2020

Infrastructure for Performance Monitoring and Analysis of Systems and Applications

Ramin Izadpanah
University of Central Florida

INFRASTRUCTURE FOR PERFORMANCE MONITORING AND ANALYSIS OF SYSTEMS AND APPLICATIONS

by

RAMIN IZADPANAH
M.S. University of Tehran, 2014
B.S. University of Tehran, 2011

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, FL

Fall Term
2020

Major Professor: Damian Dechev

ABSTRACT

The growth of High Performance Computer (HPC) systems increases the complexity with respect to understanding resource utilization, system management, and performance issues. HPC performance monitoring tools need to collect information at both the application and system levels to yield a complete performance picture. Existing approaches limit the abilities of the users to do meaningful analysis on actionable timescale. Efficient infrastructures are required to support large-scale systems performance data analysis for both run-time troubleshooting and post-run processing modes. In this dissertation, we present methods to fill these gaps in the infrastructure for HPC performance monitoring and analysis. First, we enhance the architecture of a monitoring system to integrate streaming analysis capabilities at arbitrary locations within its data collection, transport, and aggregation facilities. Next, we present an approach to streaming collection of application performance data. We integrate these methods with a monitoring system used on large-scale computational platforms. Finally, we present a new approach for constructing durable transactional linked data structures that takes advantage of byte-addressable non-volatile memory technologies. Transactional data structures are building blocks of in-memory databases that are used by HPC monitoring systems to store and retrieve data efficiently. We evaluate the presented approaches on a series of case studies. The experiment results demonstrate the impact of our tools, while keeping the overhead in an acceptable margin.

To my parents, and my lovely wife, Samaneh

ACKNOWLEDGMENTS

I would like to thank everyone who helped me during my studies, some of whom are acknowledged below.

First of all, I would like to express my sincere gratitude to my advisor, Dr. Damian Dechev, for his excellent mentorship, support, and encouragement. He is always willing to do his best whenever his help is needed and never let obstacles and restrictions to stop me from making progress. His guidance and advice have made me grow to a better version of my self, personally and professionally. For all that, I am truly grateful.

I would like to also thank my dissertation committee members Dr. Liqiang Wang, Dr. Damla Turgut, and Dr. Eduardo Mucciolo for their time and effort. I appreciate their insightful comments and encouragement that help me finish my dissertation.

I have been fortunate to collaborate with brilliant researchers from the Sandia National Laboratories, University of Central Florida, and Open Grid Computing during my Ph.D. journey. I am grateful for all the support and guidance I received from Jim Brandt, Benjamin Allan, Ann Gentile, Yan Solihin, Christina Peterson, and Nichamon Naksinehaboon.

I am also grateful for working alongside my amazing fellow lab-mates from whom I learned a lot. I would like to thank Steven, Deli, Pierre, Christina, Christopher, Lance, Kenneth, Zachary, Victor, Alexander, Justin, Ryan, Shane, Kishore, and Dipanjan for their support, friendship, and the discussions about interesting ideas.

Finally, I would like to thank my loved ones for their sacrifices and unconditional support in my life. My parents have always encouraged me by any means they could even when I was thousands of miles away from them. Thanks to my brother, Amin, and my sisters, Elham and Shima, who

have offered invaluable advice and support that guide me through my personal and professional life.

Most importantly, I want to thank my extraordinary wife and my best friend, Samaneh. She has been there for me in all ups and downs of my Ph.D. journey. I have always felt that I am accompanied by a true warrior when facing challenges. She has made sacrifices to support my decisions. Samaneh, I'm grateful for all the fun, joy, and beauty you have brought to my life. I love you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Making high-performance computing (HPC) applications efficient and reliable at high process counts is challenging and requires a comprehensive knowledge of the application behavior, resource utilization and system state. This knowledge is necessary to understand and mitigate issues in HPC application and system performance. Continuous system monitoring is necessary to gain this knowledge. HPC monitoring systems collect global system information on resource utilization and system state such as network and Lustre usage; CPU and memory utilization; hardware performance counters; environmental information, such as temperatures and fan speeds. On current large-scale systems, the collected data for monitoring can be many TB/day [34, 18].

In this dissertation, we tackle these problems in HPC system monitoring:

- Enabling real-time troubleshooting and feedback to system components and applications.
- Streaming application performance data for system monitoring.
- Providing the infrastructure support for durable transactional data structures that are needed by the in-memory databases persisting monitoring data.

By designing and building tool-sets, we show that real-time analysis and application performance data streaming can contribute to better understanding of the system and timely responses to various events in the system. Also, we show that by using the persistent memory technology, we can build efficient data processing systems. Our experimental results demonstrate the effectiveness and efficiency our proposed tools and software.

In this chapter, we describe the issues with existing approaches, the importance of these problems, and the motivation for solving them. We present a high level overview of our solutions and

contributions. Finally, a summary of the dissertation and organization is provided.

1.1 Motivation

Real-time troubleshooting and feedback to system components and applications relies on the ability to perform low latency analysis and to expose the results to application and system components, such as resource managers. While monitoring systems may support *in-situ* processing at the point of data collection (e.g., if the collection is performed by a script), more often the analysis is done in post-processing off-system (e.g., in a database). Storage and processing of large data sizes can be demanding, making it difficult to obtain results in a timely fashion. Moreover, data that could be key to understanding is either not collected or not retained for analysis. Lower latency access to results can be obtained by incorporating streaming analysis into the monitoring process, but there are trade-offs in features such as latency, overhead, and analysis complexity.

Post-processing provides the best flexibility for analysis construction since we can answer complex questions and perform multiple passes of queries through the data to extract meaningful information. This flexibility comes at the expense of having the highest latency to solution, with results not immediately exposed to platform components. Conversely, *in-situ* processing at the point of data collection can potentially expose the results to platform components. However, this type of processing imposes overhead on compute nodes and incurs complexity when the analyses rely on combinations of data from different nodes. While it can reduce the amount of data for ultimate storage, it is at the cost of losing data that could be used later. *In-transit* data processing at aggregation points on the compute platform can enable analysis at locations where performance impact is not an issue and also provide exposure of the results to the platform components. Also, such processing may reduce the complexity of and alleviate the need for sophisticated post-processing analyses.

In addition to the system level metrics, events from the inside of applications can also have critical performance implications [102] [79]. Extracting application events, especially in a production environment, involves challenges such as minimizing the interference with computations, dealing with the trade-off between the data accuracy and application efficiency, and determining the appropriate time and location of data exposure. Application profiling tools typically take approaches based on large trace collection or statistical sampling of the program counter and call stack. Capturing application behavior with full details introduces overhead in memory, network bandwidth, and storage. Furthermore, this approach negatively impacts the application's ability to execute its operations and perform computations. On the other hand, it is challenging to choose a subset of events and compile proper statistical samples that represent the application's behavior of interest with reasonable accuracy and efficiency. Many of the performance monitoring libraries accumulate data and release them after the program termination [130] [140] [134]. Revealing the performance data at the end of run could be helpful in some cases such as application tuning and optimization that is performed during development and before deployment. However, continuous system monitoring in production environments demands the collection and exposure of data during the run-time efficiently.

Continuous performance monitoring, especially on large-scale systems, produces a huge amount of data [34, 18]. The persistence and process of data with this size are challenging and require efficient infrastructures. With persistent or non-volatile memory (NVM) recently becoming available commercially, there has been a surge of interest in utilizing it not only as a high-capacity main memory (e.g., Optane DC PM with 3TB per socket [77]), but also for hosting persistent/durable data. To enable applications to rely on persistent data, approaches to construct persistent data systems have been proposed. In general, such systems can be categorized based on two aspects. One aspect is whether they support transactions as primitives or not. A transaction supports ACID: *atomicity* (all operations must all succeed or none does), *consistency* (the data structure state is

consistent before and after the transaction), *isolation* (concurrent executions of transactions appear to take effect in some sequential order), and *durability* (effects of a transaction are not lost upon a power failure). Another aspect is whether the persistent data system relies on low-level information, such as reads and writes, or relies on high-level information such as data structure semantics. Table 1.1 illustrates these aspects.

Table 1.1: Taxonomy of persistent data system.

	Transactional	Non-transactional
High-Level (Data structure semantics)	PETRA (This Work)	Persistent data structures
Low-Level (Reads/writes)	Persistent Transactional Memory (PTM)	N/A

In one approach, specific *persistent data structures*, such as list, set, tree, queue, and hash map have been proposed [32, 158, 113, 56, 110]. These data structures allow the application to execute individual operations such as node insertion and deletion in a crash atomic manner. However, while individual operations are crash atomic, transactions are not supported. A problem arises when an application may need to execute not just single operations atomically, but a sequence of operations atomically, i.e. as transactions. For example, consider a transaction that moves a node from one persistent set to another, i.e. {set1.delete(x); set2.insert(x)}. While individual crash atomic operations are useful, a transaction allows both operations to make durable changes to both sets atomically. Executing transactions on data structures is an essential functionality [131], especially in applications such as databases, data analytics tools, and solving complex graph problems [45, 87]. Furthermore, to support a broad spectrum of applications, a more general framework is needed beyond individual data structure designs.

In another approach, such as *Persistent Transaction Memory* (PTM) [142], researchers provide

transactional support by adding durability to general-purpose transactional programming models. PTMs typically use an underlying Software Transactional Memory (STM) to allow for the execution of transactions atomically. Since STM already supports ”ACI”, PTM only needs to add durability (”D”) to STM. Unfortunately, the reliance on STM results in inheriting its limitations. One issue is that STM relies on low-level information of memory accesses (reads/writes) for conflict detection, which is attributed to the problem of voluminous false aborts [68, 20], incurred by false conflicts stemming from high contention on the data structure's points of accesses, e.g., the top pointer of a stack. Transactions that have read the top of the stack will get aborted if another transaction writes to the top of the stack, even though they might not conflict based on the semantics of the data structures, leading to wasted computational resources due to restarting the transactions [68]. Furthermore, supporting durable transactions requires maintaining (undo or redo) logs, which incurs performance overheads.

1.2 Integrating Low-latency Analysis into HPC System Monitoring

To solve the first problem, we present an approach that enables both *in-situ* and *in-transit* processing to address the challenges in low overhead, low latency analysis and in the exposure of results at arbitrary locations. We implement our method within an existing HPC monitoring system, Lightweight Distributed Metric Service (LDMS) [3]. LDMS is used in monitoring large-scale HPC systems such as NCSA's Blue Waters [109] Cray XE/XK system with 27,648 nodes. Data collection intervals are order of 1 minute down to sub-second, thus resulting in substantial data to be processed for analysis. LDMS is well suited for the integration of analysis within its architecture because of its support for a) plugins that operate on the data [53], b) node-level exposure of data and c) arbitrary communication topologies. This flexibility enables us to place analysis modules at arbitrary locations in the monitored network and use the results of those analyses to

provide feedback throughout the system (e.g., application processes, resource managers).

In LDMS, plugins exist for data collection (getting data into the infrastructure) and for storage (getting data out of the system). We enhance the LDMS architecture by adding the infrastructure for streaming data processing and for handling the transformed data within the system in the same way as the collected data, thus providing a uniform format for data consumers. Our enhancement, called a *transform module*, enables authorized users to provide arbitrary data transformations, at arbitrary points within the monitoring system's communication topology. Our flexible and low-overhead method enables monitoring tools to provide low latency feedback to system components and applications. It provides the capabilities to perform run-time troubleshooting with near-past data by eliminating the need for storage before analysis. Furthermore, our approach supports research on historical data by enabling analysis results to be included in with the raw data to be stored.

1.3 Production Application Performance Data Streaming for System Monitoring

To solve the second problem, we present an approach to collect application level events and provide a production-time status of the application. Our approach can integrate with HPC monitoring tools to support continuous system monitoring in production environments. Independent components of this approach make it suitable to work with different programming paradigms and monitoring tools. In our approach, we utilize an efficient inter-process communication (IPC) method to convey data from the application profiler, which is the data provider, to the data consumer in the performance monitoring tool. The application profiler collects data through dedicated counters that are embedded in the application's points of interest. The HPC monitoring tool exposes the provided data at periodic intervals during the application execution. The data exposure at the execution time enables efficient software-level performance data streaming, which is necessary to provide

monitoring services in production systems seamlessly.

Run-time performance analysis tools employ event tracing through instrumentation or sampling approaches to monitor and evaluate HPC applications [2, 134, 130, 108, 91, 92, 140]. Methods used for instrumentation include modifying the source code, changing the binary code, and function call interception through linking with specialized libraries such as tools based on Message Passing Interface (MPI) profiling layer [88]. An extensive amount of information about the desired parts of the application with details is collected to characterize its behavior. However, these methods usually add significant overhead to the system regarding memory, network bandwidth, and storage. The high overhead of instrumentation makes these methods unsuitable to apply to many code regions in applications or in production. Sampling methods collect statistical performance data at intervals or when interrupted by external events [130]. In this approach, performance analysts should tune the sampling intervals to achieve the optimal solution based on specific efficiency goals. Profiling tools that work based on this model provide the most accurate data at high frequencies with a considerably high cost. On the other hand, they are more efficient at low frequencies, but less informative and prone to detail loss [4]. Current approaches either lack the necessary efficiency to be utilized in production systems or support only post-mortem analysis that does not present online data about application events during the execution. Also, developers face challenges when analyzing applications that scale to larger parallel systems [18].

The novelty of our approach is its efficiency in providing exposure of software level events while the application is running. In our approach, we collect events through software level counters and expose them to the data consumer in the HPC monitoring tool during the application execution. We design and implement a tool-set based on our approach and demonstrate its impact using a case study of the analysis of a scientific application. Our tool-set consists of three components: application profiler, shared memory index, and a sampler. The application profiler collects information about the software level events. The shared memory index provides a mechanism to locate the data

collected by the application profiler. The sampler utilizes the shared memory index to expose the data collected by the application profiler instances periodically.

We integrate our approach with the Lightweight Distributed Metric Service (LDMS) system [3], a monitoring system used on large-scale computational platforms [18]. LDMS provides the infrastructure to gather streams of performance data efficiently while keeping the overhead low. The scalability of LDMS allows us to build a tool-set to monitor large-scale applications. Furthermore, the design of LDMS supports plug-in sub-modules [53]. The sampler component of our tool-set operates as an LDMS sampler plug-in and is independent of the specific metrics collected or the application. We demonstrate our approach using applications implemented with MPI. MPI is one of the most common standards for the development of large-scale scientific applications. The MPI profiling interface (PMPI) [88] allows us to instrument many HPC applications without the modification of their source or binary code. The application profiler component of our tool-set is implemented to profile use of the MPI API, though our method may be adapted to any other instrumentation approach producing counters.

Our streaming based method enables run-time operational analysis and troubleshooting in HPC applications. By continuous, efficient, and direct access to application events, monitoring tools can generate software performance data stream as well as the general system state. The software level metrics exposure supports the performance analysis based on exploring correlations between the system events and application internal events. This analysis helps users to understand the application behavior and possible performance bottlenecks. Our approach is not trace-based, and it avoids generating large data sets in the application's hardware environment. It allows processing performance data elsewhere while the application is running. Data stream processing allows for gaining immediately useful insights by computing functional combinations of data. Our tool-set can benefit a researcher doing application characterization without interfering with the computation. For example, our tool-set can provide insights that can be used in combination with other

data sources in an automatic application phase detection approach. This can be achieved without looking at source code or binary instrumentation.

We demonstrate the impact of our method using an open-source HPC application, Nalu [40]. Nalu has been chosen as one of the representative simulation codes to be used as a performance benchmark for the Trinity Capability Improvement Metric [5]. It is a representative of implicit codes that have been developed under the Advanced Simulation and Computing (ASC) [6] program. We show how our tool-set enables us to efficiently identify patterns in the behavior of the application without any scientific domain knowledge or access to the source code. We leverage LDMS to collect system level performance data as well as software level data and explore the correlation between the system and application events. Also, we show how our tool-set enables quick issue mitigation by assisting in the detection of anomalies in the behavior of the application. We run tests on two different architectures to understand the interaction of our tool-set with the operating system and applications on different HPC platforms. These architectures include a system enabled with the latest generations of Intel Xeon Phi code-named Knights Landing and another system equipped with Intel Xeon processor.

Our overhead study shows our method imposes at most 0.5% CPU overhead on the application.

1.4 Persistent Transactional Non-blocking Linked Data structures

To solve the third problem, we propose a new approach for constructing a persistent transactional data system for linked data structures (PETRA). PETRA can be used as an efficient infrastructure for building in-memory databases for processing performance monitoring data. Our approach, PETRA, is the first one that relies on a transactional approach but combines it with the high-level information from the data structure semantics. Beyond having a unique approach, the goals of our

design are:

- **High performance**: low overheads added to achieve durability.
- **High scalability**: performance scaling well with increasing thread counts.
- **Non-blocking**: there is guaranteed system-wide progress.

PETRA achieves high performance by keeping the number of cache line flushes and memory fences low. The key to high performance is that data structure semantics provides substantial advantages over using low-level memory accesses. Consider descriptor objects [64], commonly used in the design of lock-free data structures [44, 52, 33, 65, 155, 156]. Our descriptor object (*transaction descriptor*) contains the information needed to execute a transaction. All nodes that are accessed by a transaction hold a reference to a shared transaction descriptor. We observe that transaction descriptors have all of the required information to execute the transaction, and can be utilized to verify the consistency of the underlying data structures after a crash and correct possible inconsistencies. These observations lead us to leverage transaction descriptors as redo logs instead of introducing additional logging constructs. Removal of explicit logging not only leads to fewer instructions to execute, but also relaxes the ordering constraints between persistent memory operations, leading to the removal of many persist barriers (cache line flushes and store fences). In contrast, PTMs typically need to enforce orderings between writes to the log and writes to the actual data structure. In our methodology, enforcing the persistence of the transaction descriptors at the end of the transaction is sufficient. Furthermore, since the transaction descriptor is already needed to manage concurrency, leveraging it to manage crash consistency adds only minor additional overheads.

PETRA achieves high scalability due to working at the data structure semantics level, hence removing false aborts. Aborts only occur when transactions conflict on nodes of the data structure based

on its semantics. Again, this is achieved thanks to the transaction descriptors that we use. Our transaction descriptors enable a node-based conflict detection scheme that does not rely on transactional memory nor require the use of an additional data structure. Furthermore, when conflicts are detected, PETRA uses the transaction descriptors to implement a conflict recovery strategy based on the interpretation of the logical status of nodes instead of explicitly revoking executed operations in an aborted transaction.

Finally, PETRA is an obstruction-free transactional persistent data system, where system-wide progress is guaranteed. It does not rely on using locks, hence deadlocks are not possible. With locks, if a thread holding a lock is pre-empted or if it crashes, no system progress can be made. With a persistent data system, if the lock is persistent, post-crash recovery is not simple as we have to recover the thread that held the lock at the crash time [32]. With PETRA, no locks are used. PETRA utilizes transaction descriptor that ensures global progress through a helping mechanism. If a thread is pre-empted while executing a transaction, another thread can help complete the transaction.

1.5 Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 presents the background and related work. The design and evaluation of the low-latency analysis integration into HPC system monitoring is described in Chapter 3. Chapter 4 proposes the the design and experimental evaluation of the production application performance data streaming for system monitoring. In Chapter 5, we present the design and evaluation of PETRA, persistent transactional non-blocking data structures. Finally, we conclude in Chapter 6.

1.6 Publications

This dissertation is written based on peer reviewed papers published by the author of this dissertation in collaboration with other authors. Chapter 3 includes content from a paper that is published in the International Conference on Parallel Processing [80]. Chapter 4 is based on a paper that is published in the ACM Transactions on Modeling and Performance Evaluation of Computing Systems journal [78]. Some material from each of these papers has been used in the current chapter, and the following chapter.

1.7 Software

The software developed during this dissertation is publicly available on GitHub repository of the Ovis project at https://github.com/ovis-hpc/ovis.

CHAPTER 2: BACKGROUND

In this chapter, we introduce the fundamental concepts and tools used in our work. Next, we place our proposed approaches in the context of related work.

2.1 LDMS Monitoring Framework

Our approaches leverage LDMS's capabilities in data collection, transport and aggregation that enable continuous monitoring in large-scale systems. In the LDMS framework, daemons run on the resources to be monitored (e.g., compute nodes), and utilize plugins for data sampling and storage. Daemons can also play the role of an *aggregator*.

Daemons can aggregate data from other LDMS daemons over various transports, including Infiniband, iWarp, and Ethernet, in arbitrary communication topologies. The purposes of aggregation can be arbitrary cases such as feeding data to other consumers, and writing out to some permanent storage system [3, 80]. Aggregators shift the load and overhead of storage and aggregation to cluster service nodes that do not run HPC applications. This offloading reduces the overhead on compute node as much as possible and eliminates possible interference between computations and storage.

Multiple aggregation points can be configured to pull data from disjoint and overlapping sources, including other aggregators, as shown in Figure 2.1. This flexible communication topology is a key for performing low latency analysis and feedback that requires bidirectional data flow. This infrastructure design allows us to perform transforms at arbitrary locations where the computational overhead is not a concern (e.g., on "aggregation" nodes), but still expose the transformed data where it is needed. Aggregation nodes in the LDMS context are nodes dedicated primarily to

aggregation of large collections of sampled data sets (metric sets). Because aggregation nodes are dedicated to this functionality, the computational intensity of analytics performed on these nodes has no adverse effect on application performance within the computational infrastructure.

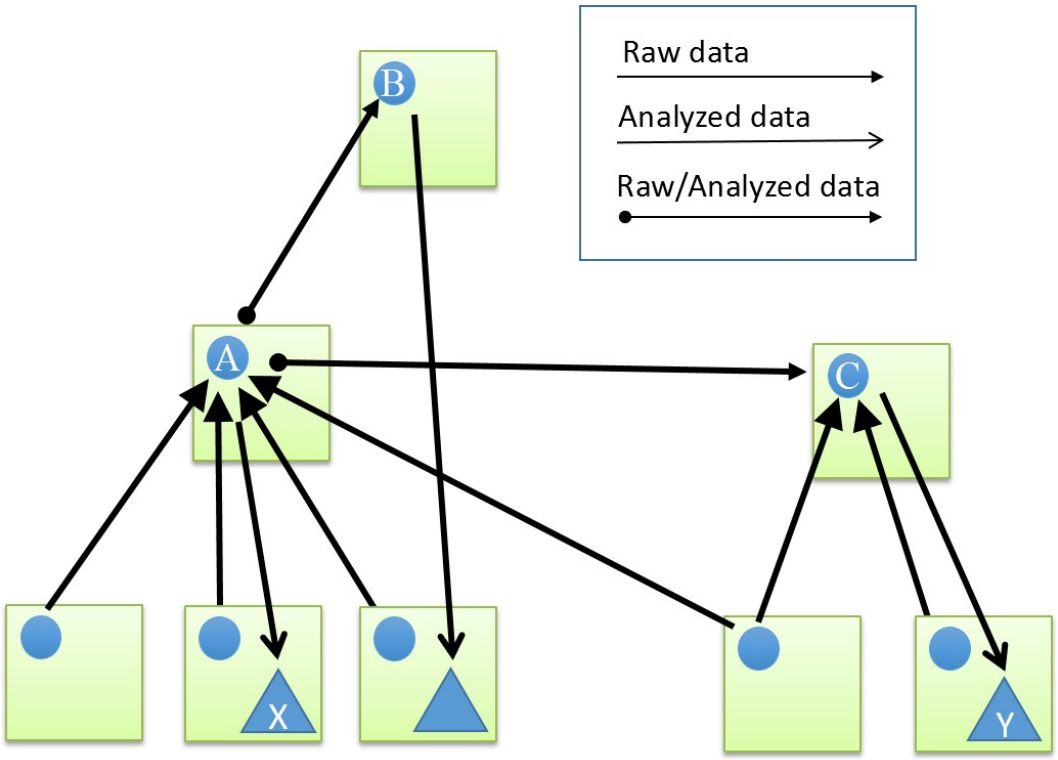


Figure 2.1: LDMS supports arbitrary communication topologies. Green squares indicate nodes; blue circles indicate LDMS daemons; blue triangles indicate applications. Arrows indicate the direction of aggregation and data accessing by applications. For example, A can aggregate metric sets from 4 of LDMS daemons; B can aggregate any metric sets generated on A or aggregated by A; application X can access metric sets on A; application Y can access metric sets on C. A command line query tool can also query any daemon remotely to obtain its data.

The typical process of collecting data, or *metric* values, from compute nodes is as follows. LDMS daemons on compute nodes, which are configured as *sampler daemons*, create in-memory data structures, called *metric sets*, to store the collected data. They periodically sample new metric values using *sampler plugins*. An *aggregator* connects to a set of sampler daemons and then periodically reads and stores, in local memory, metric sets from sampler daemons. An aggregator

might also then store[1] (e.g., write out to a file, or a named pipe for forwarding to a disjoint architecture such as a named pipe to syslog [3]) the metric sets using a *store plugin*. Daemon instances, per-daemon plugins, aggregation setup (including topology and sets to be aggregated), rates, and store parameters are all configuration options.

A metric set consists of meta-data and data sections. The meta-data section retains the description of the set (e.g., metric names, metric types, size of the set). The data section stores both *meta metrics*, which have values that are either constant or rarely change (e.g., component ID corresponding to a metric set), and *data metrics* which store values of frequently changing metrics.

An LDMS daemon stores only a single set of values for its current metric sets. An LDMS daemon may be queried to get its current metric sets either by an aggregator or via ldms_ls, a query tool that works similarly to how aggregators collect sets from sampler daemons. An annotated example of the ldms_ls output of a metric set is shown in Figure 2.2, including meta-data vs. data sections, and, for the data, metric data types, names, and values.

A store plugin is notified every time an LDMS daemon obtains an update to a metric set for which it has been configured for storing. Before our presented approach, in LDMS, some limited streaming computations and data transformations have been performed using store plugins called *function store* plugins. These plugins [17] perform limited computations and filtering on the metric set data before writing the raw or computed data to storage.

Expanding the store plugin for more general exposure of data and access to the resultant computations would not be as useful and flexible as the ability to support and expose transformed data within the metric set context already handled by the infrastructure. In our analysis integration approach, we overcome this limitation by designing a flexible method to support streaming analy-

[1]Note the different use of *store* to write out as opposed to the daemons which *store* data in memory. We believe this standard terminology will be clear to the reader.

sis. This enables us to take advantage of LDMS's support for arbitrary communication topologies, including bi-directional communications, to minimize the impact of the computations within the compute environment while still supporting access, by both system services and applications, to the transformed data. We can then opt to place the transforms at locations where the computational impact is not a concern while the resultant transformed set can then be pulled to a node and only incur the transport cost.

```
nid00004/cray_gemini_r_sampler: consistent, last update: Tue May 16 00:34:36 2017 [3060us]
  METADATA --------
    Producer Name : curie
    Instance Name : nid00004/cray_gemini_r_sampler
      Schema Name : cray_gemini_r_sampler_nid00004
             Size : 8312
     Metric Count : 144
               GN : 2
  DATA ------------
        Timestamp : Tue May 16 00:34:36 2017 [3060us]
         Duration : [0.000793s]
       Consistent : TRUE
             Size : 1192
               GN : 243772
  -----------------
M u64        component_id                              4
D u64        job_id                                    0
D u64        nettopo_mesh_coord_X                      0
D u64        nettopo_mesh_coord_Y                      0
D u64        nettopo_mesh_coord_Z                      2
D u64        X+_traffic (B)                            1533417918654
D u64        X-_traffic (B)                            0
D u64        Y+_traffic (B)                            12243567675
...
D u64        client.lstats.read_bytes_llite.snx11024   98955581392
D u64        client.lstats.write_bytes_llite.snx11024  131235178
D u64        client.lstats.open_llite.snx11024         18478
D u64        client.lstats.close_llite.snx11024        18478
D u64        client.lstats.flock_llite.snx11024        0
D u64        client.lstats.getattr_llite.snx11024      9124
D u64        client.lstats.statfs_llite.snx11024       24
D u64        client.lstats.alloc_inode_llite.snx11024  9683
....
D u64        loadavg_latest(x100)                      0
D u64        loadavg_total_processes                   190
```

Verbose listing of metric sets shows the MetaData vs Data sizes, the amount of time (Duration) to sample the data and build the set, and the Timestamp of the set.

Metric Set Instance naming convention: <component>/<sampler name>

Data metric values are presented by (type, name, value).

meta metrics (M) vs data metrics (D)

Metrics used in the transform analyses are boxed in red.

