/or more detailed simulations result in an insatiable demand for compute power. High Performance Computing (HPC) service providers counteract the demand by installing more powerful computing systems, leading to a soon break through of the exascale barrier [1]. For decades, Moore's law [2], which describes the observation that the number of transistors per fixed area doubles approximately every two years, allowed hardware manufacturers to provide ever more powerful systems. While Moore's law is reportedly predicted to decelerate in the near future [3, 4] single CPU performance improvements already hit their limits years ago [5]. To evade the single CPU performance barrier a trend towards heavy parallelization has emerged. Manufactures employ increasing numbers of CPU cores per processor while HPC systems use ever rising numbers of nodes. Apart from the natural increase of CPU cores this trend requires upscaling of devices all along the infrastructure such as server racks, network switches, storage disks, power supply, and cooling systems. The deployment of more system components, together with the adoption of new technologies like liquid cooling, link increasing compute power inseparably with overall higher system complexity.
At the same time, higher system complexity also significantly increases the risk of component failure and misconfiguration. To achieve reliable, continuous, and efficient service HPC systems must therefore be permanently monitored. System components commonly offer access to certain monitoring data of integrated sensors. With the ongoing trend of digitalization the amount of available sensor data is ever increasing and allows to satisfy the needs of all stakeholders. HPC stakeholders, i.e. operators, administrators, and users of HPC centers all share an interest in certain monitoring data. Operators require data of component utilization to optimize them for their most efficient operating point or to reduce overall operating costs. Administrators require notification of component failure or metric anomalies which may indicate malfunctioning devices. Users require exhaustive performance data of their applications for future optimizations. To effectively make use of the available sensor data, also despite the ever more complex systems, and provide stakeholders with required monitoring data one generally deploys a monitoring solution. A monitoring solution collects and stores the available sensor data from one or multiple different systems and allows for easy data access through unified high-level interfaces.

1.1 Problem Statement

The increasing system complexity results in a high diversity of devices. As HPC systems constantly get upgraded, extended, and replaced by new systems there coexist different generations of devices as well as components from a diverse set of vendors. Unfortunately, many of them allow access to their monitoring sensor data only via protocols that are proprietary or incompatible to each other. Further on, most existing monitoring tools

are fixed on certain sensor data sources or are tailored towards a specific stakeholder. Therefore one is required to deploy multiple monitoring solutions to serve all stakeholders and gather data from all devices. Deployment of multiple monitoring tools results in a set of disadvantages such as accumulating overhead and competition for the same data source which can result in mutual inferences. Also, every tool usually uses a different format for its data complicating the correlation of multiple data sources to the point of impossibility.
To overcome the fragmentation into multiple differing monitoring solutions there is a need for a holistic monitoring tool. Deployment of one single solution allows to eliminate possible inference, reduce overhead, and unify all data in one common format and therefore allow for simple and efficient correlation. To keep up with ever more complex future systems, the monitoring tool should be highly scalable and adaptable for new devices aka data sources.

1.2 Contribution

The Data Center DataBase (*DCDB*, cf. Chapter 2) [6, 7] project takes on all of the stated challenges. *DCDB* is a modular monitoring framework that aims to allow holistic monitoring from facility to application level. To achieve this goal, an adaptable and extendable data acquisition component is required that supports data collection from all kinds of data sources. Components of *DCDB* that acquire data for the framework are called *Pusher*.
Before this thesis, the author developed a new *Pusher* component that unifies and enhances previous *Pusher* implementations. The following parts were implemented:

- a new *Pusher* that provides a framework infrastructure for actual data acquisition through plugins and forwards the data to other *DCDB* components,
- four framework plugins, each supporting a new distinct data source,
- porting of the previous *Pusher* components to framework plugins, and
- a general HTTPs Server infrastructure for RESTful APIs (RestAPI) [8] and based upon it a RestAPI was integrated into the new *Pusher* framework for runtime control.

The purpose of this document is twofold: It serves as documentation of the author's previous work on *DCDB* and exhaustively describes the actual thesis work. Namely, an additional hybrid plugin based on the *Caliper* [9] toolbox for application introspection.
This thesis presents *Pusher*'s functionality, design principles, and implementation in detail. The functionality and implementation of *Pusher*'s various plugins in general and the *Caliper* application introspection plugin in particular will be outlined. Further on, *Pusher*'s general overhead and the impact of the *Caliper* plugin on a production system are depicted in this thesis.
The remainder is as follows. Chapter 2 gives an general overview of the *DCDB* project and its parts relevant for *Pusher*. In Chapter 3 other existing monitoring tools are presented. *Pusher*'s design goals, functionality and implementation details are introduced in Chapter 4. The functionality and implementation of *Pusher*'s accompanying plugins in general and the *Caliper* plugin in particular are detailed in Chapters 5 and 6. Chapter 7 presents an evaluation of overhead measurements for *Pusher* and the *Caliper* plugin. Finally, Chapter 8 sums up the thesis and states future work.

2 Background: DCDB

The *Pusher* component presented in this thesis is developed for the *DCDB* project. *DCDB* aims to offer a software architecture for holistic monitoring from facility to application level data. It is primarily targeted at HPC facilities. *DCDB* is presented in detail in [6, 7]. Following, relevant parts are summed up.

2.1 Design Principles

The *DCDB* project employs a few design principles that also apply to *Pusher*. Further on, technical decisions that affect the whole project, like the selected inter-component communication protocol, impact *Pusher*'s implementation. Therefore, relevant design principles and decisions are described in the following.

Holism

DCDB is intended to avoid the fragmentation into multiple custom monitoring systems for each use case. Therefore it aims to be holistic. *DCDB* is kept as generic as possible to be applicable for all relevant use cases. It is not tailored towards a specific use case other than general applicability for HPC facilities. It rather provides only a monitoring infrastructure which may be adapted and deployed as required.

Scalability

HPC facilities usually host one or more large HPC systems. Current systems comprise thousands of CPU cores accompanied by adequate infrastructure hardware. Most of those devices have built in sensors that are accessible to the outside and therefore allow to monitor their operational state. The large quantity of devices results in an exceptional amount of available raw monitoring data that can be expected to further grow in the future as HPC systems get ever larger. To keep up with this enormous data stream, the *DCDB* framework tries to achieve high scalability by employing a distributed modular architecture and consistent use of a hierarchical push principle.

Modularity

Components of *DCDB* run independently of each other and are related to each other only through well-defined APIs or protocols. Access to *DCDB* monitoring data is abstracted through an library called *libdcdb*. Underlying components can therefore be exchanged or supplemented through custom implementations as required. Likewise, monitoring data is pushed into *DCDB* via MQTT, allowing for employing any data collection tool that speaks MQTT. This allows for adaption for a diverse set of use cases. Also, thanks to the independent modularity, multiple instances of one component can be run in parallel. This way, components of *DCDB* can be scaled independently of each other as required to overcome bottlenecks.

Push Principle

Acquired data gets pushed by the data collector (source) to the data sinks. As a result, required computations to gather data are distributed among data sources, avoiding possible bottlenecks of a central data collection service and keeping overhead of an individual instance to a minimum.

Sensor

In the *DCDB* context, a single data point is referenced as *Sensor*. A *Sensor* is the smallest possible data unit in the framework. Multiple readings or measurements of the same *Sensor* result in a timeline of this data point. The *Sensor* unit is part of all *DCDB* components. Two or more *Sensors* can be arithmetically combined to retrieve derived metrics, which is called a *virtual Sensor*. *Virtual Sensors* will not be considered further, however.

MQTT

For communication among *DCDB* components the *Message Queuing Telemetry Transport* (MQTT) [10] protocol is designated. MQTT bases on the publish/subscribe communication pattern and therefore fulfills the need for push-based data forwarding. It satisfies the need for modularity as components can join and leave the MQTT communication domain anytime at own will. The only interconnecting point is the MQTT broker. The protocol is very lightweight and intended to be able to run on embedded devices. As it is widely used, many implementations are available and libraries for all important programming languages exist. Therefore almost no hard- or software limits are imposed on components for *DCDB*'s infrastructure.
Within the MQTT protocol, all published messages are associated with a topic. Data consumers receive data from all topics they are subscribed to. To make the potential huge amount of *Sensors* distinguishable, each *Sensor* has to be assigned its own unique topic. The topic is used as unique identifier among the whole monitoring framework. Additionally, topics allow for file system-like hierarchical ordering. Although not strictly necessary it is strongly recommended to make use of this feature to organize *Sensors*. This way, they can be retrieved by selections with wildcard patterns later on. For example, *Sensors* can be sorted for their location within a system ("Cluster/Rack/Node/Socket").

2.2 Components

The *DCDB* project consists of three integral abstract parts: *Pusher*, *Collect Agent*, and *Storage Backend*. The realization of the components is not fixed and the current implementation may be replaced with other variants at ones own discretion. Specifically for the current implementation of the three core components, Figure 2.1 shows an exemplary deployment of *DCDB*.
In addition to the three core components, an operational data analytic framework called *Wintermute* [11] is integrated into *DCDB*. *Collect Agent* and *Storage Backend* are already implemented in the *DCDB* project and are taken for granted. The *Wintermute* component was independently developed by collaborators in parallel to the presented *Pusher*.

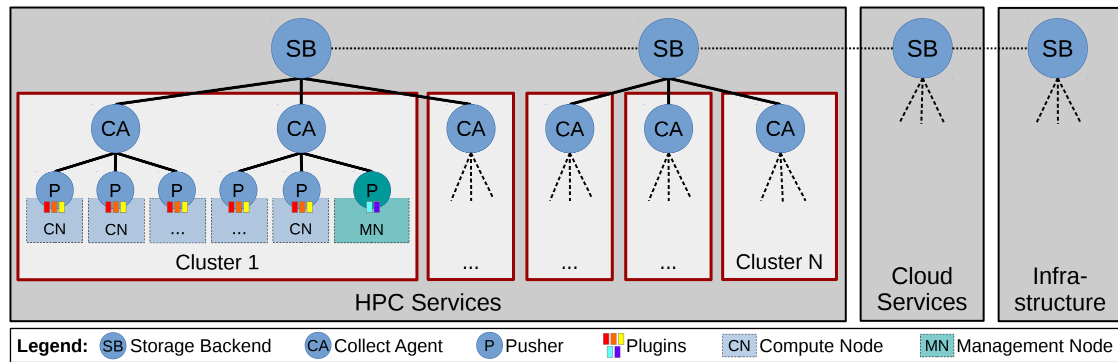


Figure 2.1: A deployment example of *DCDB* visualizing its different components and their hierarchical structure.

Pusher

Acquisition of *Sensor* data in the *DCDB* infrastructure is task of *Pusher*. *Pusher* acts as MQTT publisher and therefore publishes the acquired data from *Sensors* under their respective MQTT topic. To achieve holistic monitoring, *Pusher* should be deployed on all relevant data sources, e.g. compute and management nodes as well as on dedicated servers to gather data remotely from infrastructure devices such as chillers.

Collect Agent

The *Collect Agent* is a custom MQTT broker and therefore acts as data intermediary. It receives published data from one or more *Pusher*. The *Collect Agent* in turn stores the received data in the *Storage Backend* via *libdcdb*. As of now, there are no subscribers for *Sensor* data except the *Storage Backend*. Therefore the subscriber related functionality is not implemented in the *Collect Agent* to reduce unnecessary overhead. The missing logic, however, can be added in the future if required.

Storage Backend

Storage Backend is the *DCDB* part responsible for storing all acquired data and serving requests for historic (non-live) data. One or more *Collect Agents* can write data to a single *Storage Backend* and there can be multiple *Storage Backends*. All read or write accesses to a *Storage Backend* are abstracted by the *libdcdb* library, making it independent of a specific storage solution. As of now, a Cassandra [12] database is employed as *Storage Backend* solution.

Wintermute

While the three aforementioned components make up the core of the *DCDB* infrastructure, the *Wintermute* framework as additional data analytic component is also part of the *DCDB* project. *Wintermute* is integrated in the *Pusher* and *Collect Agent* components and extends their functionality for operational data analyses. It offers in-band or out-of-band and online or on-demand data analyses. *Wintermute*'s internal structure is similar to *Pusher*'s; the framework forms the basis for *Operator* plugins which implement actual data analyses. Plugin data analyses operate on *Sensors* as smallest possible item. Although tightly integrated, the core components of *DCDB* run independent of *Wintermute* and it can be switched off if required.

Data source	P[1]	cf.
Building Automation and Control Networks [13]		5.3.2
Lawrence Livermore National Laboratory's (LLNL) *Caliper* [9] tool		6
General Parallel File-System [14] monitoring data		5
Intelligent Platform Management Interface [15]	X	5.3.1
Intel model-specific register [16]		5
Intel Omni-Path [17]		5.3.3
RESTful APIs [8]		5.3.4
Linux *Perf Events* [18]		5.3.5
Linux `/proc` file system [19]		5
Simple Network Management Protocol [20]	X	5.3.1
Linux `/sysfs` file system [21]	X	5.3.1

[1] Already realized in an existing *Pusher* instance

Table 2.1: Overview of required data sources that should be realized in plugins.

2.3 Shortcomings

Before the work described in this thesis started, there existed no generic *Pusher* framework yet. Instead, following the modularity principle, for each of the three data sources supported at the time a custom *Pusher* component was implemented. The intention was to create multiple differing *Pusher* implementations, each adapted and optimized for its respective protocol. In the long term, however, this approach resulted in significant drawbacks. At the core, all of the separate *Pusher* implementations had the same functionality (gathering data from sensor instances and pushing them to the *Collect Agent*), resulting in huge parts of duplicated code. Changes to the common code base would require adaption of all other *Pusher* instances, significantly impacting development costs for future plans to expand *Pusher*'s functionality. Also, running multiple similar instances induces extra (management-)overhead that could be avoided by running the core functionality only once.
Therefore it was decided to unify the core functionality of all *Pusher* instances into one generic framework. Data source specific code should be outsourced into framework plugins. The already existing *Pusher* implementations should be ported to plugins for the new framework. In general, more data sources should be supported by the development of further framework plugins. A summary of all required data sources is presented in Table 2.1.
This thesis describes the new generic *Pusher* framework and associated plugins for the required data sources. Design, functionality, and implementation of the framework in general are presented in Chapter 4. Details on the plugins and their realization are given in Chapter 5. The *Caliper* plugin is particularly highlighted in Chapter 6.

3 Related Work

The *Pusher* as introduced in this thesis is developed for the *DCDB* monitoring framework [6, 7]. An operational data analytics framework [11] is closely integrated into the *DCDB* infrastructure. Besides *DCDB*, there is a number of other monitoring tools available. Most of them are specialized for a certain domain or focus on one HPC stakeholder. Still, there are also approaches to a continuous and scalable monitoring solution.
Ganglia [23] is a widespread system monitoring tool. A daemon is run locally on each node to retrieve monitoring data. Among each other, the daemons exchange data via a multicast-based listen/announce protocol. Data of node federations can be retrieved from independent aggregating daemons which retrieve data from nodes in a poll-manner. Although it states scalability of up to 2,000 nodes, this is not sufficient anymore for HPC systems as of today. Also, it does not allow for subsecond sampling frequencies which is possibly not fine-grained enough.
The *Lightweight Distributed Metric Service* (LDMS) [24], which is part of the OVIS [25] project, is a more recent HPC system monitoring tool. Its design is very similar to *DCDB*. LDMS runs separate sampler, aggregation and storage instances. All instances are based on the same *ldsmd* daemon, which is adapted by configurable plugins for each use case. However, LDMS' design does not account for customization. Development of new sampling plugins or extension of storage options requires significant effort. Also, the employed custom communication protocol complicates integration of other components into the LDMS infrastructure.
Another rather job-centered system-wide monitoring tool is *TACC Stats* [26, 27]. Similar to the other approaches, a data collection instance called `monitor` is run on each node. The `monitor` allows for exhaustive monitoring of the node's hardware metrics. Collected data can either be harvested and permanently stored once a day or pushed via a RabbitMQ [28] server to data consumers in real-time. Further on, data analysis and visualization tools are offered. Although monitoring with *TACC Stats* is system-wide, it is not holistic. Collectable metrics are limited to compute node hardware.
Performance Co-Pilot (PCP) [29] uses an almost identical design approach to *DCDB*. Its plugin-based architecture allows for customization and system-wide holistic monitoring. Instead of the push-principle as employed by *DCDB*, PCP forwards data in a poll-manner. Reliable key-figures regarding PCP's overhead could not be found.
There is a number of commercial monitoring solutions available, e.g. *Nagios* [30] or *Splunk* [31]. Most cloud service providers are known to offer monitoring options as well. Details about their internals are usually sparse, however. All of them have in common, that they are proprietary, closed-source, and rather alert-oriented.
Some of the system monitoring tools, like *TACC Stats* and *LDMS*, allow to collect hardware data from compute nodes that is closely related to a user's application. None of the tools, however, allows for the integration of actual software introspection data. On application side, there is a vast amount of performance analysis tools available that allow for exhaustive software introspection. Namely *Score-P* [32], *TAU* [33], *perf* [18], *Intel VTune Amplifier* [34], and the *Caliper* toolbox [9] among others. Most of them support instrumentation and/or sampling of applications. All have in common, though, that they are

primarily tailored towards HPC application users. The tools usually store the gathered data in a proprietary data file for retrospective analyses. They do not support forwarding of their data or integration of other facility data. The main work of this thesis, the hybrid *Caliper* plugin, allows to integrate application introspection data into the holistic *DCDB* monitoring framework. To the author's knowledge, no other solutions to unify system hardware monitoring and user software performance analysis data into one monitoring tool exist, yet.