Figure 2.2: Output from the ldms_ls command.It shows part of a metric set produced by a sampler plugin written to collect a variety of metrics from a Cray XE/XK system. In Chapter 3, we utilize transform plugins to perform some analyses using the metrics related to the Lustre file system.

LDMS has been used in monitoring large-scale HPC systems such as NCSA's Blue Waters [109] Cray XE/XK system with 27,648 nodes. LDMS has performed efficiently on large-scale production systems, and overhead assessments have demonstrated no significant detrimental system impact [3, 34, 35, 109]. LDMS enables low-overhead performance data streaming using the pull model of aggregators that periodically fetch data from the samplers. This pulling mechanism utilizes the Remote Distributed Memory Access (RDMA) protocol to unburden the compute nodes of the required functionality and overhead for sending data, storage, and failover handling. Clients can connect to aggregators and consume the data stream for various purposes such as analysis, and storage.

2.2 Integrating low-latency analysis

Many widely used HPC monitoring frameworks are intrinsically designed as one-way communication constructs, thus limiting the ability to feed back the data and analysis results to arbitrary consumers. Ganglia [105] and Nagios [122] are invoked periodically on the nodes with the data typically aggregated to a central location. Ganglia is designed to use rrdtool [112] as a back-end database, which can then be used for off-system analysis and visualization. Nagios supports some limited failure alert features based on predefined thresholds. ElasticStack [43] ingests input data into a publish-subscribe message bus, LogStash, and does server-side analysis with ElasticSearch. This model can ingest data from sources such as Ganglia and Nagios but does not address analysis on the compute platform. Even if on-node analysis were supported, the message bus interaction and message parsing would incur additional overhead, as opposed to LDMS's RDMA, primarily pull-based model. Collectl [28] can be configured to report delta rather than raw values but not to perform arbitrary analyses. It is not designed for easy general configuration of arbitrary communication topologies. TACCStats [46] has in the past collected data on the node to a file which has then

been collected off the machine nightly. While they have recently enabled run-time collection [47] via a daemon-based version of TACCStats to an off-platform site, it still does not intrinsically enable streaming analysis using a general approach as we did in this work. Instead, some limited analyses like maximum and average for certain metrics have been provided. The SOS [147] project appears to share our goals with respect to enabling system wide information sharing, low latency analysis and feedback to both applications and system software. A significant difference is that SOS is a new framework which incorporates an additional daemon per-node that communicates with applications and other data collection entities for data acquisition and uses a SQLite database for both on node and aggregator storage. Its functional scalability, including application performance impact at large scale, has yet to be established. Published information about SOS's online data analysis and automated information migration is insufficient for comparison currently. Our work leverages an existing HPC monitoring framework with proven scalability to 10s of thousands of nodes. Storage of data values for this work is in native ldmsd metric set data structures and whatever backend storage is configured for a particular system. Performance libraries such as PAPI [19] and the perf tool provide limited support for presenting some derived metrics such as IPC (instructions per cycle) on a local node. Our scalable tool enables flexible analysis on application and system resources using various transport protocols in arbitrary communication topologies at runtime.

Communication architectures and tools such as MRNet [124] and AMQP [111] could be used for the transport part. However, all the capabilities for data collection, analysis, and exposure of both raw and transformed data in a uniform way would have to be built. Note that MRNet targets a tree-based overlay and hence the setup to enable arbitrary and bi-directional communications could become quite complex. It does not currently support RDMA which therefore increases its innate overhead for data transfer and feedback based on analyses. MRNet explicitly targets filtering of data, which is an analysis, at the tree aggregation points to reduce message size. It was used in

collecting platform data with reduction [16]. Here, we integrate low latency streaming analysis, and not merely reduction, at arbitrary locations in the entire HPC system, and support building complex analysis units from basic transform plugins using the chaining capability. AMQP theoretically would support more arbitrary communication topologies because of its publish-subscribe architecture. However, typical applications of this model use self-describing messages, which are inherently of larger size than the LDMS messages that send only data.

Various tools for big data systems and streaming databases exist (e.g., [1][31][115][62][24]). These tools provide one-way communication constructs using query interfaces for analyses. This limits the ability to feed back the data and analysis results to the compute platform. In our work, we leverage the bidirectional infrastructure of the LDMS framework to enable information feed back to various system components and applications. This makes the decision-making process in the system software and applications more informed and environment-aware.

Pipeline capabilities that support filters for analysis and visualization exist in architectures such as the Visualization Tool Kit, VTK [126]. VTK has a relatively sophisticated model for handling the pipeline due to the need to handle possibly complex issues such as visible components in the 3D rendering of objects. Our work seeks to incorporate a much more limited capability for chaining analyses without the added complexity of writing code utilizing VTK's language bindings.

Analysis capabilities such as SciPy [129] tools are being applied to computations in HPC analysis (e.g., [101]) as are no-SQL databases (e.g., [14]) in support of data storage. Efficiency in the analysis, insertion, and retrieval can provide a performance benefit for data processing, however they would not entirely obviate the desire to compute and expose data on the platform. Similarly, the innate collection and transport data capabilities would need to be integrated. Such analysis capabilities could, however, be used to facilitate the building of the analyses required in the plugins.

2.3 Application performance data streaming for system monitoring

A wide range of performance monitoring and analysis tools exist that rely on tracing or profiling approaches to collect data.

Cloud based tools such as Graphite [38] and Prometheus [120] provide a different set of services for applications and systems that are hosted on the cloud. Graphite relies on third party tools for data acquisition services and is mainly used for storage and visualization purposes [146, 86] in conjunction with monitoring data collection tools in the cloud environment. While Graphite and its data collectors provide a convenient monitoring mechanism by supporting string format communication and script based commands, these characteristics limit the frequency, accuracy, and latency of data collection. Prometheus text-based exposition format does not separate meta-data from data, limiting the sampling frequency and latency due to the parsing. These limits in addition to the overhead on the bandwidth and computing resources availability to the primary applications prevent wide applicability of these tools in HPC environments. While these tools require extremely capable nodes for data aggregation, our tool builds upon LDMS, which can aggregate on the same class of hardware the collectors run on. In addition, unlike the equipment requirements of cloud based solutions like these tools, we build our tool upon LDMS that has demonstrated that it requires only two aggregator nodes to serve more than 27000 clients on NCSA Blue Waters [3, 109] and a similar number of nodes on Trinity [34, 35, 100].

Some general purpose system monitoring tools mainly focus on monitoring and analyzing resource utilization in the system and not the application performance. Some tools such as Ganglia [105] have limitations in scalability, and Nagios [122] has a different purpose of failure alerting. LIKWID Monitoring Stack [123], which targets small-to medium-sized commodity clusters, focuses on utilizing hardware performance counter data and does not provide software level performance metrics. GUIDE [138] is another scalable tool for data collection and analysis that focuses on

the entire HPC ecosystem and does not provide application level insights. Our tool-set streams software level performance data and has a broader purpose than these tools.

The gprof tool [60] supports both sampling and instrumentation mechanisms separately for sequential programs. It creates a report at the end of execution and is not considered a scalable tool for monitoring large-scale parallel applications. HPCToolkit [2] provides a performance analysis framework that directly analyzes the binaries of applications without instrumentation. From this perspective, HPCToolkit and other tools based on this model [57, 75, 127] are mostly used for MPI call-path profiling and performance tuning during development, releasing the performance data after program termination. We demonstrate our approach using MPI applications. However, our design is not limited to a specific programming pattern. Our tool-set combines sampling and profiling to stream accurate data efficiently and supports run-time analysis.

Periscope [13] takes a distributed search approach to detect performance issues specified by users and relies on other libraries to collect the performance data. The iterative strategy of Periscope requires the application to follow a specific programming pattern with a region, such as the main loop in scientific simulations, that repeats during the execution. TAU [108] offers offline performance analysis of applications based on tracing and profiling approaches and generates the result after the application termination. Vampir [91] provides visualization of performance traces and profiles generated by other tools. Scalasca [58] performs post-run analysis using parallel trace replay for specific performance issues.

TAU could be used as a partner tool in the instrumentation of modifiable applications. However, the binary or source rewriting mechanisms used by tools like TAU limit their applicability. These limitations make the tools not suitable choices for many production HPC settings where application builds have been through specific verification and validation steps. Our streaming based approach enables run-time analysis and does not enforce a specific programming paradigm.

Periscope, TAU, Vampir, and Scalasca use Score-P [92] measurement system as the underlying layer to record performance data. Score-P instruments applications and stores the performance information in forms of profiles or traces. Besides the support for the offline analysis, Score-P provides the interface to Periscope to perform the online analysis. However, it does not generate a stream of performance data during the execution of the application. Tools that use Score-P must follow a specific work-flow, which typically involves recompiling and relinking the application using the Score-P compiler. Our implemented tool-set does not require the user to recompile or relink their application for monitoring.

MPItrace [130], IPM [134], and mpiP [140] specifically support the performance analysis for MPI applications. MPItrace supports both tracing and sampling based approaches. The sampling mechanism is built on top of PAPI [19] and supports sampling based on available hardware counters. IPM and mpiP collect detailed and statistical performance data respectively, and both generate the results after the program termination.

We implement a tool-set based on our approach and demonstrate it using MPI applications. Our tool-set enables operational run-time analysis using performance data streaming. By integrating this approach with LDMS, we can explore the correlations between events that are happening in different layers of the software and hardware stack.

2.4 Persistent Transactional Non-blocking Linked Data structures

In this section, we provide an overview of the concepts, techniques, and tools used by PETRA, and discuss the related work that proposed transactional executions of data structures.

2.4.1 Related Work

2.4.1.1 Non-Blocking Progress Assurance

Concurrent data structures can either be *blocking* or *non-blocking* based on the progress guarantee they provide. Blocking data structures do not provide system-wide progress guarantees, such as the completion of one or more operations. Non-blocking data structures allow scalable and thread-safe access to shared data while providing progress guarantees that are not possible with the use of locks. The correctness of such algorithms is typically established by relying on a key correctness condition, linearizability [67], and its more relaxed derivatives.

One of the techniques employed in non-blocking data structures is the use of descriptor objects [44, 52, 33, 65, 155, 156], which are shared objects that keep the required information for executing data structure operations and allow several updates to take effect atomically. This shared object allows for cooperation between threads. When a thread stalls, another thread can read the descriptor object for it and execute its operations according to the information provided by its descriptor. Such a thread is referred to as a *helper thread* and the act as *helping*.

2.4.1.1.1 Persistent Data Structures

We consider a concurrent data structure "persistent transactional" if it provides the full ACID guarantee [61]. Specific persistent data structures such as list and set [32, 158], tree [113, 23, 96, 139, 151, 32, 144, 97], queue [56], hash map [110, 128, 32] have been proposed, with each operation designed to keep data in the containers persistent. The idea of building the structure of the containers upon recovery have been proposed recently [158, 106]. However, to the best of our knowledge, they do not provide native support for transactions, i.e. not allowing programmers to

arbitrarily group multiple operations to execute with ACID properties. To execute transactions on these data structures, they can be integrated with a PTM-based approach. PETRA introduces a methodology for adding transactional persistency to concurrent linked data structures, instead of specific non-transactional data structures.

2.4.1.1.2 (Persistent) Transactional Memory

The typical transactional programming model allows for the transactional execution of atomic blocks specified by the programmer. Transactional memory was first proposed as a hardware primitive (HTM) [69] that relies on the cache to buffer a thread's speculative state. Implementations on real processors (e.g., Intel TSX) do not guarantee forward progress as a transaction with size exceeding the cache size aborts the transaction. STM [132] is its software counterpart that is not restricted with transaction size but incurs higher performance overheads. It performs conflict detection using memory-level read and write sets. STM's lack of knowledge about the high-level data structure semantics introduces false aborts, which restricts concurrency and scalability.

Recently, researchers have added durability to HTM [85, 133], or STM to form PTM [142, 9, 8]. To support durability, software PTMs rely on a (redo or undo) logging mechanism. The log structure introduces substantial performance overhead. For example, an undo log must persist before data structure can persist and in the case of using redo logs, a traversal of the write set for concurrent read operations is needed [103]. Examples include PMDK [116], Atlas [21], JUSTDO [81], iDO [98], NV-Heaps [27], Mnemosyne [142], Romulus [30], and generic STM transformation methods [152]. Some PTMs such as JUSTDO [81], PHTM [9], and PHyTM [8] require special hardware support not available in commercial processors. Mnemosyne [142] is built on top of TinySTM [50] and uses a redo log. Romulus [30] relies on data redundancy instead of persistent logs. OneFile [121] is a variant of Romulus based on a universal construction [66], which tends

to be expensive because of the overhead that is mainly incurred by its maintenance and instrumentation [67, 49, 26, 99]. There are multiple problems with PTMs. First, PTMs are built on top of STMs or HTMs, hence relying on low-level information of memory accesses (reads/writes) for conflict detection, which leads to the problem of voluminous false aborts that are not caused by actual conflicts at the data structure semantic level. Second, PTMs rely on logging; log updates add performance-robbing cache line flushes and memory fences, and additional ordering constraints. Finally, many STMs rely on locks, hence most software PTMs are blocking. In contrast, PETRA removes false aborts by utilizing high-level data structure semantics, adapts transactional descriptors instead of traditional logging, and provides non-blocking transactional behavior.

2.4.2 PETRA Baseline Selection

2.4.2.1 Choice of the Non-Durable Baseline

Recall that the goals of PETRA design are high performance, high scalability, and non-blocking progress. Given that there are many choices of non-durable transactional data structure methodologies (Table 2.1), we must select one that is most relevant for PETRA design goals. While all the listed transactional data structure methodologies provide atomicity, isolation, and consistency, they differ in their conflict detection, transaction logging, and progress guarantee. STM does not use data structure semantic conflict detection, hence is not suitable for PETRA. Transaction logging is an appealing feature for enabling durability since persisting the log is sufficient for recovery. Lock-free Transactional Transformation (LFTT) [156] and STMs such as Word-based Software Transactional Memory (WSTM) [55] and Object-based Software Transactional Memory (OSTM) [55] provide transaction logging. Software Transactional Objects (STO) [55] logs a transaction's actions such as validation, installation, and rollback in a tracking set. Transactional Data Structure Libraries (TDSL) [136] does not explicitly log transactions, but their methodology could

be extended to log the read/write set per transaction for recovery. Transactional Boosting [68] is a lock-based approach that maintains a log of inverse operations for rollbacks that is updated as the transaction executes. If the transaction does not finish executing, then only a subset of the transaction's operations are included in the log. To extend this approach for the recovery of an entire transaction, the complete list of transaction operations should be logged regardless of its execution status. LFTT and lock-free variants of STM are the only methodologies that provide non-blocking progress. From the table, we decided to choose LFTT to build PETRA on, due to its non-blocking progress, transactional logging, and data structure semantic conflict detection.

Table 2.1: Comparison of Non-durable Transactional Data Structure Methodologies. They all support Atomicity, Isolation, and Consistency, but not Durability. Check-marks indicate existing features.

Transactional Methodology	Semantic Conflict Detection	TX Logging	Non-blocking Progress
LFTT [156]	✓	✓	✓
TDSL [136]	✓		
Transactional Boosting [68]	✓		
STO [71]	✓	✓	
STM [55]		✓	✓

2.4.2.1.1 Overview of LFTT

LFTT enables developers to build non-blocking transactional data structures using existing non-blocking containers. Unlike generic STM-based approaches, LFTT leverages the semantic knowledge of the data structure to allow *commutative operations*, i.e., operations that have no dependencies on each other, to proceed concurrently in a non-blocking manner. This eliminates most false aborts due to access conflicts. LFTT also uses this knowledge to find conflicts in non-commutative operations through a node-based conflict detection mechanism. The progress guarantee in transac-

tions based on the LFTT approach could degrade to obstruction-freedom if concurrent transactions access the same keys in reverse order. To guarantee lock-freedom in such cases a pre-processing technique can be used to ensure that transactions access the keys in the same order. To synchronize transactions, LFTT uses a cooperative technique [10] allowing threads that share a node in their transactions to help complete each other's operations by including the required information in a descriptor. The helping scheme reduces not only false aborts, but also many true aborts, by allowing the thread that detects the conflict to execute the delayed transaction associated with the conflicting node. Also, in case of an abort, LFTT uses a logical rollback technique that cancels the effects of an aborted transaction by reversing the logical interpretation of the status of the nodes. This method eliminates the overhead of physical rollback and wasting CPU cycles, while guaranteeing system-wide progress. Several variants of LFTT extend this methodology to support more linked data structures such as dictionary, and binary search trees with features such as dynamic transactions, wait-freedom, transactions among multiple data structures, and transactions on non-linked data structures such as dynamic arrays [95, 94, 157, 153, 93].

CHAPTER 3: INTEGRATING LOW-LATENCY ANALYSIS INTO HPC SYSTEM MONITORING

In this chapter[1], we explain our approach to integrate low latency analysis into HPC system monitoring. We start by discussing the motivation for integrating low latency analysis. We explain the design of the transform module and its components. Next, we demonstrate the impact of our approach by performing experiments and analyses using different performance data sources.

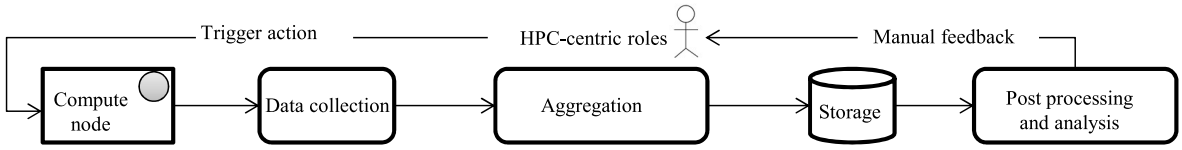
3.1 Motivation

Large-scale HPC systems utilize a variety of resources (e.g., network and file systems) that are shared by both processes of a parallel application and those of other concurrently running applications. Contention for these resources can create congestion that can severely impact application performance and system efficiency. While monitoring and storage of system data can enable root cause analysis through post-processing when problems have been identified (typically after a failure or apparent lack of forward progress of an application), this approach is not well suited for run-time feedback to utilize the results of such analysis.

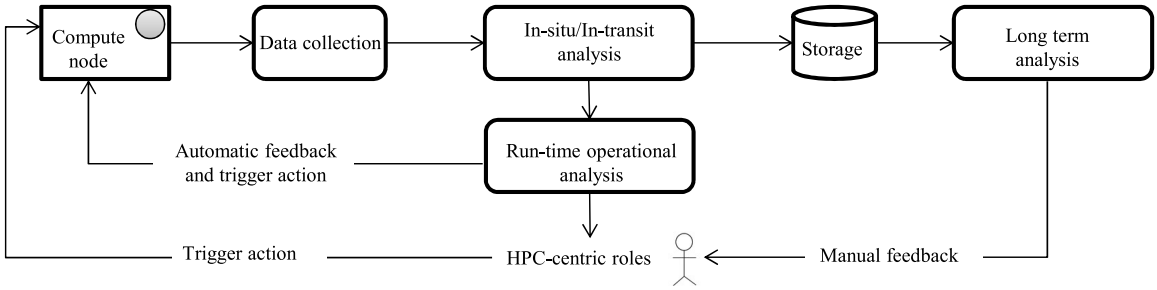
Figure 3.1 provides a comparison between typical resource utilization and performance analysis in HPC monitoring systems based on post-processing (Figure 3.1a) versus our approach based on integrating in-situ processing on the node or in-transit processing at data aggregation points (Figure 3.1b). In both approaches, lightweight processes collect data (e.g., error counters, network performance counters, file system access counters) on resources (e.g., compute nodes, LNET routers, admin nodes). Data flows from sampling points (shown as compute nodes here) to aggregators

[1]This Chapter includes content from a paper that is published in the International Conference on Parallel Processing [80].

that manage its disposition after collection. An off-platform machine or cluster typically stores this data for future use for analysis for troubleshooting and feedback. The processing performed by the data collector is typically minimal. Typical use cases of the stored data are troubleshooting and threshold-based feedback (e.g., component temperature too high, therefore take some action). Troubleshooting is typically driven by a failure of some sort and therefore post-processing with a human in the loop can be feasible, timewise. Defensive threshold-based, automated low latency feedback is typically incorporated into system components and not exposed to system administrators.



(a) Traditional post-processing data analysis approach. The data is collected on the compute node and then flows to the aggregation nodes, which manage and send data to the storage system. The analysis is performed on the historical information and HPC-centric roles, such as application developers/users and system administrators, manually trigger required actions based on the feedback.



(b) Proposed integrated streaming analysis approach. We enhance the approach with run-time operational analysis, which can be performed either *in-situ* on node or *in-transit* at data aggregation points. Based on this information, on-node consumers, such as applications can receive feedback and automatically trigger appropriate actions. This information can also enable users/admins to make more informed decisions with the additional run-time analysis. Long-term analysis is still performed with the same approach as described in Figure 3.1a.

Figure 3.1: Differences in the processes of performance analysis: post-processing vs integrated analysis.

A missing monitoring-related capability is the utilization of the monitored data to enhance appli-

cation and system efficiency through run-time analysis and exposure of appropriate information. Most situations would only benefit from a reasonably low latency feedback cycle that could incorporate functional combinations of data from multiple, possibly global, sources. Examples include the use of global network utilization/congestion assessment by a workload manager for job placement and of global Lustre file system utilization by queuing systems for job launch decisions or by application processes for open/close/read/write timing decisions. While HPC monitoring systems globally collect the types of data that can be used for these analyses, none provide utilities for run-time sharing of such information with applications or system access to the results. Other information such as power consumption, thermal information, storage bandwidth, memory contention, and CPU utilization can enable certain applications to realize benefits using a low latency feedback approach. For example, in multi-core applications, memory contention has been used to manage concurrency [119], and thread contention analysis has helped to tune non-blocking algorithms [79]. Computations, based on hardware performance counters, of node and job level flop rates, cache misses rates, and cycles per instruction are used in assessing application resource utilization [11].

In the remainder of this chapter, we describe our modular and extensible approach to providing capabilities for streaming analysis on either counters or state data in the context of the Lightweight Distributed Metric Service HPC monitoring framework.

3.2 Low latency Analysis Integration

In this section, we describe our approach to integrate low latency analysis into HPC system monitoring. We explain the design of the transform module and its components. Also, we discuss the considerations and challenges in our design.

3.2.1 Transform Design

We enhance LDMS by designing and implementing the transform module to enable streaming analysis. LDMS collects and transports metrics in a well-defined format. We design the transform module to take inputs as metric sets and generate output also in the metric set format. This decision adds flexibility to our design and enables transformed sets to be transported and stored in the same way as raw sets.

Some of the functionalities that the transform module adds to the LDMS framework are displayed in Figure 3.3. In this figure, several sources provide data streams. Transform modules take the provided metric sets and perform the labeled operations using different components within this module.

Our transform module consists of two components: transform management and transform plugins. *Transform management* handles the data flow from receiving an update of a locally obtained or remotely pulled metric set to obtaining the final output of a transform chain. The *transform plugin* is a new plugin type in the LDMS framework. It takes metric sets or individual metrics as input and derives a transformed set, of one or more metrics, as the output.

3.2.1.1 Transform Management

We enhance LDMS daemons with the transform management functionality. The transform management creates transform instances according to the user configurations, chains the transform instances, and optionally passes the output of transform chains to a store plugin to retain the derived metrics. Transform management supports use of both locally obtained and remotely pulled metric sets for transformation and manages all the transform configurations.

Some calculations in analyses need input from multiple data sources and metric sets. The transform management component supports multiple transform plugin instances. This enables application of transform plugins to a variety of configurations and multiple input sets.

Configuration of a transform includes specification of its input sets. Upon receiving a set update, an LDMS daemon passes the updated set to each transform instance. Each instance uses the configuration to determine whether it needs the set for its calculation or not. When all metric values for the transforms' output set have been updated, the transform instance notifies the host LDMS daemon that the updated output set is ready to use.

While our enhancement to the LDMS infrastructure enables users to develop a single transform plugin that can perform all desired computations, the ability to chain plugins can provide reusable elements from which more complex calculations can be built. This flexibility may entirely obviate the need for a user to design and implement any transform plugin. For example, in the Lustre analysis presented in Section 3.3.2, the computed values include ratios of the rates of some metrics (e.g., *file opens*) per compute node relative to the total number of opens from all compute nodes. The transform management component supports this by providing a path from the output of one transform plugin to another. The LDMS daemon, enabled with transform management, passes the output set to each transform instance in the chain according to the provided configurations.

Using a uniform format for the transform's input and output allows us to treat these just as any regular metric set in the LDMS framework. This includes the final output as well as any intermediate output sets, which are input sets of another transform instance. LDMS daemons can pass transform output sets to a store plugin as well to store either the final output set or the intermediate sets.

We discuss the challenges introduced by the design decisions described here in Section 3.2.2.

3.2.1.2 Transform Plugin

Transform plugins are responsible for parsing and interpreting their specific configuration, generating transformed sets, performing mathematical manipulation, and letting the transform management know when a transformed set is ready. Each plugin receives an input set(s), performs its mathematical manipulation if metrics of the set are needed in the derivation, and then updates the corresponding output set. To reduce clutter when the intermediates in a chain of transforms are not useful as an end goal, transformed sets can be marked as *unpublished*. Only *published* sets can be aggregated and will appear in the $\mathrm{ldms\_ls}$ output. For example, in Figure 3.4, which shows a sequence of transforms performed on an aggregator, only the final per-node sets need to be published.

Our flexible design allows the user to develop transform plugins and perform arbitrary analyses on the performance data stream. More complex analyses are feasible through chaining the basic plugins together. In this work, we implement several transform plugins to demonstrate and evaluate the capabilities of our enhancements to LDMS using a case study.

The following list defines the transform plugins implemented for our case study and evaluation. In all equations, capital letters represent a metric set where a subscript shows a metric within the metric set and a superscript shows the timestamps attributed to a value.

- **delta plugin** calculates the difference of a metric between two consecutive timestamps.

$$delta^{(t)}(M) = N, \;\; where$$
$$\forall i < |M| \;\; N_i = M_i^{(t)} - M_i^{(t-1)}$$

- **rate plugin** calculates the ratio of the delta and the difference of two consecutive timestamps.

$$rate^{(t)}(M) = N, \;\; where$$
$$\forall i < |M| \;\; N_i = \frac{M_i^{(t)} - M_i^{(t-1)}}{\Delta t}$$

- **ratio plugin** calculates the ratio of different metrics in the same metric set at the same timestamp.

$$ratio^{(t)}(M\_num, M\_den) = N, \;\; where$$
$$\forall i < |M\_num| \;\; N_i = \frac{M\_num_i^{(t)}}{M\_den_i^{(t)}}$$

- **sum vector plugin** assumes that input metrics are vectors. It calculates the sum of all elements in a vector at the same timestamp.

$$sum\_vector^{(t)}(M) = \sum_{i=0}^{|M|} M_i^{(t)}$$

- **windowed minimum plugin** calculates the minimum of the metrics over a defined window of timestamps.

$$min\_n^{(t)}(M) = N, \;\; where$$
$$\forall i < |M| \;\; N_i = \min_{\forall s \in [t-n,t]} M_i^{(s)}$$

- **windowed maximum plugin** calculates the maximum of the metrics in a defined window

of timestamps.

$$max\_n^{(t)}(M) = N, \;\; where$$
$$\forall i < |M| \;\; N_i = \max_{\forall s \in [t-n,t]} M_i^{(s)}$$

- **windowed average plugin** calculates the average of the metrics in a defined window of timestamps.

$$avg\_n^{(t)}(M) = N, \;\; where$$
$$\forall i < |M| \;\; N_i = \frac{\sum_{s=t-n}^{t} M_i^{(s)}}{n}$$

- **combine plugin** combines multiple metric sets into a single set according to the user configuration. The plugin assumes that all sets have the same sampling interval in order to ensure it combines the input sets from comparable times.

- **global ratio plugin** operates on a single metric set. Simple user definable associations of the metrics in the set are used to determine a group of metrics to sum. The output is a set with the ratio of each of the individual values to the sum(s). For example, if the set contains the same two metrics (e.g., Active, MemFree) for each of N nodes, the output will contain for each metric 1) the sum of all the nodes' values and 2) the ratio of each node's individual value relative to that sum.

- **separate plugin** separates a single metric set into multiple metric sets according to the user configuration.

These basic plugins can be chained together to compute the quantities of interest given in Section 3.1. In Section 3.3.2, we show how we utilize transform plugins for performing an analysis

on the Lustre file system, based on the raw data displayed in Figure 2.2.