4 DCDB Pusher

The parts responsible of originally acquiring monitoring data for the *DCDB* infrastructure are called *Pusher*. They are the most critical part of *DCDB* as their number of deployed instances can be expected to be the most of all *DCDB* components. To achieve holistic data center monitoring one has to deploy *Pusher* instances on all compute and management servers as well as additional instances to monitor facility infrastructure data.
Before this thesis, the author developed a new *Pusher* component that employs a framework infrastructure and supports considerably more data sources through plugins than previous *Pushers*. The framework structure of the new *Pusher* is presented in the following. Its functionality and implementation as well as its underlying design goals are described.

4.1 Design Goals

The very first design goal for the new *Pusher* component (only called *Pusher* in the following) is to be as easy to use as possible for all involved stakeholders. They are as following, ordered for their priority:

1. User,
2. Developer, and
3. Maintainer.

For users, the complexity to setup and configure *Pusher*, also during runtime, should be as low as possible. Possibilities for the user to misconfigure *Pusher* should be strictly avoided. Note that the term user usually refers to HPC system operators in the *DCDB* context and not to the actual HPC system users. Most commonly, HPC system users lack the permission to install facility wide monitoring solutions, although they might be given limited access through exposure of *DCDB*'s outside interfaces. The term developer refers to the "using" developers, which want to utilize the *Pusher* framework for their needs, e.g. by creation of a custom plugin. Maintainers are the "maintaining" developers behind the *DCDB* project. For developers, all interfaces should be powerful enough to satisfy their needs and at the same time simple to use. They should be able to build upon the framework with only minimal effort but without them having to cut back their requirements. Maintainers require the whole *DCDB* project including the *Pusher* part to be comprehensible at all times. Interfaces should be expandable for future requirements and framework-internal revisions should only require minimal expenses. All of the stakeholders unites the need for complete and comprehensible documentation.
Furthermore, the same design principles as stated in Section 2.1 for the *DCDB* project apply for *Pusher*. The design should be modular and holistic, while the functionality must obey the push-principle. *Pusher*'s implementation shall realize *Sensors* as smallest possible unit and must use MQTT to publish its acquired sensor data.

```
global {                              ;general settings
    mqttBroker 127.0.0.1:1883         ;address of CollectAgent
    mqttprefix /test                  ;global MQTT prefix (can be overwritten)
    verbosity 3                       ;verbosity level of log messages
}

restAPI {                             ;settings for the integrated RestAPI
    address     127.0.0.1:8000        ;address to listen on
    certificate ca-cert.pem           ;settings for HTTPS encryption
    privateKey  ca-key.pem
    dhFile      dh2048.pem

    user admin {                      ;user which is allowed to make requests
        password  admin
        PUT                           ;is allowed to do PUT requests
        GET                           ;is allowed to do GET requests
    }
}

plugins {                             ;plugins that are loaded on startup
    plugin caliper {
        path    ./path/to/binary
        config  ./path/to/config
    }
}
```

Listing 4.1: Excerpt from a configuration file for *Pusher*.

4.2 Functionality

From a user's point of view, *Pusher* allows for configuration of framework settings through a configuration file. It uses Boost's custom *INFO* file format [35]. An exemplary configuration file for the *Pusher* framework can be seen in Listing 4.1. Selected *Pusher* options can also be set via command line parameters passed on start-up. Usage of the framework as well as all configurable options for *Pusher* are documented in a `README` file within the code repository [22].
Pusher ships with plugins for eleven different data sources. More details on them can be found in Chapter 5.
Further on, *Pusher* runs a RESTful API (RestAPI) [8] that allows for runtime configuration of the plugins. The RESTful API can be leveraged by developers to control *Pusher* from external applications. For example, one could automatically halt certain plugins to avoid inference with other software.
For developers, the framework provides all infrastructure to develop custom plugins. Plugins are realized as shared libraries. Users only need to provide appropriate configuration files for the plugin. The framework will then take care of loading and running the plugin, as well as making it accessible through the RestAPI. Plugins will get periodically invoked according to their configuration to poll their data. *Pusher* will take care of the acquired data to be forwarded to the *Collect Agent*. To ease the process of developing plugins, enforce a uniform plugin structure, and eliminate the need to repeatedly write the same code skeleton, plugin generator scripts are included. On invocation they generate all source files required for a new plugin and fill them with the necessary code structures.

Developers only need to take care of the constituting plugin code parts pointed to by the `TODO` comments.
For maintainers, code is documented as complete and comprehensible as possible. *Doxygen* [36] is used to create an interconnected HTML documentation.

4.3 Implementation

The *DCDB* project including *Pusher* is written in C++11 and publicly available as open source [22] under the GNU GPL license. To not re-invent the wheel for all functionality, *Pusher*'s implementation employs a set of external libraries. They are listed in Appendix A. Further on, the usage of external libraries for certain functionality greatly increases the maintainability of *Pusher*'s actual code base.
The *Pusher* framework itself consists of five rather loosely coupled components (compare Figure 4.1):

- *Configuration*,
- *MQTTPusher*,
- *PluginManager*
- *RestAPI*, and
- *dcdbpusher* ("main"-method).

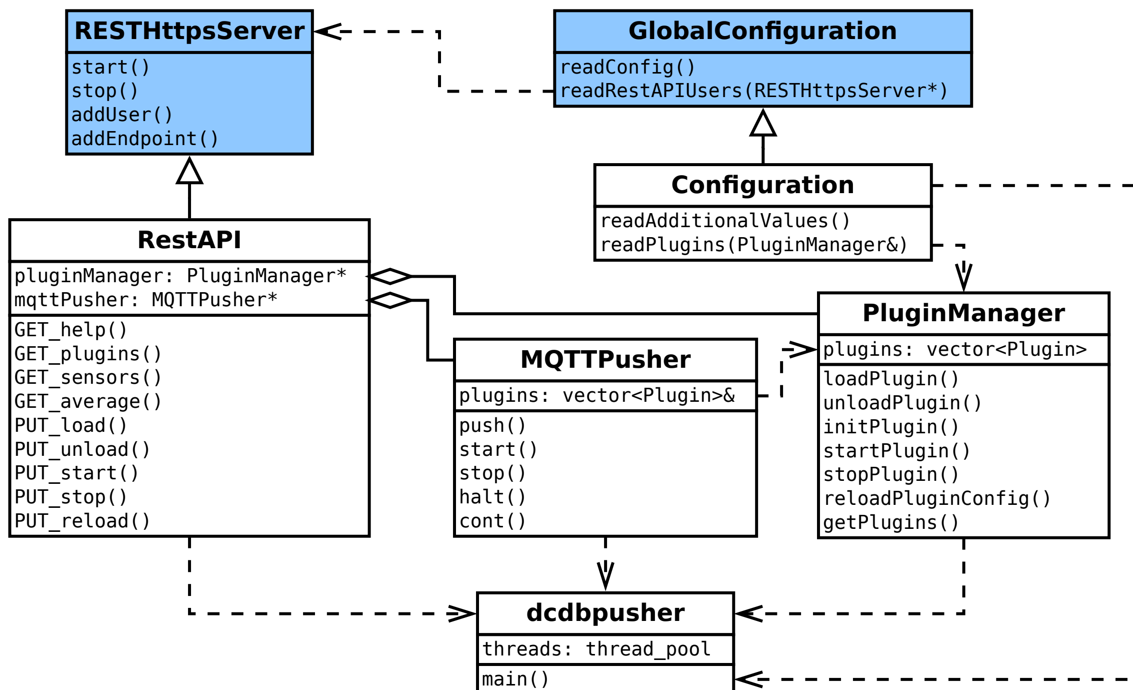


Figure 4.1: UML class diagram depicting the components of the *Pusher* framework and their relations. Classes with blue background are global to the *DCDB* project. For the sake of simplicity, implementation details are left out.

PluginManager

Central to the framework is the *PluginManager* component. It is responsible of administering the framework plugins. The corresponding `PluginManager` class offers various functionality to operate on single plugins. A new plugin can be loaded, i.e. its shared library is opened, making it accessible for further operations. Opposite, plugins can also be unloaded, i.e. its dynamic library is closed and completely unlinked from the framework. After a new plugin was loaded, it has to be initialized. Initialization leaves the plugin in a stopped state, meaning it is fully operational but currently does not collect data. After initialization a plugin can be started, i.e. its sensors start collecting data, and stopped again any number of times. To allow for adaption of a plugin during runtime without the need to fully unload and load it again one can alter the plugin's configuration file at will and then reload it during runtime. The new configuration file is read in and the plugin is set up accordingly, inducing a short interruption of the plugin's data collection. During a typical plugin lifecycle, operations are called in the following order:

1. `loadPlugin()`
2. `initPlugin()`
3. `startPlugin()`
4. (optional: `reloadPluginConfig()`)
5. `stopPlugin()`
6. (optional: go back to 3.)
7. `unloadPlugin()`.

One can also retrieve direct access to all currently loaded plugins via *PluginManager*'s `getPlugins()` method and operate on a plugin's components directly.

MQTTPusher

MQTTPusher is the actual "pushing" component sending all data as MQTT messages to the *Collect Agent*. The main functionality of *MQTTPusher* is implemented in the `push()` method of the `MQTTPusher` class which is portrayed in Listing 4.2. It is run by its own thread which continually cycles over all loaded plugins in an endless loop. Initial start and final termination is controlled via start and stop methods which set the `_keepRunning` flag accordingly. Within one cycle all sensors of a plugin are accessed. Sensor data which was buffered since the last cycle is retrieved and used to construct a MQTT message with the sensor's MQTT topic. The message is then sent to *Collect Agent*. For MQTT communication the open source Mosquitto [37] library is used. To reduce overhead, `MQTTPusher` keeps a reference to `PluginManager`'s plugins instead of retrieving them each cycle via `getPlugins()`. In doing so, `MQTTPusher`'s plugin list is also up to date at all times. However, modifications made to the plugin list from elsewhere, namely (un-)loading plugins, or plugin-internal changes to its groups or sensors require the `MQTTPusher` to be paused beforehand to avoid race conditions. For this reason, functionality to temporarily halt and later on continue the execution of the primary `push` cycle is offered.

```
push() {
    while (keepRunning)
        foreach (p : plugins)
            foreach (g : p->getSensorGroups())
                foreach (s : g->getSensors())
                    msgBuf = s->getReadings();
                    topic  = s->getMqttTopic();
                    mosquitto_publish(msgBuf, topic);
                end
            end
        end
    end
}
```

Listing 4.2: Pseudocode of `MQTTPusher`'s primary `push` method.

RestAPI

The primary user interface during runtime is the RestAPI offered by *Pusher*. As other components of the *DCDB* project offer a RestAPI, too, common functionality is merged in the project global `RESTHttpsServer` class. The class is not instantiable on its own but only forms a base class with common functionality. The common base class leverages the Boost.Beast library [38] to provide a HTTPS server for RestAPIs. Derived classes only have to implement their endpoint functionality and register the endpoints as well as the associated endpoint handler function with the `RESTHttpsServer`. In this context, an endpoint specifies an unique functionality offered by the RestAPI. An endpoint is identified by its unique URI. If a client makes a HTTPS request to an endpoint, the `RESTHttpsServer` takes care of the server-client communication like accepting the connection, sanity checks, error handling, authenticating users, checking user access permissions and determining the correct API endpoint. On success, the registered handler function is called and provided with all request parameters. The handler has to process the query and return a HTTPS response which will be dispatched from the `RESTHttpsServer`. Pseudocode of the whole procedure can be seen in Listing 4.3. Internally, functionality to start/stop the server threads, i.e. controlling availability of the RestAPI, add authorized users and add new RestAPI endpoints is provided to external and derived classes.
Pusher's `RestAPI` class derives from the common `RESTHttpsServer` class. It currently provides nine endpoints, each realized by its own handler. They are listed in Table 4.1. The *RestAPI* component provides five PUT endpoints wrapping functionality of the *PluginManager*. In doing so, *PluginManager*'s functionality gets accessible from the outside during runtime. To realize the PUT endpoints, `RestAPI` keeps a pointer to the `PluginManager` and `MQTTPusher`. Access to the latter one is required whenever the plugins are modified and `MQTTPusher` has to be paused to avoid race conditions. The other four endpoints are associated with GET methods and are of informational character. They respond with a small endpoint cheatsheet, currently loaded plugins, all sensors of a plugin and the average of one sensor's last readings respectively. The GET endpoints allow for user-space access to *Sensor* data through a single interface at runtime. This increases security, as no access to the actual underlying data sources has to be granted. Instead, a user only has to be supplied with appropriate credentials for the RestAPI. Further on, data access is simplified in general for external parties, as all data can be retrieved through the same RestAPI.

```
connectionHandler(socket) {
    connection = socket.accept();
    request    = connection.receive();

    //check provided user credentials
    if( !userValid(request.credentials) ) {
        //fail: unauthorized request
    }

    //look up requested endpoint within
    //all registered endpoint handlers
    reqHandler = endpointHandlerMap[request.endpoint];

    if( reqHandler == NULL ) {
        //fail: no handler registered for requested endpoint
    }

    if( reqHandler.method != request.method ) {
        //fail: REST methods (e.g. GET or PUT) of request
        //      and endpoint handler do not match
    }

    //delegate further processing to endpoint handler
    response = reqHandler.fun(request);

    connection.send(response);
    socket.close(connection);
}
```

Listing 4.3: Pseudocode of `RESTHttpsServer`'s functionality that will be invoked whenever a request is received.

RM[1]	URI Path	Description
GET	`/help`	Return a cheatsheet of possible REST API endpoints.
GET	`/plugins`	List all loaded *Pusher* plugins.
GET	`/sensors`	List all sensors of a specific plugin.
GET	`/average`	Get the average of the last readings of a sensor. Also allows access to *Wintermute*'s analytic sensors.
PUT	`/load`	Load and initialize a new plugin but do not start it. Use the `/start` request to kick off the plugin's data collection.
PUT	`/unload`	Unload a plugin, removing it completely from *Pusher*. To use the plugin again one has to `/load` it first.
PUT	`/start`	Start a plugin, i.e. its sensors start polling.
PUT	`/stop`	Stop a plugin, i.e. its sensors stop polling.
PUT	`/reload`	Reload a plugin's configuration (includes fresh creation of a plugin's sensors and a plugin restart).

[1] RestAPI Method

Table 4.1: Overview of all RestAPI endpoints supported by *Pusher*.

Configuration

The component responsible for reading in configuration files is the *Configuration*. Its implementation class is similar to the `RestAPI` as it is based on a project global parent class called `GlobalConfiguration`. Many settings, like log verbosity, temporary file directory, and MQTT prefix are common to multiple components of *DCDB*. This includes the RestAPIs' server settings, as all of them are based on the common `RESTHttpsServer` base class. The logic to read in the common settings is bundled in `GlobalConfiguration`. The derived `Configuration` class only has to provide a method to read in additional values. This method parses *Pusher* specific configuration settings, e.g. the address of the MQTT broker aka *Collect Agent*, and will be called from the parent class after common settings have been parsed. Separated from the general setting parsing is the code to read in the users and their corresponding permissions of the *RestAPI* and *Pusher*'s plugin configuration. Both parts rely on other components, namely `RESTHttpsServer` and *PluginManager*, to be instantiated first as the parsed values will be directly set within the components. Therefore, those settings cannot be parsed at the very beginning together with the general settings but only at a later point of time.

Dcdbpusher

The last component, containing the `main()` method, is *dcdbpusher*. It initiates all other components, starting with *Configuration*. After the global configuration is read in, *PluginManager* is invoked and all initial plugins loaded and configured. *Dcdbpusher* sets up a pool of threads which will execute the data reading tasks issued by the plugins (see Section 5.2 for details). The thread pool allows for asynchronous execution of data readings. Hence, read intervals of *Sensors* can freely be adapted to the underlying data source. The number of associated threads in the pool can be configured depending on the number of data sources and their read cycle as well as the hardware *Pusher* is run on and therefore allows for scalability. Also, *dcdbpusher* kicks off *RestAPI* and *MQTTPusher* and takes care of a graceful shutdown on termination.

An overview of all *Pusher* components, including those of the plugins, can be seen in Figure 5.2.

Logging

During the development of *Pusher* a logging infrastructure was integrated and adapted to other components of the *DCDB* project. The logging infrastructure is based on the Boost.Log [39] library, which allows for thread-safe efficient logging with different levels of severity. A component requires only a local instance of a boost logger to participate. Convenient macros for recording messages are provided. The infrastructure logs all messages to a file and the terminal. One can specify the verbosity of the log messages, i.e. set the severity level a message must exceed to be logged.

5 Pusher Plugins

The new *Pusher* framework only provides the general infrastructure to push *Sensor* data towards other *DCDB* components. It does not implement any logic to acquire data itself. Instead, it relies on plugins for this task. The plugin-based approach allows to achieve modularity and extensibility. Depending on the machine *Pusher* is deployed on, different data sources can be tapped into by loading the appropriate plugin. Also, future data sources can easily be integrated as only a plugin that implements the data reading functionality has to be developed.

5.1 Functionality

Currently, eleven *Pusher* plugins for different data sources are provided. Out of the eleven plugins, four were developed by the author before this thesis. Additionally, the preceding *Pusher* implementations for the first three data sources have been ported to plugins for the new framework. During the thesis work itself, the complex *Caliper* plugin to gain user application introspection data was developed. The remaining three plugins were developed in parallel by *DCDB* collaborators. An overview of all plugins can be seen in Table 5.1. Plugins that were developed by the author are described in detail below.
Plugins either acquire data locally from the same host they are run on (in-band) or from remote machines (out-of-band). Just like the framework, plugins are configured through individual configuration files in Boost's *INFO* file format [35]. One can specify *Sensors* as smallest possible unit, *SensorGroups* that bundle multiple *Sensors*, and optionally *Entities*. More details on those components can be found in Section 5.2. The configuration files also allow to specify template blocks. These are equal to normal *Sensors*/*SensorGroups*/*Entities* but will not be actually instantiated. Instead, they can be used to set default values and allow one to skip the repeated specification of unvarying configuration parameters. Usage of plugins as well as their individual configurable options are documented in the code repository's [22] `README` file. An exemplary excerpt from a plugin configuration file can be seen in Listing 5.1.

5.2 Implementation

Plugins are realized as shared dynamic libraries. Just like the *Pusher* framework they are written in C++11 and are available as open source [22]. A number of plugins relies on third-party software. See Appendix A for more information. Each plugin is based on the same general component architecture that is presented in the following.
A plugin includes the components as follows (compare Figure 5.1):

- *SensorBase*,
- *SensorGroup*,
- *Entity* (optional), and
- *Configurator*.

```
global {                            ;Plugin global settings
    mqttprefix /plugin              ;MQTT prefix
                                    ;(overwrites Pusher global prefix)
}

template_entity temp1 {             ;Template entity which is not used
                                    ;in live operation.
    ...                             ;Here go entity attributes

    group g1 {
        interval    1000            ;Group wide attributes
        minValues   3
        mqttPart    /aa             ;Will be appended to global prefix

        sensor s11 {
            mqttsuffix  /s11        ;Appended to prefix and group part,
                                    ;must result in an unique MQTT topic.
            ...                     ;Usually the sensor would require
        }                           ;additional attributes.

        sensor s12 {
            mqttsuffix  /s12
            ...
        }
    }
}

entity ent1 {
    default     temp1               ;Use temp1 as template Entity

    group g2 {                      ;ent1 has now two groups (g1 and g2)
                                    ;with a total of 3 sensors
        sensor s21 {                ;(s11, s12, s21).
            mqttsuffix  /s21
            ...
        }
    }
}

entity ent2 {                       ;Enity with only one sensor
    mqttPart    /ent2               ;Entities can also add an part
                                    ;to the MQTT topic.
    single_sensor s1 {              ;Specify a single sensor that
        interval    2000            ;does not belong to a group.
        mqttsuffix  /s3
    }
}
```

Listing 5.1: Excerpt from a configuration file for a generic plugin that uses the optional *Entity* component.

Plugin	Data source	In-band	Out-of-band	Entity	DBA[1]	Ported
BACnet	Building Automation and Control Networks [13]		X	X	X	
Caliper	Lawrence Livermore National Laboratory's (LLNL) *Caliper* [9] tool	X			X	
GPFSmon	General Parallel File-System [14] monitoring data	X				
IPMI	Intelligent Platform Management Interface [15]		X	X		X
MSR	Intel model-specific register [16]	X				
OPA	Intel Omni-Path [17]	X			X	
REST	RESTful APIs [8]		X	X	X	
PerfEvent	Linux *Perf Events* [18]	X			X	
ProcFS	Linux `/proc` file system [19]	X				
SNMP	Simple Network Management Protocol [20]		X	X		X
SysFS	Linux `/sysfs` file system [21]	X				X

[1] Developed by the author

Table 5.1: Overview of all plugins and their features.

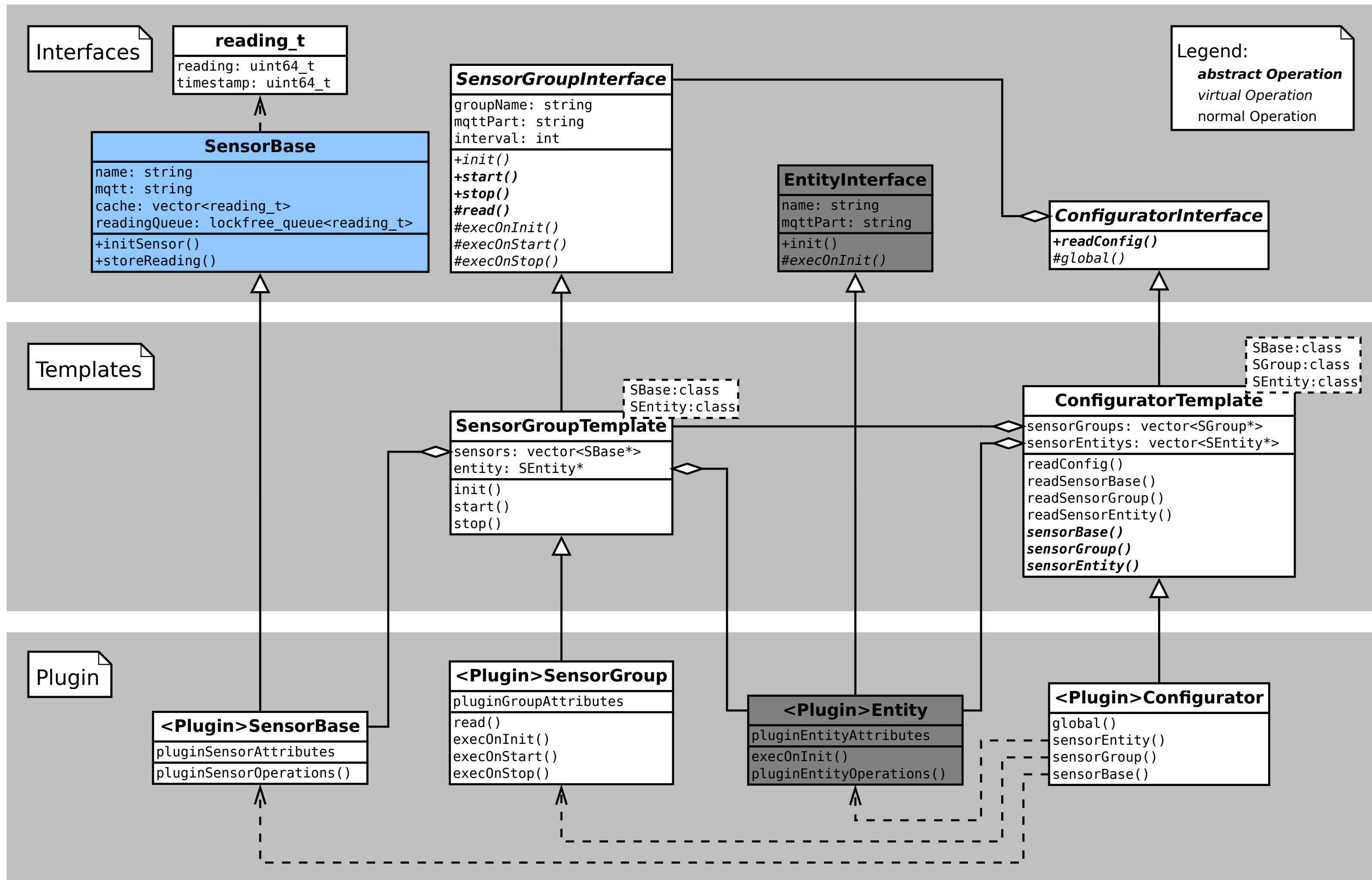


Figure 5.1: UML class diagram depicting all involved plugin components and their relations. Classes with blue background are global to the *DCDB* project. Classes with dark gray background are optional. For the sake of simplicity, implementation details are left out.

Components are implemented according to the *Template Method* design pattern [40]. The design pattern is useful to keep common functionality in a general interface class while allowing or even forcing derived concrete classes to customize parts of the functionality. Therefore it perfectly suits the realization of plugins. Each plugin part is made up of an abstract interface class and a concrete, plugin-specific implementation class. The interface defines a component's functionality that is accessible to the framework. Also, common code is implemented in the interface. Plugin specialization is achieved by abstract and/or virtual protected methods which are implemented in concrete plugin classes and are called from the public interface functions. If an operation has to be overwritten by a plugin it will be pure virtual, aka abstract. Methods that can be optionally specialized from a plugin are virtual and the interface provides an usually empty default implementation. The concrete implementation classes contain all plugin specific code, namely all code that is required and/or specific to access the plugin's target data source.
During implementation of the first plugins for *Pusher* it became obvious, that the *SensorGroups* and *Configurators* of all plugins share common code. Unfortunately, some code details rely on plugin specifics, e.g. the presence of an entity, which prevents unification in the interfaces. Therefore an additional class layer of templates is introduced. The templates allow to join the common code while keeping it plugin specific at the same time by using plugin implementation classes as template parameters. To account for the optional entity component, partial template specializations are implemented. They come into effect if no class was specified for the entity template parameter. The templates cannot fully replace the abstract interfaces, however, as `C++` does not allow to reference a template class in general without plugin specialization. Therefore, the use of templates as interfaces to the framework is prohibited. The templates rather extend the interface level of the *Template Method* design for an additional layer.

Configurator

The *SensorBase*, *SensorGroup* and *Entity* components are responsible for the plugin's data collection. Each of them has a name attribute for human readable identification. The *Configurator* component is the primary access point to a plugin. Other components of a plugin can only be accessed through the plugin's *Configurator* (operations not shown in Figure 5.1). When a plugin is loaded the first and only component created by the *Pusher* framework is the plugin *Configurator*. The *Configurator* is subsequently responsible of reading in a plugin's configuration file and creating the other components accordingly. If a plugin is requested to reload its configuration during runtime, the *Configurator* is the only component to persist. All other plugin parts are recreated according to the new configuration file. The `ConfiguratorInterface` only implements logic to parse common plugin attributes, like the plugin-global default MQTT prefix. More specialized logic to read in the *Entity*, *SensorGroup* and *SensorBase* structure is realized in the template. The concrete plugin `Configurator` implementation has to parse all plugin-specific attributes.

SensorBase

SensorBase realizes the *Sensor* concept, i.e. a *SensorBase* represents the smallest possible data source unit within the *DCDB* project. It may represent the number of cache accesses, branch mispredictions, CPU temperature, or the energy consumption of a single power

outlet. As this unit is also used by other components the general base class has a project global scope. The most important attributes of the `SensorBase` are its assigned MQTT topic and the storage for data readings. The MQTT topic is usually made up of multiple parts. Each hierarchy level (plugin, *Entity*, *SensorGroup*) may add an identifying MQTT part. Those parts make up the prefix which is completed from a custom MQTT suffix assigned to each *SensorBase*. A *SensorBase* has two storages for data readings, the cache and the reading queue. A data reading in turn consists of two components, the actual read value and the corresponding timestamp. A single data reading is produced every read cycle (see also the *SensorGroup* paragraph below). The cache is a ring buffer which stores all data readings for a configurable amount of time. It is used to speed up internal operations like serving *RestAPI* requests or to enable the *Wintermute* analytics framework for fast online analysis without making a detour to the *Storage Backend*. The reading queue contains all readings which should be dispatched by the *MQTTPusher*. The reading queue may be only a subset of the cache. As only one data producer (the *SensorBase* itself) puts data into the queue while the *MQTTPusher* is the only data consumer, it is implemented as efficient lock free single-producer single-consumer queue.
Plugin implementations of the *SensorBase* usually do not have to provide any special logic except plugin specific attributes and their associated `get` and `set` methods. Therefore no additional template layer is required for this component.
The `SensorBase` base class offers additional logic through its `storeReading()` method for more fine grained processing of a new data reading:

- Storage of only delta values, i.e. instead of the absolute reading just the difference to the preceding value will be stored. Selected plugins, e.g. for *Perf Events*, that are known to only provide monotonically increasing data have this feature activated by default.
- Skipping of constant values. Values that did not change since the last read cycle will be skipped completely from storing. If sensors, e.g. error counters, are known to change only occasionally this can save storage space.
- Subsampling only stores every n-th value. Can be used to reduce pressure on the network and/or storage space as only every n-th read value will be pushed. Naturally, this will also reduce granularity of data in the *Storage Backend*.

In all cases all readings will still be stored in the cache. The previously listed features only affect the reading queue. The additional logic can be individually dis- and enabled through settings in the configuration file.

SensorGroup

Within *Pusher* a *SensorBase* object never exists on its own. It is always part of a *SensorGroup*. *SensorGroups* aggregate one or more related *SensorBases* whose data sources will always be read together. This allows to correlate data readings of different *SensorBases*, like cache misses to overall cache accesses, and reduces overhead. The logic to read values from a plugin's data source is implemented within the *SensorGroup*. The read functionality will be periodically executed with a frequency depending on a configurable read interval. Each read cycle an asynchronous task to execute the *SensorGroup*'s `read()` method is issued to be picked up by one of the threads in the *Pusher* framework's pool. An exemplary implementation of the relevant `read()` method is depicted in Listing 5.2. The *SensorGroup* will iterate over all of its *SensorBases* and trigger a read of the corre-

```
read() {
    while (keepRunning)
        reading_t reading;
        reading.timestamp = now();

        //entity.openConnection();
        foreach (s : sensors)
            reading.value = acquireData(s.attributes);
            //reading.value = entity.acquireData(s.attributes);
            s.storeReading(reading);
        end
        //entity.closeConnection();
        sleep(interval);
    end
}
```

Listing 5.2: Pseudocode of a *SensorGroup*'s data reading functionality that is executed during runtime. Code in comments shows an alternative use case with an *Entity* involved.

sponding data source value. The value is then stored within the *SensorBase*.
A *SensorGroup* allows for some additional settings which will affect all of its associated *SensorBases*. To reduce network overhead caused by the *MQTTPusher* one can set a minimum number of readings a *SensorBase* should contain before they are processed into a MQTT message. This avoids network congestion by MQTT messages that only contain one data reading at a time. Naturally, this increases the latency until data gets available in the *Storage Backend* and can be accessed through *libdcdb*. Hence, this setting is a trade-off between latency and network overhead.
One can also instruct a *SensorGroup* to synchronize its read interval with other groups. In this case, the absolute timestamp of the next reading time will be rounded down to the next multiple of a *SensorGroup*'s read interval. Hence, all groups with the same interval will start their next read cycle at the same absolute timestamp. Synchronized data readings result in improved post-analysis, as data points do not have to be interpolated to correlate them. As the clocks of all nodes, and therefore the time of all *Pushers* running on them, are synchronized via the *Network Time Protocol* (NTP) [41], the data reading overhead of synchronized *SensorGroups* will occur on all instances (nodes) at the same point of time. Synchronized readings between nodes, however, only work for sufficiently large read intervals, as the nodes' clocks are subject to drift and the NTP synchronization may not be precise enough for fine-grained read intervals.
To accommodate the need for single independent *Sensors*, a special *SingleSensor* option can be used in the configuration files. A *SingleSensor* block allows to specify the attributes for a *SensorBase* and *SensorGroup* at once. Internal, the *Configurator* takes care of creating a *SensorGroup* accordingly which contains only one *SensorBase*.

Entity

A plugin can contain a second, optional aggregation level above *SensorGroups*, namely *Entities*. *Entities* are intended for plugins that gather their data from remote locations, e.g. power delivery units that are accessible via network or the Baseboard Management Controller (BMC) of a remote server. If multiple *SensorGroups* need to communicate with

the same server an *Entity* can be introduced. As *Entities* are an optional component, they do not control associated *SensorGroups*, but instead a *SensorGroup* will get a pointer to its associated *Entity* set from the plugin's *Configurator*. The *Entity* can then provide logic to communicate with the remote server which has to be used from the *SensorGroups* to read their data. This way, the *Entity* is the solely controller of the connection to "its" server and can eliminate race conditions if multiple *SensorGroups* want to read data at the same time.

A complete overview of all framework and plugin components can be seen in Figure 5.2.