3.2.2 Challenges and Considerations

We enhance the LDMS monitoring infrastructure with a flexible design for the transform module to support low latency analysis within the monitoring system. This enables the authorized user to perform low latency analyses using multiple metric sets from different data sources. It supports the modularity of the required calculations for analysis by chaining a series of transform plugins. The ability to place any of the transforms at any location along the data communication path where its input set(s) are available and from where its output can be used adds an extra level of flexibility to our design. This flexibility enables target where to apply memory and CPU in the system to perform analyses. Note that memory and CPU will be approximately the same globally but we can define where it happens and do it on the fly using this functionality.

With increased flexibility, comes increased need for consideration in transform design. Considerations include:

3.2.2.1 Location variation: Time skew between nodes

Sets are timestamped with the transaction time of the plugin generating the set. Time skew between a node creating an input set and a different node performing the transform can result in timestamp offsets that make associations between sets difficult. We include a flag that enables inclusion of time metrics from the input set(s) into the output set(s) to facilitate such associations. This becomes more complex as transforms are chained and/or more input sets are supported.

3.2.2.2 Data Types

Generically, transforms must address computations for all data types in handling the input, within the computation, and in the output type. Overflow must be recognized and handled. LDMS supports signed and unsigned 8, 16, 32, and 64-bit integers, floats, doubles, chars, arrays of such, and generic data blobs. In the example transform plugins mentioned in Section 3.2.1.2, we support multiple numerical input types and generate output in double to provide consistency, particularly for mixed input types where multiple values are used in a computation. However, this is not a restriction in the transform management and infrastructure. New plugins can be developed that support other data types.

3.2.2.3 Invalidating data/computations in a transform chain

Invalid results must be flagged and propagated throughout the chain of computations. For example, divide by zero in a ratio transform or negative delta time in a rate computation due to a clock reset would result in invalid results if used as input in a subsequent transform. Such cases may be indicated by values such as NaN or inf, depending on the type, or by a validity flag carried with each variable. In the *function store*, there is a validity flag carried throughout every computation. This doubles the output size of the data, but it is immaterial since the store plugin functions off-host. For the current set of transforms, we handle the invalidity in the data; where necessary we will implement it as an optional feature on a per-metric basis to enable the user to keep as small as metric set as desired.

3.2.2.4 Missing input sets or multiple input sets with time offsets

A transform plugin with multiple input sets faces additional complexity in addressing combinations of data that may be offset in time. These types of plugins handle such offsets based on the operations within the transform plugin and the implication of the offset on computations. For example, in our combine transform plugin, we check the timestamps of the input sets to avoid two possible issues: (1) combining input sets significantly mismatched in time and (2) lack of progress if a particular input set does not arrive in a timely fashion (e.g., due to node failures). Timely output is ensured with use of the validity flag to indicate results based on incomplete data.

Judicious writing of the collector can minimize the need to write multiple input set transforms, since metrics collected via the same collector will be in the same metric set. For example, the full set of metrics in the sampled set in Figure 2.2 includes Lustre, network, GPU, and CPU data. This design choice was motivated by the desire to have a single timestamp associated with all metrics [18], which eases some processing, particularly for large-scale systems. The legitimacy of doing this is dependent on the time required for collection of all metrics. Since all plugins (samplers, transform, and store) carry with them the time of the full transaction of the plugin, this can be used to get an idea of the possible time offset between the collection of the first metric in a set and the last. In the case of this set, sampling takes around 425us without GPU data, and 800us with GPU data, both of which are small compared to the 1 to 60-second intervals of collection typically used.

3.3 Experimental Evaluation

In this section, we demonstrate the impact of our approach by performing experiments and analyses using different performance data sources. We study the overhead introduced by our enhancement

to the LDMS framework. Also, we present a case study of using the transform module for performance monitoring and analysis of an HPC file system.

3.3.1 Overhead evaluation

In this section, we evaluate the overhead impact of the transform module. The transform module is integrated within the LDMS framework. LDMS is proven to perform efficiently on large-scale production systems, and overhead assessments have demonstrated no significant detrimental system impact [3, 34, 35, 109]. Transform plugins run on LDMS daemons that operate as a part of the LDMS infrastructure to leverage its efficiency and scalability.

To assess the impact of the transform module, we utilize a 2 chassis, 64 node Cray XE/XK testbed system, called Curie, with a Cray Gemini Interconnect and a Sonexion Lustre file system. This testbed is representative of the type of hardware of one of our target platforms, Blue Waters. For our overhead analysis assessments we utilize datasets from $/\mathrm{proc}/\mathrm{meminfo}$ and $/\mathrm{proc}/\mathrm{vmstat}$ sources as a representative sets. These sets have 43 and 97 metrics respectively. We utilize the *rate* transformations in the experimental configurations shown in Figure 3.2. Each experiment runs for 30 minutes with a sampling interval of 1 second. Baseline experiments were intended to measure the inherent CPU time of sampling and aggregation. We repeat the same experiments with the *rate* transform running on nodes to measure the CPU overhead of transform plugins.

The additional overhead of running the transform on sampler nodes can contribute to interferes with the operations of applications running on compute nodes. By moving the transform operations to aggregator nodes, we eliminate the overhead from the compute nodes, while still enabling low latency access to the transformed data. Furthermore, aggregators have the additional benefit of having access to additional producer's metric sets, which can enable generation of metric sets with global information aggregates. This information can then be utilized by nodes in assessing current

global levels of shared resource utilization.

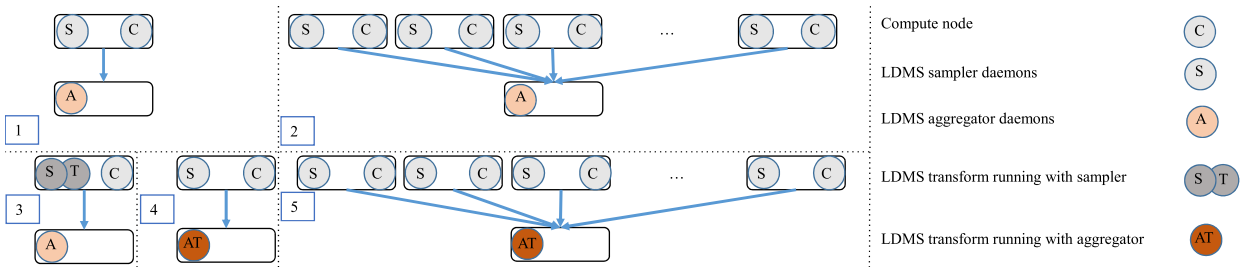


Figure 3.2: Experimental setup for measuring transform module's overhead. Samplers are collecting two metrics set on each compute node. Aggregators are pulling data from either one sampler, specified by experiments 1,3, and 4, or ten samplers, specified by experiments 2 and 5. The experiments specified by sections 1 and 2 of the figure are run without any transform plugins. Experiments 3-5 are run with active transform plugins on nodes.

Table 3.1 shows the total CPU time in microseconds per metric set on each node. In the first row, the aggregator is pulling data from one sampler node, and in the second row, ten sampler nodes are providing data to a single aggregator. Around 16 microseconds of overhead for running the *rate* transform plugin on the aggregator is seen. By chaining multiple transform plugins, this overhead increases linearly due to the sequential wiring of transform plugins.

Table 3.1: CPU time per sampled metric set for the baseline and the case that is running a *rate* transform plugin. Different number of compute nodes are used to show the efficiency of using the in-transit approach on aggregators compared to the in-situ analysis on compute nodes.

Experiment type		Baseline	Transform
# of aggregator nodes	# of compute nodes	(μs)	(μs)
1	1	63.9	71.4
1	10	57.3	73.5

We run the transform plugins on the aggregators pulling data from different number of compute nodes. Running the transform plugin on one aggregator enables low latency analysis from all of

the compute nodes. Achieving the same results using the in-situ approach requires running one instance of the transform plugin on each compute node. In addition, the overhead per sample on sampler daemons is higher than the overhead per sample on aggregators.

For the baseline, for each sample, a sampler daemon needs to parse /proc/meminfo and /proc/vmstat to update each metric in the meminfo and vmstat sets. By contrast, for each sample, an aggregator only performs an RDMA read operation which does not consume any CPU cycles on the sampling host. Since the read operation is per set, the aggregator does not iterate through each metric in each set. Hence, the overhead per sample on aggregator daemons is lower than the overhead per sample on a sampler.

3.3.2 Case study: Lustre file system analysis

A case of general interest is discovering and assessing contention in shared parallel file systems. Since Lustre is a popular shared parallel file system utilized extensively on large-scale HPC systems, we focus on contention for both meta-data services and read and write bandwidth. Note that there are caching effects on the client side that we do not address here. In this section, we demonstrate the applicability of our work to a transform analysis of Lustre metrics. Providing the types of analyses demonstrated here could be of use to applications and system services running on the platform hosts in load balancing, partitioning, and scheduling if low latency exposure to applications and system services were possible.

The goal of this analysis is to make available to consumers on each node and off-platform, data about each node's relative use of the file system. To minimize the impact on the compute nodes, we leverage the bi-directional transport capabilities to perform the computations entirely on the aggregators for all nodes, and we present small memory footprint output metric sets to per-node consumers. Figure 3.3 presents the design of our experiment to perform an analysis on Lustre

metrics using transform plugins. We chose the instantaneous read, write, open, and close values from the original metric set provided by the LDMS sampler monitoring the Lustre file system status displayed in Figure 2.2.

The first layer of transform plugins shown in Figure 3.3 is the *rate* plugin that operates on the performance data streams generated by the compute nodes. The results generated by the *rate* plugin are fed into the windowed function plugins to calculate the average, minimum, and maximum values reported by the *rate* plugin in the previous step. In the third layer of transform plugins, the *combine* transform merges all of the results from the previous step into one metric set. The *global ratio* plugin works on this single metric set and calculates each node's share of the system resource utilization. Next, a *separator* plugin operates on the *global ratio*'s output and extracts metrics from it to output multiple metric sets, one for each compute node. Finally, compute nodes receive this information.

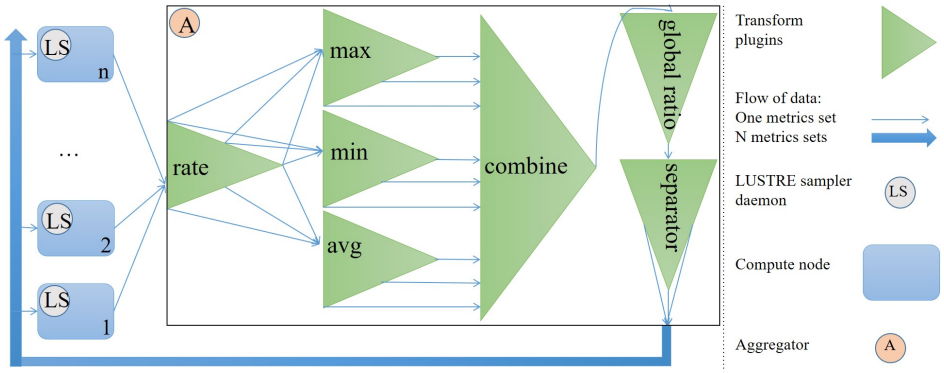


Figure 3.3: Transform sequence and positions in the computation of the transformed Lustre metrics.

Figure 3.4 shows representative subsets of intermediate and final transformed metric sets. Naming conventions for the set instances are determined by the transform, for example, the rate transform appends _rate to the input set name (top of the figure); support for more flexibility in handling naming conventions is in work. The final analysis results for node nid00004 indicate that its

average read operations are roughly 10% of all read operations performed in this Lustre system (marked in green). The final, smaller (size is shown in the final meta-data), per-node metric sets are available to be pulled back to or queried by LDMS daemons or system software on the compute nodes.

This global knowledge of Lustre system component utilization, provided by transform plugins, can be leveraged by application processes for open/close/read/write timing decisions. In addition, queuing systems can make informed decisions for launching jobs based on this information. Also, this global knowledge can improve load balancing [41]. Run-time determination of the relative per-node file system demands can play an important role in system administration. Run-time availability and exposure of such data would be of benefit to those seeking to resolve issues. These data can be used to identify the causes of high load on the file system and to identify imbalances in an application's resource demands.

Our approach for the in-transit analysis at aggregation points enables run-time operational analysis with no overhead on compute nodes. Our experimental evaluations demonstrate the capability of the transform module to support low latency analyses within the monitoring system efficiently.

```
nid00004/cray_gemini_r_sampler_rate: consistent, last update: Tue May 16 00:34:35 2017 [154528us]
D d64        client.lstats.read_bytes_llite.snx11024    0.000000
D d64        client.lstats.write_bytes_llite.snx11024   0.000000
D d64        client.lstats.open_llite.snx11024          0.000000
D d64        client.lstats.close_llite.snx11024         0.000000
```

(1) Rate transform operates on each node individually over last time step

Rate transform output metric set is input to the windowed avg, min, and max transforms

```
nid00004/cray_gemini_r_sampler_rate_avg_n: consistent, last update: Tue May 16 00:34:35 2017 [155962us]
D d64        client.lstats.read_bytes_llite.snx11024    1256485.542593
D d64        client.lstats.write_bytes_llite.snx11024   1256485.542593
D d64        client.lstats.open_llite.snx11024          0.099954
D d64        client.lstats.close_llite.snx11024         0.099954
```

(2) Window avg/min/max transforms operates on each node individually over time window

Windowed avg, min, and max output metrics sets for all nodes are input to the combine transform which merges them into one dataset (unshown). (3)

Output of the combine transform is input to the global ratio transform

```
curie/lnet_rates_all_global_ratio: consistent, last update: Tue May 16 00:34:36 2017 [203812us]
  METADATA --------
    Producer Name : curie
    Instance Name : curie/lnet_rates_all_global_ratio
      Schema Name : lnet_rates_all_global_ratio
             Size : 14112
     Metric Count : 132
               GN : 1
  DATA ------------
        Timestamp : Tue May 16 00:34:36 2017 [203812us]
         Duration : [0.001161s]
       Consistent : TRUE
             Size : 1104
               GN : 331717
  ----------------
D d64        cray_gemini_r_sampler_nid00010_rate_avg_n__client.lstats.read_bytes_llite.snx11024 0.198067
D d64        cray_gemini_r_sampler_nid00010_rate_avg_n__client.lstats.write_bytes_llite.snx11024 0.396454
...
D d64        cray_gemini_r_sampler_nid00004_rate_avg_n__client.lstats.read_bytes_llite.snx11024 0.100202
D d64        cray_gemini_r_sampler_nid00004_rate_avg_n__client.lstats.write_bytes_llite.snx11024 0.075414
...
D d64        cray_gemini_r_sampler_nid00004_rate_max_n__client.lstats.read_bytes_llite.snx11024 0.115037
D d64        cray_gemini_r_sampler_nid00004_rate_max_n__client.lstats.write_bytes_llite.snx11024 0.101542
...
D d64        rate_avg_n__client.lstats.read_bytes_llite.snx11024_global 12539496.433430
D d64        rate_avg_n__client.lstats.write_bytes_llite.snx11024_global 16661235.373810
D d64        rate_avg_n__client.lstats.open_llite.snx11024_global 0.899944
D d64        rate_avg_n__client.lstats.close_llite.snx11024_global 0.799949
```

(4) Global ratio transform operates on a single input metric set. It calculates per-node window avg, min, max values relative to the total node sums. It outputs a single metric set

Global ratio needs all nodes' data for the computation. Output set has all nodes and global data as metrics

Output of the global ratio transform is input to the separator transform

```
curie/lnet_rates_all_global_ratio_nid00004: consistent, last update: Tue May 16 00:34:36 2017 [210050us]
  METADATA --------
    Producer Name : curie
    Instance Name : curie/lnet_rates_all_global_ratio_nid00004
      Schema Name : lnet_rates_all_global_ratio_separate
             Size : 2104
     Metric Count : 24
               GN : 1
  DATA ------------
        Timestamp : Tue May 16 00:34:36 2017 [210050us]
         Duration : [0.000002s]
       Consistent : TRUE
             Size : 240
               GN : 60313
  ----------------
D d64        rate_avg_n__client.lstats.read_bytes_llite.snx11024 0.100202
D d64        rate_avg_n__client.lstats.write_bytes_llite.snx11024 0.075414
D d64        rate_avg_n__client.lstats.open_llite.snx11024 0.111067
D d64        rate_avg_n__client.lstats.close_llite.snx11024 0.124951
D d64        rate_max_n__client.lstats.read_bytes_llite.snx11024 0.115037
D d64        rate_max_n__client.lstats.write_bytes_llite.snx11024 0.101542
...
D d64        rate_avg_n__client.lstats.read_bytes_llite.snx11024_global 12539496.433430
D d64        rate_avg_n__client.lstats.write_bytes_llite.snx11024_global 16661235.373810
```

(5) Separator transform operates on a single input metric set and and outputs a subset of data to each of several output sets. The smaller output sets are then available to on-node consumers

Output set for nid00004. Read operations are 10% of the total.

Figure 3.4: Example metric sets at stages in a transform sequence. The goal is per-node sets of windowed avg, min, and max of quantities relative to the set of nodes' total usage. (1) Rate output for nid00004 (2) Windowed avg output for nid00004 (3) Combine transform (unshown) (4) Global ratio output. The component for the metric set instance is the system, with the nodes encoded in the metric names. (5) Separator output produces per-node sets – nid00004 shown. Computationally and memory intensive transforms can be done on aggregation nodes, and the final smaller set is then available to be pulled back to or queried by the compute nodes for use by system software and applications. All sets are exposed in the same way.

3.4 Chapter Summary

In this chapter, we presented the design of chaining transform capabilities to support streaming analysis within an existing production HPC monitoring framework, LDMS. The transformed data is supported by the same structures as the collected data, thus enabling the transformed data set the same flexibility in transport and the same exposure as the collected data. We leverage the transport flexibility of LDMS to enable placement of computationally intensive transformations on hosts where the overhead would not adversely affect an application and yet be able to transport the result to hosts, including those hosting applications, where the results are needed.

We have shown the viability of our implementation for a case with production-relevance: run-time determination of the relative per-node filesystem demands. Run-time availability and exposure of such data would be of benefit to those seeking to identify the causes of high load on the filesystem and to identify imbalances in an application's resource demands.

CHAPTER 4: PRODUCTION APPLICATION PERFORMANCE DATA STREAMING FOR SYSTEM MONITORING

In this chapter[1], we describe our approach to collect and expose performance data. We start with an overview of our model and the system components. We introduce each component and explain the design decisions. Next, we explain the work-flow and how different components work together to stream performance data. We discuss the challenges that we face in our tool-set's design. Finally, we present case studies of using our tool-set for performance monitoring and analysis of Nalu and evaluate the overhead impact of using our tool-set when Nalu is running.

4.1 System Overview

Figure 4.1 displays a high-level overview of the system. Our design consists of three components: an application profiler, a shared memory index, and a sampler. The application profiler, which is described in Section 4.2, collects information about the software level events. The shared memory index, which is described in Section 4.3, provides a mechanism to access the data collected by the application profiler. The sampler, which is described in Section 4.4, utilizes the shared memory index to expose the data collected by the application profiler periodically.

The shared memory index is identified using a name that is assigned to a region in the shared memory area on the system. The sampler and the application profiler are configured with the same index name to access the shared memory index. The shared memory index consists of entries that each correspond to an application process being monitored. The application profiler instance puts data collected about its events in a specific location in the shared memory. Each index entry

[1]This Chapter is based on a paper that is published in the ACM Transactions on Modeling and Performance Evaluation of Computing Systems journal [78].

contains information about this shared memory location. The sampler stores the information about each application in an instance of a data type called *box*. A *box* holds information such as the metric set description and the data collected by the application profiler.

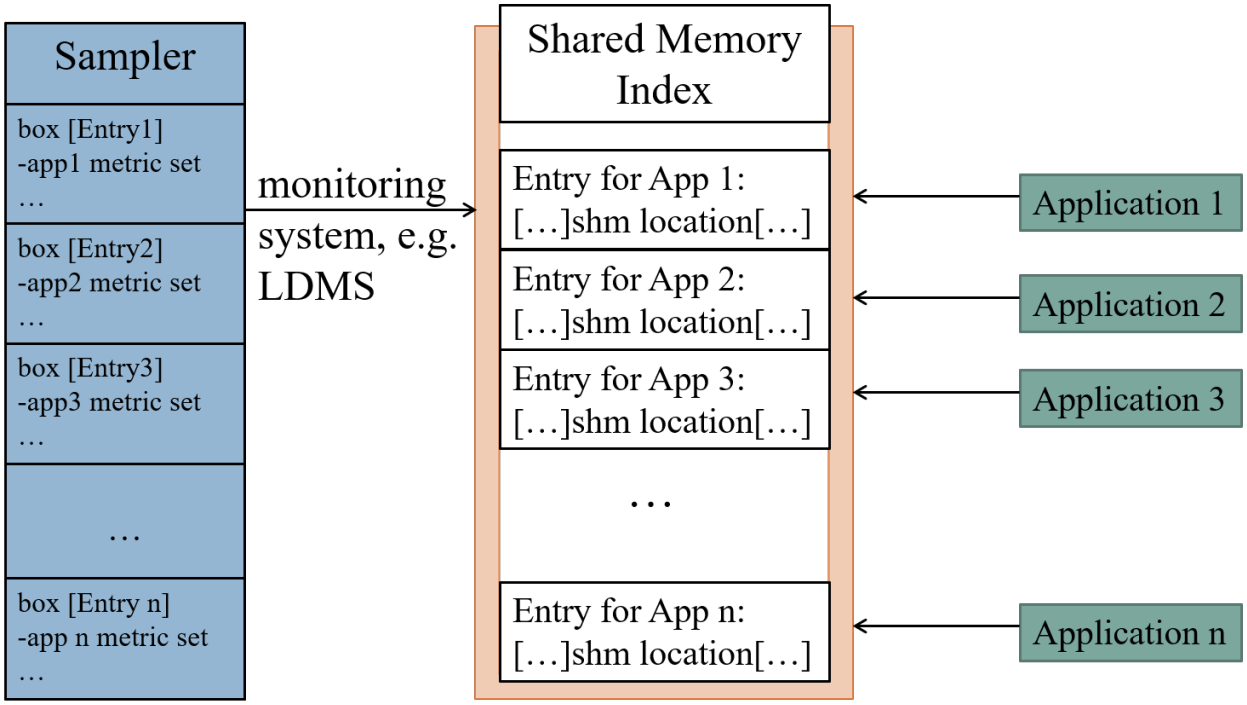


Figure 4.1: A high-level overview of the system components. Application profilers collect performance data and share them with the sampler using the shared memory index. Shared memory index consists of one entry per application. Sampler allocates a box for each entry in the index.

As we explain in Section 4.5, these components cooperate with each other in our tool-set to stream performance data that is collected from the application. In Section 4.6, we discuss the challenges involved in our design.

4.2 Application Profiler

In our approach, the application profiler is the data provider to the sampler. The profiler collects information about application level events. The application profiler assigns dedicated counters to each application event defined at the software level to perform the data collection. The profiler provides information about events to the sampler via the shared memory index. This flexible design enables us to develop customized profilers to accommodate any specific type of application and programming pattern. The programming paradigm used and the application developer determine the types of events to be collected and monitored.

We design and develop an MPI profiler as a proof of concept. Our MPI profiler leverages PMPI to collect data from the application. This method requires no modifications of either MPI applications or libraries. We rely on the Linux library preloading feature to inject our profiler at run-time. This mechanism imposes no burden on the programmer to recompile the application, except in statically linked applications.

Listing 4.1: Example pseudo-code that demonstrates the functionality of MPI profiler query interface

```
SELECT EventID, Rank, count (Bytes), count(Calls)
FROM MPIEvents
WHERE MessageBased = True AND MessageSize GreaterThan 10
GROUP BY EventID, Rank
```

Our MPI profiler provides a query interface to retain flexibility and convenience to study various events with different properties. In this interface, the user can specify events with particular characteristics and features that need to be measured. This mechanism enables the user to focus on the essential features that have impacts on a specific aspect of application behavior rather than being

exposed to a massive amount of data that can be generated. Multiple aspects may be specified as shown by an example pseudo-code in Listing 4.1. This query instructs the MPI profiler to perform the measurement per MPI rank and only measure the message size and number of calls for message-based events where the message handled by that event is greater than 10 bytes.

The application profiler uses the shared memory index to expose events to the sampler. In Section 4.3, we explain the shared memory index design.

4.3 Shared Memory Index

To access the information about software level events in applications we need to employ a method of inter-process communication (IPC). Various communication facilities allow processes to exchange data with one another. These facilities can be divided into two categories: *Data-transfer* and *Shared memory* [90]. The first type of communication involves kernel functions in the data transfer, but in shared memory approaches, processes can communicate with each other by placing data in a shared memory region. This direct communication makes the shared memory approach a fast method because it requires few system calls or kernel operations. Unlike data-transfer facilities, shared memory allows one process to make the data visible to any number of processes that share the same memory region. The shared memory approach enables a process to access shared data like any other memory area in its virtual address space. This model fits well with application designs that need to maintain a shared state between multiple processes, such as the approach we present in this chapter.

Most UNIX based systems provide the support for shared memory. This type of IPC should be employed cautiously as the operations on the shared memory may need to be synchronized. The synchronization could negatively impact the advantages of fast communication in shared memory

approaches depending on the use case. In our implementation, there are a few scenarios, e.g., initial configurations, that we use synchronization facilities such as semaphores to protect shared resources.

We choose POSIX shared memory over System V shared memory in our design as POSIX features better fit our implementation needs. The advantages of POSIX IPC include the simplicity of the interface, consistency with the traditional UNIX file mode, and support for reference counted objects [90]. Furthermore, POSIX IPC mechanisms are guaranteed to be thread-safe, but System V IPC techniques do not provide such a guarantee [125]. In our approach, multiple processes, including parallel applications, utilize shared resources. In Section 4.6, we discuss how we use reference counts to manage shared resources.

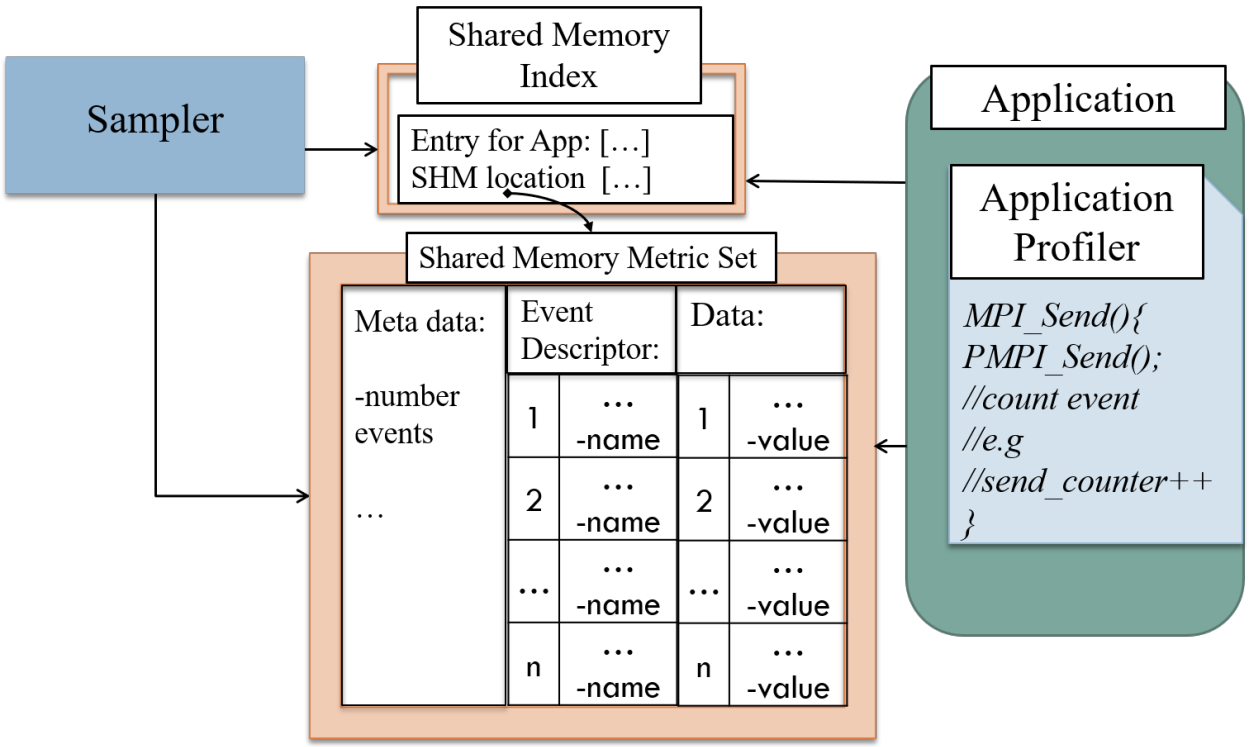


Figure 4.2: The structure of the metrics set in the shared memory index.