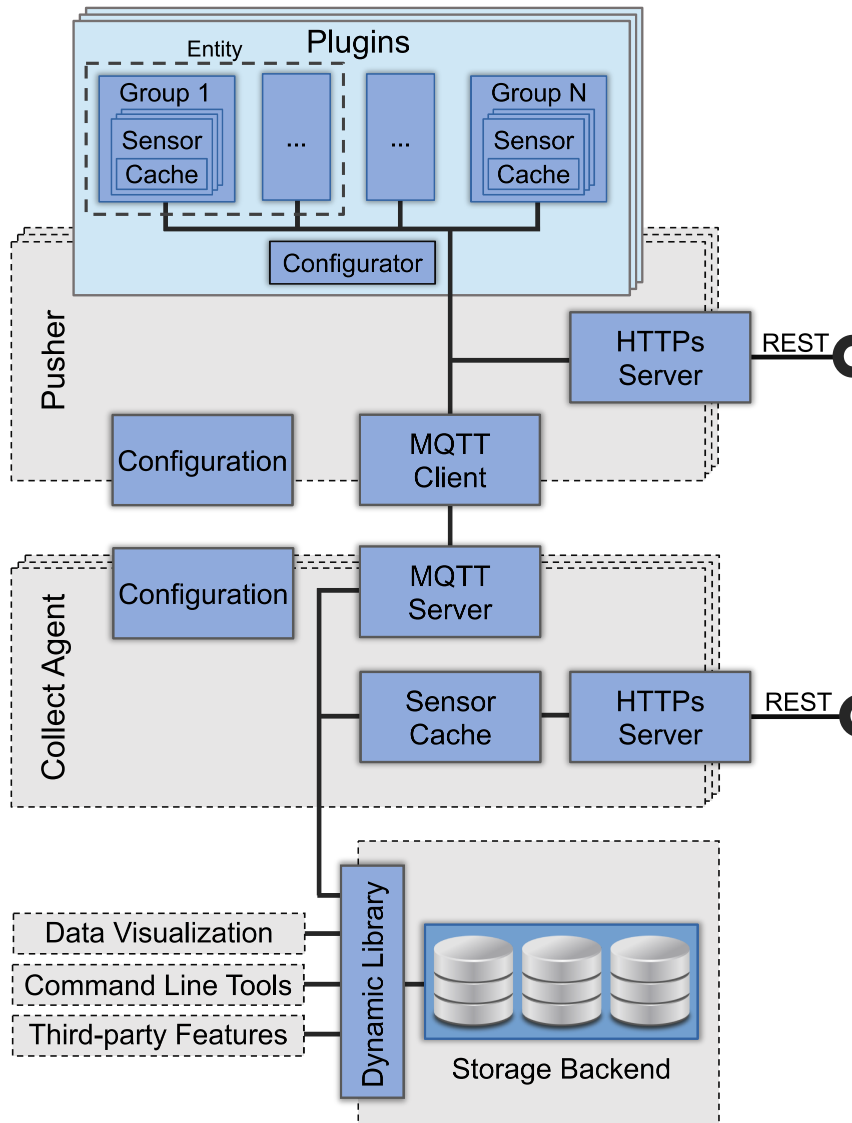


Figure 5.2: Overview of all components of the *Pusher* framework and its plugins highlighting the data flow. This figure also includes an introspection into *Collect Agent* and *Storage Backend*.

5.3 Plugin Details

All plugins which are developed by the author are implemented according to the structure presented above. Still, a handful of plugins also require implementation of special features that are unique to them and are therefore not accounted for in the framework but are kept in the plugins. Those special features are presented in the following sections. The *Caliper* plugin, that allows to gather application introspection data, is highlighted in particular in Chapter 6 as it does not fit the common value-timestamp data point format and uses a hybrid approach for data acquisition.
The fact that this highly diverse set of adaptions is possible within the plugins demonstrates the flexibility and permissiveness of the *Pusher* framework.

5.3.1 IPMI, SNMP, SysFS

The three plugins *IPMI*, *SNMP*, and *SysFS* are historically the very first data sources for *DCDB*. Before the start of the *Pusher* framework and its plugin based architecture they existed as separate binaries. Functionality that is unified by *Pusher* as of now was implemented by each of them separately, resulting in major code duplication. During the development process of *Pusher* they were ported and rewritten to be the first compatible plugins.

The *IPMI* plugin allows to query data from a server mainboard's Baseboard Management Controller (BMC) via the Intelligent Platform Management Interface (IPMI) [15] protocol. The *SNMP* plugin offers access to data from devices that are accessible via the Simple Network Management Protocol (SNMP) [20], e.g. routers, switches, and printers. *IPMI* and *SNMP* both make use of the *Entity* feature to access their remote data sources. The *Entity* abstracts one host machine and is employed by one or multiple *SensorGroups* to read remote data. It allows to reuse the same established connection to the remote machine for multiple queries. This significantly reduces overhead and delay, as initialization of an IPMI or SNMP connection is comparatively slow and expensive.

The *SysFS* plugin acquires data by parsing the content of files in the local `/sysfs` file system [21]. During its porting, it was enriched with support to parse files with *regular expressions* (*regex*) [42]. In fact, the *SysFS* plugin could be (ab-)used to parse arbitrary files to acquire data and make use of its *regex* feature.

5.3.2 BACnet

The *BACnet* plugin implements functionality to acquire data from devices that support the Building Automation and Control Network (BACnet) [13] protocol. It is deployed in data centers to control and monitor infrastructure like water pumps, cooling towers, or air handlers. The *BACnet* plugin uses the *Entity* feature to query its remote data sources. Like most of the plugins that are implemented to acquire data via a specific protocol, *BACnet* uses an external open source library that provides a protocol communication stack (see also Appendix A). The library used for *BACnet*, however, is not reentrant, i.e. it does not support more than one connection at once. Therefore the number of *BACnet Entities* is also limited to one. This property is ensured by the plugin's *Configurator*. All *BACnet* connections have to pass through the single *Entity* instance. In turn, the *Entity* supports connections to multiple differing remote devices.

5.3.3 OPA

To gather data from Intel Omni-Path (OPA) [17] network interfaces the *OPA* plugin is implemented. It allows to read data from the various status counters of the local machine's OPA network interfaces. Such status data comprise for example the number of network transmit or receive errors, if the network link is up or down, and the overall transmitted data packets in general.

5.3.4 REST

The *REST* plugin allows to query RestAPIs [8], parse their response, and match the parsed data with *Sensors*. RestAPI queries are dispatched through *Entities*. Each *Entity* is usually associated with an unique host and one *SensorGroup* matches a specific request. The response may contain data for multiple *Sensors*. The corresponding *SensorBases* are part of the requesting *SensorGroup*. Each read cycle, the *SensorGroup* dispatches its request through its associated *Entity* to the target host, parses the response, and serves the data readings of all associated *SensorBases* with the parsed data.

5.3.5 PerfEvent

The *Perf Events* plugin supports data acquisition from CPU internal performance counters [18]. *Perf Events* allow for high frequency, low level sampling of a CPU core's key figures. As a single processor can consist of dozens of CPU cores, one node can have multiple processors, and a HPC system has thousand of nodes, the amount of data produced by *Perf Events* is huge. Although the *DCDB* infrastructure can handle the tremendous data flow per se, requests to reduce the pressure on the required storage space were brought forward. To meet the requests, an option is implemented that allows to accumulate the data of CPU cores by specifying an aggregation value. All CPU core numbers that are a multiple of this value will be aggregated, i.e. only one *Sensor* exists for all aggregated CPU cores that stores the sum of all of their readings. This is especially useful to aggregate values of logical cores as they occur in processors that implement simultaneous multithreading (SMT) [43]. Further on, *Perf Events* allows to limit the data acquisition to certain CPU cores. Cores that are permanently pinned to management tasks may therefore be excluded from the data sampling.

6 Caliper: A Hybrid Plugin

For a fully holistic monitoring of HPC system facilities one also requires introspection capabilities into user applications. This allows to automatically provide users with monitoring data correlated to their application code without them taking further actions. Also, the HPC system operators have a fair interest in the application performance data. Most HPC software makes use of precompiled libraries provided by the operators. Access to application performance data would allow the providers to tune critical functions within those libraries. This way, all of the users profit as own optimizations to those libraries are out of their area of influence. Also, access to user application performance data allows for a better requirements analysis when purchasing new hardware.
The intention was to develop a plugin for *Pusher* that provides user application introspection data. While doing so, the solution should obey two primary design goals:

1. minimal user application overhead, and
2. no requirement for application developer involvement.

Acquisition of application introspection data always incurs overhead. It should be kept to a minimum to avoid negative impacts on an software's runtime and increase acceptance among users. To automatically gather data from all running applications without relying on the good will of the software developer, no action should be required from their side to gather application data.
A *Pusher* plugin itself has no access to an application's introspection data by default. In fact, it would be possible for the plugin to "hijack" user binaries. However, this would require significant development effort and increased process privileges which induces a security flaw. Instead, to acquire introspection data, a hybrid solution is developed. On user application side the *Caliper* [9] performance analysis toolbox developed at LLNL is employed to gather introspection data in the first place. A custom service developed for the *Caliper* framework allows to access the data and forward it to *Pusher* via inter-process communication (IPC). On *Pusher*'s side, a special plugin is implemented that will receive the application data and therefore make it available to the *DCDB* infrastructure.

6.1 The Caliper Framework

Caliper per se is a framework for program instrumentation and performance measurements and is intended to be used by application programmers for performance analysis and optimizations.
The most basic data processing unit within *Caliper* is a snapshot. It contains all information provided by the application developer and the *Caliper* framework about the user program at a certain point of time. Main features of the framework stem from its services which digest snapshots. Depending on their functionality a service will be invoked during different stages of a snapshot's lifecycle. Services can trigger the creation of a snapshot, add additional information, process snapshots at runtime or store them for later analysis. The framework as well as all required services are compiled into a shared

library which has to be linked with binaries that want to use *Caliper*'s functionality. Runtime configuration is achieved by setting environment variables or by configuration files. *Caliper*'s original target user group are application developers. Its strength is performance analysis of applications via instrumentation. Annotations have to be incorporated by developers into an application's source code and trigger snapshot creation synchronously whenever they are encountered if the *Event* service is activated. Although supported in *DCDB*, the *Event* data does not suffice as sole data source. It relies on the user to fully annotate its code for holistic application introspection, which may require excessive effort and cannot be enforced after all. Therefore, the *Sampler* service is used for primary snapshot creation to acquire application introspection data. *Caliper*'s *Sampler* service allows for asynchronous snapshot creation with a configurable frequency, i.e. it provides low overhead analysis via a sampling approach. Even more important, apart from initialization of *Caliper* no further software adaption is required on application side to start the *Sampler* service. This way, *Caliper*'s application introspection capabilities can be integrated into the software almost invisible to users and require as little involvement from them as possible. The sampling approach may produce less accurate profiling results than precise instrumentation. This drawback is considered of little weight as HPC software usually correlates with hour-long runtimes that allow for enough samples to form a precise statistical profile of the application. Also, by adjustment of the sampling frequency, fast and fine grained control of the induced overhead is possible.

6.2 Use Cases

For the implementation of the *Caliper* plugin two use cases are considered.

Sampler

In the first case, the goal is to gather information on how much time an application spends in which function. This information can point the user directly to the most critical methods that offer the most optimization potential. Also, the data can help system operators to identify frequently used libraries. Special focus can then be put on the most used libraries for further support.
For this use case, the *Sampler* service is used to gather sampling introspection data from every thread. The data can be sorted into a histogram to directly present the most encountered function. An exemplary histogram for a single node High-Performance Linpack (HPL) [44, 45] run can be seen in Figure 6.1. One can see there, that primarily the *Caliper*-enriched HPL binary *xhpl_cali* itself and the *Intel Math Kernel Library* (MKL) [46] are encountered during runtime. In particular, the AVX-512 [47] version of the MKL library is used as the HPL run is conducted on a Skylake system with AVX-512 support. The fact, that by far the most samples are taken from MKL's "dgemm_kernel" method meets the expectations, as it makes up the main computation of HPL and therefore should also require the most execution time.
This use case is the default, as it allows almost invisible monitoring of application internals with minimal expenses for the user. One only has to initialize *Caliper* once in the application. Sampling on all threads is then automatically set up by the *Caliper* framework. In the future, it is intended to integrate the *Caliper* initialization invisibly to the user in every application. The long-term goal is to gather sampling data from all user software without requiring further involvement from the developers.

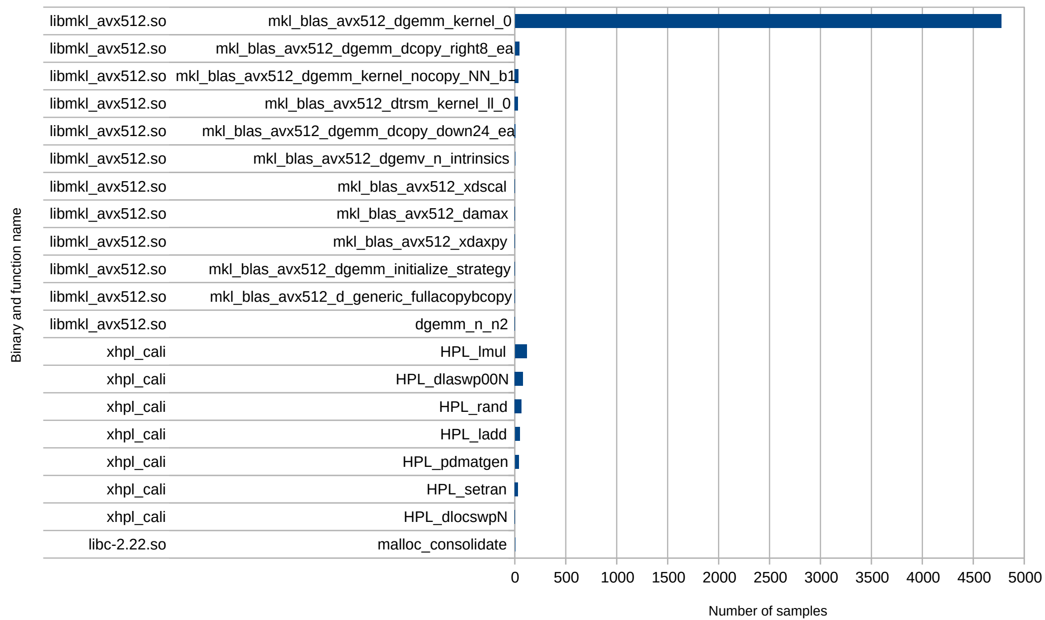


Figure 6.1: A histogram of the most meaningful function samples sorted by binaries, as acquired with the *Caliper* plugin for a single-node HPL run on an AVX-512 Skylake system.

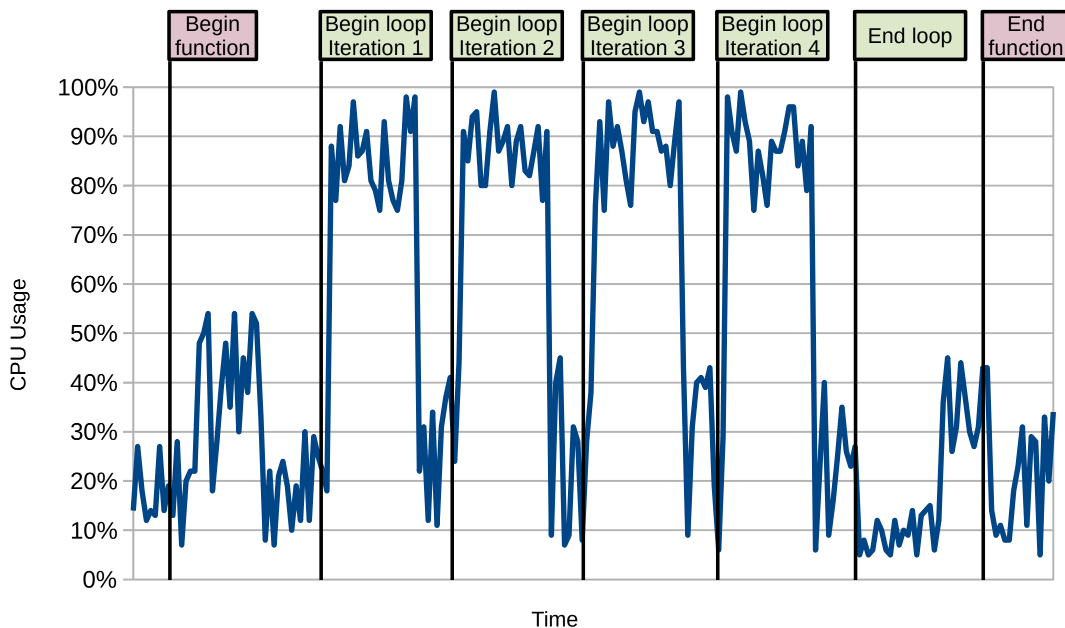


Figure 6.2: Visualization of a fictional use case where CPU monitoring data is enriched with *Event* annotations.

Sampler with Events

The second use case is an extension of the first. It is intended to additionally allow the user to enrich their application with *Caliper* annotations that can later be correlated with other monitoring data. This way, the user can easily comprehend the flow of their software and can correlate each section with detailed resource usage statistics. Also, the annotation data can be used to gain more fine-grained information than the first use case may provide as annotations allow to distinguish different sections within the same function. For this case, the *Event* service is used in addition to the *Sampler* as it triggers a snapshot upon encountering a *Caliper* annotation.
An exemplary depiction of such a use case can be seen in Figure 6.2. It presents the fictional correlation of function and loop annotations with CPU usage data. In this example, one can easily deduce that the loop iterations are by far the most CPU intensive parts of the functions. Or vice versa, the function parts outside the loop do not fully use the CPU. Such a deep introspection down to the loop iteration level can not be achieved with the *Sampler* use case alone.