In our implementation, the shared memory index consists of entries corresponding to application processes being monitored. Each entry stores information about the application instance including the location in the shared memory of the application instance's profiling data. This information is shared with the sampler using the assigned index entry for the application. The data layout that is shared between the application profiler and sampler is determined dynamically at run-time. This data layout is called metric set, which defines a collection of metrics and provides data about it.

Figure 4.2 displays the general structure of a metric set that is stored in the specified shared memory region within our tool-set. There are three chunks of contiguous memory associated with each metric set. First is the meta-data that provides general information about the metric set as a whole. The number of monitored events is an example of such information that is stored in the meta-data. Second is the general information describing the elements of the data chunk, such as the event name. The final piece of data represents the counter values correspond to collected events by the application profiler. No history is retained within a profiler or the shared memory index, and, the allocated memory is overwritten as the profiler provides new information about an event.

The shared memory index allows the application profiler and sampler to access the events information simultaneously. In Section 4.4, we describe the role of sampler within our tool-set.

4.4 Sampler

We leverage the Lightweight Distributed Metric Service (LDMS) for data collection, transport, and aggregation. The typical process of data collection starts with running LDMS daemons on compute nodes. These daemons, which are configured as sampler daemons, store the collected data in LDMS metric sets. A sampler daemon stores only the latest set of values for each of its metric sets and no sample history is retained within a plugin or the host daemon.

The streaming part of our methodology relies on the sampler component. The sampler is a daemon plug-in that periodically copies data provided by a source on a monitored computing node and exposes the collected data to consumers. The sampling frequency, which is defined by the user, determines the resolution of data. Running a sampler that utilizes the application profiler's shared memory as the data source allows us to stream application performance data related to software level events while the program is running.

We implement the plug-in *shm sampler* within the LDMS performance monitoring framework. This sampler stores the information from each application profiler instance in an instance of a data type called *box*. A *box* holds information such as the metric set collected by the application profiler and other information that is required by LDMS infrastructure. The sampler dynamically discovers the metric set defined by the profiler and creates the *box* corresponding to it.

In Section 4.5, we explain how different components within our tool-set cooperate to generate the stream of application-level performance data.

4.5 Work-flow

Figure 4.3 displays the typical work-flow in our approach, annotated with the associated actions for each component. Both sampler and application profiler should agree on an index name that is used to obtain the access to the shared memory index. This index is created at the first call to *open index* by any process. Unix *shm_open* API provides a file descriptor that can be used to handle the shared memory region with the specified name. Subsequent calls to open index do not allocate a new area in the shared memory, and the file descriptor will be used to access the index. Each process creates a memory mapping in its virtual address space using the provided file descriptor. All processes can access the same index and use the mapped memory to share information.

The user configures the application profiler to monitor events of interest. After the configuration, the profiler provides the required information about events to the shared memory index. At this step, the application profiler registers itself as the updater of a metric set in the index using the provided information. The shared memory index allocates an entry to the application and creates a shared memory metric set as specified in Figure 4.2.

The sampler periodically scans the index to find new or modified entries and updates its local *box*es. The sampler detects changes in the metric set using a generation number associated with each index entry allocated to an application. When an application profiler modifies the configurations of events, the generation number of that entry is incremented. The sampler, as a data consumer, determines if the generation number it has stored locally matches that associated with the current produced data by the profiler.

For each new application, the sampler registers itself as a reader of its metric set. Also, the sampler configures a new metric set in the performance monitoring library infrastructure. After the setup, the application profiler counts the specified events, as they happen in the application, and updates the counters associated with events. The sampler reads the counter values at intervals and exposes them to the consumers.

This work-flow within our tool-set yields application performance data streaming for system monitoring. This cooperation between multiple processes in a shared environment is not without challenges. In Section 4.6, we explain some challenges involved in this work-flow and how we deal with them.

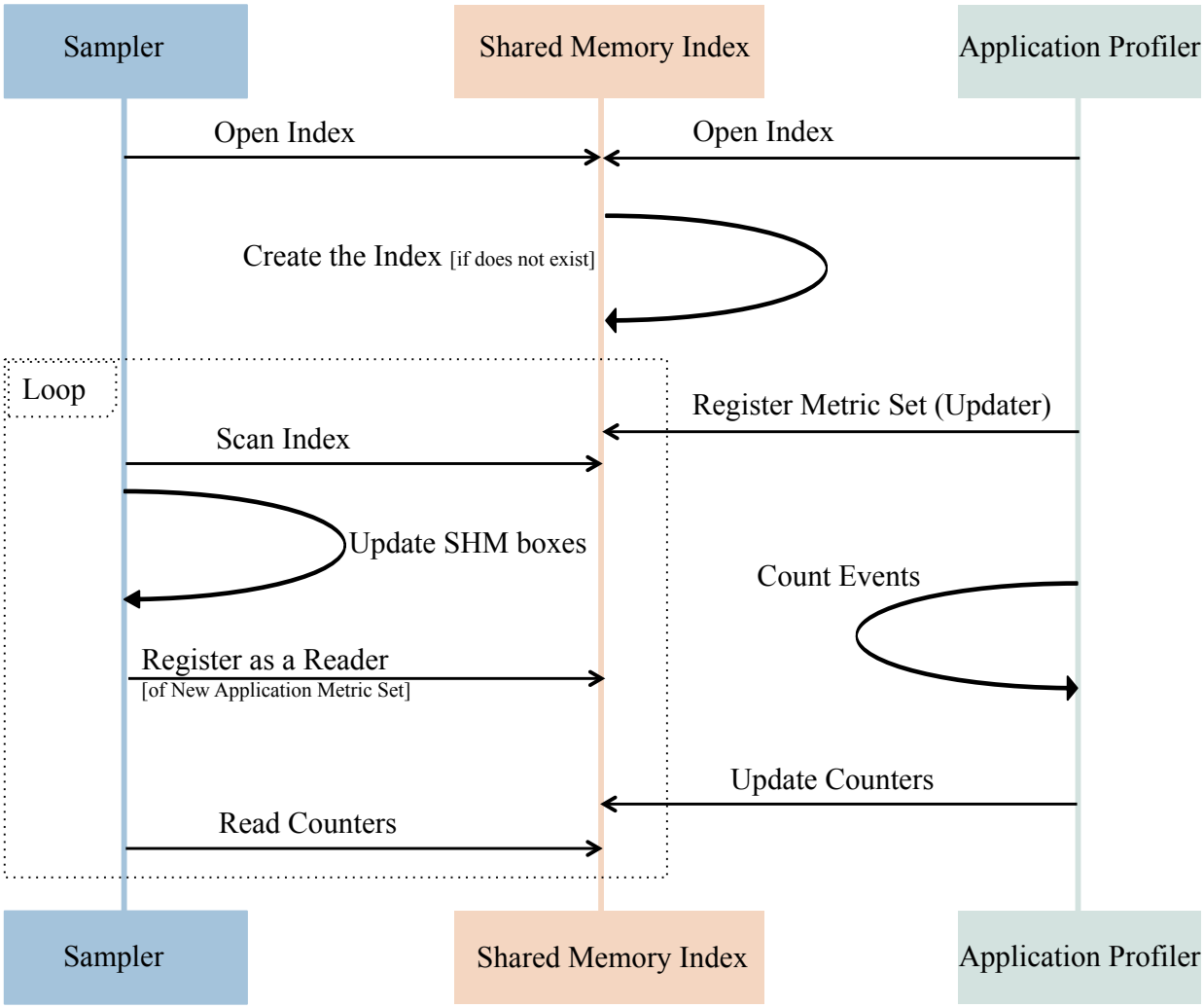


Figure 4.3: The work-flow of our methodology

4.6 Handling Process Failures

In our approach, the index is shared between the sampler and the application processes (e.g., MPI ranks). The shared resources used in the index and its entries need to be handled carefully in different scenarios such as crashes. Figure 4.4 exhibits different scenarios using the sampler and two applications. We explain how we deal with shared resources in different situations. In this

figure, we have 12 different sets of states and each set shows the state of the sampler, two sample applications called app 1 and app 2, and an index named I.

We start from the initial state where applications and the sampler are not available. We assume that the index name is somehow defined in this environment (e.g., defined by the user). Next, we start the sampler and the applications, and we assume that the sampler wins the race and starts first. Since the sampler starts first and finds out that the index is not available, it creates index I.

Now, we are in state 2 that the sampler and applications are started, and the index is created. Applications open the index and register their metric sets in the index. We assign reference counts to metric sets defined in index entries to keep track of each indexed set. In our example, state 4 displays that the metric set M with two references and the metric set N with one reference is defined in the index. App 1 is writing into the metric set M and app 2 is writing into the metric sets M and N. The sampler is reading all sets included in index I.

Now, let us assume that app 2 finishes its execution. App 2 deregisters metric sets M and N before it terminates. We decrement the reference count for any set that is deregistered. For any set that the reference count value is zero, the shared resources can be cleaned. If the reference count value for a set is not zero, it means that another application is writing into this set. In our example, state 7 shows that app 2 is no longer running and app 1 is still writing into set M. Index I contains a metric set M with one reference, but metric set N does not exist anymore because its reference count dropped to zero and was cleared from the index.

Starting from state 7, we cover two crash scenarios. First, we assume that app 1 crashes in the middle of its run. This crash causes a transition to state 8. At this state, none of the applications are running. Index I still exists with the metric set M with one reference and the sampler is still reading from this index. To handle this type of crash we use a timeout concept. After a period of inactivity since the last update in the metric set by the application, the sampler may assume that

the set is no longer in use. At this step, the sampler notifies the index to remove the stale set. We end up in state 9 where the sampler is running, and the index is empty.

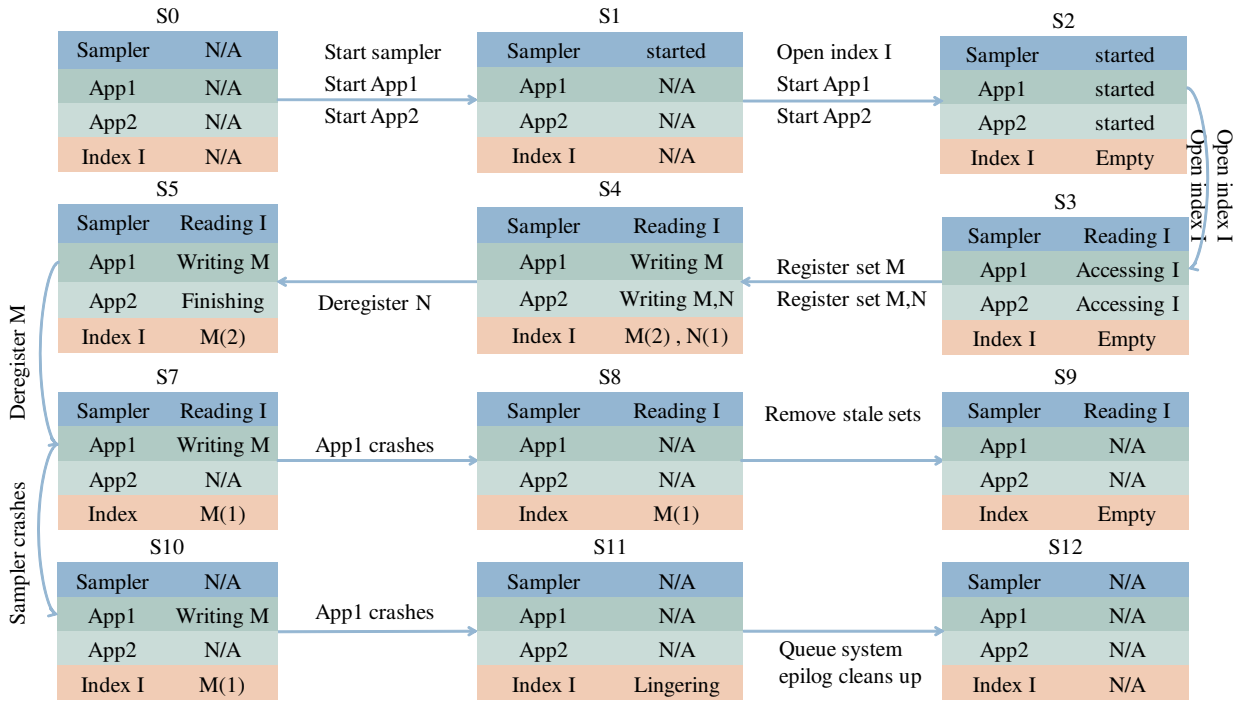


Figure 4.4: Index management. The second column of each table represents the state for the sampler, applications and the index. Transitions between states are displayed using arrows, which are annotated with the associated action.

Another possible crash scenario, starting from state 7, happens when the sampler quits unexpectedly in the middle of the execution. The unexpected quit results in a transition to state 10, where index I exists with the metric set M and app 1 is running and writing into set M. If app 1 finishes its execution safely, before terminating, the application profiler cleans the shared resources. If the last writer upon its deregistration finds out that there are no active readers, it can unlink the index file. The profiler figures this out using either read timeout or reference counts. However, if app 1 crashes, we will end up in state 11 where no program is running and index I still exists without any owner. At this step, the queue system epilog cleans up the idle index file between jobs.

Utilizing reference counts, and read and write timeout mechanisms enables us to manage the allocated resources in different use cases including failures in any of the involved processes.

4.7 Experimental Evaluation

In this section, we demonstrate the impact of our method using an open source HPC application, Nalu. We start with a brief introduction of Nalu. Next, we present a case study of using our tool-set for performance monitoring and analysis of Nalu. Finally, we evaluate the overhead impact of using our tool-set when Nalu is running.

4.7.1 Nalu

Nalu is a massively parallel computational fluid dynamic (CFD) application built on top of the Sierra Toolkit and the Trilinos solver Tpetra stack. Nalu has been chosen as a representative simulation code and performance benchmark for the Trinity Capability Improvement Metric [5]. It is a representative of implicit codes that have been developed under Advanced Simulation and Computing (ASC) program [6].

The results presented in Sections 4.7.2.1 and 4.7.2.2 are extracted from the *waleElem* model, which is available in the Nalu repository. We run this model using eight MPI processes on a KNL system as specified in Table 4.1. In Section 4.7.2.3, we use the *milestoneRun* model to run Nalu with thirty MPI processes.

Table 4.1: System specifications

System	Processor Model Clock	Cores (Physical) Threads	Memory	OS	Compiler
KNL	Intel XEON Phi 7250 @ 1.4GHZ	68 272	16 GB MCDRAM 96 GB DDR4	CentOS 7.3	Intel 17.0.1
Xeon	Intel Xeon E5-2683 @ 2.10GHz	32 32	128 GB DDR4	CentOS 7.1	Intel 17.0.1

4.7.2 Case Study

In this section, we present the case study we use to demonstrate one intended use of our tool-set. First, we explain how our tool-set can provide insights to assist with the process of revealing application phases. Next, we demonstrate how software level performance data provided by our tool-set contributes to a better understanding of the application behavior by correlating to hardware level metrics. Third, we show how our tool-set can help detect issues in the application execution in a timely manner. Finally, we demonstrate how our tool-set enables HPC users to monitor multiple scientific applications.

4.7.2.1 Application Phases

Scientific applications typically execute in several phases. They usually start with an initialization phase and end with a final phase. In parallel scientific applications the intermediate computation phase typically consists of nested iterations. The computation phase usually dominates the application efficiencies and load balance. In this section, we show our tool-set provides information that can be used as one data source in an automatic phase detection method.

Figure 4.5 exhibits the rate of changes in the number of calls to *MPI_Issend* function during the execution of Nalu on the waleElem model. Nalu starts with an initialization phase, which involves

initial communications, initial value assignments to variables, and decomposition. We do not see any repetitive patterns during the initialization phase (which lasts for about six minutes). After finishing this initialization, the computation phase begins.

In Figure 4.5, regions surrounded by dashed bubbles demonstrate a periodic behavior in this application. We attribute this repetitive pattern to application iterations executed during the computation phase. To validate this, we add information about the application iterations to the log file produced by Nalu application. Figure 4.6 displays this information extracted from Nalu log file.

In both figures, we observe six minutes of initialization. Next, the computation phase begins, which consists of five time-steps as configured in Nalu input file. At each time-step, Nalu runs three nonlinear iterations. During each iteration, the application solves three equation systems. In Nalu application's structure, these equations are identified as MomentumEQS, ContinuityEQS, and MixtureFractionEQS. According to Nalu's manual, these systems are used by the WALE model to capture the asymptotic behavior for flows. By aligning Figures 4.5 and 4.6, we can derive application phases with the granularity of equation system iterations from the MPI data collected by our tool-set.

Exposing performance data in a streaming manner by our tool-set enables us to monitor the level of progress in the application as it is running. We do not need knowledge of the log file location or its contents to observe detailed application behavior and compare it to previous runs of the application. To facilitate the phase detection, our tool-set can be used as one data provider in combination with other data sources in an automatic application phase detection approach. This is an essential feature, particularly for system support staff lacking specific application knowledge. Our tool-set provides this information using collected MPI data with a negligible interference with computation. In Section 4.7.3, we study the overhead of using our approach.

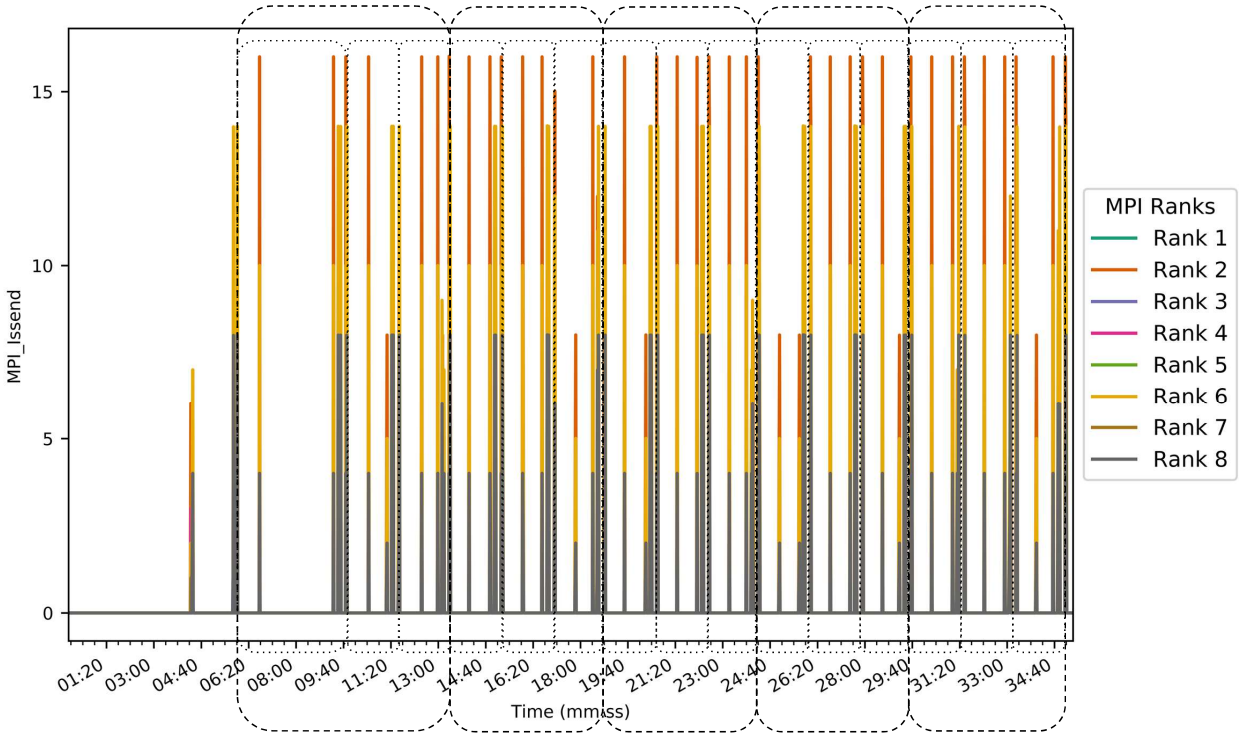


Figure 4.5: Nalu Phases.

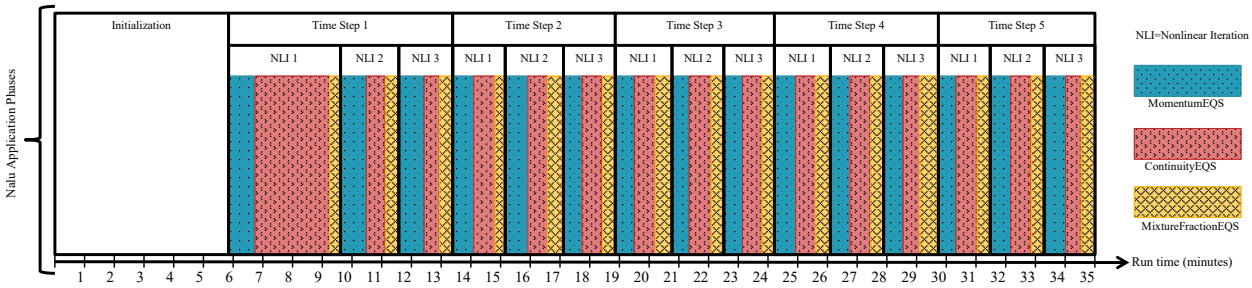


Figure 4.6: Nalu Phases extracted from log files.

4.7.2.2 *Correlations of application-level events with other performance data representing system events*

Integrating our software performance counter collection approach with LDMS allows us to study how events happening at the hardware-level are related to the application-level events. The LDMS framework provides several sampler plug-ins for the data collection from hardware performance counters. We can run several sampler plug-ins at the same time as our sampler. In this section, we use our tool-set to demonstrate a cursory exploration of the correlation between hardware and software metric time series.

In Figure 4.7, six graphs are plotted together. All graphs share the same x-axis that represents the run time in minutes. The first graph from the top with ”MPI_Issend” label on the y-axis displays the rate of calls per second to *MPI_Issend* function, which represents the application activities using software level metrics. The second graph with ”procnet_tx_bytes” label on the y-axis shows the byte rate of the data transmission by the network interface. The third graph with the y-axis labeled ”meminfo_Dirty” represents the total amount of memory waiting to be written back to the disk. The fourth graph with ”nfs_numcalls” label on the y-axis displays the rate of total RPC calls per second to NFS. The 5th graph with ”nfs_read” label on the y-axis shows the rate of NFS read calls per second. The last graph with the y-axis labeled ”nfs_write” shows the rate of NFS write calls per second. These metrics represent the I/O activities of the application running on the system.

As we can see from the NFS data and based on the discussion in Section 4.7.2.1, the application starts with reading from input and configuration files. After this input activity, the initialization phase begins, which takes around six minutes. The initialization phase mostly involves mesh distribution and communication setup. Most of the calls to functions like *MPI_Reduce* and *MPI_-Scatter* happen during this phase. After the initialization, the computation phase starts, which consists of five time-steps.

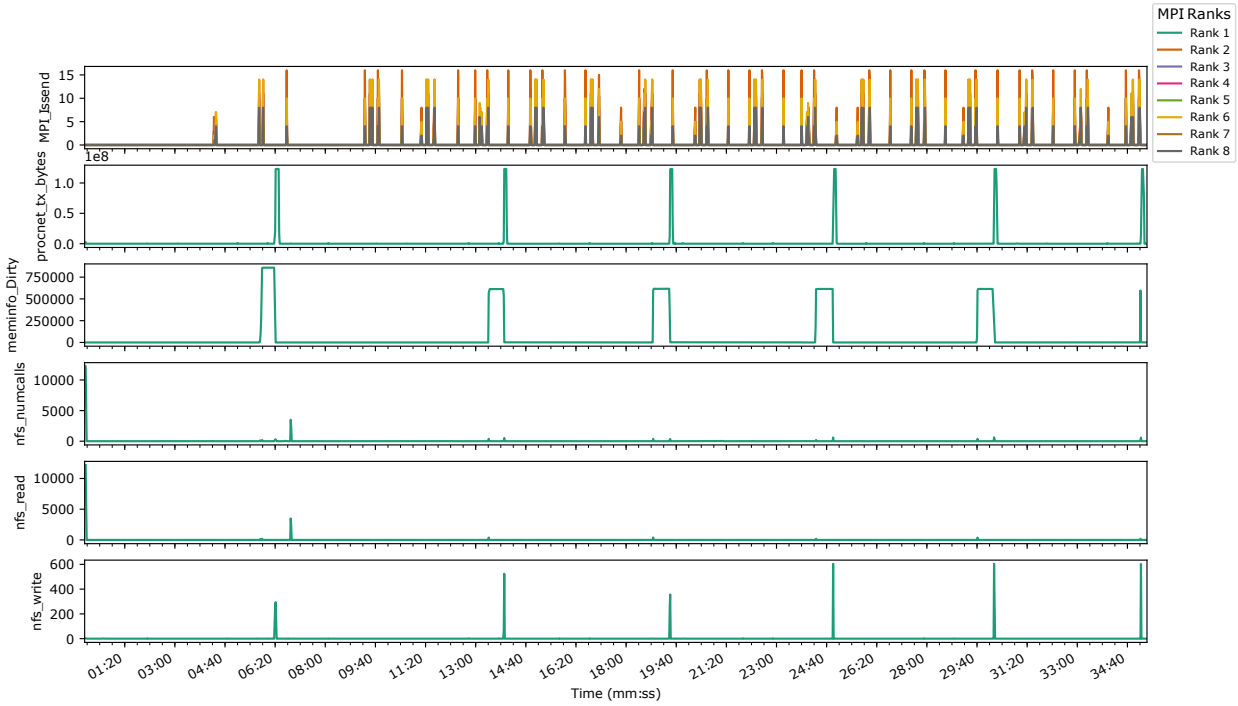


Figure 4.7: Correlations between hardware and software data

The first time-step in Nalu application begins with a sudden increase in the number of *read* calls to NFS. Here we can see how the application is interacting with the system using I/O operations, while it exhibits no signal of the MPI event. NFS reading is followed by a peak in *Dirty* metric of memory. This buffering in the NFS software stack is completed with a sudden increase in writing. This write event involves flushing the data, which is accumulated in memory and indicated by *Dirty*, to the disk. The next event is a sudden increase in the amount of data transmitted by the network interface, which is followed by a considerable number of calls to *MPI_Issend*. All of these events happened from the beginning of the time-step to the end of the MomentumEQS in the first nonlinear iteration.

From this point, we can see repeated computation periods interrupted by chunks of file output. The same pattern with a similar sequence of events repeats. Due to the 1 Hz relatively infrequent

sampling rate, the profile peaks vary somewhat from one time-step to the next.

Using our tool-set, we can monitor the behavior of applications. Integrating with LDMS enables us to correlate application level events with metrics that indicate system utilization. Streaming this performance data enables us to track application progress and detect unexpected behaviors as soon as they happen during the application run-time.

4.7.2.3 Anomaly detection

One of the challenges in HPC system management is application performance variations that surprise the user. Resource contention in the presence of other processes is an anomaly example that can cause performance variations and potentially lost computing cycles [15]. The quick detection of performance variations is critical to mitigate the issues and improve resource utilization in a timely manner. Typical profiling tools fail to help detect such issues as they occur, because only post-run analysis methods are available. In this section, we show how our tool-set can help detect an abnormal behavior in Nalu.

We design an experiment to evaluate the capability of our tool-set to support the anomaly detection. In this experiment, we run Nalu in an environment where another application might be run simultaneously. We run this experiment on the Xeon system as specified in Table 4.1. Nalu application takes the *milestoneRun* problem as the input and utilizes thirty MPI processes to run. We use our tool-set to investigate how the behavior of Nalu application changes in the presence of other processes.

In Figure 4.8, two graphs are plotted together. Both graphs share the same x-axis that represents the run time in minutes. The y-axis on both graphs displays the rate of calls per 0.1 second to *MPI_Send* function. The first graph on the top represents the data collected from the normal run of

Nalu application. The other graph on the bottom shows the data collected from the abnormal run where there is contention for system resources in the presence of other processes.