6.3 Caliper-Pusher Communication

IPC between the custom *Caliper* service and the *Pusher* plugin (only called service and plugin respectively in the following) is realized via POSIX [48] shared memory and POSIX Unix domain sockets for initial setup. It is assumed that per compute node not more than one *Pusher* is run, but that multiple applications with *Caliper* integrated can be run in parallel, e.g. by shared node usage or execution of multiple MPI [49] processes per node. Therefore, several instances of the service may want to connect to the same plugin. To avoid inferences, each service instance uses a distinct shared memory file for communication with the plugin. File names are based on a common schema which involves the unique process identifier (PID) to avoid name collisions. The plugin continuously listens on a local Unix socket whose name is predefined and known to the service. On application start, the service sends its PID to the plugin. By receiving the PID on its listening socket, the plugin is informed about the new process and is capable to deduce the shared memory file name for the IPC. As both sides now can determine the file name all following communication is done via shared memory. The shared memory file contains a ring buffer to send data from the service to the plugin. An entry in the queue consists of a string that contains snapshot data from the application and an associated timestamp. In case of a *Sampler* triggered snapshot the data string comprises the name of the sampled function and its containing library. For *Event* snapshots the string contains the triggering annotation name and its optionally associated value. Shared semaphores are realized in the memory file to synchronize read/write accesses to the queue.

6.4 Caliper Service

The service attaches to the snapshot processing level of *Caliper*. It will be invoked every time a new snapshot is created and retrieves information of interest from the snapshot to forward it to the plugin. The service relies on four other services shipped with *Caliper* to function correctly:

- *Sampler*,
- *Event*,
- *Timestamp*, and
- *Pthread*.

The *Event* and *Sampler* services are required to trigger the snapshot creation in the first place. They can be either used concurrently or only one of them on its own.
The *Sampler* service provides the application's PC value right before the software is interrupted for snapshot creation. The PC allows to determine the program function name for the sample with the binary's symbol data. A symbol is the string name of an attribute, function, or other labels in the binary. Depending on debug options used during compilation, each binary comprises more or less exhaustive symbol data that allows to resolve a string name to a memory address or vice versa.
The *Event* service enriches a snapshot with an ID of the triggering annotation. The ID can then be resolved to the unique annotation name and its optional value.
The *Timestamp* service enriches every snapshot with the current timestamp of its creation, which is required to attain a complete value-timestamp data point for *DCDB*. The service is therefore required in any case. The *Timestamp* service, however, did previously not offer sub-second precision which is required for *DCDB*. Therefore, a patch for the *Timestamp* service to provide nanosecond timestamps was submitted to *Caliper*'s GitHub repository [50]. The patch was accepted and therefore sufficiently precise timestamps are provided as of now.
The *Pthread* service takes care of starting the *Sampler* service for every new thread that was created via POSIX threads (pthreads) [51], the core API for multithreading in UNIX. The service achieves this by using *GOTCHA* [52] to wrap *pthread_create* to create a *Caliper* thread scope for every new thread. *Pthread* must be enabled whenever the *Sampler* service is used to achieve fully holistic application sampling.
On invocation, the snapshot processing routine of the custom *Caliper* service retrieves the relevant data from a snapshot and processes it to a data string for the IPC queue. In case of *Event* triggered snapshots, the ID of the triggering annotation is retrieved. The ID will be resolved to the annotation's name and its optionally associated value. Resolution can be done through *Caliper*'s framework API. In case of *Sampler* snapshots, the custom *Caliper* service retrieves the provided PC value. The PC value will be resolved to the associated function name and its containing library (see below for details). The gathered function information will then be processed into a data string. In both cases, the custom *Caliper* service also retrieves the timestamp of the snapshot and the CPU number it is currently executed on. Assuming that thread pinning is enabled, one can safely expect that the current CPU equals the one the snapshot was created on. Otherwise, one has to trust that the thread was not rescheduled to another CPU in the meantime of snapshot creation and its actual processing in this service. The CPU value is added to the data string. Together with the timestamp a queue entry is then formed which is eventually written to the shared memory queue.
Due to address space layout randomization (ASLR) [53], an application is mapped to different memory addresses each execution. PC values of the *Sampler* service can therefore only be resolved to concrete function symbol names as long as the current memory mapping of the application is known. Consequently, PC values have to be resolved to function symbols at runtime. Therefore, the service reads all function symbol names and their corresponding addresses from its application binary and associated loaded libraries on startup. The data is parsed and stored in a local data structure (the "symbol index")

that is searchable for address values. PC values can then be easily looked up in the symbol index and resolved to actual function names during runtime.
Although it is not very common for HPC software, it may occur that new libraries are loaded during runtime. The new libraries include symbol data that is not yet present in the index. To account for this case, the service will rebuild the symbol index at runtime if a PC value could not be resolved to a symbol name. To limit the significant overhead of this operation in case a PC value persistently cannot be resolved, a temporal cooldown for the symbol index rebuild can be configured.

6.5 Pusher Plugin

The plugin consists of only one single *SensorGroup*. The single *SensorGroup* processes the data of all *Caliper* applications. Every read cycle it iterates over all known *Caliper* processes and their corresponding shared memory files. It will read all buffered data from the ring buffer and further process it before pushing it to the *DCDB* infrastructure. If consistently no new values are available in the buffer, the associated application is assumed to have terminated and will be removed from the plugin's internal list. This timeout based approach allows for guaranteed release of resources allocated by the plugin for the application, even if the corresponding process crashed.
The value-timestamp data point scheme of *DCDB* does not allow for sending data strings. Therefore the data strings have to be encoded in MQTT topics. *Sampler* and *Event* data is treated differently in this context.

Sampler

Data from the *Sampler* service is stored with all other *Sensor* data in the *Storage Backend*. For every unique function name encountered, a new *SensorBase* is constructed and added to the single *SensorGroup*. The function name is used to customize the *SensorBase*'s MQTT topic following the form "*globalPrefix / cpu / caliper / library name / function name*". For example, if a *Pusher* instance uses the hostname "mpp2r08c01s05" as global prefix, and the function "libstdc::start" was sampled on CPU 2, the corresponding *Sensor* topic will be "*mpp2r08c01s05/cpu2/caliper/libstdc/start*". On future encounters of the same function, a simple value of one will be stored as data reading with its corresponding *SensorBase* object. To avoid excessive network overhead by having to send many readings of one for each function encounter, an aggregation optimization is implemented. All encounters of the same function within the same read cycle are summed up. At the end, only the total value of function encounters in this read cycle is stored with the *SensorBase*. As a drawback, the exact timestamps for aggregated samples are lost. The time granularity is therefore reduced to the *SensorGroup*'s read interval. Hence, the choice of the read interval is a trade-off between network overhead and sample time precision. One has to keep in mind, though, that the read interval can also affect CPU usage. However, in Chapter 7 it is shown that *Pusher*'s CPU usage is sufficiently low. Hence, the neglect of CPU usage for the read interval consideration is justifiable in this case.
In retrospect, the overall number of function calls in a certain time frame can be calculated by retrieving and summing up all corresponding *Sensor* readings in this time frame. The function name to MQTT topic correlation also allows to aggregate statistics for shared library functions that are used among multiple applications.

Event

Data from *Event* snapshots is stored in a separate table within the *Storage Backend*. The key value in the table will be of the form "*hostname / cpu / caliper*". As data entries the annotation name and value from the *Event* data strings are used. The table can then be queried for a stream of annotations that were encountered on a node's CPU in a certain time frame, e.g. during the time a specific application was run. The annotation data is sent from *Pusher* to *Collect Agent* by encoding it in a MQTT topic which requires the creation of a *SensorBase* object per topic. The MQTT topic consists of a fixed prefix, the key for the *Event* data table, and the actual event data, for instance "*CALI_EVT_DATA / mpp2r08c01s05 / cpu2 / caliper / begin_function / main*". The topic prefix will then be recognized in the *Collect Agent* and the message handled differently from usual *Sensor* data messages. *Collect Agent* will split up the topic in its key and data part and store the data within the *Event* data table in the *Storage Backend*.

To limit the memory usage by possibly infinite creation of new *SensorBase* objects, two mechanisms exist. First, if no applications are currently connected to the plugin anymore, e.g. in the meantime between two user jobs, all existing *SensorBase* objects will be cleared. Second, one can specify a maximum number of *SensorBase* objects that can exist simultaneously. If this threshold is reached, all current *SensorBase* objects are destroyed and have to be reconstructed on future encounters.

7 Evaluation

A key challenge of monitoring is to gather meaningful data while keeping the overhead introduced by the monitoring system to a minimum. In the HPC domain mostly scientific software is run. Resource budget and time of the researchers is usually limited. User applications should complete as soon as possible, hence excessive overhead introduced by a monitoring system may be unacceptable. In this context the question arises how much additional overhead is introduced by the *DCDB* system. In particular, the impact of the previously presented *Pusher* component collecting in-band data on compute nodes is of interest. The impact of *Pushers* that collect out-of-band data, *Collect Agents*, and *Storage Backends* is of subordinate significance as they can be run on dedicated hardware that does not affect user applications.
This chapter is devoted to the performance evaluation of *Pusher* and seeks to answer the question how much additional overhead is introduced by the *Pusher* framework in general and the *Caliper* hybrid plugin in particular. The measurements for Section 7.2 were conducted as part of a previous publication [7]. Following up, the results for Section 7.3 were collected as part of this thesis work.

7.1 Setup

The exact hard- and software setup for the evaluation as well as the evaluated metrics are presented in the following.

Hardware

Evaluation tests are conducted on the *SuperMUC-NG* (SNG) [54] system and the *CoolMUC-2* (CM2) [55] Linux cluster at LRZ. A short summary of their respective hardware is given in Table 7.1. If not stated otherwise, measurements are conducted on SNG, as it is the primary test system. For the measurements, no hyper-threading is used and in case of SNG only thin nodes are employed. Threads are pinned, although no specific thread to core assignment is specified.
Collect Agents are run on dedicated hardware as listed in Table 7.2.

Software

For the evaluation, a varying set of benchmarks from the CORAL-2 [56] suite is used to represent user applications. Overall Kripke [57], AMG [58], LAMMPS [59], Quicksilver [60], and Pennant [61] are used. All benchmarks except LAMMPS are configured to instantiate one MPI [49] process per node and use as many OpenMP [62] threads as there are physical cores. As representation of multiple processes on the same node, LAMMPS is configured to start one MPI process per every physical core and not use any OpenMP threads. Tests are conducted on various system sizes with a weak-scaling approach. Additionally, on SNG single node High-Performance Linpack (HPL) [44, 45]

	SNG	CM2
Processor	Intel Platinum 8174	Intel E5-2697 v3
Architecture	Skylake	Haswell
Cores per Node	2 x 24	2 x 14
Memory per Node	96 GB	64 GB
Number of Nodes	6,480	384
Interconnect	Intel Omni-Path	FDR14 Infiniband

Table 7.1: Overview of the system hardware used for the evaluation.

Collect Agent	SNG	CM2
Processor	Intel Gold 6148	Intel E5-2690 v3
Architecture	Skylake	Haswell
Cores	2 x 20	2 x 12
Memory	768 GB	128 GB

Table 7.2: Overview of the system hardware used for the *Collect Agents*.

runs are conducted. The HPL version used for this runs uses Intel's MKL [46] library with AVX-512 [47]. Just like with LAMMPS, HPL uses one MPI process per every physical core but does not use any OpenMP threads.
A separate *Pusher* instance is launched on every node, each using two sampling threads. For each system, SNG and CM2, a single *Collect Agent* is run in a special benchmark mode. In the benchmark mode, the *Collect Agent* does not store any messages in a *Storage Backend* but instead discards all messages after receiving them. As the *Collect Agent* is run on separate hardware anyway, and still receives all messages, the evaluation should not be affected by its benchmark mode. During the measurements, it is recorded that the *Collect Agent* never receives more than 80,000 messages per second which is well below the maximum message rate tested in [7]. Hence, the *Collect Agent* can be ruled out as bottleneck for this evaluation. A *Storage Backend* is not run for this tests.
A list of used compilers is shown in Table B.1. All measurements are repeated ten times and from the results median values are taken to account for possible outliers.

Metrics

To assess *Pusher*'s impact on an application's performance, runtime overhead and additional memory usage are consulted as evaluation metrics.
For runtime overhead the wall clock execution time of the benchmarks is consulted. Overhead is defined as $O = (T_p - T_r)/T_r$. In this context, T_r refers to the reference value as measured during an unmodified benchmark run. T_p denotes the measured time of a benchmark run with *Pusher* and *Caliper*, if applicable.
Additional memory usage is determined by $M = M_p - M_r$. M_r refers to the memory required by an unmodified benchmark while M_p denotes the memory used by a benchmark with *Caliper* integrated. Memory usage is determined by the average Residual Set Size (RSS) as reported by the Linux *ps* tool throughout a run.
Further on, CPU and memory usage of the *Pusher* process as measured by *ps* are reported.

7.2 Pusher Framework Overhead

Pusher's runtime overhead is already evaluated for another *DCDB* related publication [7]. For the sake of completeness, the runtime overhead results are briefly repeated here.
Two *Pusher* configurations are tested. The first uses a special *Tester* plugin that allows to create *Sensors* which generate dummy data with negligible overhead. This configuration is therefore used to isolate the overhead of the *Pusher* framework itself from its plugins. The second configuration uses the four plugins *Perf Events*, *ProcFS*, *SysFS*, and *OPA* to gather in-band data. In both cases almost 2500 *Sensors* with a sampling rate of 1 Hz are created. The results are depicted in Figure 7.1. If no bar is visible, no overhead could be measured at all.

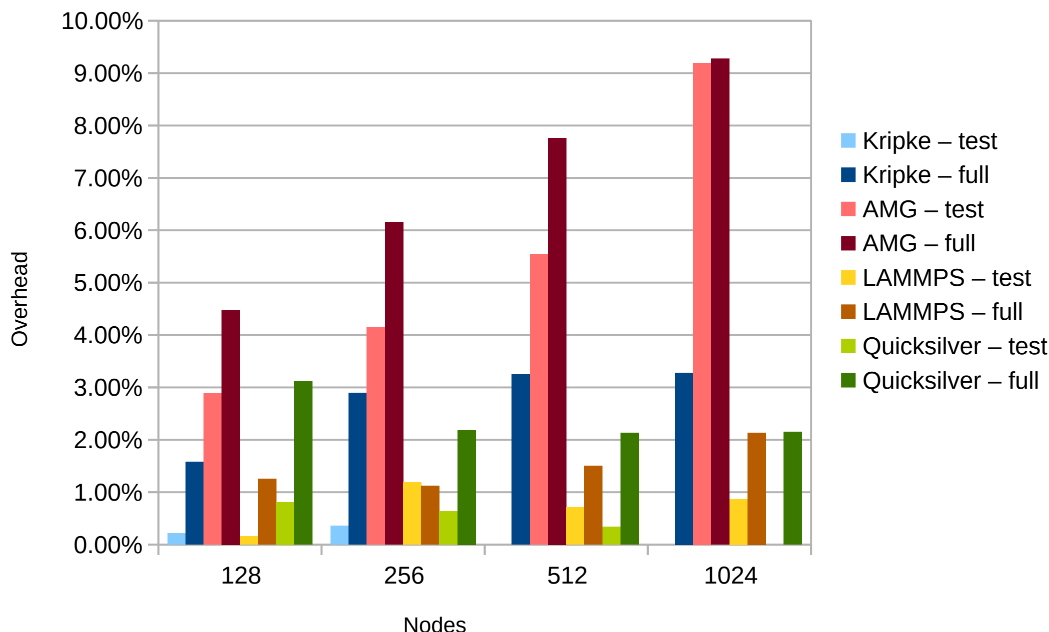


Figure 7.1: *Pusher*'s overhead on different benchmarks and system sizes on the SNG system.

Regarding Kripke, LAMMPS, and Quicksilver one can see that the overhead introduced by *Pusher* is below 5%. The overhead of the *Pusher* test configuration is well below the full setup. Hence, most of the overhead seems to be induced by the actual data collection of the plugins while the overhead of the *Pusher* framework itself is negligible.
Regarding the AMG benchmark, however, overhead goes up to over 9% which exceeds the other benchmark quite well. Also, the difference between test and full setup is not as distinct as with the other benchmarks. The root cause for the additional overhead lies therefore within the *Pusher* framework itself. In [7] it is already deduced that the unusual high overhead for AMG can be attributed to network inferences. Both configurations, test and full, produce the same amount of messages which have to be sent to the *Collect Agent*. As AMG is a network sensitive benchmark, both configurations infer with it in equal parts resulting in overly high overhead in both cases. One may consider the usage of a separate network interface to forward *Pusher*'s data messages to reduce the network overhead.

7.3 Caliper Overhead

Part of the work for this thesis is the exhaustive evaluation of the *Caliper* plugin. As it is a hybrid plugin that has to be integrated into a user's application, its overhead may be much more critical than *Pusher* one's. Two configurations, each representing a distinct use case as described in Section 6.2, are evaluated. The *Pushers* are instantiated with only the *Caliper* plugin enabled. The exact *Pusher* and *Caliper* configuration used for the tests is also shown in Appendix B.

7.3.1 Benchmarking Problems

The process of conducting the *Caliper* evaluation was hindered by quite a number of problems. At first, it was intended to run all measurements on CM2. The cluster, however, is usually crowded and jobs include up to several days of prior waiting time. After the required infrastructure was set up on CM2, and first results were already gathered, the cluster underwent a three day maintenance. Unfortunately, the updates rolled out during the maintenance resulted in an unstable cluster operation that rendered further measurements impossible. After these instabilities persisted for more than two weeks, it was decided to switch to the SNG system for the measurements. The setup on SNG in turn was hindered by inferences between *Caliper* and Intel libraries. Although an issue was submitted to *Caliper*'s GitHub repository [63], only a provisional solution for the problem could be found. Namely *Caliper*'s initialization is only done after the MPI initialization. Eventually, the results on SNG and CM2, whose operation stabilized in the meantime, showed to be not fully rational, as detailed below. Although further investigations were undertaken, the root cause was not found.
Overall, the aforementioned obstacles resulted in a remarkable slow down of the whole evaluation process. The initial time budget was by far exceeded. Therefore, the following results are not as polished as they could be. Also, not all unreasonable results can be appropriately explained as no time was left for further investigations.

7.3.2 Sampler

For the evaluation of the *Sampler* use case all benchmarks were modified to initialize *Caliper* at program start. By default, a sampling rate of 10 Hz is used.

Runtime

One can see in Figure 7.2 that in the case of Kripke and AMG the runtime overhead lies well below 10%. In case of Pennant, the runtime overhead almost vanishes in the measurement fluctuations. For LAMMPS and Quicksilver the measurements yield quite strange results. Contrary to all expectations, LAMMPS runtime notably improves by the *Caliper* integration. Quicksilver in turn reveals some great overhead fluctuations, ranging from almost -45% to over +25%. Both benchmarks strongly suggest that the setup and/or conduction of the experiments is erroneous. Despite triple checking the experimental setup and various further tests to exclude any other possible environmental inference a root cause or other explanation for the LAMMPS and Quicksilver results could not be found and would require an exhaustive performance analysis that is out of the scope of

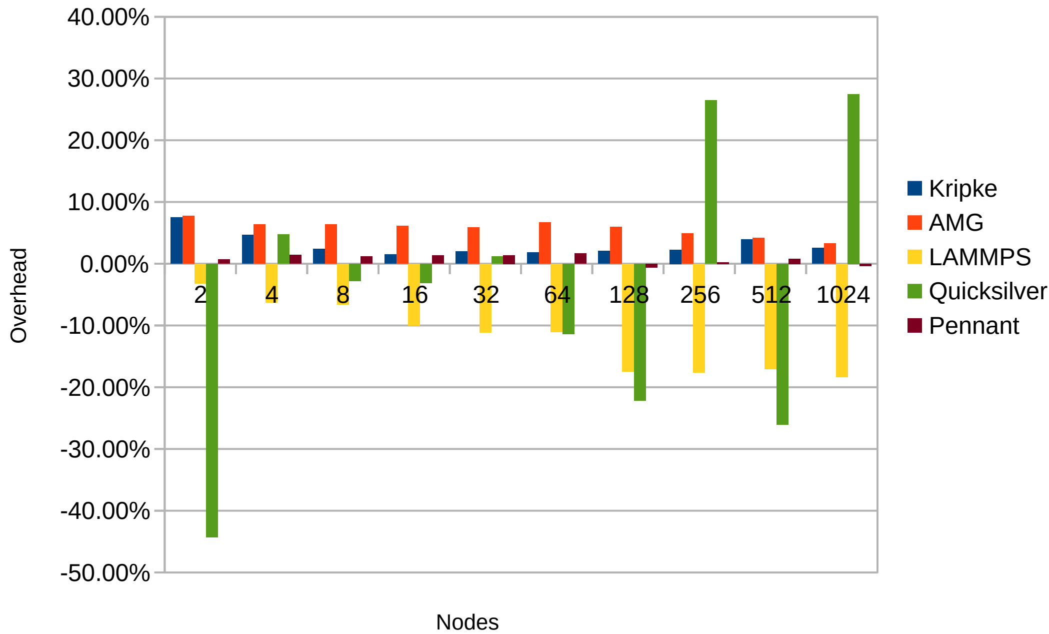


Figure 7.2: Runtime overhead induced by *Pusher* and *Caliper* using the *Sampler* configuration on the SNG system.

this thesis. Although the results for LAMMPS at least could be considered to cast a very favorable light on the *Caliper* set up, the author does not assess them as correct.

To further investigate on the unreasonable runtime overhead results, the experiment is repeated at smaller scale on the CM2 system. The resulting runtime overheads are depicted in Figure 7.3. One can see that on one side Kripke's overhead improved on CM2 and is almost negligible. On the other side, AMG's overhead got worse and even increases with larger node configurations. Pennant's overhead in turn remains unchanged except a significant outlier with the 32 node configuration. For the aforementioned results, a statement on generally better or worse performance on CM2 can not be given.
LAMMPS' overhead appears to be more sound than on SNG, despite one extreme outlier at the 16 node configuration. Quicksilver repeatedly shows greatly varying results, which may indicate a general high fluctuation of runtimes. Although the overhead never exceeds 5%, the Quicksilver results should be interpreted carefully, as an overhead of less than -20% in the two node configuration does not seem reasonable.
One should also recall into mind that all measurements are repeated ten times and only median values are reported. Outliers and fluctuations are therefore persistent.

For better classification of the results, the overhead caused by *Pusher* and *Caliper* is compared to the overhead of *Caliper* integration alone. For this test, the runtime overhead on single node HPL is measured on SNG with two configurations. The first uses *Pusher* and *Caliper* in the same setup as for the CORAL-2 benchmarks above. The second configuration leaves out *Pusher* and sets up *Caliper* to use the *Trace*, *Symbollookup*, and *Recorder* services instead of the *dcdbpusher* service. This way, *Caliper* achieves similar sample data results but without being integrated into the *DCDB* infrastructure. The exact *Caliper* configuration is given in Listing B.4.
The HPL runtimes with the different configurations reveal that *Caliper* together with *Pusher* induces an overhead of 3.4% while *Caliper* alone only adds 0.3%. Therefore, only

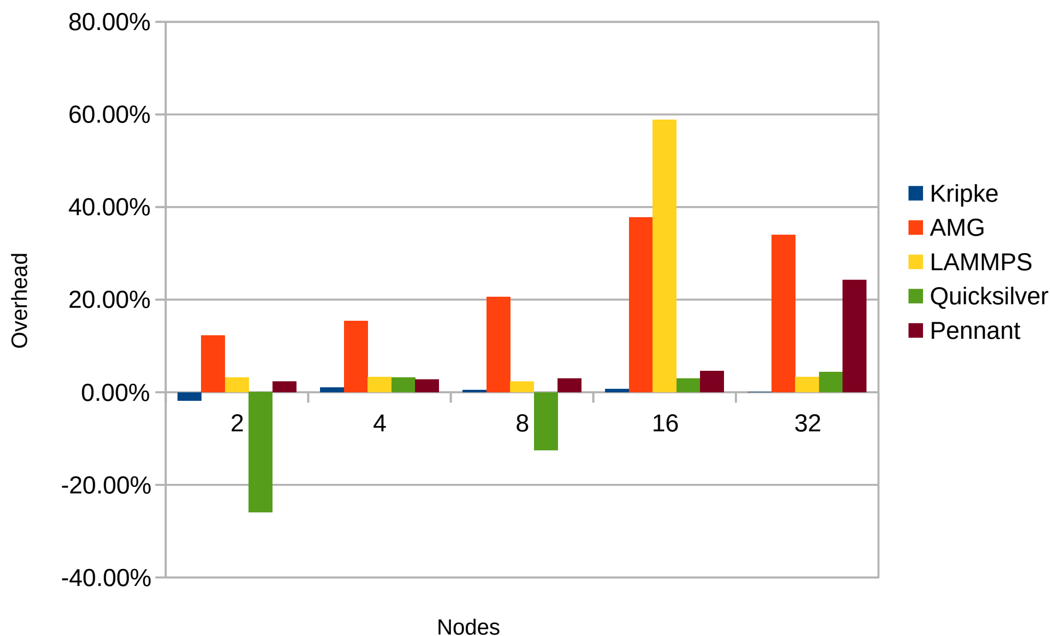


Figure 7.3: Runtime overhead induced by *Pusher* and *Caliper* using the *Sampler* configuration on the CM2 system.

a minor part of the runtime overhead seems to be caused by the *Caliper* integration on its own. Instead, the majority has to be accredited to the *dcdbpusher Caliper* service and the presence of *Pusher*.

Overall, the runtime overhead caused by *Pusher* and the *Caliper* integration has to be interpreted carefully. Only regarding meaningful benchmark results, the overhead lies mostly below 5%. Nevertheless, 5% are also significantly exceeded especially in the case of AMG, but also other benchmarks revealed extreme outliers. Taking the unusual and currently unexplainable results from LAMMPS on SNG and particularly Quicksilver into account, doubts about the validity of the measurements remain.

Memory

The induced additional memory consumption of the CORAL-2 benchmarks due to the *Caliper* integration as measured on SNG is depicted in Figure 7.4. Note that *Pusher*'s memory usage is not included in Figure 7.4 but is disclosed separately below. One can see that *Caliper* usually requires an additional 500 MiB of memory. In the case of AMG some significant negative outliers can be observed. This may be accredited to AMG's general high memory fluctuations. The benchmark's memory usage varies within a few GiB that can easily make up for the extra memory usage of *Caliper*. The same reasoning can be applied to Kripke's significant fluctuations, although they are not quite as high as with AMG.
Results from the measurements on the CM2 system are left out, as they do not expose any new insights. *Caliper*'s memory usage stays unchanged with around 500 MiB on CM2.
The 500 MiB of additional memory required by *Caliper* seem to be acceptable for most cases, as current HPC system's main memory is comparatively large, rendering *Caliper*'s requirements negligible. Here for example, even the most memory intensive benchmark,

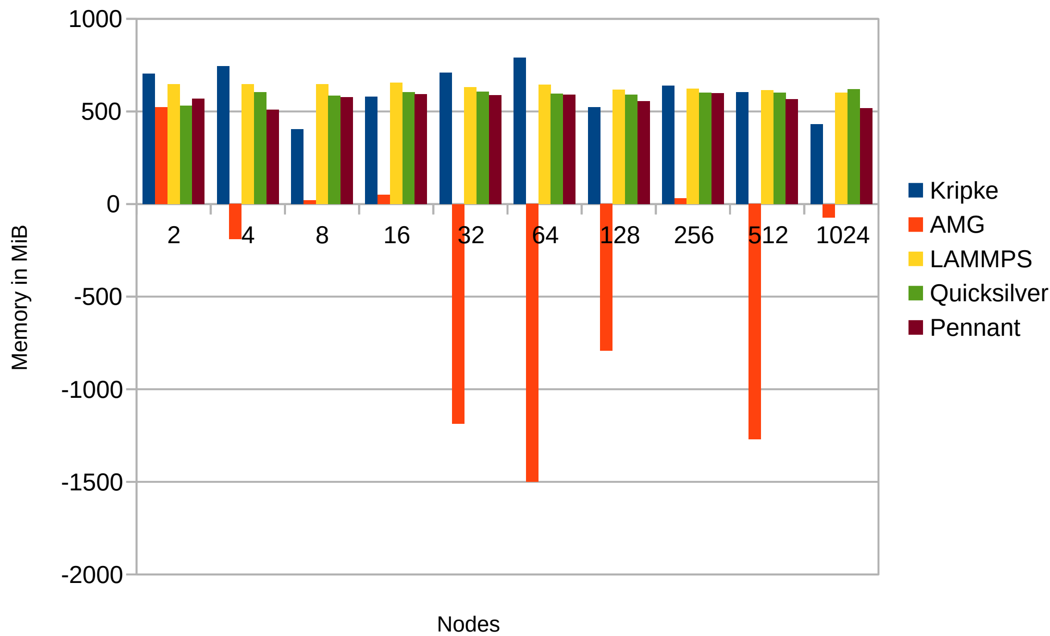


Figure 7.4: Additional memory usage of benchmarks induced by the *Caliper* integration using the *Sampler* configuration on the SNG system.

Kripke, still did not use half of the memory that is offered on SNG. Still, extreme memory affine applications, that can completely use a system's memory on their own may be impacted by the *Caliper* integration. In such cases, it may be necessary to account for *Caliper*'s additional memory usage and reduce the problem size accordingly.

Pusher

Pusher's resource usage, namely its CPU and memory usage are depicted in Figures 7.5 and 7.6 respectively. One can see, that *Pusher*'s CPU usage is below 1% in every case and is therefore on the verge of being negligible (100% CPU usage correspond to one fully loaded CPU core). *Pusher*'s memory usage lies between 90 and 160 MiB. This matches the memory measurements with other plugins as reported in [7]. Unequal memory usages with different benchmarks can be explained by the amount of differing functions that are sampled. The more different functions are sampled, the more *Sensor* objects are created by the *Caliper* plugin, hence memory usage rises. Even in the worst case, namely Quicksilver, *Pusher*'s memory usage of at most 160 MiB should be acceptable as current HPC server systems comprise dozens of GiB of main memory. Similar results for *Pusher*'s resource usage could be observed on the CM2 system. CPU usage is below 1% at all times while no more than 160 MiB of memory are required. Altering resource usage on different system sizes can not be observed and is also not to be expected, as a separate *Pusher* instance is run on every node. Hence, the number of *Pusher* instances scales with the system size while individual resource usage of a single *Pusher* stays constant.

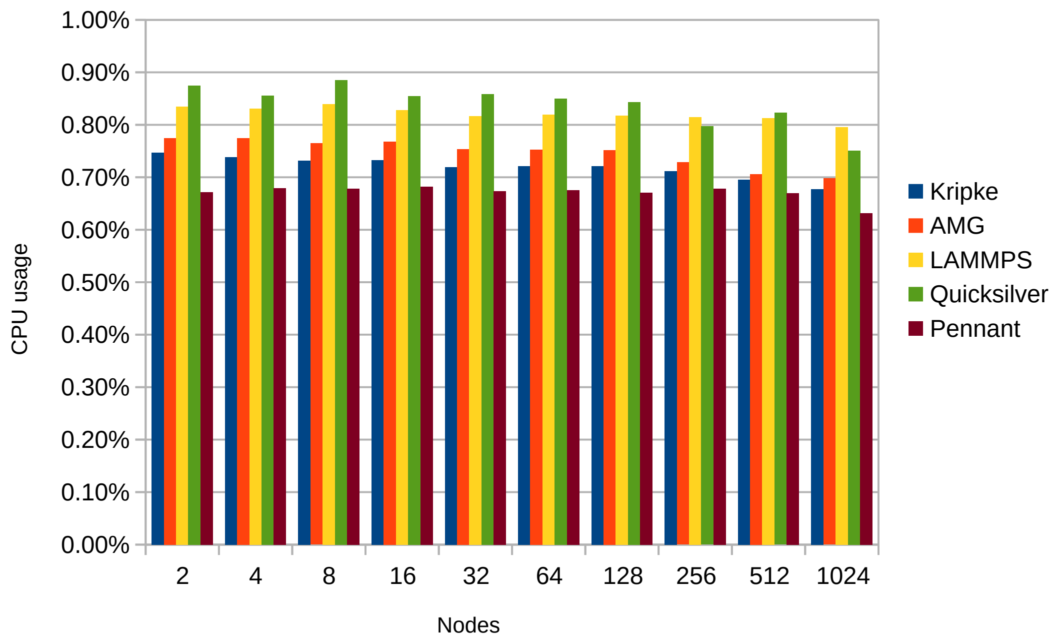


Figure 7.5: *Pusher*'s average CPU usage using the *Sampler* configuration on the SNG system.

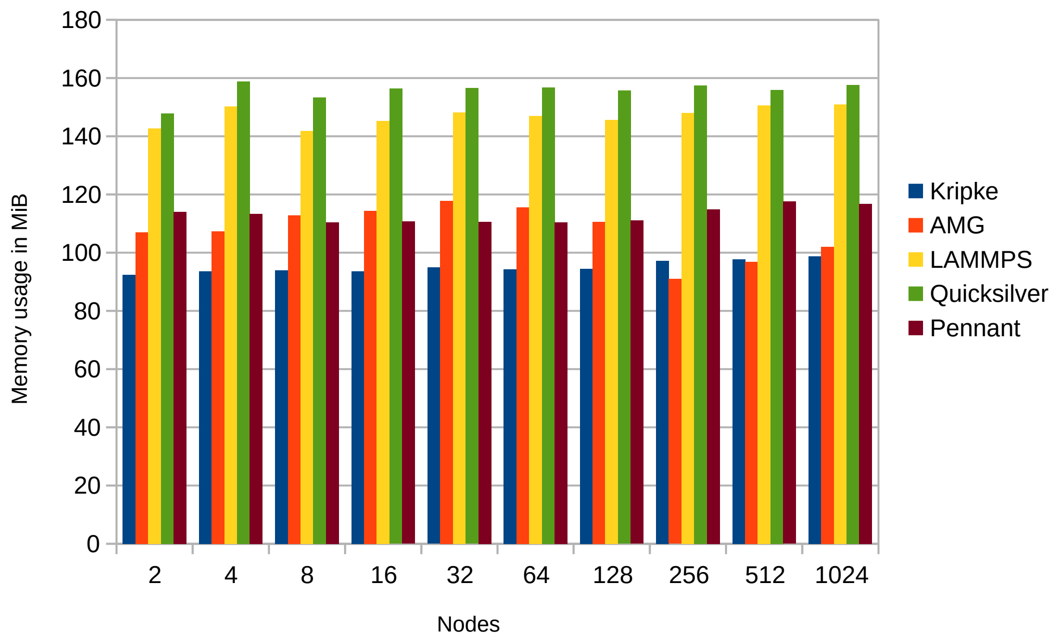


Figure 7.6: *Pusher*'s average memory usage using the *Sampler* configuration on the SNG system.