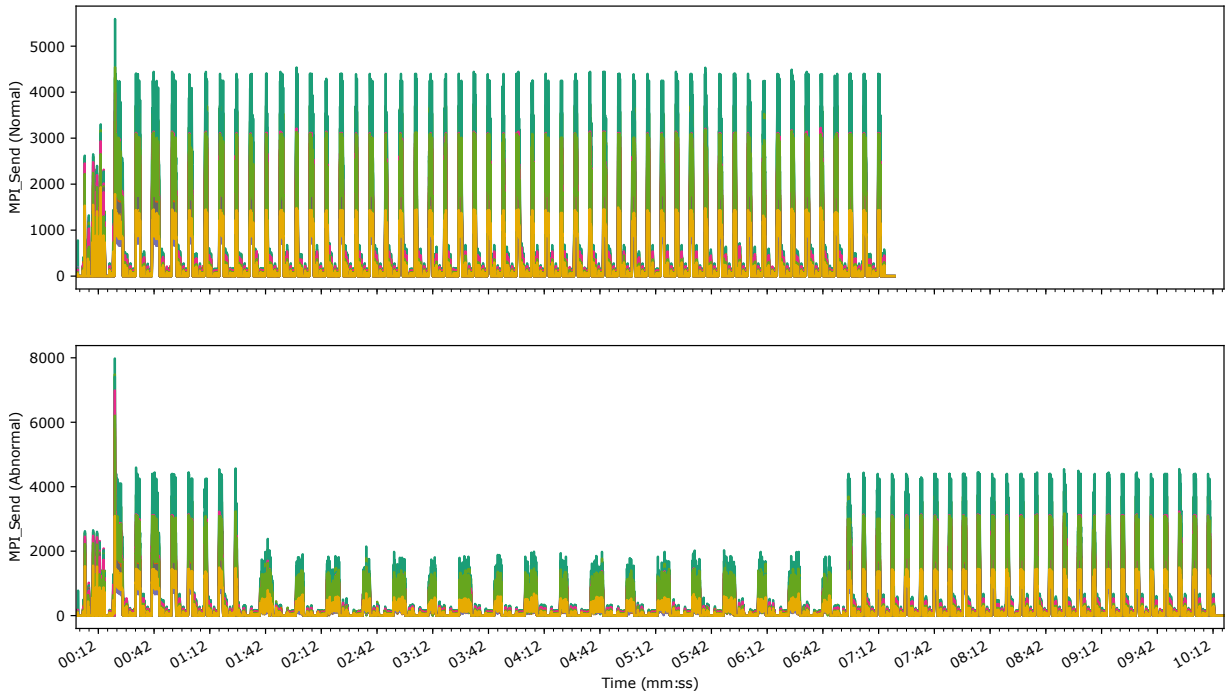


Figure 4.8: Revealing abnormal behavior

At the beginning of this experiment, we have Nalu and LDMS daemons as the only processes running on the system. After about 90 seconds from the beginning of the execution of Nalu, other processes start on the system. The presence of other processes introduces a contention on system's resources. We can see this impact on the bottom graph in Figure 4.8 immediately after it occurs. The resource contentions cause a performance degradation. We can see that the duration of each application major iteration has increased by a factor of two. At each phase, the rate of calls to *MPI_Send* function has decreased compared to the normal execution. The termination of the competing processes after five minutes restores normal operation. We can see that the rate of calls to *MPI_Send* function in the abnormal run has decreased by half compared to the regular execution.

Our tool-set can help detect an abnormal behavior in an application as soon as it occurs. The presented results demonstrate the benefits of our tool-set over the tracing tools in the anomaly detection. By streaming application-level events we can decrease the latency in the application analysis and issue mitigation. As in this case, determining the *normal* behavior for a set of metrics is application-specific. Our tool-set can be coupled with automatic anomaly detection methods such as statistical data analysis approaches to automate the process.

4.7.2.4 Monitoring multiple applications

Our approach is designed to enable monitoring of multiple applications. The tool-set interacts with application processes and these processes can belong to different applications. In this section, we demonstrate how our tool-set can be used for streaming performance data collected from multiple applications running at the same time.

We design an experiment to evaluate the capability of our tool-set to monitor four running applications at the same time. In this experiment, we run one instance of Nalu and three instances of an MPI based scientific mini-application, MiniMD. MiniMD is a Molecular Dynamics (MD) mini-application in the Mantevo mini-application project [72]. It represents the computation and communications of the Lennard-Jones method used in the LAMMPS application [104]. We run this experiment on the Xeon system as specified in Table 4.1.

In our experiment, Nalu application takes the *milestoneRun* problem as the input and utilizes sixteen MPI processes to run. Each instance of the miniMD application is run using the default test with different problem sizes and number of MPI processes. The smallest instance is run using two MPI processes and 400000 atoms. The next instance utilizes four MPI processes and has 13500000 atoms in the input configurations. The largest instance of the miniMD uses eight MPI processes to work on 32000000 atoms that specified in the input configurations.

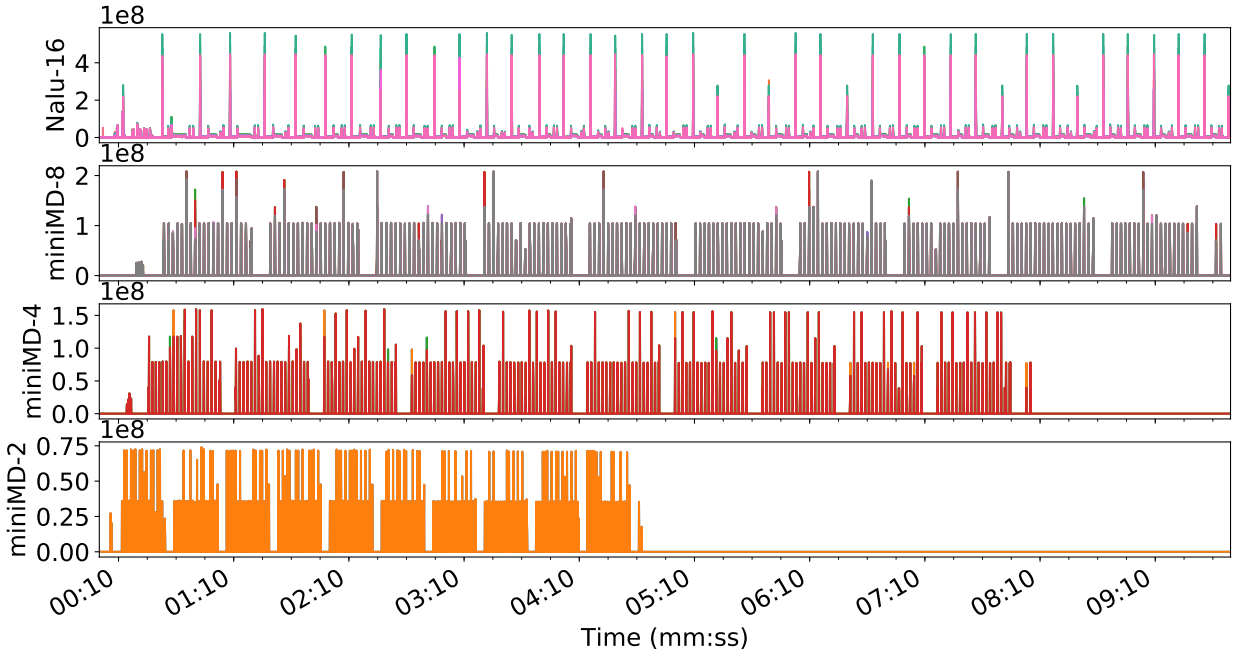


Figure 4.9: Monitoring multiple applications. Y axis represents the rate of messages sent using the MPI_Send function.

In Figure 4.9, four graphs are plotted together. All graphs share the same x-axis that represents the run time in minutes. The y-axis on all plots displays the rate of changes in the volume of messages that are communicated using the *MPI_Send* function. The first graph from the top with "Nalu-16" label on the y-axis represents the rate of messages for the Nalu application that is running using 16 processes. The second graph with "miniMD-8" label on the y-axis shows the message rate for the miniMD application when is run with 8 processes. The third graph with the y-axis labeled "miniMD-4" displays the rate of messages for the miniMD application when is run with 4 processes. The last graph with the y-axis labeled "miniMD-2" shows the message rate for another instance of the miniMD application that is run with two MPI processes.

As we can see in Figure 4.9, all applications start at the same time. Although the repetitive pattern can be seen in all of them, each application instance exhibits a different behavior depending on the

problem and configuration. The miniMD-2 instance executes each iteration faster and send fewer messages compared to others, e.g., less than 50% of the miniMD-8 messages. This application finishes its execution after around 5 minutes. The miniMD-4 instance, which has longer iterations, finishes 3 minutes later. The long iteration and high number of messages of the miniMD-8 instance indicates that it is solving a larger problem compared to other instances of miniMD.

Another notable point is the difference between the behavior of applications in Figures 4.8 and 4.9. In Section 4.7.2.3, an anomaly is injected to interfere with the computations of the main application and limit its available resources. Figure 4.8 shows how this interference leads to the abnormal behavior and the run-time slow-down. In our experiment in this section, we ensure the availability of computing cores to all applications. 50% of available cores have been used by the Nalu application. Three miniMD instances use 14 of the remaining 16 CPU cores. As a result, we do not observe a significant slow-down in this experiment.

Our tool-set enables HPC users to monitor multiple applications. This is achieved by our approach's design that utilizes a specific channel for data collection from each application.

4.7.3 Overhead Evaluation

Our approach is intended for the deployment in a production environment on large-scale HPC systems. As a result, performance and scalability of the implementation become critical requirements. We investigate the overhead of using our tool-set to validate that our design meets the requirements. We design a set of experiments to investigate the overhead in different configurations. In our analysis, we perform the following impact assessments:

- Impact of using our tool-set compared to the base case with no monitoring.
- Impact of changing sampling frequency and data resolution.

- Impact of the presence of other LDMS samplers.
- Impact of using different strategies in process placements.

We run the experiments in the environments specified by Table 4.1. To avoid hardware variability within result sets that may obscure small overhead costs, we run the related experiments in series on the same compute node.

We use two types of systems to run the experiments to understand the interaction of our tool-set with the operating system and applications on different HPC platforms. The first system includes compute nodes that are equipped with Intel Xeon Phi 7250 1.4GHZ processors (68 cores), 16 GB high-speed cache, and 96 GB DDR4 memory. This architecture is used on some HPC platforms (e.g., [35]) and may still be a representative of future architectures. CentOS 7.3 is running as the operating system. We build Nalu and its dependencies using Intel Compiler 17.0.1 enabled with the level three optimization. We refer to this system as KNL. The next system, which we refer to as Xeon, includes compute nodes that are equipped with Intel Xeon processors, one of the mainstream processors utilized in HPC systems. On this system, 32 Intel Xeon E5-2683 processors are running at 2.10 GH with 128 GB memory. CentOS 7.1 is the operating system, and we build Nalu using Intel Compiler 17.0.1.

We run our tool-set alongside Nalu application to measure the overhead based on the *wall time* reported by Nalu's output log file. Nalu performs the simulation utilizing a computational mesh. In our experiments, we use an R2 mesh with 2725802 elements and 77 MB file size. We choose a Nalu problem that is called *milestoneRun* from Nalu's regression test suite [39]. This problem has been used for Trinity acceptance tests [5]. We can use this problem to run Nalu with an arbitrary number of MPI processes.

We use t-test to verify the statistical significance of these results [74]. For each set of experiments,

we compare two groups of results. One is the control group which is defined according to the experiment and the type of impact assessment. The second group includes the results that are generated by changing the parameter related to the impact assessment experiment. For example, consider the experiment that we evaluate the impact of the process placement. In this experiment, we choose placement 1 as the baseline and the group of placement 1 results is the control group. The changing parameter in this experiment is the type of process placement. Any experiment with a new process placement generates a new set of results. Every set of results form the second group for t-test to compare with the control group. This test demonstrates statistically significant results of our experiments with the p-value less than the chosen threshold of 0.01.

4.7.3.1 Process Placement Strategies

We used several process placement strategies to measure Nalu's performance in different situations. Table 4.2 presents these strategies. In the first three placements, all MPI processes and LDMS daemons are unpinned. In the first placement, we do not reserve any processor for LDMS daemons, and the application utilizes all available processors, i.e., 272 on KNL and 32 on Xeon. We keep one and two processors idle in placements 2 and 3, respectively, and the MPI application uses the rest of the processors. In placements 4-6, we pin all MPI processes and LDMS daemons. The MPI application utilizes all processors in placement 4, and both LDMS sampler and aggregator are pinned to the first processor, which results in it being used by three processes. In placement 5, we do not pin any MPI process to the first processor and use it for both sampler and aggregator. In placement 6, we dedicate the first and second processors to the sampler and aggregator and pin 270 MPI ranks to the next 270 processors on KNL or 30 MPI ranks to the next 30 processors on Xeon. In placements 7-9, we apply strategies similar to placements 4-6 with a difference that the last two, instead of the first two, processors are used for LDMS sampler and aggregator.

Table 4.2: Process placements used in experiments. Numbers on the second column represent the number of MPI ranks we use to run Nalu. The third column states whether the processes were pinned or not. The fourth and fifth columns show core number that we use to run the sampler and aggregator daemons with pinning.

Placement Version	# MPI Ranks KNL — Xeon	Pinned/ Unpinned	LDMS Sampler Daemon Processor Place	LDMS Aggregator Daemon Processor Place
1	272 — 32	Unpinned	Free	Free
2	271 — 31	Unpinned	Free	Free
3	270 — 30	Unpinned	Free	Free
4	272 — 32	Pinned	First	First
5	271 — 31	Pinned	First	First
6	270 — 30	Pinned	First	Second
7	272 — N/A	Pinned	Last	Last
8	271 — N/A	Pinned	Last	Last
9	270 — N/A	Pinned	Last	Second to last

Previous generations of Intel Xeon Phi processors were known for reserving the first core for the operating system [84]. In our experiments, we take two approaches based on the reserving first or last cores for OS activities and LDMS daemons (if available) to investigate this impact. Since this is not the case for Xeon processors, we do not run experiments in placements 7-9 for Xeon.

Figures 4.10 and 4.11 exhibit Nalu's performance regarding run-time using different process placement strategies on both KNL (Figure 4.10) and Xeon (Figure 4.11). The vertical axis on both plots shows run-time in milliseconds and the horizontal axis presents different placement strategies as described in Table 4.2. The rotated plot outside of the box shows the probability density of the data at different values [73]. In this figure, we exclude all data from the cases where we run LDMS.

In KNL experiments, Nalu demonstrates the best run-time when using the placement 6, where we keep the first two processors idle and use the rest of processors (270) as MPI ranks to run the application. Since we did not run LDMS in these experiments, the first two processors are free to be used by the system processes. The availability of these processors reduces the OS noise and

its impact on the application performance and run-time variability. OS activity has been known as one of the primary causes of the variability [12] and some methods exist to address this issue on Intel Xeon Phi processors [25].

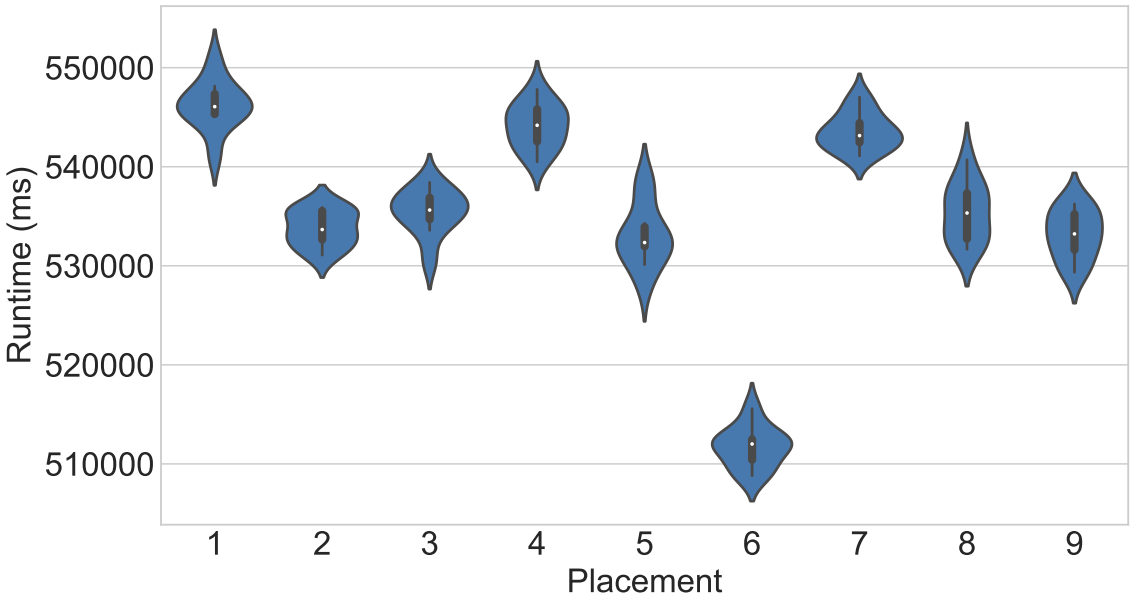


Figure 4.10: Run-time of Nalu application with different process placement strategies when running on the KNL cluster without any LDMS daemons running. The run-time plot for each placement features a kernel density estimation of the underlying distribution of run-time derived from 10 samples. Table 4.2 provides a brief overview of different placements.

We observe the highest run-time in placements 1, 4, and 7. The overhead introduced by using these strategies are in the range of 6.5% on average. In all of these placements, we use the maximum available processors to run Nalu application, and system noise and necessary OS activities do not have any choices other than interrupting tasks being carried out by the MPI application. Choosing any of placements 2, 3, 5, 8, and 9 roughly adds 4.5% overhead on average to the application.

In Xeon experiments, Nalu exhibits the best run-time when we use all 32 available processors as MPI ranks. In general, using more processors on the Xeon machine allows Nalu to leverage the

maximum parallelism. Pinning MPI processes does not seem to provide any benefits on Xeon. Placements 3 and 6 that keep two processors idle show the highest run-time with an overhead of 4.7% on average. Choosing any of placements 2, and 5 roughly adds 3.2% overhead on average to the application's run-time. Placement 4 performs slightly slower by adding 0.2% overhead to the run-time of the placement 1 that differs only in pinning the MPI processes.

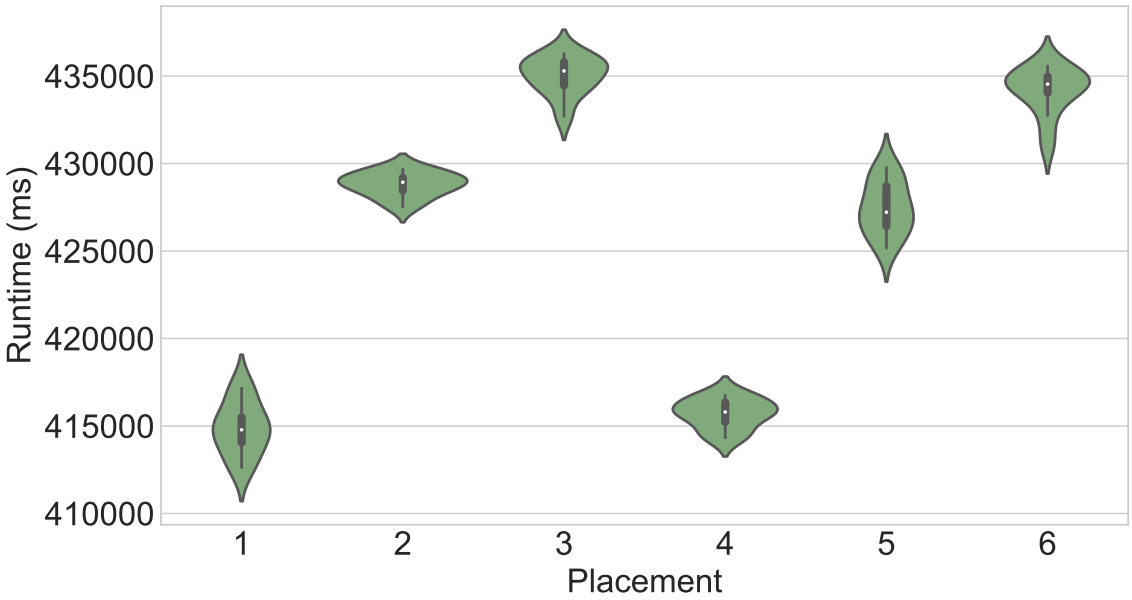


Figure 4.11: Run-time of Nalu application with different process placement strategies when running on the Xeon cluster without any LDMS daemons running. The run-time plot for each placement features a kernel density estimation of the underlying distribution of run-time derived from 10 samples. Table 4.2 provides a brief overview of different placements.

Given the best results we get from the placement 6 on KNL, and placement 1 on Xeon, we use these placements for the impact analysis of using our tool-set for monitoring Nalu.

4.7.3.2 *Sampling frequency impact*

In this section, we study how using our tool-set affects the performance of Nalu. We run experiments in two cases: regular Nalu run with no LDMS daemons running and a Nalu run where LDMS MPI sampler is running at the same time and data is collected from the application. We run the LDMS MPI sampler with four different frequencies: 0.1 HZ, 1 HZ, 2 HZ, and 10 HZ.

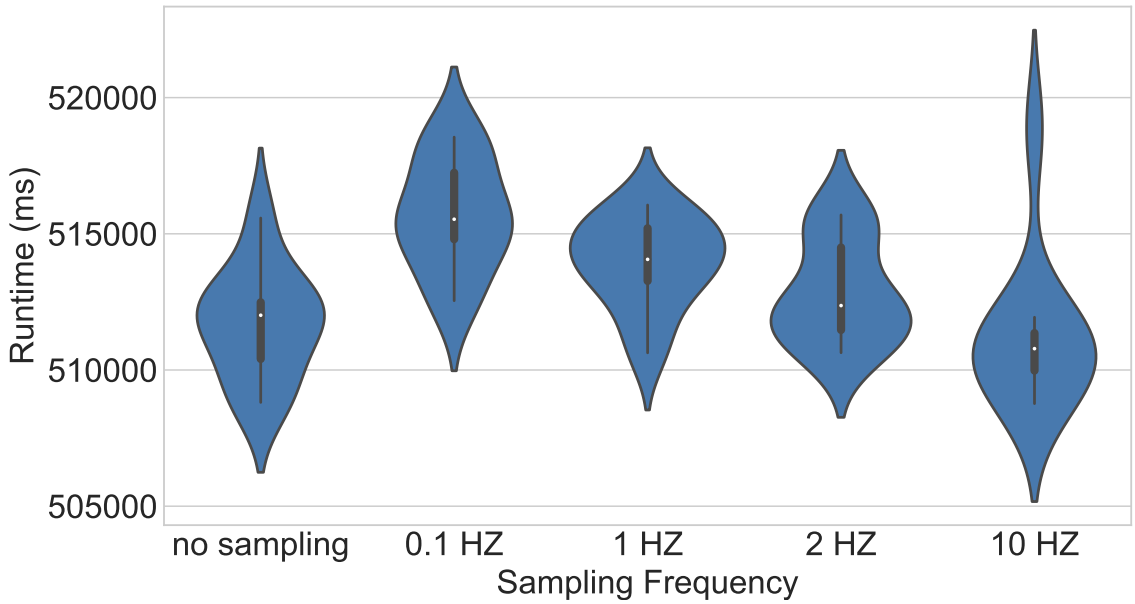


Figure 4.12: Run-time of Nalu application on the KNL cluster using placement 6 in the cases of running without monitoring and when LDMS daemons are running on the system with different MPI sampling frequencies. The run-time plot for each sampling case features a kernel density estimation of the underlying distribution of run-time derived from 10 samples.

Figures 4.12 and 4.13 depict Nalu's performance regarding run-time on both KNL (Figure 4.12) and Xeon (Figure 4.13) when LDMS daemons are not running and the cases where the application is monitored with four different sampling frequencies. The vertical axis on both plots shows run-time in milliseconds and the horizontal axis displays different cases based on the sampling ap-

proach. For KNL, we run these experiments using the placement 6 where the first two processors are used by LDMS daemons and the MPI application utilizes the last 270 processors. We use the placement 1 for running experiments on Xeon where we use all 32 available processors and do not pin any processes.

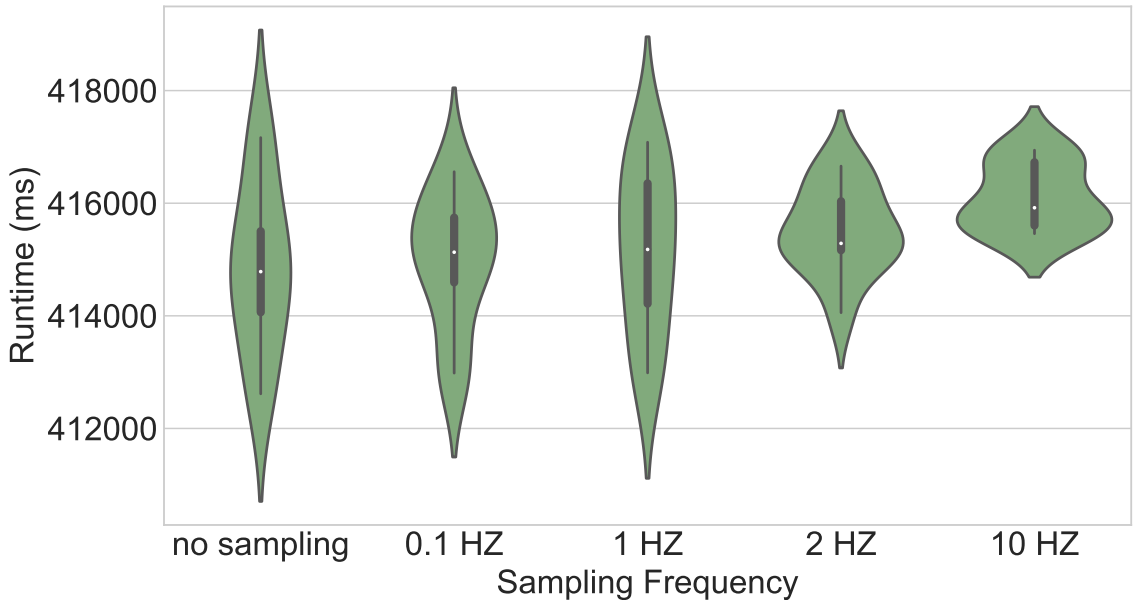


Figure 4.13: Run-time of Nalu application on the Xeon cluster using placement 1 in the cases of running without monitoring and when LDMS daemons are running on the system with different MPI sampling frequencies.The run-time plot for each sampling case features a kernel density estimation of the underlying distribution of run-time derived from 10 samples.

In KNL experiments, the overhead introduced by using our tool-set on average is within the range of 0.78% in different sampling frequencies. Running the sampler with the high frequency of 10 HZ, on average, yields a surprisingly acceptable run-time. However, the run-time variability is high in this case. Running the sampler at 0.1 HZ does not gain performance and, in some cases, it slightly slows down the application more than higher frequencies. Furthermore, we have lower data accuracy in this case.

Running the sampler at 1 HZ and 2 HZ leads to both reasonable performance and data accuracy. The run-time overhead introduced by both is less than 0.5%. We can see some differences in the data distribution between these two frequencies. The 2 HZ case shows another small peak in higher run-time values in addition to the main peak that we have in all cases. On the other hand, the 1 HZ case demonstrates a distribution closer to normal with one peak.

In Xeon experiments, using our tool-set with the high frequency of 10 HZ and 2 HZ respectively adds 0.3% and 0.16% overhead on average to the run-time. The run-time overhead of running the sampler at 1 HZ and 0.1 HZ are both less than 0.1%. Other than the lower overhead, the higher accuracy in data collected by the sampler at 1 HZ and its distribution distinguish this case from other cases.

This outcome suggests that we choose to run the samplers at 1 HZ for our next study to get a fair result.

4.7.3.3 Impact of the presence of other samplers

Collecting data from software and different hardware components at the same time helps gain a comprehensive understanding of the application behavior.

In this section, we study the impact of running the MPI sampler alongside other samplers on the run-time of Nalu application.

We run the experiments with four different versions of sampling for this study. Table 4.3 presents the information about each version. In version 1, we run all of the samplers including the MPI sampler. In version 2, we only run the MPI sampler. All samplers but the MPI sampler are evaluated in the version 3. We also have a regular application run in version 4, where no sampler is running. Other samplers provide statistics related to the virtual memory, RAM and CPU utilization,

networking, and file systems. The last column of this table indicates the number of metrics that are collected and exposed by samplers at each second during the execution.

Table 4.3: Sampling versions.