7.3.3 Sampler Frequency

The used *Sampler* frequency of 10 Hz is a rather arbitrary choice. To assess the impact of the *Sampler* frequency on the evaluation and gather a small indication on how severe the impact of higher frequencies is, a small experiment is conducted. The benchmarks Kripke and AMG are run on two SNG nodes with the same configuration as before but with increasingly higher *Sampler* frequencies.

Looking at the runtime overhead as reported in Figure 7.7, both benchmarks seem to be only marginally affected by increased *Sampler* frequencies. Although the applications themselves are rather unaffected by higher *Sampler* frequencies, *Pusher*'s CPU usage linearly correlates with the increasing snapshot rate as can be seen in Figure 7.8. This seems logical, as the *Pusher* plugin is the place where the snapshots have to be processed and assigned to *Sensors*. Therefore, more snapshots that have to be processed require more processing power. In this test *Pusher*'s previously almost negligible CPU overhead of less than 1% rises to significant 15% in the worst case.

The additional memory usage of the benchmarks is subject to high variations as can be seen in Figure 7.7. Fluctuations within the results can realistically be credited to overall measurement variations. Especially AMG is already described as having a highly varying memory usage. Still, the high memory fluctuations do not allow for a meaningful interpretation of the results, although in general no rising memory usage with increasing *Sampler* frequencies can be observed. Memory usage staying the same for increasing frequencies is to be expected, as the generated snapshots do not get stored within the application but get directly forwarded to the *Pusher* plugin. Higher frequencies result in more snapshots per second to be forwarded to the plugin but they do not require a significant part of additional memory.

Pusher's memory usage does not correlate linearly with the *Sampler* frequency as can be seen in Figure 7.8. Although there is a slight increase of memory usage until 100 Hz, it seems to be capped for higher frequencies. This behavior meets the expectations. The main factor of *Pusher*'s increasing memory is the creation of new *Sensor* objects for every newly sampled function. At some point, however, under the assumption of finite user software, all possible functions are sampled already. No new *Sensors* have to be created as all samples can be accounted to an existing object. Therefore the rise of memory usage comes to a halt.

Overall, it can be recorded, that an increase of the *Sampler* frequency does not directly affect an user's application. However, the software is still affected indirectly, as it has to share a node's resources with a *Pusher* whose CPU usage increases linearly with the frequency. Especially compute bound applications are therefore prone to performance impacts from rising *Sampler* frequencies. Still, a frequency of up to 100 Hz seems to be an acceptable choice, as its induced increased resource usage lies within a small acceptable range.

7.3.4 Sampler with Events

The second use case makes use of the *Event* service in addition to the *Sampler*. Once again, the *Sampler* uses a sampling rate of 10 Hz. The benchmarks Kripke, AMG, and Quicksilver are enriched with two annotations respectively that enclose the main compute loop. Further on, a heavily annotated Quicksilver version from the *Caliper* example repository [64] is evaluated ("Quicksilver Exam" in the following). The annotations make up for 5% (Kripke), 30% (AMG), 15% (Quicksilver), and 90% (Quicksilver Exam) of all processed snapshots respectively.

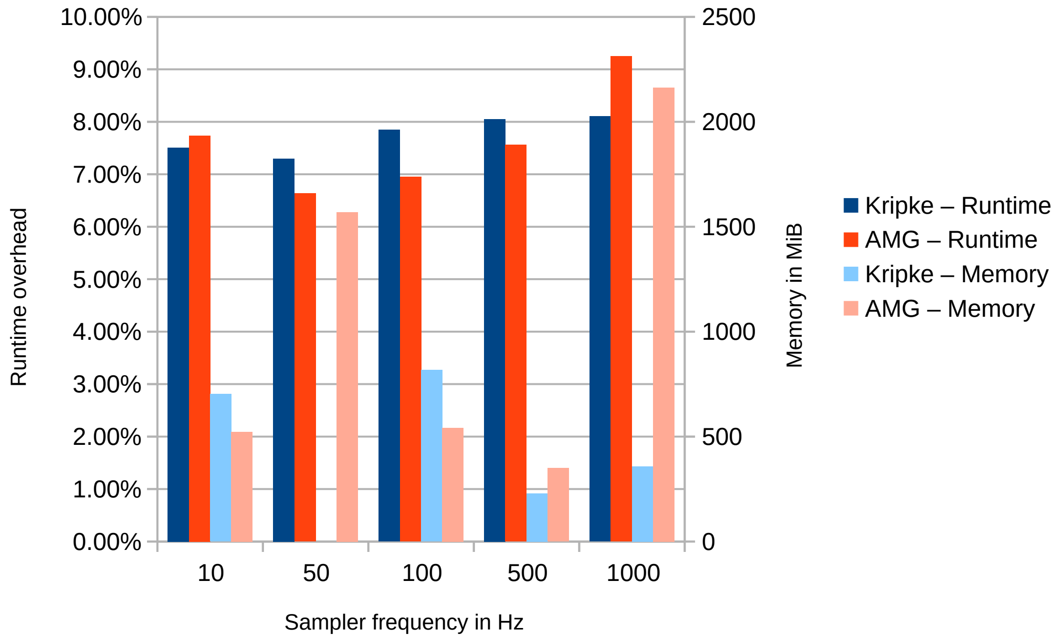


Figure 7.7: Runtime overhead induced by *Pusher* and *Caliper*, as well as additional memory usage of benchmarks caused by *Caliper*'s integration. Both measured with different *Sampler* frequencies on the SNG system.

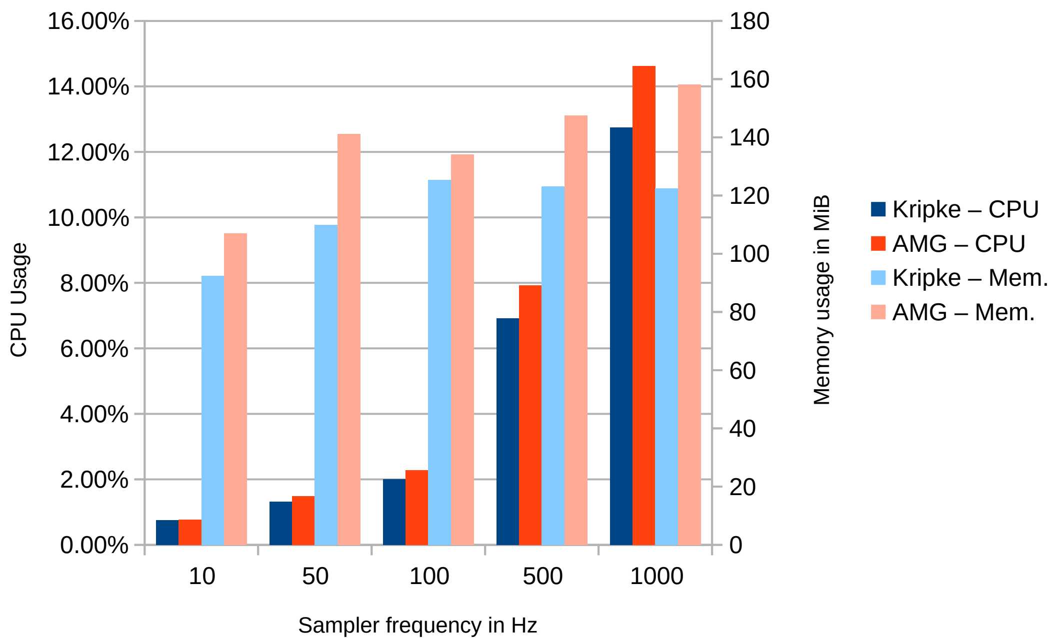


Figure 7.8: *Pusher*'s average CPU and memory usage with different *Sampler* frequencies on the SNG system.

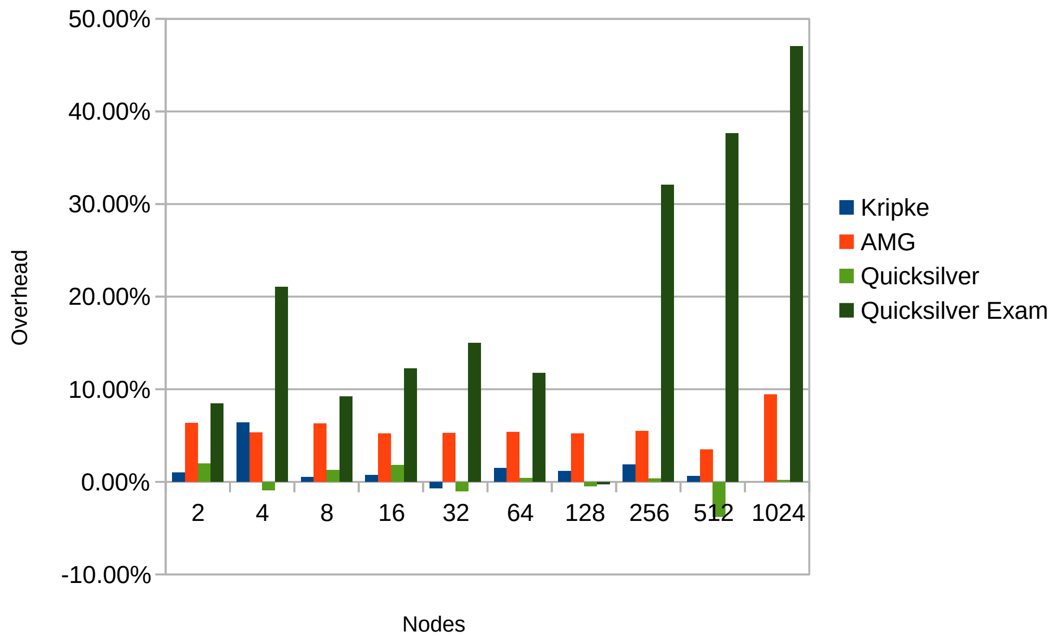


Figure 7.9: Runtime overhead induced by *Pusher* and *Caliper* using the *Sampler* with *Events* configuration on the SNG system.

The runtime overhead caused by *Caliper* with the *Sampler* and *Event* configuration is depicted in Figure 7.9. Visible outliers are credited to the same unexplainable problem as reported in Section 7.3.2. Kripke, AMG, and Quicksilver all seem to be rather unaffected by the additional annotations. Their overhead does not significantly exceed the numbers shown in Figure 7.2 and is in line with the overhead reported in [9]. This may also be partly credited to the fact, that the annotations only make up for a minority of the snapshots. Quicksilver Exam, however, shows a great performance impact caused by its annotations of up to almost 50%. This is to be expected, as the annotation triggered events make up 90% of its snapshots and result in an average event rate of 4,500 events/second. One has to keep in mind, though, that the Quicksilver results are not as indicative as they could be, because of Quicksilver's known performance variations. For an accurate statement, further measurements would be required.
In general, additionally introduced overhead greatly depends on the number of annotations for the *Sampler* with *Events* configuration. Therefore, acceptable overhead has to be determined by the users themselves and annotations have to be deployed accordingly.
Additional memory usage remains unchanged in this configuration, presumably for the same reasoning as presented in Section 7.3.3. Therefore, the corresponding graphs are left out.

The impact of the *Sampler* with *Events* configuration on *Pusher*'s resource usage appears to be limited. Its CPU usage as shown in Figure 7.10 did not significantly increase in any case in comparison to Figure 7.5 and is still well below 1%. Only in the case of Quicksilver Exam a slightly increased CPU usage can be noted. It seems that the additional *Event* snapshots are negligible even in the case of Quicksilver Exam, where they make up 90% of all snapshots. This appears reasonable, as a significant increase of *Pusher*'s CPU usage in Figure 7.8 could only be observed with per-thread *Sampler* frequencies that induce overall many more snapshots than the per-process annotations from Quicksilver Exam.
Regarding *Pusher*'s memory usage as presented in Figure 7.11, no significant rise in com-

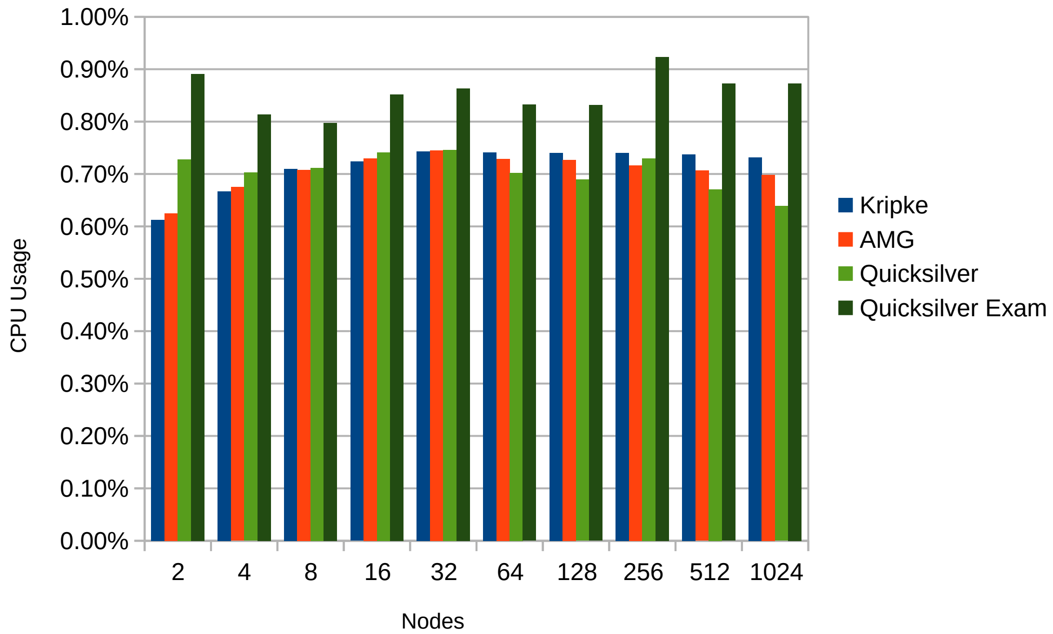


Figure 7.10: *Pusher*'s average CPU usage using the *Sampler* with *Events* configuration on the SNG system.

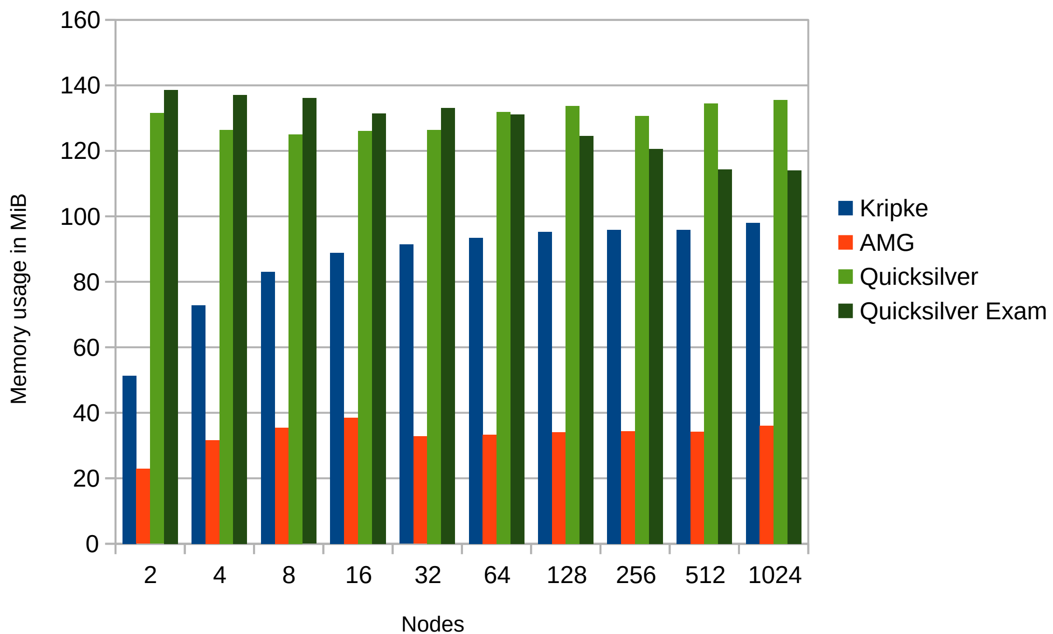


Figure 7.11: *Pusher*'s average memory usage using the *Sampler* with *Events* configuration on the SNG system.

parison to Figure 7.6 can be observed. It seems that the additional *Sensor* objects required for the annotation data are negligible in comparison to the sampling induced *Sensors*. Surprisingly, in the case of AMG, *Pusher* requires persistently less than half of the memory than with the *Sampler* only configuration. Results for the other benchmarks, however, testify that the measurements in general are valid. Currently, no sound explanation for this odd behavior can be given.