Sampling Version	Running Samplers	# Collected Metrics per Second KNL — Xeon
1	MPI Sampler, vmstat, meminfo, procnetdev, procnfs, procstat	5912 — 903
2	MPI Sampler	2700 — 320
3	vmstat, meminfo, procnetdev, procnfs, procstat	3212 — 583
4	None	0 — 0

Based on the findings from previous sections, for KNL experiments, we use placement 6, where the first two processors are reserved to be used by LDMS daemons if present. For Xeon experiments, we use placement 1, where we use all available processors with no pinning involved. When we run LDMS samplers, i.e., versions 1-3, we set the sampling frequency to 1 HZ.

Figures 4.14 and 4.15 display Nalu's performance regarding run-time on both KNL (Figure 4.14) and Xeon (Figure 4.15) when LDMS daemons are not running and the cases where the application is monitored with many LDMS samplers. The vertical axis on both plots shows run-time in milliseconds and the horizontal axis represents different sampling versions as described in Table 4.3.

In KNL experiments, the overhead introduced by the samplers in different cases is within the range of 0.5% of the base case, i.e., no sampling. We observe that running all samplers, where we roughly collect 6000 metrics every second, causes the run-time to be more scattered across the data range. The non-MPI samplers collect more than 3200 metrics from different data sources every second. Most of these hardware level data are provided in /proc file system. However, MPI sampler uses the shared memory index to expose the 2700 metrics collected by the MPI profiler. This seems to impact the application's run-time more than other hardware-level samplers. The overhead is still

within the range of 0.5% of the original application run. A group of HPC users at SNL provided bounds on acceptable overhead to be less than 1% slowdown [3].

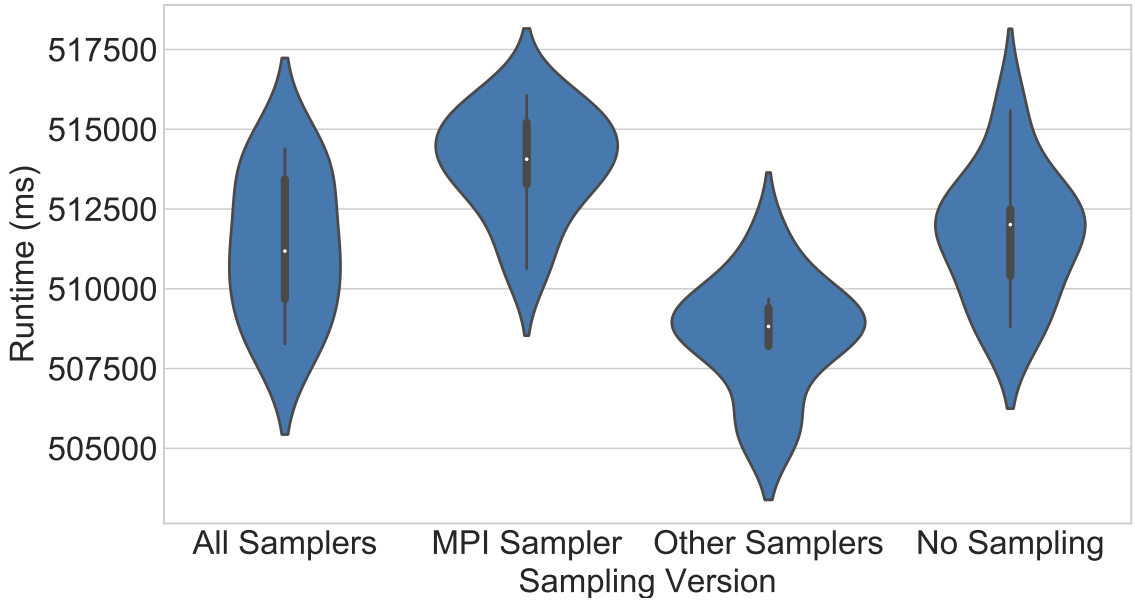


Figure 4.14: Run-time of Nalu application on the KNL cluster in the cases of running without monitoring and with LDMS samplers running on the system. The run-time plot for each sampling case features a kernel density estimation of the underlying distribution of run-time derived from 10 samples. Table 4.3 provides a brief overview of different sampling versions. All of the samplers collect data at 1 HZ.

In Xeon experiments, the overhead introduced by the samplers in different cases, on average, is within the range of 0.3% of the base case, i.e., no sampling. When LDMS collects 900 metrics per second using all samplers, we observe the highest overhead of 0.29%. Using LDMS MPI sampler alone increases the run-time by less than 0.1%.

Impact analysis results show that the overhead introduced by our tool-set is within the acceptable overhead range. This makes our tool-set a suitable choice to deploy in a production environment for continuous monitoring.

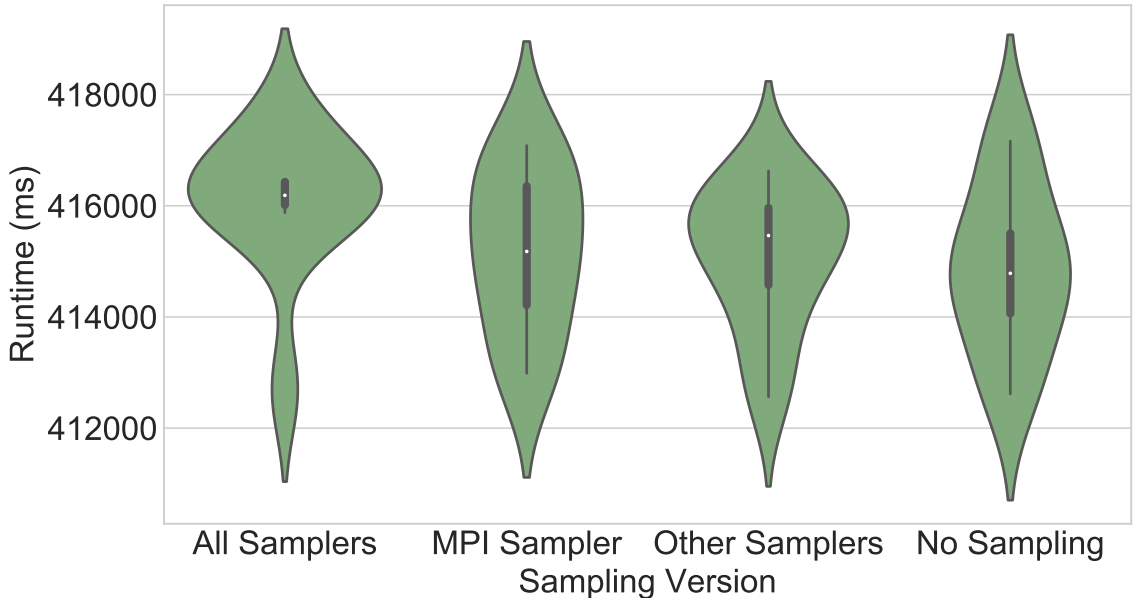


Figure 4.15: Run-time of Nalu application on the Xeon cluster in the cases of running without monitoring and with LDMS samplers running on the system.The run-time plot for each sampling case features a kernel density estimation of the underlying distribution of run-time derived from 10 samples. Table 4.3 provides a brief overview of different sampling versions. All of the samplers collect data at 1 HZ.

Experimental evaluation results demonstrate that our low overhead tool-set helps understand the behavior of applications by streaming software level performance data. Our approach enables run-time operational analysis by exploring the correlations between hardware and software level data.

4.8 Chapter Summary

In this chapter, we developed and demonstrated a hybrid approach to HPC application monitoring that supports performance analysis in production conditions. This approach takes advantage of

the low overhead of shared memory and LDMS to provide insights into the application's behavior using profiling during the execution. By taking this approach, we avoid the overhead of heavy tracing methods. We have implemented a tool-set to evaluate its impact. While we demonstrate our approach using MPI applications, it does not require a specific programming paradigm and the design is generally applicable. We integrated our tool-set with the LDMS framework to create a scalable hybrid tool that streams application performance data using profiling. By taking advantage of the shared memory approach, components within our tool-set can communicate efficiently. We have resolved challenges of using this method and provided mechanisms for fault management.

We have evaluated our approach using test cases of the open-source HPC code, Nalu. We have shown how our tool-set provides insights that can be used in combination with other data sources to reveal application phases during the execution without access to application source code. By integrating with LDMS, we have run experiments that stream several types of data using hardware samplers in addition to the software level metrics collected by our tool-set during the execution. We provided examples that show how this tool-set helps explore the correlations between different events happening during the execution. Also, we demonstrated how our tool-set helps us to detect abnormal behavior in an application during the execution.

Overhead results show a slight increase in run-time, which is within the acceptable range. This feature makes our approach a suitable choice for continuous performance monitoring in a production environment. Integrating this approach with LDMS enables streaming performance data to storage efficiently and supports run-time analysis and feedback to the application.

CHAPTER 5: PERSISTENT TRANSACTIONAL NON-BLOCKING LINKED DATA STRUCTURES

5.1 PETRA Methodology

In this section, we present our PETRA methodology to design non-blocking durable transactional data structures. Building on top of LFTT, we discuss how we add durability to transactions while avoiding additional logging overheads.

5.1.1 Overview of the Methodology

Figure 5.1 presents high-level steps to execute transactions using PETRA. Every thread starts the execution of a transaction by creating a descriptor object that includes information about the transaction status, data structure and transaction's operations.

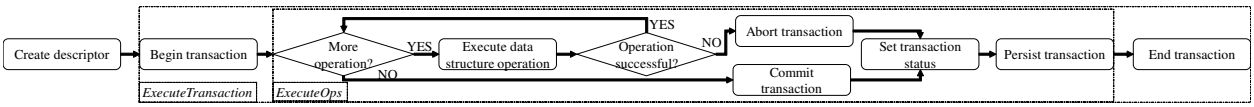


Figure 5.1: Methodology overview

A transaction begins after calling the EXECUTETRANSACTION function that performs initialization, such as preparing the helping scheme for executing the transaction's operations. Next, the EXECUTEOPS function executes the sequence of operations specified by the transaction descriptor (Section 5.1.4). The results of these executions determine whether the transaction is committed or aborted. After setting the transaction status atomically[1], we perform the required actions to

[1]Compare-And-Swap (CAS) is an atomic instruction used to perform a conditional update on a shared variable. CAS succeeds if the contents of the shared variable match an expected value.

durably commit (or abort) the transaction (Section 5.1.2). Finally, the transaction finishes with post-execution activities such as marking the removed nodes for deletion. We use the pointer marking approach to indicate logically deleted nodes [63]. If a node is bit-marked, the key associated with it is not part of the list. We also assign a logical status to each node based on the transaction status and type of operations. Further description is provided in Section 5.1.3.

Algorithm 5.1: Type Definitions

```
1  enum TxStatus
2      Active;
3      Commited;
4      Aborted;
5  enum PersStatus
6      Maybe;
7      InProgress;
8      Persisted;
9  enum OpType
10     Insert;
11     Delete;
12     Find;
13 struct Operation
14     OpType type;
15     int key;
16 struct Desc
17     int size;
18     int txid TxStauts status
         PersStatus pstatus
         Operation ops[ ];
19 struct NodeInfo
20     Desc* desc;
21     int opid;
22 struct Node
23     NodeInfo* info;
24     int key;
25     ...
```

We apply the PETRA's methodology on different linked data structures such as linked list and skiplist based sets, a multi-dimensional (md) list, and a hash map and present the evaluation results in Section 4.7. Without losing generality, we illustrate PETRA using a set abstract data type with three standard operations (INSERT, DELETE, and FIND). We list the constants and data type definitions in Algorithm 5.1. Each node of the data structure has a pointer, named $info$, to an object of type NODEINFO that keeps track of the latest executed transaction on this node. This information is provided by a reference to the transaction descriptor ($desc$) and the index of the most recent operation in the transaction that accessed the node ($opid$). Other important information in the transaction descriptor determines the status of the transaction and its durability status. The transaction status may be Active (being executed), Aborted (a data structure conflict has been detected necessitating an abort) and Committed (transaction execution is successful). The

durability status may be Persisted (transaction already persisted), InProgress (transaction being persisted), and Maybe (persistence status unclear). A transaction becomes visible to other threads when the transaction status is Committed and persistence status is Persisted.

5.1.2 Durability via Transaction Descriptors

In LFTT, transaction descriptors are used for managing consistency and ensuring progress. When considering adding durability in PETRA, we deviate from the typical PTM approach of explicit logging. A PTM may rely on undo logging to keep track of old values of memory locations that will be written by the transaction. For correct recovery, the log itself must be made durable before the data structure is written. Alternatively, a PTM may rely on redo logging to record all memory writes of a transaction that needs to be made durable at transaction commit. The logs are used after a crash to recover to a consistent state. Thus, traditional logging incurs two types of overheads: the additional instructions that manage the log and persist the log, and the additional ordering that requires the log to persist before data structure modification. Note that if crashes are infrequent, traditional logging is very expensive: each transaction is slowed down even when the log is needed only when a crash occurs.

A key point of PETRA is our observation that the transaction descriptor object contains sufficient information of all data structure operations that a transaction must execute. Hence, we can re-purpose the transaction descriptor as a redo log to support durability. As such, our redo log has high-level information of data structure operations, rather than low-level information of memory accesses. Due to this high-level information, in PETRA, persisting a transaction is achieved by persisting its transaction descriptor, but the data structure itself does not need to be persisted, i.e. flushed out of the cache. We let the memory system naturally handle the durability of the data structure and resolve any inconsistency during the recovery using the transaction descriptors. In

contrast, PTMs must persist both the log and then the data structure in that order. Note that as computation progresses, changes made by a completed transaction will become durable as cache blocks modified in the transaction will be gradually evicted from the cache. Thus, in contrast to PTMs, data structure changes are persisted *lazily*, as opposed to *eagerly* in PTMs [7].

Since only transaction descriptors are persisted, if a crash occurs, recovery needs to visit each past transaction to validate whether all operations specified in the persisted transaction descriptor have been reflected durably in the data structure. If they have, nothing else is needed. This is likely the case for most transactions because modified data blocks in the cache will get evicted over time. Otherwise, the transaction must be repeated, and here the descriptor serves as a redo log that specifies which operations need to be performed. Recovery procedure details are discussed in Section 5.1.5.

In addition to the benefits discussed above, a transaction descriptor serves the following additional purposes. First, by keeping all the necessary information to complete a transaction, descriptors enable threads to help each other when a transaction is delayed. Delays in transactions can happen for reasons such as contention on shared resources and the operating system interrupts [156]. Second, it reflects the latest status of the transaction and makes it accessible to all threads that are executing transactions on common nodes. Finally, it enables *detectable execution* [56], the ability to determine after recovery whether a specific operation was executed.

Figure 5.2 presents an example that illustrates how PETRA uses transaction descriptors for helping, detecting conflicts, and ensuring durability. In this example, the set data structure consists of keys 1 and 3, which were inserted by Thread 1 through transaction $t1$. $t1$ was committed and persisted, as indicated by its $Status$ and $PStatus$. Next, threads 2 and 3 execute their transactions $t2$ and $t3$ concurrently. Transaction $t2$ specifies two insert operations with keys 4 and 2, while $t3$ attempts to delete keys 3 and 4. Thread 3 performs its first operation and updates the $info$ pointer on node

3 with a new NODEINFO. During the execution of its second operation, a conflict is detected with $t2$, which has not finished its operations. Because an active transaction is working on node 4, $t3$ does not modify it. Instead, this transaction first helps transaction $t2$ to complete its remaining operations. This helping mechanism is possible because PETRA has the semantic knowledge of the data structure and divides the transaction into multiple steps, i.e., data structure operations. This division allows for keeping track of the transaction progress. Note that this helping mechanism can be prone to a livelock problem when circular dependencies between helper threads exist. This problem is avoided through the use of a per-thread helping stack that contains the descriptors of the transactions that are being helped and checking for duplicates [156].

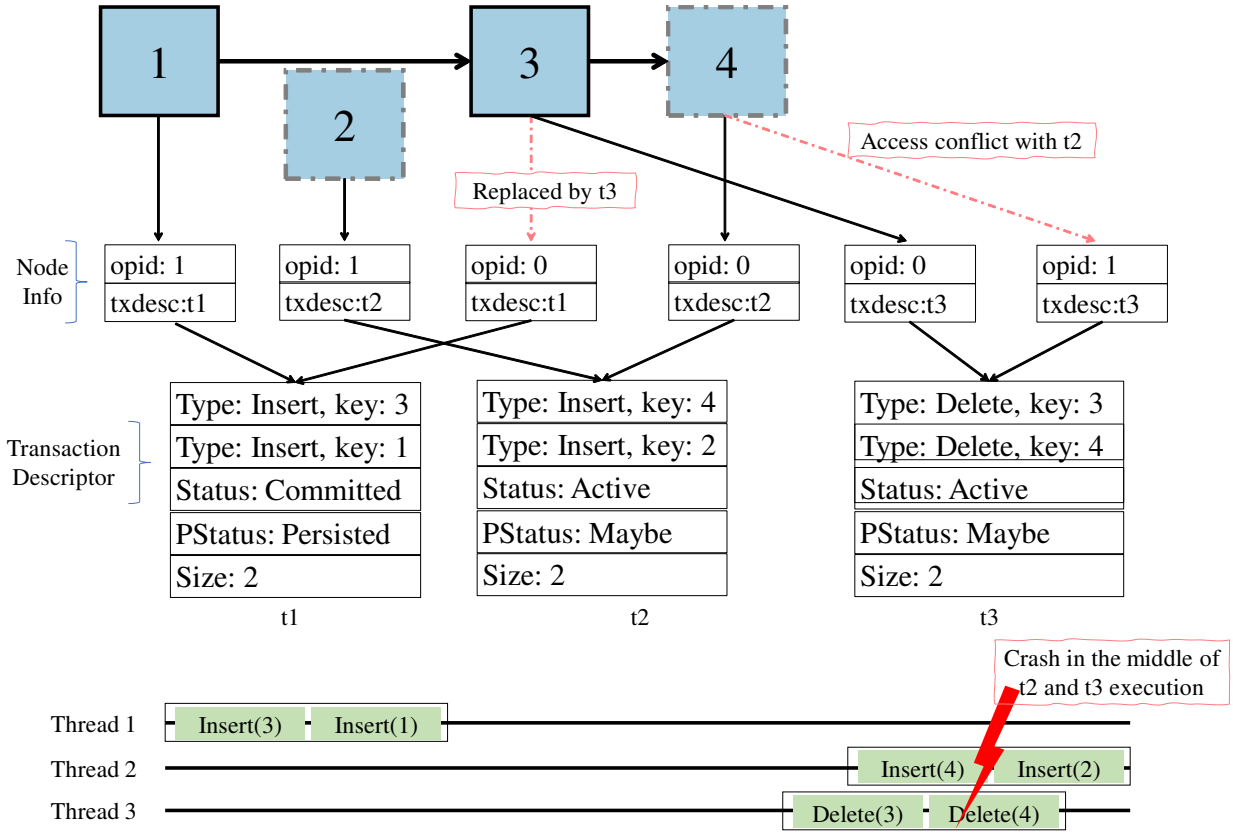


Figure 5.2: Using transaction descriptors for helping, conflict detection, and ensuring durability.

Suppose that a crash happens in the middle of the execution of $t2$ and $t3$. Neither of these transactions completed before the crash and their effects are not visible to other threads. During recovery, the members of the set must reflect only the outcome of transactions that were completed and persisted before the crash. In the example, we only accept keys 1 and 3 as the members of the set. This state of the data structure is verified by the completed and persisted transaction descriptors, i.e., $t1$ in this example. If any effect from $t2$ and $t3$ remained in the data structure, they would be canceled during the recovery, because those impacts were not visible before the crash. If any of the keys 1 and 3 does not exist in the data structure, they will be inserted using the information provided by the transaction descriptor of $t1$, which serves as a redo log.

5.1.3 Determining the Logical Status

We adapt the logical transaction management capabilities of LFTT [156] for building durable transactional data structures with ACID properties. We assign a logical status to the nodes to ensure atomicity and isolation. The status of each node is inferred based on the status of the latest transaction that accessed that node. This logical status allows us to hide the intermediate state of the shared data from concurrent transactions. Modifications are visible to other threads when the transaction is complete and can guarantee durability. Also, upon abort, a transaction can revoke the modifications made by the completed operations to guarantee atomicity. One approach to cancel the effects of the completed operations in transactional data structures is to invoke their inverse operations [68]. This method increases contention among threads in accessing the shared data structure without contributing to the overall throughput. Instead, in our logical mechanism, a transaction inverts its interpretation of the logical status of a node that was last accessed by an aborted transaction.

Algorithm 5.2 provides the details of our method to determine the node's status. On line 2, the

physical presence of a node with the specified key is verified. Determining the logical status of a key is done by the function IsKeyPresent. This function returns a boolean value that indicates the logical presence of the key in the abstract state of the data structure. This function uses the information from the last transaction that accessed the node and the descriptor object of the current transaction. We know that the state of a node is not altered if the last transaction that accessed it was a Find operation. We report this node as present in this case (line 6).

Algorithm 5.2: Logical Status

```
1  Function IsNodePresent(Node* n, int key)
2      return n.key = key
3  Function IsKeyPresent(NodeInfo* info, Desc*desc)
4      OpType op ← info.desc.ops[info.opid]
5      if op = Find then
6          return True
7      TxStatus status ← info.desc.status
8      PersStatus pstatus ← info.desc.pstatus
9      switch status do
10         case Active do
11             if info.desc = desc then
12                 return op = Insert
13             else
14                 return op = Delete
15         case Committed do
16             return op = Insert and pstatus = Persisted
17         case Aborted do
18             return op = Delete and pstatus = Persisted
```

In the next step, we read the status of the last transaction that accessed the node. If the last transaction is still active and the node was inserted by an operation in the current transaction, we reveal the presence of the node only to the subsequent operations in the same transaction (line 12). If the last transaction executed a Delete operation but is not committed yet, we declare the node as present.

If the last transaction has finished its execution on the node, we determine whether its effect is observable by the current transaction by examining two cases. In the first case (line 16), the key logically exists only if the last transaction has executed an INSERT operation, committed successfully, and made its descriptor durable. In the second case (line 18), we consider a DELETE operation. By definition, a successful DELETE operation must remove the key from the set. To report a key as present, if the last transaction has executed a DELETE operation, it must have aborted and persisted its descriptor. If any of the above conditions are not met, the key does not logically exist from the point of view of the current transaction.

Algorithm 5.3: Update Info

```
1  Function UpdateInfo(Node* n, NodeInfo*info, bool wantkey)
2      NodeInfo* oldinfo ← n.info
3      if IsMarked(oldinfo) then
4          Do_Delete(n)
5          return retry
6      if oldinfo.desc ≠ info.desc then
7          ExecuteOps(oldinfo.desc, oldinfo.opid + 1)
8      else
9          if oldinfo.opid ≥ info.opid then
10             return success
11     bool haskey ← IsKeyPresent(oldinfo)
12     if (!haskey and wantkey) or (haskey and !wantkey) then
13         return fail
14     if info.desc.status ≠ Active then
15         return fail
16     if CAS(&n.info, oldinfo, info) then
17         return success
18     else
19         return retry
```

Function ISKEYPRESENT is called by function UPDATEINFO, which starts at line 1 of Algorithm 5.3. An operation in the underlying data structure needs to update the $info$ pointer of its

active node before making changes. This update is necessary as the $info$ pointer of the node is used to determine its logical status. The active operation calls function UPDATEINFO to perform this modification (Figure 5.3). If the target node is logically marked for deletion, we complete the operation by invoking the base data structure delete method and inform the caller to retry the current operation. Before updating the node info, the current thread first helps complete other transactions if needed (line 7 (EXECUTEOPS is illustrated in Figure 5.1)). Also, if a helper thread already executed the current operation, we can ignore this operation and continue the rest of the transaction (line 9). Next, we check if the logical presence of a key is matched with the need of the operation. For example, a DELETE operation expects that the key to be present in the list and an INSERT operation requires that the key to not be a part of the list. UPDATEINFO evaluates these conditions in line 11. After verifying the liveness of the current transaction, $n.info$ is updated by using a CAS (line 16) and the data structure operation can proceed.

5.1.4 Executing Durable Transactions

Algorithm 5.4 presents our method of ensuring the durability of transactions starting at line 1 using the PERSISTTRANSACTION function. Since we only need to ensure the durability of the transaction descriptor object, the descriptor is all that PERSISTTRANSACTION needs as the input. A thread in this function first declares its intent to persist the transaction descriptor. This declaration prevents possible helper threads from re-persisting the descriptor by executing expensive flush and fence operations. If a thread commits or aborts its transaction, but gets delayed in the middle of the persistence, another thread can help persist the transaction. The need for help can be inferred based on the delayed transaction's status.

If the current thread successfully declares its intent to persist the descriptor, it traverses over all the operations and flushes the information related to each operation. In the next step, the transaction

id is assigned to the descriptor. The recovery procedure (Section 5.1.5) uses the transaction id to determine the order of transactions executed on each key in a data structure. Next, the descriptor object is flushed to store the remaining information about the transaction. To guarantee that the persisted transaction is visible, we use the SFENCE instruction. Finally, we set the persistency status of the transaction to PERSISTED to notify other transactions. After the execution of line 8, the effect of the current transaction is globally visible. Note that we do not need to ensure the persistence of the PSTATUS itself, as its value is implied by a transaction that is persisted.

Algorithm 5.4: Persistence of Transactions

```
1 Function PersistTransaction(Desc* desc)
2     if (CAS(&desc.pstatus, Maybe, InProgress)) or (desc.pstatus == InProgress)
      then
3         for op ∈ desc.ops do
4             FLUSH(&op)
5         desc.txid ← GetNextTXID()
6         FLUSH(desc)
7         SFENCE()
8         desc.pstatus ← Persisted
```

Function PERSISTTRANSACTION provides durability at low cost by reducing the number of flushes and fences. In total, the number of flushes corresponds to the size of the transaction plus one more flush to store the transaction descriptor. Finally, for each transaction, we explicitly execute one fence instruction regardless of its size.

As illustrated by Figure 5.1, each transaction executes the data structure operations specified by the descriptor object. Figure 5.3 presents how our methodology executes this step. Each data structure operation features a CAS-based while loop, which is the typical approach for implementing non-blocking data structures. Each thread attempts to apply updates on the shared object atomically, and if it fails, it retries the operation execution if needed. Functions that start with a prefix DO_- represent the methods typically implemented by a linked list-based set, which is the underlying

data structure in our example. For example, DO_LOCATEPRED returns the required nodes and variables for handling the linkage in the structure, e.g., the predecessor node. DO_OPERATION could be any of DO_INSERT and DO_DELETE functions that add and remove the necessary links to perform their operations respectively.

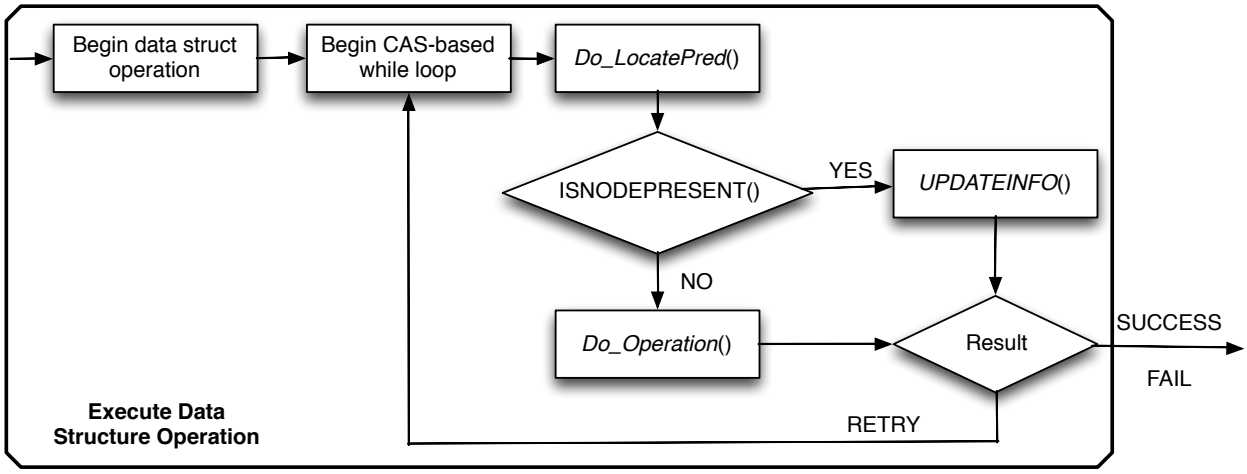


Figure 5.3: Executing data structure operation

If the node exists in the structure of the linked list, we call the UPDATEINFO function (line 1 of Algorithm 5.3) before making changes. This step is necessary to interpret the logical status of a node and update it to prevent unsafe access by concurrent transactions. Based on the results of the call to this function, we determine whether another attempt is needed to perform the operation or return the result. If the node does not exist, no call to UPDATEINFO is needed, and the operation can proceed.