7.3.5 Conclusion

In general, approval of the results depends on what threshold one is willing to accept. Runtime overhead of the *Pusher* framework collecting general in-band data is below 5% in most cases except AMG. It is predicted that also AMG's overhead can be further reduced by sending *Pusher*'s data over a dedicated network. Therefore, the overhead of *Pusher* collecting in-band data is classified as acceptable for production use.
Results for *Pusher* collecting application introspection data with *Caliper* in the *Sampler* configuration have to be interpreted carefully. The measured runtime overheads appear to be likely inaccurate in the case of LAMMPS on SNG and Quicksilver in general. Regarding the reasonable results, the conclusion is mixed. Depending on the benchmark the runtime overhead may exceed 5% and reach more than 20%. It is unlikely that the overhead is caused by *Pusher*'s resource usage per se. It is shown that *Pusher* does not exceed 1% CPU and 160 MiB memory usage. Runtime overhead also appears to be rather indifferent towards the used sampling frequency, as no overhead changes could be observed for frequencies of up to 1000 Hz. Still, *Pusher*'s CPU usage linearly increases with the *Sampler* frequency and is therefore likely to cause runtime inference at some point.
Regarding measurements with *Caliper* in the *Sampler* with *Events* configuration, it is shown that the runtime overhead significantly depends on the amount of *Event* triggering annotations. The overhead can increase greatly with *Event* snapshots. However, the users themselves are in direct charge of the amount of *Event* annotations.
An applications additional memory usage induced by *Caliper* is determined to be around 500 MiB. The memory usage is largely independent of the configuration. Although current HPC systems employ multiple dozens of GiB as main memory per node, the additional memory usage should be taken into consideration for memory affine applications.

8 Summary and Outlook

Holistic monitoring is key to efficient operation of current and future HPC systems. In this thesis, a new, generic data acquisition module called *Pusher* for the *DCDB* monitoring infrastructure is presented and evaluated. As a background, *DCDB*'s design principles and main components, namely *Pusher*, *Collect Agent*, and *Storage Backend*, are introduced. Further on, the need for a new generic *Pusher* implementation is stated.
A new *Pusher* component was developed. It is designed as general framework, that allows for the attachment of plugins. Actual data acquisition from specific data sources is outsourced into those plugins. The framework provides the infrastructure to forward the data to other *DCDB* components. All design goals and the full functionality of the framework like the integrated *RestAPI* are presented. Further on, an in-depth overview of the actual implementations and the various involved components is given. Following, the implementation structure of plugins is presented. All currently implemented plugins and their functionality are showcased. The hybrid *Caliper* plugin is highlighted in particular, as it allows to gather application introspection data for the monitoring framework.
In conclusion, the *Pusher* framework in general and the *Caliper* plugin in particular are evaluated. It is shown, that the resource footprint of *Pusher* collecting generic in-band data is viable for production use. The runtime overhead of the *Caliper* plugin on different benchmarks may exceed a 5% in certain configurations. Acceptance is therefore subject to personal opinions. Additional memory usage induced by *Caliper* is determined to be around 500 MiB. The resource usage of *Pusher* itself supports *Caliper* sampling frequencies of up to 100 Hz without exceeding 2% CPU and 160 MiB memory usage.
For the future, some tasks still remain. The integration of *Caliper* into user software should be done automatically and invisible to the user to gather sampling data from applications at all times. Possibly, this can be done by providing a custom library that overwrites a program's start method and initializes *Caliper* before actual application start. Feasibility of this approach still has to be explored, though. Especially the inferences between *Caliper* and Intel libraries have to be resolved first, as they form a significant hurdle for the automatic *Caliper* integration in general. Also, further investigations of the LAMMPS and Quicksilver benchmark results may be necessary to dispel doubts about the evaluation results. It may also be useful to do further experiments to fully explore the impact of *Sampler* triggered snapshots in comparison to *Event* triggered ones.

A Software Dependencies

Pusher and its shipped plugins depend on a set of third-party software. Those dependencies are listed in the following.

Name	Version	Source	Required for
BACnet Stack	0.8.6	[65]	*BACnet* plugin
Boost	1.70.0	[66]	Among others: Threadpool, Logging, *SensorBase* reading queue, `RESTHttpsServer`
Elfutils	0.177	[67]	*Caliper* plugin
FreeIPMI	1.6.3	[68]	*IPMI* plugin
Mosquitto	1.5.5	[69]	*MQTTPusher*
Net-SNMP	5.8	[70]	*SNMP* plugin
OPA Software	10.6.0.0.134	[71]	*OPA* plugin
OpenSSL	1.1.1c	[72]	`RESTHttpsServer`, *REST* plugin

Table A.1: External software dependencies *Pusher* and its plugins rely on.

B Additional Evaluation Information

This chapter is a collection of all relevant information related to Chapter 7 that did not fit the text flow very well.

Compilers

To compile the evaluation applications as well as *Pusher* and *Caliper* themselves, different compilers and versions are used. They are listed in the following.

Name	Version	Used for
GCC	4.8.5	*Pusher* in Section 7.2
GCC	7.3.0	*Pusher* and *Caliper* in Section 7.3
Intel (including Intel MPI)	18.0.5	All benchmarks in Section 7.2
Intel (including Intel MPI)	19.0.4	All benchmarks in Section 7.3

Table B.1: Compiler versions used for the evaluation.

Configuration Files

The exact *Pusher* and *Caliper* configurations as used for the *Caliper* evaluation are shown below.

```
global {
    mqttPrefix  /System/Rack/Chassis/Node
}

group cali {
    interval    100
    maxSensors  1000
    timeout     250
}
```

Listing B.1: Configuration for *Pusher*'s *Caliper* plugin as used in Section 7.3.

```
global {
    mqttBroker 172.16.224.164:1883
    mqttprefix /test
    threads 2
    maxMsgNum 1000
    verbosity 0
    daemonize false
    tempdir /tmp/
    cacheInterval 120
    qosLevel 0
}

plugins {
    plugin caliper {
        path
        config
    }
}

operatorPlugins {
}
```

Listing B.2: *Pusher*'s configuration file.

```
CALI_SERVICES_ENABLE=event:sampler:timestamp:pthread:dcdbpusher;
CALI_SAMPLER_FREQUENCY=10;
CALI_TIMER_TIMESTAMP=true;
```

Listing B.3: Runtime configuration of the Caliper toolbox as used in Section 7.3.

```
export CALI_SERVICES_ENABLE=sampler:timestamp:pthread:trace:
    symbollookup:recorder
export CALI_SAMPLER_FREQUENCY=10
export CALI_TIMER_TIMESTAMP=true
export CALI_TRACE_BUFFER_POLICY=flush
export CALI_SYMBOLLOOKUP_LOOKUP_FUNCTIONS=true
export CALI_SYMBOLLOOKUP_LOOKUP_SOURCELOC=false
export CALI_SYMBOLLOOKUP_LOOKUP_FILE=true
```

Listing B.4: Runtime configuration of the Caliper toolbox as used for the HPL run without *Pusher* in Section 7.3.2.

Bibliography

[1] "Aurora." [Online]. Available: `https://www.anl.gov/article/us-department-of-energy-and-intel-to-deliver-first-exascale-supercomputer`, last accessed 16.09.2019.

[2] G. E. Moore, "Progress in Digital Integrated Electronics," *IEEE International Electron Devices Meeting (IEDM)*, pp. 11–13, 1975.

[3] "The law that's not a law," *IEEE Spectrum*, vol. 52, no. 4, pp. 38–57, 2015.

[4] "Intel pushes 10nm chip-making process to 2017, slowing Moore's Law." [Online]. Available: `https://www.infoworld.com/article/2949153/intel-pushes-10nm-chipmaking-process-to-2017-slowing-moores-law.html`, last accessed 08.10.2019.

[5] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, 2005.

[6] A. Auweter, *The DCDB Framework*. Dissertation, Technische Universität München, München, January 2019.

[7] A. Netti, M. Müller, A. Auweter, C. Guillen, M. Ott, D. Tafani, and M. Schulz, "From Facility to Application Sensor Data: Modular, Continuous and Holistic Monitoring with DCDB," *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, August 2019.

[8] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000.

[9] D. Boehme, T. Gamblin, D. Beckingsale, P. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, "Caliper: Performance Introspection for HPC Software Stacks," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 550–560, Nov 2016.

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

[11] A. Netti, M. Müller, C. Guillen, M. Ott, D. Tafani, G. Ozer, and M. Schulz, "DCDB Wintermute: Enabling Online and Holistic Operational Data Analytics on HPC Systems," *Submitted to 34th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, October 2019.

[12] G. Wang and J. Tang, "The NoSQL Principles and Basic Application of Cassandra Model," in *2012 International Conference on Computer Science and Service System*, pp. 1332–1335, 2012.

[13] "BACnet-A Data Communication Protocol for Building Automation and Control Networks," Standard 135-2016, American Society of Heating, Refrigerating and Air-Conditioning Engineers (ASHRAE), 2016.

[14] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing

Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, (Berkeley, CA, USA), USENIX Association, 2002.

[15] "Intelligent Platform Management Interface Specification v2.0 rev. 1.1," industry specification, Intel Corporation, Hewlett-Packard Company, Dell Computer Corporation, NEC Corporation, 2013.

[16] *Intel 64 and IA-32 Architectures Software Developer's Manuals*, vol. 4. May 2019.

[17] M. Feldman and A. Snell, "A New High Performance Fabric for HPC," white paper, Intersect360 Research, 2016.

[18] V. M. Weaver, "Linux perf_event features and overhead," in *Proc. of the FastPath Workshop 2013*, vol. 13, 2013.

[19] "proc Filesystem." [Online]. Available: `http://man7.org/linux/man-pages/man5/proc.5.html`, last accessed 02.10.2019.

[20] J. Case, M. Fedor, M. L. Schoffstall, and D. James, "A Simple Network Management Protocol (SNMP)," RFC 1157, Internet Engineering Task Force (IETF), 1990.

[21] "sysfs Filesystem." [Online]. Available: `http://man7.org/linux/man-pages/man5/sysfs.5.html`, last accessed 02.10.2019.

[22] "Data Center DataBase (DCDB)." [Online]. Available: `https://dcdb.it`, last accessed 23.09.2019.

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

[24] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, vol. 2015-January, pp. 154–165, 2014.

[25] J. M. Brandt, B. J. Debusschere, A. C. Gentile, J. R. Mayo, P. P. Pébay, D. Thompson, and M. H. Wong, "Ovis-2: A robust distributed architecture for scalable RAS," in *IPDPS Miami 2008 - Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium, Program and CD-ROM*, 2008.

[26] T. Evans, W. L. Barth, J. C. Browne, R. L. Deleon, T. R. Furlani, S. M. Gallo, M. D. Jones, and A. K. Patra, "Comprehensive resource use monitoring for HPC systems with TACC stats," in *Proceedings of HUST 2014: 1st International Workshop on HPC User Support Tools - Held in Conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 13–21, 2014.

[27] R. T. Evans, J. C. Browne, and W. L. Barth, "Understanding application and system performance through system-wide monitoring," in *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016*, pp. 1702–1710, 2016.

[28] "RabbitMQ." [Online]. Available: `https://www.rabbitmq.com/`, last accessed 18.09.2019.

[29] "Performance Co-Pilot." [Online]. Available: `https://pcp.io/`, last accessed 19.09.2019.

[30] W. Barth, *Nagios: System And Network Monitoring*. Open Source Press GmbH, 2 ed., 2008.

[31] D. Carasso, *Exploring Splunk - Search processing Language (SPL) Primer and Cookbook*. CITO Research, 1 ed., 2012.

[32] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*, pp. 79–91, Springer Berlin Heidelberg, 2012.

[33] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[34] "Intel VTune Amplifier." [Online]. Available: `https://software.intel.com/vtune`, last accessed 14.10.2019.

[35] "Boost INFO File Format." [Online]. Available: `https://www.boost.org/doc/libs/1_71_0/doc/html/property_tree/parsers.html#property_tree.parsers.info_parser`, last accessed 02.10.2019.

[36] "Doxygen." [Online]. Available: `http://www.doxygen.nl/index.html`, last accessed 02.10.2019.

[37] R. A. Light, "Mosquitto: server and client implementation of the MQTT protocol," *The Journal of Open Source Software*, vol. 2, no. 13, p. 265, 2017.

[38] "Boost.Beast." [Online]. Available: `https://github.com/boostorg/beast`, last accessed 25.09.2019.

[39] "Boost.Log." [Online]. Available: `https://github.com/boostorg/log`, last accessed 28.09.2019.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[41] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Transactions on communications*, vol. 39, no. 10, pp. 1482–1493, 1991.

[42] J. E. F. Friedl, *Mastering Regular Expressions*. O'Reilly Media, Inc., 3 ed., 2006.

[43] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism," in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pp. 392–403, 1995.

[44] J. Dongarra, J. Bunch, C. Moler, and G. Stewart, *LINPACK Users' Guide*. Other Titles in Applied Mathematics, Society for Industrial and Applied Mathematics, 1979.

[45] "High-Performance Linpack." [Online]. Available: `http://www.netlib.org/benchmark/hpl/`, last accessed 05.11.2019.

[46] "Intel Math Kernel Library." [Online]. Available: `https://software.intel.com/mkl`, last accessed 07.11.2019.

[47] J. Reinders, "Intel AVX-512 Instructions," 2013. [Online]. Available: `https://software.intel.com/articles/intel-avx-512-instructions`, last ac-

cessed 07.11.2019.

[48] "Portable Operating System Interface (POSIX) Base Specifications, Issue 7," Standard 1003.1-2017, Institute of Electrical and Electronics Engineers (IEEE), 2017.

[49] "MPI: A Message-Passing Interface Standard," Standard Version 3.1, Message Passing Interface Forum, 2015.

[50] "Caliper GitHub Pull Request." [Online]. Available: `https://github.com/LLNL/Caliper/pull/222`, last accessed 07.11.2019.

[51] "pthreads man-page." [Online]. Available: `http://man7.org/linux/man-pages/man7/pthreads.7.html`, last accessed 07.10.2019.

[52] "GOTCHA GitHub Repository." [Online]. Available: `https://github.com/llnl/GOTCHA`, last accessed 07.11.2019.

[53] "Address space layout randomization (ASLR)." [Online]. Available: `https://pax.grsecurity.net/docs/aslr.txt`, last accessed 09.10.2019.

[54] "SuperMUC-NG." [Online]. Available: `https://doku.lrz.de/display/PUBLIC/SuperMUC-NG`, last accessed 24.10.2019.

[55] "CoolMUC-2 Linux cluster." [Online]. Available: `https://doku.lrz.de/display/PUBLIC/CoolMUC-2`, last accessed 01.11.2019.

[56] "CORAL-2 Benchmarks." [Online]. Available: `https://asc.llnl.gov/coral-2-benchmarks/`, last accessed 24.10.2019.

[57] 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.

[58] U. M. Yang *et al.*, "BoomerAMG: a parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155–177, 2002.

[59] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of computational physics*, vol. 117, no. 1, pp. 1–19, 1995.

[60] D. F. Richards, R. C. Bleile, P. S. Brantley, S. A. Dawson, M. S. McKinley, and M. J. O'Brien, "Quicksilver: a proxy app for the Monte Carlo transport code mercury," in *Proc. of CLUSTER 2017*, pp. 866–873, IEEE, 2017.

[61] C. R. Ferenbaugh, "PENNANT: an unstructured mesh mini-app for advanced architecture research," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 4555–4572, 2015.

[62] "OpenMP Application Programming Interface," Specification Version 5.0, OpenMP Architecture Review Board, November 2018.

[63] "Caliper GitHub Issue." [Online]. Available: `https://github.com/LLNL/Caliper/issues/223`, last accessed 06.11.2019.

[64] "Quicksilver example repository." [Online]. Available: `https://github.com/LLNL/caliper-examples/tree/master/apps/Quicksilver`, last accessed 01.11.2019.

[65] "BACnet Stack." [Online]. Available: `http://bacnet.sourceforge.net/`, last accessed 10.10.2019.

[66] "Boost." [Online]. Available: `https://www.boost.org/`, last accessed 10.10.2019.

[67] "Elfutils." [Online]. Available: http://elfutils.org/, last accessed 06.11.2019.

[68] "GNU FreeIPMI." [Online]. Available: https://www.gnu.org/software/freeipmi/, last accessed 10.10.2019.

[69] "Mosquitto." [Online]. Available: https://mosquitto.org/, last accessed 10.10.2019.

[70] "Net-SNMP." [Online]. Available: http://www.net-snmp.org/, last accessed 10.10.2019.

[71] "Intel Omni-Path Software." [Online]. Available: https://downloadcenter.intel.com/download/27220/Intel-Omni-Path-Software-Including-Intel-Omni-Path-Host-Fabric-Interface-Driver, last accessed 10.10.2019.

[72] "OpenSSL." [Online]. Available: https://www.openssl.org/, last accessed 10.10.2019.