5.1.5 Recovery Management

5.1.5.1 System Support and Memory Addressing

When a system crash, objects in persistent memory need to be found and mapped back into the process address space. This requires system support, such as memory-mapped files [27, 29, 42, 76, 143, 141, 148], persistent memory-aware file systems [42, 29, 148, 149], or system-managed objects in memory [150]. Once found, the region may be remapped to the process address space at a different virtual memory location, hence relocatability needs to be supported [107, 89], such as using new relocatable pointer formats [145, 22], and persistent page table [150]. Addressing these issues is orthogonal to PETRA and beyond the scope of this work. Note that there is nothing that fundamentally prevents these ideas from being applied to PETRA.

5.1.5.2 Recovery Procedure

Recall that PETRA explicitly persists transaction descriptors at the end of each transaction. The recovery procedure rebuilds the underlying data structure, verifies its consistency using the transaction descriptors, and fixes possible inconsistencies that might have occurred as a result of a crash. Figure 5.4 presents the steps to recover a data structure (linked list-based set) after a crash. Upon recovery, the initial set is built by loading the head node. Any node reachable from the head node is a part of this initial list. Next, the transaction descriptors that were persisted by each thread are read in order to figure out transaction execution records ①. Based on the transaction descriptors, we build the key-descriptor map (*KDMap*) ②. This involves visiting each committed/persisted transaction to find the transactions that accessed each key in the data structure. If we have more than one transaction that is executed on a key, we use TXID of the descriptor to identify the transaction that happened last. Note that TXID is assigned by a global monotonically increasing generator

before persisting the transaction descriptor (line 5 of Algorithm 5.4). No transaction is visible to other threads unless TXID of its descriptor is assigned and persisted. The ordering mechanism here does not need to enforce a global ordering on all transactions. It is only sufficient to know the order of transactions executed on each key, which can be achieved using TXID generated by tools such as simple FETCH-AND-ADD operations, time-stamps, or other similar techniques.

Next, we traverse the loaded set ③ and determine the logical status of a key based on the last valid transaction that is executed on the node with that key. *KDMap* provides the descriptor for this transaction. If the descriptor pointer of the node, is not persisted before crash, it does not match the descriptor found by KDMap. In this case, we remove the node and execute the corresponding operation based on the data provided by the valid descriptor and we end up in a valid state for the node. If the descriptor found by KDMap matches the node's descriptor there are two cases to consider: 1) node contents are valid, i.e., the value is correct and 2) node contents are invalid. In the second case, to restore the consistent state, we remove the node and execute the descriptor's operation. To fix other possible inconsistencies, we insert the items that, according to the KDMap, should be present in the data structure but are not ④. After this step, the data structure is restored to a consistent state and the recovery procedure is complete ⑤.

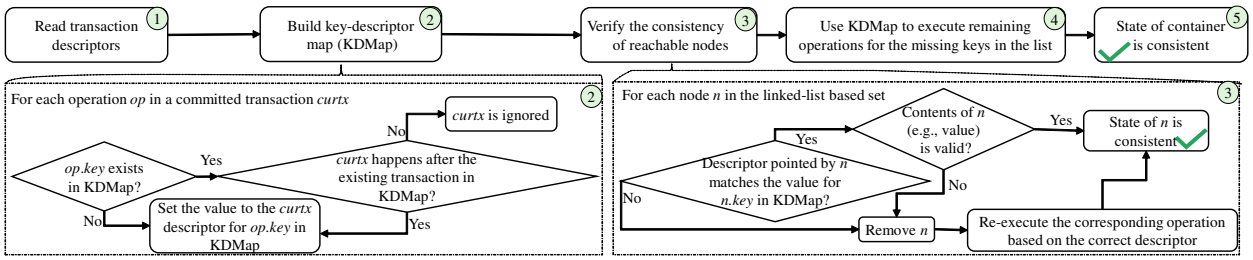


Figure 5.4: Recovery steps

To guarantee consistency, we do not need to use any of the transaction descriptors that are not persisted, even for those transactions that are completed. As we describe in Section 5.2, we use

durable linearizability [82], which is the strictest correctness property to ensure a consistent state of the data structures. Durable linearizability requires that the state of the data structure after a crash includes a consistent subhistory of the operations that actually occurred and were globally visible before the crash. As we discuss in Section 5.1.3, the effect of a completed transaction is visible to other threads only when its transaction descriptor becomes persistent.

5.2 Correctness

We now show that PETRA satisfies durable linearizability. Section 5.2.1, presents definitions known from published work that we use in Section 5.2.

5.2.1 Correctness definitions

Definitions are provided to facilitate reasoning about durable linearizability. An execution of a concurrent system is modeled by a *history*, a finite sequence of method *invocation* and *response* events [70]. A response *matches* an invocation if they are called by the same thread on the same object. A *method call* in a history H is a pair consisting of an invocation and next matching response in H, also referred to as an *operation*. An invocation is *pending* in H if no matching response follows the invocation. An *extension* of H is a history constructed by appending responses to zero or more pending invocations of H. The notation $complete(H)$ denotes the subsequence of H consisting of all matching invocations and responses. A *sequential specification* for an object is a set of sequential histories for the object. A sequential history H is *legal* if each object subhistory is legal for that object.

Definition 1. *A history H is linearizable if it has an extension H' and there is a legal sequential history S such that 1) $complete(H)$ is equivalent to S, and 2) if m_0 precedes method call m_1 in*

H, then the same is true in S [70].

Legal sequential history S in Definition 1 is referred to as a *linearization* of H.

Definition 2. *Given an execution E, an operation O is durable at step t of the (extended) execution E if the following holds. For any legal execution E', which equals E in the first t steps, if the execution of the recovery of O completes in E', then for any linearization of E', O is linearized.*

An operation is considered durable if there is sufficient information in NVM such that the recovery procedure causes this operation to be linearized.

Definition 3. *Given an extended execution E, the durability point of operation O is the first point t in the execution when the operation O becomes durable.*

Definition 4. *Given an execution E, the durability points of the operations in the execution E imply an order on the operations, called durability order.*

Definition 5. *A linearizable object is durably linearizable if for all executions E of the object, 1) the durability point of each operation is between its invocation and response, and 2) there exists a linearization of E whose order of operations is the same as the durability order of operations in E [56].*

Definition 6. *A history H is strictly serializable if the subsequence of H consisting of all events of committed transactions is equivalent to a legal sequential history S in which these transactions execute sequentially in the order they commit [114].*

Legal sequential history S in Definition 6 is referred to as a *strict serialization* of H.

We extend the notion of durable linearizability to transactions by considering an "operation" in Definition 5 to be a transaction and a "linearization" in Definition 5 to be a strict serialization.

5.2.2 Durable Linearizability

To prove that PETRA is durably linearizable, it must be shown that for all multithreaded executions E, 1) the durability point of each transaction is between its invocation and response, and 2) there exists a strict serialization of E whose order of transactions is the same as the durability order of transactions in E.

Theorem 1. *PETRA is durably linearizable.*

Proof. First, it is shown that the durability point of a transaction occurs between its invocation and response. When EXECUTETRANSACTION is invoked for transaction T_1, the operations listed in T_1's transaction descriptor are executed according to operation order. If T_1 detects a conflict with transaction T_2, then T_1 helps complete T_2 prior to proceeding with its own operations. If transaction T_3 detects a conflict with T_1, then T_3 helps complete T_1. Once T_1's operations have been completed, a CAS is attempted to either commit or abort T_1. If the CAS fails, then some other thread must have either committed or aborted T_1. After T_1 has either committed or aborted, it is persisted by invoking PERSISTTRANSACTION. Since T_1 is guaranteed to be durable once it returns, the durability point for T_1 occurs between its invocation and response.

Next, it is shown that there exists a strict serialization of E whose order of transactions is the same as the durability order of transactions in E. The ISKEYPRESENT function prevents transaction T_1's operations from being visible to other transactions until T_1 is persisted due to the return value on line 5.2.16 and line 5.2.18. Since the effects of T_1's operations are visible to other transactions at the instant it is persisted and PETRA is strictly serializable by the LFTT methodology [156], there exists a strict serialization of E whose order of transactions is the same as the durability order of transactions in E.

If a crash occurs, the recovery procedure is invoked by the main thread to restore the state of

the PETRA-based data structure. It now must be shown that the restored state reflects a strict serialization of E whose order of transactions is the same as the durability order of the operations in E. As described in Section 5.1.5, *KDMap* is a map where the key is the node key and the value is the most recent committed/persisted transaction that accesses the node key in the data structure. We now show that a valid state of the data structure can be recovered from *KDMap*. Since set operations that access different nodes are commutative, the order of the set operations relative to different keys does not affect the outcome of the node state. Let $T_1, T_2, ..., T_j, ..., T_{n-1}, T_n$ be the history of committed/persisted transactions in persist order as described in Section 5.1.5. The IsKeyPresent function only enables committed transactions that have persisted to be visible to other transactions, so the commit order is equivalent to the persist order. Let T_j be the last committed/persisted transaction to access some node k. Let op_j be the last operation in T_j to access node k. Since T_j commits, this implies that op_j succeeds. Let S be the set of nodes that exist in the list. If op_j is Find or Insert, $node\ k \in S$. If op_j is Delete, $node\ k \notin S$. The same reasoning applies for all other nodes in the data structure. Therefore, the state of the data structure consistent with a strict serialization of E whose order of transactions is the same as the durability order of the operations in E can be recovered from *KDMap*.

□

5.3 Experimental Evaluation

In this section, we evaluate our approach and compare it against the state-of-the-art PTM platforms using various benchmarks.

5.3.1 Experimental setup

5.3.1.1 Machine Testbed

We conduct our tests on a machine equipped with Intel Optane DC Persistent Memory (DCPM). The machine has Intel's most recent second-generation Xeon Scalable processors (codenamed Cascade Lake) with 48 cores (2 sockets), supporting 96 threads. The main memory consists of Optane DCPM with 6TB total capacity, plus 768GB DRAM. In all experiments, we place persistent data structures in the DCPM; DRAM is used to store everything else (e.g. code). The machine is configured to run in 100% App Direct Mode [83], which allows applications byte-addressable access to the persistent memory. The OS is Ubuntu 18.04 LTS. The application and micro-benchmarks were compiled using gcc 7.4 with the -O3 optimization flag and C++14 standard flags.

5.3.1.2 Micro-benchmarks

We conduct our evaluations on four transactional non-blocking data structures: three different sets based on linked list, skiplist and multi-dimensional list (mdlist), and hash map. In the linked list-based set experiments, each thread performs $100,000$ transactions and the key range is set to $10{,}000$. In the experiments for other data structures, each thread performs $1{,}000{,}000$ transactions and the key range is set to $1{,}000{,}000$.

In micro-benchmarks, we compare the overhead and scalability of PETRA against three state-of-the-art PTMs: OneFile (lock-free version) [121], Romulus (LR version) [30], and PMDK (libpmemobj++ protected using read-write locks) [116]. We also ran experiments using Mnemosyne [142], but we did not include the results, because it exhibits the lowest throughput and does not support more than 31 threads [30]. Romulus was reported to outperform PMDK and Mnemosyne

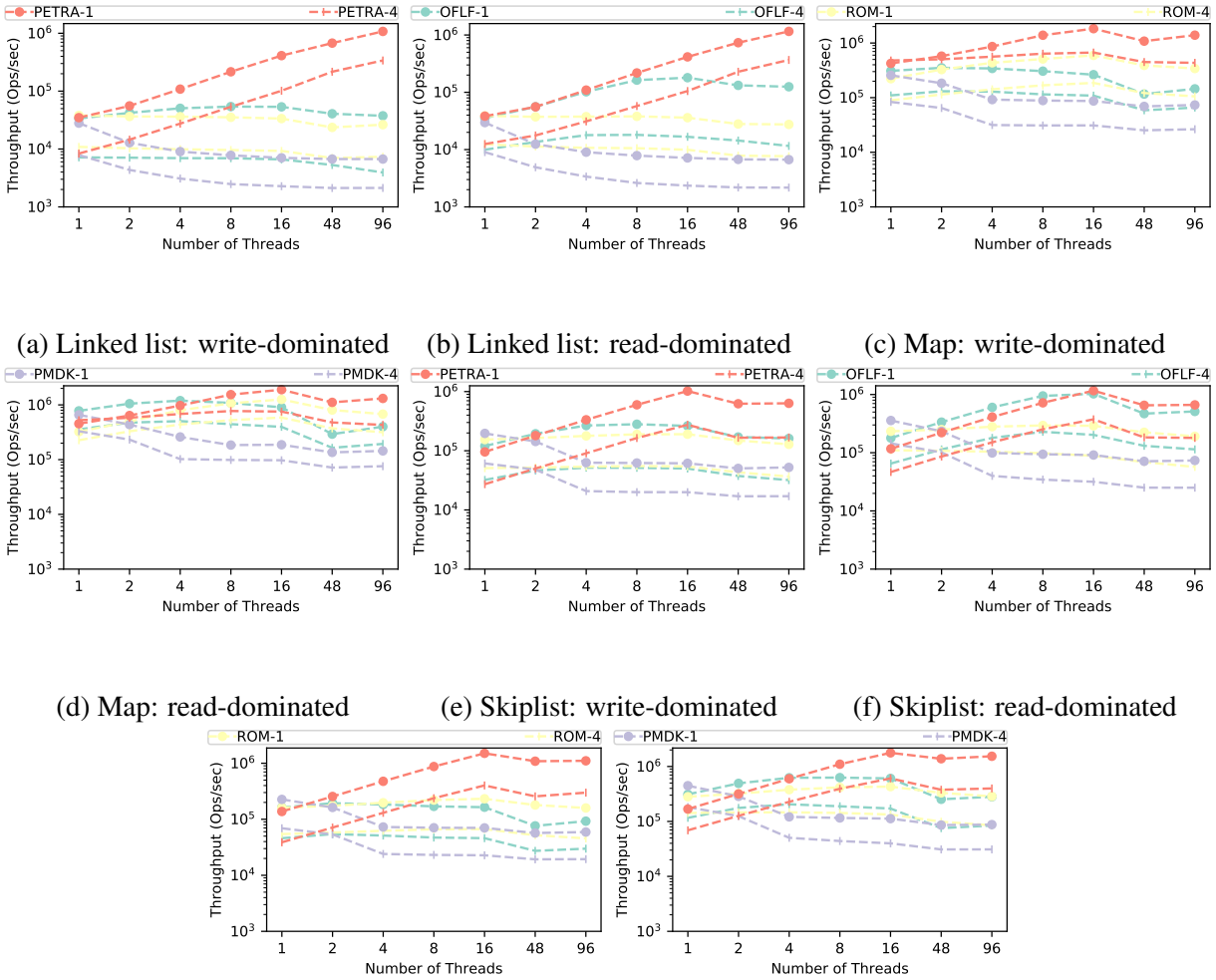
and OneFile shows a slightly better throughput compared to Romulus in some cases in the literature [30, 121]. We run our micro-benchmark experiments to evaluate the overall performance using various workloads based on the ratio of read and write operations. This method of evaluation, commonly used in the literature [63, 30, 56, 156, 121], consists of a loop that randomly chooses a transaction to execute with a mixture of read and write operations according to a uniform distribution, and operation ratio and workload type.

5.3.2 Micro-benchmark evaluation results

Figure 5.5 displays the throughput for the transactional linked list (a,b), map (c,d), skiplist (e,f), and mdlist (g,h) implementations using different workloads (note the logarithmic scales). Throughput (y-axes) reflects the number of completed operations per second. In all plots, our scheme is denoted by PETRA, OneFile by OFLF, Romulus by ROM, and PMDK by PMDK. The transaction size (number of operations in a transaction) varies from 1 to 16. In Figure 5.5, we report the results based on the transaction sizes of 1 and 4 to present clear plots and the rest of the results are reported in Figure 5.6. The transaction size appears as a suffix to each set (e.g., PETRA-4 means transaction size 4 for PETRA). Each thread allocates memory from a pre-allocated pool. The number of threads varies from 1 to 96.

Figure 5.5 (a) displays results for a write-dominated workload for the linked list-based set. For a single thread, all approaches perform close to each other. As the thread count increases, PETRA's throughput increases substantially, while the throughput of other approaches stagnates or declines. The structure of the SET abstract data type makes it a suitable choice to exploit the parallelism of a multi-threaded system by distributing contention across nodes. PETRA exhibits high throughput and scalability in this case that can be attributed to its non-blocking approach that keeps abort rates low. The high abort rates due to false aborts in the alternative approaches keep them from increas-

ing their throughput. At 48 and 96 threads, PETRA outperforms the next performing technique, OneFile, by more than one order of magnitudes.



(a) Linked list: write-dominated (b) Linked list: read-dominated (c) Map: write-dominated

(d) Map: read-dominated (e) Skiplist: write-dominated (f) Skiplist: read-dominated

(g) MDlist: write-dominated (h) MDlist: read-dominated

Figure 5.5: Throughput for transactional data structures for transactions of size 1 and 4. Operation ratio for write-dominated workload in lists: 40% Insert, 40% Delete, 20% Find and maps: 40% Insert, 30% Delete, 10% Update, 20% Find. Operation ratio for read-dominated workload in lists: 10% Insert, 10% Delete, 80% Find and maps: 10% Insert, 10% Delete, 5% Update, 75% Find. Key range for linked list: $10K$, other data structures: $1M$.

We show the results from read-dominated workloads in Figure 5.5 (b). The results for these work-

loads follow a similar trend as the write-dominated intensive workload, but OneFile and Romulus exhibit better performance compared to the read-dominated workloads. Romulus uses lighter synchronization mechanisms to optimize read-only operations that enable reader scalability, with throughput slightly increasing with thread counts. PETRA uses transaction descriptors for all operations and updates the references even for read operations such as FIND, hence its scalability remains the same as in write-dominated workload. As a result, PETRA's throughput advantage over OneFile and Romulus decreases, but it is still larger than one order of magnitude with 96 threads.

For hash map experiments, in part (c) with the write-dominated case, PETRA outperforms all the alternative approaches, again thanks to not suffering from many transaction aborts due to helping and not having false aborts. In part (d) with mostly read operations, similar to the linked list experiments, the throughput of other approaches is improved. PETRA performs not as well for lower thread counts, but it scales better at higher thread counts and outperforms alternative transactional implementations. We also evaluate the performance of PETRA's hash map using a database benchmark. These results are presented in Section 5.3.4.

Transactional skiplist and mdlist display a similar trend to the transactional hash map. The base data structures in both cases [54, 154] have logarithmic search times and execute transactions more efficiently compared to the linked list-based set. Although for these types of data structures, in read-dominated cases with lower thread counts all approaches exhibit close throughput, overall PETRA performs 3 times better than the next best PTM.

5.3.2.1 Impact of transaction size

The general trend of the baseline comparison in transactions of size one can be observed for larger transactions too. In general, smaller transactions reduce the probability of transaction conflicts and

boost scalability. However, larger transactions are often needed and convenient to the programmer.

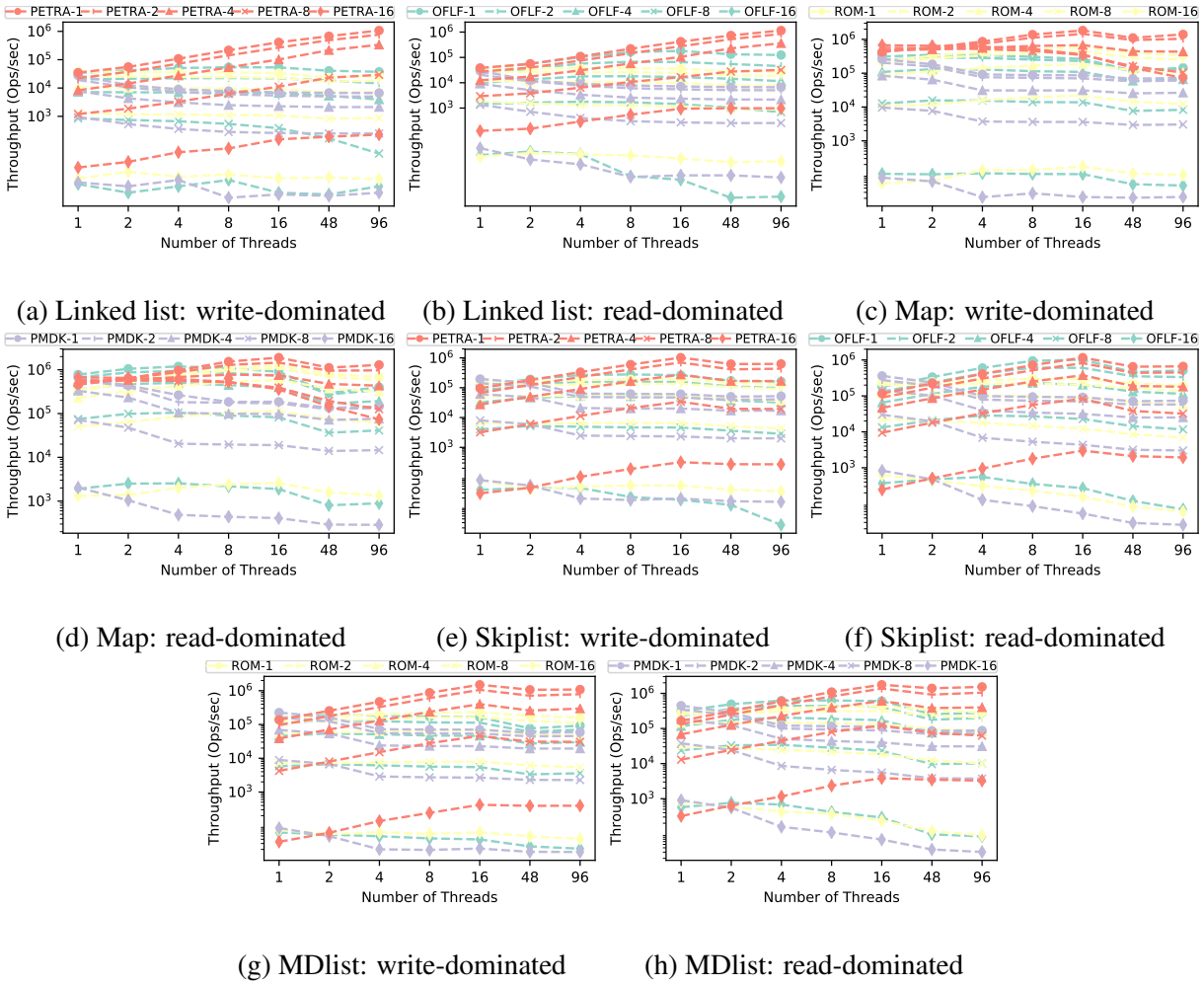
As expected, larger transactions are more vulnerable to conflicts based on the data structure semantics, hence throughput decreases with transaction size for all approaches. For example, going from size 2 to 4, we observe around 50% reduction in throughput. At the extreme (size of 16), the throughput is at about 1% compared to that of size 8. Scalability is the key difference between PETRA and other approaches. The scalability of PETRA across thread counts holds regardless of the transaction size. In contrast, the throughput of all other PTMs decreases with a higher thread count.

Although increasing the transaction size results in more aborts, using more threads can compensate for the loss of throughput. For example, consider the write-dominated workload in the linked list-based set experiments. At transaction size of 4, using 96 threads results in higher throughput than executing transactions of size 2 with just 16 threads. The performance loss resulted from increasing the transaction size for Romulus, OneFile, and PMDK is more severe compared to PETRA. PETRA with 48 threads and transactions of size 8 outperforms other PTMs in almost all combinations of transaction sizes and number of threads. By increasing the size to 16, other approaches almost fail to execute transactions.

Figure 5.6 presents the micro-benchmark results for all transaction sizes. We vary the transaction size from 1 to 16. As we discussed, large transactions have a higher chance to abort because of the possible conflicts between the data structure operations. Because of these aborts, the throughput is decreased across all approaches.

The performance drop in PETRA is lower compared to other approaches. For example, consider the read-dominated workload in transactional hash map (Figure 5.6(d)), where Romulus has the best performance between approaches other than PETRA. While PETRA, on average, exhibits about 45% higher throughput for transactions with one operation, it performs extremely better

than Romulus for transactions of size 16 and shows more than 200 times higher throughput.



(a) Linked list: write-dominated (b) Linked list: read-dominated (c) Map: write-dominated

(d) Map: read-dominated (e) Skiplist: write-dominated (f) Skiplist: read-dominated

(g) MDlist: write-dominated (h) MDlist: read-dominated

Figure 5.6: Throughput for transactional data structures for larger transactions. Operation ration and key ranges similar to Figure 5.5.

5.3.3 TATP benchmark

We evaluate our transactional map in the TATP benchmark [137] by testing *UpdateLocation* transactions and compare its performance with generic PTMs proposed in the recent literature [152]. Figure 5.7 presents these results. Throughput reflects the number of millions of transactions ex-

ecuted per second. While other approaches exhibit poor scalability, TLRW [37] and Orec [36, 51, 135] perform as good as PETRA for low thread counts but fail to scale as we increase the number of threads (TLRW crashed when running with 96 threads). Orec uses ownership records with variants of undo/redo logging, the locking mechanisms, and lazy/eager approaches. TLRW is an eager algorithm with readers/writer locks that does not require quiescence to ensure safety during commit. This feature and other optimizations, such as fence pipelining, contribute to the better scalability. Similar to write-dominated workloads in Figure 5.5, PETRA demonstrates its scalability and shows over 9 times higher throughput compared to the best PTM at 96 threads. This advantage happens as a result of leveraging the data structure semantic knowledge to manage both concurrency and durability efficiently, which also reduces the number of required flushes and fences.

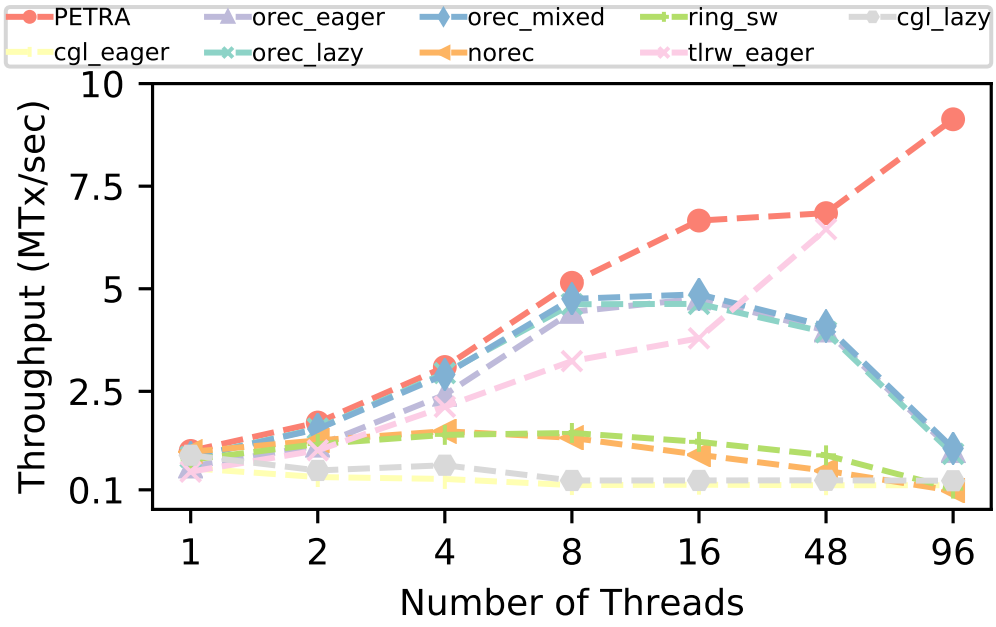


Figure 5.7: Performance comparison of PETRA with general-purpose PTMs in TATP benchmark.

5.3.4 Database benchmark

We demonstrate the application of our methodology in a persistent key-value store by using PETRA's transactional map. We integrated our transactional hash map with *pmemkv* [117], a key/-value datastore for persistent memory. We evaluate and compare it against an implementation based on Intel TBB concurrent hash map. To add transactional capabilities to the implementation based on the TBB map, we use *abstract locking* with undo logs, analogous to transactional boosting [68]. We use a benchmark named *pmemkv_bench* from *pmemkv-tools* [118], which provides a collection of standard read and write benchmarks. The benchmarks are based on the *db_bench* utility, which is integrated with popular databases such as LevelDB [59] and RocksDB [48].

In all benchmarks, we utilize integer keys and values and each thread executes one million transactions and each transaction performs four operations. In the *fillseq* benchmark, each thread executes insert-only transactions using sequential keys. The *fillrandom* benchmark performs the same but with random keys per thread. The *overwrite* benchmark performs the insertions similar to *fillrandom*, but works on a database that is filled with the key-value pairs. The *readseq*, *readrandom*, *deleteseq*, and *deleterandom* benchmarks are similar to their *fill* versions, but perform read and delete transactions. The *readmissing* benchmark reads N missing values in random order. In the *readrandomwriterandom* benchmark, all threads carry out transactions with both types of operations randomly. In this benchmark, 90% of operations are read and 10% of them are write operations.

Figure 5.8 presents the results, with the y-axis showing the time (in microseconds) to execute an operation, while the x-axis shows two sets of bars: *cmap* represents Intel TBB's concurrent hash map, and our approach is denoted with *PETRA*. Each set contains 7 bars corresponding to the following thread counts: 1, 2, 4, 8, 16, 48, 96.

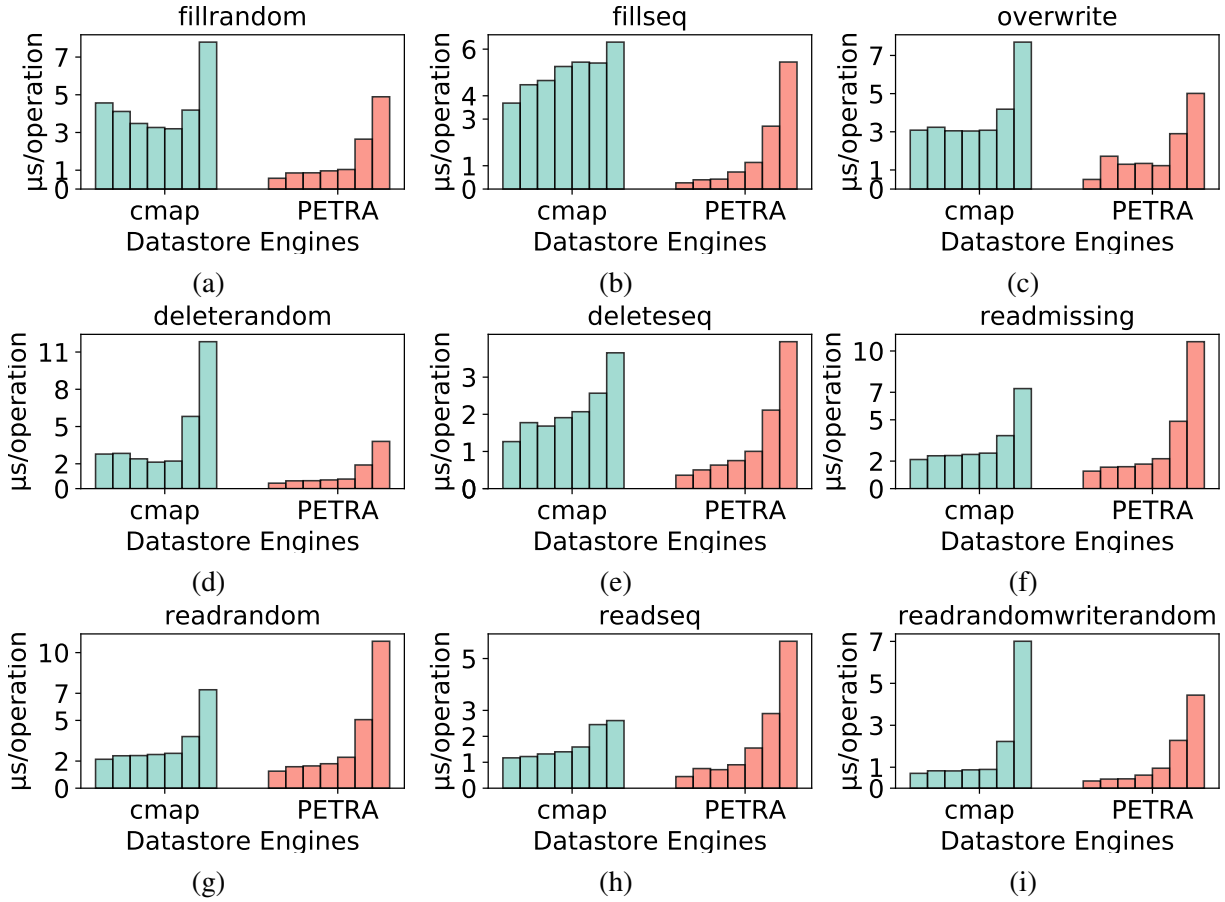


Figure 5.8: Database benchmark. Number of threads in all plots (1,2,4,8,16,48,96).

For write-only workloads, (a-e), PETRA allows faster database transaction execution in all cases. For read-only workloads, (f-h), PETRA outperforms cmap in low thread counts except when the system uses threads on both CPU sockets. PETRA's engine outperforms Intel TBB's concurrent map engine in the mixed workload, (i), that each transaction can execute both read and write operations.

5.4 Future Work

PETRA brings the benefit of high performance at the cost of space amplification ($2 - 3\times$). Most of the amplification is due to LFTT. To achieve persistence, PETRA itself only adds 12-35% space overheads on top of LFTT, depending on the transaction size. This is a reasonable trade-off especially since persistent memory capacity is much higher than DRAM. In this work, we assumed that small objects are used in the transactions and operation data fit in the cache-line. To guarantee failure-atomicity for transactions with larger objects, we need to ensure the durability of the large object before persisting the transaction, and to persist data that do not fit in a single cache-line, more flushes are needed. We also assumed that a crash is rare and our methodology follows the principle of optimizing of the failure-free execution at the expense of possibly slower recovery. In future work, we plan to employ a *periodic checkpointing* mechanism to put an upper bound on the number of past persisted transactions to validate. This mechanism can also improve the persistent memory space overhead. We also plan to apply our approach to the extended versions of LFTT to support features such as wait-freedom, dynamic transactions, and more data structures. We will also apply our techniques in implementing an in-memory database.

5.5 Chapter Summary

In this chapter, we presented PETRA, a new technique to create persistent non-blocking transactional data structures with ACID properties. We leveraged descriptor objects to implement an efficient scheme that manages concurrency and durability. PETRA achieves high performance by keeping the number of cache line flushes and memory fences low, persisting a transaction by only persisting its descriptor, and by persisting data structures lazily without using flushes and fences. It uses the transaction descriptors as redo logs. It achieves high scalability by eliminat-

ing false aborts (by utilizing high-level knowledge of data structure semantics) and reducing true aborts (through helping). PETRA also preserves LFTT's non-blocking progress guarantee. Our performance evaluation demonstrates that our approach, on average, exhibits $17\times$ and $3\times$ higher throughput compared to the state-of-the-art PTM, for mixed workloads that utilize set and other data structures, respectively.

CHAPTER 6: CONCLUSION

In this dissertation, we presented approaches that enable the efficient resource utilization and performance analysis in HPC and cloud environments. We presented methods for enabling real-time troubleshooting of system components and applications, streaming application performance data for system monitoring, and efficient durable transactional data structures that can be utilized by the in-memory databases persisting monitoring data.

In Chapter 3, we presented the design of integrating low-latency analysis into system monitoring. This design enables low-latency access to results both off-platform and on-platform where they can be used to provide feedback to applications and system services. The transformed data is supported by the same structures as the collected data, thus enabling the transformed data set the same flexibility in transport and the same exposure as the collected data. We demonstrated the effectiveness of our implementation for a case with production-relevance: run-time determination of the relative per-node filesystem demands. Run-time availability and exposure of such data would be of benefit to those seeking to identify the causes of high load on the filesystem and to identify imbalances in an application's resource demands.

In Chapter 4, we developed and demonstrated a hybrid approach to HPC application monitoring that supports performance analysis in production conditions. This approach takes advantage of the low overhead of shared memory and LDMS to provide insights into the application's behavior using profiling during the execution. We have implemented a tool-set based on this design to evaluate its impact. Our experimental evaluations demonstrate the impact of our low-overhead tool-set in understanding the application behavior during run-time under different situations.

In Chapter 5, we presented PETRA, a new technique to create persistent non-blocking transactional data structures with ACID properties. We leveraged descriptor objects to implement an efficient

scheme that manages concurrency and durability. We discussed how PETRA achieves high performance, high scalability, and non-blocking progress. Our performance evaluation demonstrated that our approach, on average, exhibits $17\times$ and $3\times$ higher throughput compared to the state-of-the-art PTM, for mixed workloads that utilize set and other data structures, respectively. PETRA enables the development of in-memory databases which are capable of handling large amount of data efficiently.

LIST OF REFERENCES

[1] L. Abraham et al. Scuba: diving into data at facebook. *Proceedings of the VLDB Endowment*, 6(11):1057–1067, 2013.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[3] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, and J. Stevenson. The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 154–165. IEEE, 2014.

[4] A. Agelastos, B. Allan, J. Brandt, A. Gentile, S. Lefantzi, S. Monk, J. Ogden, M. Rajan, and J. Stevenson. Continuous whole-system monitoring toward rapid understanding of production hpc applications and systems. *Parallel Computing*, 58:90–106, 2016.

[5] A. Agelastos, M. Rajan, N. Wichmann, R. Baker, S. Domino, E. Draeger, S. Anderson, J. Balma, S. Behling, M. Berry, et al. Performance on trinity phase 2 (a cray xc40 utilizing intel xeon phi processors) with acceptance applications and benchmarks. *Cray User Group CUG, May*, 2017.

[6] A. M. Agelastos and P. T. Lin. Simulation information regarding sandia national laboratories' trinity capability improvement metric. *Sandia National Laboratories, Albuquerque, New Mexico*, 87185, 2013.

[7] M. Alshboul, J. Tuck, and Y. Solihin. Lazy persistency: A high-performing and write-efficient software persistency technique. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 439–451, Washington, DC, USA, June 2018. IEEE.

[8] H. Avni and T. Brown. Persistent hybrid transactional memory for databases. *Proc. VLDB Endow.*, 10(4):409–420, Nov. 2016.

[9] H. Avni, E. Levy, and A. Mendelson. Hardware transactions in nonvolatile memory. In Y. Moses, editor, *Distributed Computing*, pages 617–630, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[10] G. Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 261–270, New York, NY, USA, 1993. ACM.

[11] G. H. Bauer et al. Dynamic model specific register (MSR) data collection as a system service. In *Cray Users Group (CUG)*, 2016.

[12] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 11(1):3–16, 2008.

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

[14] S. Benedict, R. Rejitha, and C. Bright. Energy consumption analysis of HPC applications using nosql database feature of energyanalyzer. In *International Conference on Intelligent Cloud Computing*, pages 103–118. Springer, 2014.

[15] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs. There goes the neighborhood: performance degradation due to nearby jobs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 41. ACM, 2013.

[16] S. Bohm, C. Englemann, and S. Scott. Aggregation of real-time system monitoring data for analyzing large-scale parallel and distributed computing environments. In *IEEE Int'l Conference on High Performance Computing and Communications (HPCC)*. IEEE, 2010.

[17] J. Brandt, E. Froese, A. Gentile, L. Kaplan, B. Allan, and E. Walsh. Network performance counter monitoring and analysis on the Cray XC platform. In *Cray Users Group (CUG)*, 2016.

[18] J. Brandt, A. Gentile, M. Showerman, J. Enos, J. Fullop, and G. Bauer. Large-scale persistent numerical data source monitoring system experiences. In *IEEE Int'l Parallel and Distributed Processing Symposium Worksshops (IPDPSW)*. IEEE, 2016.

[19] S. Browne et al. A portable programming interface for performance evaluation on modern processors. *The international journal of high performance computing applications*, 14(3):189–204, 2000.

[20] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):40:46–40:58, Sept. 2008.

[21] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[22] G. Chen, L. Zhang, R. Budhiraja, X. Shen, and Y. Wu. Efficient support of position independence on non-volatile memory. In H. C. Hunter, J. Moreno, J. S. Emer, and D. Sánchez, editors, *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, pages 191–203. ACM, 2017.

[23] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, Feb. 2015.

[24] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *In CIDR*. Citeseer, 2003.

[25] S. Chunduri, K. Harms, S. Parker, V. Morozov, S. Oshin, N. Cherukuri, and K. Kumaran. Run-to-run variability on xeon phi based cray xc systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 52. ACM, 2017.

[26] P. Chuong, F. Ellen, and V. Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 335–344, New York, NY, USA, 2010. ACM.

[27] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[28] Collectl. Collectl, 2014.

[29] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *ACM Symposium on Operating Systems Principles*, Oct. 2009.

[30] A. Correia, P. Felber, and P. Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 271–282, New York, NY, USA, 2018. ACM.

[31] H. Cui, K. Keeton, I. Roy, K. Viswanathan, and G. R. Ganger. Using data transformations for low-latency time series analysis. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 395–407. ACM, 2015.

[32] T. David, A. Dragojević, R. Guerraoui, and I. Zablotchi. Log-free concurrent data structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 373–386, Boston, MA, 2018. USENIX Association.

[33] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-free dynamically resizable arrays. In M. M. A. A. Shvartsman, editor, *Principles of Distributed Systems*, pages 142–156, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[34] A. DeConinck et al. Design and implementation of a scalable HPC monitoring system for Trinity. In *Cray Users Group (CUG)*, 2016.

[35] A. DeConinck et al. Runtime collection and analysis of system metrics for production monitoring of Trinity phase II. In *Cray Users Group (CUG)*, 2017.

[36] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In S. Dolev, editor, *Distributed Computing*, pages 194–208, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[37] D. Dice and N. Shavit. Tlrw: Return of the read-write lock. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 284–293, New York, NY, USA, 2010. Association for Computing Machinery.

[38] J. Dixon. *Monitoring with Graphite: Tracking Dynamic Host and Application Metrics at Scale*. " O'Reilly Media, Inc.", 2017.

[39] S. Domino. Nalu's milestonerun test at master. https://github.com/NaluCFD/Nalu/tree/master/reg_tests/test_files/milestoneRun, 2018.

[40] S. Domino. Sierra low mach module: Nalu theory manual 1.0. https://github.com/NaluCFD/NaluDoc, 2018.

[41] B. Dong, X. Li, Q. Wu, L. Xiao, and L. Ruan. A dynamic and adaptive load balancing strategy for parallel file system with large-scale i/o servers. *Journal of Parallel and Distributed Computing*, 72(10):1254–1268, 2012.

[42] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[43] Elasticsearch, BV. The elastic stack, 2018.

[44] F. Ellen, P. Fatourou, J. Helga, and E. Ruppert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 332–340, New York, NY, USA, 2014. ACM.

[45] D. Eppstein and E. Havvaei. Parameterized Leaf Power Recognition via Embedding into Graph Products. In C. Paul and M. Pilipczuk, editors, *13th International Symposium on Parameterized and Exact Computation (IPEC 2018)*, volume 115 of *Leibniz International*

Proceedings in Informatics (LIPIcs), pages 16:1–16:14, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[46] T. Evans, W. Barth, J. Browne, R. DeLeon, T. Furlani, S. Gallo, M. Jones, and A. Patra. Comprehensive resource use monitoring for HPC systems with TACC stats. In *First Int'l Workshop on HPC User Support Tools*, pages 13–21. IEEE Press, 2014.

[47] T. Evans, J. Browne, and W. Barth. Understanding application and system performance through system-wide monitoring. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1702–1710. IEEE, 2016.

[48] Facebook. Rocksdb: A persistent key-value store for flash and ram storage. https://github.com/facebook/rocksdb, 2020.

[49] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 325–334, New York, NY, USA, 2011. ACM.

[50] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.

[51] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, page 237–246, New York, NY, USA, 2008. Association for Computing Machinery.

[52] S. Feldman, C. Valera-Leon, and D. Dechev. An efficient wait-free vector. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):654–667, March 2016.

[53] S. Feldman, D. Zhang, D. Dechev, and J. Brandt. Extending LDMS to enable performance monitoring in multi-core applications. In *IEEE Int'l Conference on Cluster Computing*, pages 717–720. IEEE, 2015.

[54] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[55] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25(2):5, 2007.

[56] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. *SIGPLAN Not.*, 53(1):28–40, Feb. 2018.

[57] T. Gamblin, M. Schulz, B. R. de Supinski, F. Wolf, B. J. Wylie, et al. Reconciling sampling and direct instrumentation for unintrusive call-path profiling of mpi programs. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 640–651. IEEE, 2011.

[58] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[59] Google. Leveldb: a fast key-value storage library. https://github.com/google/leveldb, 2020.

[60] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[61] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

[62] A. Hall, O. Bachmann, R. Büssow, S. Gănceanu, and M. Nunkesser. Processing a trillion cells per mouse click. *Proceedings of the VLDB Endowment*, 5(11):1436–1446, 2012.

[63] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In J. Welch, editor, *Distributed Computing*, pages 300–314, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[64] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In D. Malkhi, editor, *Distributed Computing*, pages 265–279, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[65] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215, New York, NY, USA, 2004. ACM.

[66] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[67] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, Nov. 1993.

[68] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM.

[69] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[70] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[71] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 31, New York, NY, USA, 2016. ACM, ACM.

[72] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical report, Sandia National Laboratories, 2009.

[73] J. L. Hintze and R. D. Nelson. Violin plots: a box plot-density trace synergism. *The American Statistician*, 52(2):181–184, 1998.

[74] T. Hoefler and R. Belli. Scientific benchmarking of parallel computing systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[75] T. Ilsche, J. Schuchart, R. Schöne, and D. Hackenberg. Combining instrumentation and sampling for trace-based application performance analysis. In *Tools for High Performance Computing 2014*, pages 123–136. Springer, 2015.

[76] Intel. Persistent memory programming. http://pmem.io, Aug. 2016.

[77] Intel. Intel optane dc persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2020.

[78] R. Izadpanah, B. A. Allan, D. Dechev, and J. Brandt. Production application performance data streaming for system monitoring. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 4(2):8:1–8:25, Apr. 2019.

[79] R. Izadpanah, S. Feldman, and D. Dechev. A methodology for performance analysis of non-blocking algorithms using hardware and software metrics. In *Int'l Symposium on Real-Time Distributed Computing (ISORC)*, pages 43–52. IEEE, 2016.

[80] R. Izadpanah, N. Naksinehaboon, J. Brandt, A. Gentile, and D. Dechev. Integrating low-latency analysis into hpc system monitoring. In *Proceedings of the 47th International Conference on Parallel Processing*, page 5. ACM, 2018.

[81] J. Izraelevitz, T. Kelly, and A. Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 427–442, New York, NY, USA, 2016. ACM.

[82] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In C. Gavoille and D. Ilcinkas, editors, *Distributed Computing*, pages 313–327, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[83] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic performance measurements of the intel optane dc persistent memory module, 2019.

[84] J. Jeffers and J. Reinders. *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*. Morgan Kaufmann, 2015.

[85] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. Atom: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372, Washington, DC, USA, Feb 2017. IEEE Computer Society.

[86] I. Kadochnikov, N. Balashov, A. Baranov, I. Pelevanyuk, N. Kutovskiy, V. Korenkov, and A. Nechaevskiy. Evaluation of monitoring systems for metric collection in intelligent cloud scheduling. In *CEUR Workshop proceedings*, volume 1787, pages 279–283, 2016.

[87] N. D. Kallimanis and E. Kanellou. Wait-Free Concurrent Graph Objects with Dynamic Traversals. In E. Anceaume, C. Cachin, and M. Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*, volume 46 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–17, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[88] E. Karrels and E. Lusk. Performance analysis of mpi programs. In *Proceedings of the Workshop on Environments and Tools For Parallel Scientific Computing*, pages 195–200, 1994.

[89] T. Kelly. Persistent memory programming on conventional hardware. *Queue*, 17(4), Aug. 2019.

[90] M. Kerrisk. *The Linux programming interface*. No Starch Press, 2010.

[91] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008.

[92] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer, 2012.

[93] P. LaBorde. *Practical Dynamic Transactional Data Structures*. PhD thesis, University of Central Florida Orlando, Florida, 2018.

[94] P. LaBorde, L. Lebanoff, C. Peterson, D. Zhang, and D. Dechev. Wait-free dynamic transactions for linked data structures. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'19, pages 41–50, New York, NY, USA, 2019. Association for Computing Machinery.

[95] K. Lamar, C. Peterson, and D. Dechev. Lock-free transactional vector. In *The 11th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '20, New York, NY, USA, 2020. ACM.

[96] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 257–270, Santa Clara, CA, 2017. USENIX Association.

[97] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery.

[98] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270, Washington, DC, USA, Oct 2018. IEEE.

[99] Y. Liu. *Crafting Concurrent Data Structures*. PhD thesis, Lehigh University, 2015.

[100] Los Alamos National Laboratory. Trinity. https://www.lanl.gov/projects/trinity, 2018.

[101] J. Lothian et al. Synthetic graph generation for data-intensive HPC benchmarking: Background and framework. Technical report, Oak Ridge National Laboratory, 2013.

[102] A. Marathe, H. Gahvari, J.-S. Yeom, and A. Bhatele. Libpowermon: A lightweight profiling framework to profile program context and system-level metrics. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1132–1141. IEEE, 2016.

[103] V. Marathe, A. Mishra, A. Trivedi, Y. Huang, F. Zaghloul, S. Kashyap, M. Seltzer, T. Harris, S. Byan, B. Bridge, and D. Dice. Persistent memory transactions, 2018.

[104] S. A. J. Marsden, L. S. S. Wiggins, L. Glass, R. Kohn, and S. Sastry. *Interdisciplinary Applied Mathematics*, volume 3. Springer, 1993.

[105] M. L. Massie et al. The Ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[106] A. Memaripour, J. Izraelevitz, and S. Swanson. Pronto: Easy and fast persistence for volatile data structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 789–806, New York, NY, USA, 2020. Association for Computing Machinery.

[107] A. Memaripour and S. Swanson. Breeze: User-level access to non-volatile main memories for legacy software. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 413–422, 2018.

[108] B. Mohr, D. Brown, and A. Malony. Tau: A portable parallel program analysis environment for c++. In *Parallel Processing: CONPAR 94—VAPP VI*, pages 29–40. Springer, 1994.

[109] National Center for Supercomputing Applications. Blue Waters. https://bluewaters.ncsa.illinois.edu, 2019.

[110] F. Nawab, J. Izraelevitz, T. Kelly, C. B. M. III, D. R. Chakrabarti, and M. L. Scott. Dalí: A Periodically Persistent Hash Map. In A. W. Richa, editor, *31st International Symposium*

on Distributed Computing (DISC 2017), volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[111] OASIS. AMQP. https://www.amqp.org, 2018.

[112] T. Oetiker. RRDTool, 2014.

[113] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386, New York, NY, USA, 2016. ACM.

[114] C. H. Papadimitriou. Serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[115] T. Pelkonen et al. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[116] pmem team. Pmdk: Persistent memory development kit. https://github.com/pmem/pmdk/, 2020.

[117] pmem team. pmemkv: Key/value datastore for persistent memory. https://github.com/pmem/pmemkv, 2020.

[118] pmem team. pmemkv-tools: Optional tools and utilities for pmemkv. https://github.com/pmem/pmemkv-tools, 2020.

[119] A. Porterfield, S. Olivier, S. Bhalachandra, and J. Prins. Power measurement and concurrency throttling for energy reduction in OpenMP programs. In *IEEE Int'l Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 884–891. IEEE, 2013.

[120] Prometheus authors. Prometheus - monitoring system and time series database. https://prometheus.io, 2018.

[121] P. Ramalhete, A. Correia, P. Felber, and N. Cohen. Onefile: A wait-free persistent transactional memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 151–163, Washington, DC, USA, June 2019. IEEE Computer Society.

[122] J. Reams. Extensible Monitoring with Nagios and Messaging Middleware. In *Strategies, Tools, and Techniques: Proceedings of the 26th Large Installation System Administration Conference, LISA*, pages 153–162, 2012.

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

[124] P. C. Roth, D. C. Arnold, and B. P. Miller. Mrnet: A software-based multicast/reduction network for scalable tools. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 21–21. IEEE, 2003.

[125] S. M. Sarwar and R. M. Koretsky. *UNIX: the textbook*. CRC Press, 2016.

[126] W. J. Schroeder, B. Lorensen, and K. Martin. *The visualization toolkit: an object-oriented approach to 3D graphics*. Kitware, 2004.

[127] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford. Open—speedshop: An open source infrastructure for parallel performance analysis. *Scientific Programming*, 16(2-3):105–121, 2008.

[128] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner. Nvc-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proceedings of the 3rd VLDB Workshop*

on In-Memory Data Mangement and Analytics, IMDM '15, pages 4:1–4:8, New York, NY, USA, 2015. ACM.

[129] SciPy. SciPy - scientific computing tools for python, 2018.

[130] H. Servat, G. Llort, J. Giménez, and J. Labarta. Detailed performance analysis using coarse grain sampling. In *European Conference on Parallel Processing*, pages 185–198. Springer, 2009.

[131] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 51–64, New York, NY, USA, 2011. ACM.

[132] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, Feb 1997.

[133] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin. Proteus: A flexible and fast software supported hardware logging approach for nvm. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 178–190, Washington, DC, USA, Oct 2017. IEEE Computer Society.

[134] D. Skinner. Performance monitoring of parallel scientific applications. *Lawrence Berkeley National Laboratory*, 2005.

[135] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. *SIGPLAN Not.*, 44(4):141–150, Feb. 2009.

[136] A. Spiegelman, G. Golan-Gueta, and I. Keidar. Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 682–696, New York, NY, USA, 2016. ACM.

[137] T. Strandell et al. *Open source database systems: Systems study, performance and scalability*. VDM Publishing, University of Helsinki, 2010.

[138] S. S. Vazhkudai, R. Miller, D. Tiwari, C. Zimmer, F. Wang, S. Oral, R. Gunasekaran, and D. Steinert. Guide: a scalable information directory service to collect, federate, and analyze logs for operational insights into a leadership hpc facility. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 45. ACM, 2017.

[139] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[140] J. Vetter and C. Chambreau. mpip: Lightweight, scalable mpi profiling, 2005.

[141] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible File-System Interfaces to Storage-Class Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, 2014.

[142] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *SIGPLAN Not.*, 47(4):91–104, Mar. 2011.

[143] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Pro-*

gramming Languages and Operating Systems, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[144] T. Wang, J. Levandoski, and P. Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472, Washington, DC, USA, April 2018. IEEE Computer Society.

[145] T. Wang, S. Simbasivam, Y. Solihin, and J. Tuck. Hardware Supported Persistent Object Translation. In *Proc. of the International Symposium on Microarchitecture*, 2017.

[146] J. S. Ward and A. Barker. Observing the clouds: a survey and taxonomy of cloud monitoring. *Journal of Cloud Computing*, 3(1):24, 2014.

[147] C. Wood, S. Sane, D. Ellsworth, A. Gimenez, K. Huck, T. Gamblin, and A. Malony. A scalable observation system for introspection and in situ analytics. In *Extreme-Scale Programming Tools (ESPT), Workshop on*, pages 42–49. IEEE, 2016.

[148] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, Santa Clara, CA, 2016. USENIX Association.

[149] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. D. Silva, S. Swanson, and A. Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[150] Y. Xu, Y. Solihin, and X. Shen. MERR: improving security of persistent memory objects via efficient memory exposure reduction and randomization. In J. R. Larus, L. Ceze, and K. Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020 [ASPLOS 2020 was canceled because of COVID-19]*, pages 987–1000. ACM, 2020.

[151] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, Santa Clara, CA, 2015. USENIX Association.

[152] P. Zardoshti, T. Zhou, Y. Liu, and M. Spear. Optimizing persistent memory transactions. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–231, Washington, DC, USA, Sep. 2019. IEEE Computer Society.

[153] D. Zhang. *High-performance composable transactional data structures*. PhD thesis, University of Central Florida Orlando, Florida, 2016.

[154] D. Zhang and D. Dechev. An efficient lock-free logarithmic search data structure based on multi-dimensional list. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 281–292, Washington, DC, USA, June 2016. IEEE Computer Society.

[155] D. Zhang and D. Dechev. A lock-free priority queue design based on multi-dimensional linked lists. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):613–626, March 2016.

[156] D. Zhang and D. Dechev. Lock-free transactions without rollbacks for linked data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 325–336, New York, NY, USA, 2016. ACM.

[157] D. Zhang, P. Laborde, L. Lebanoff, and D. Dechev. Lock-free transactional transformation for linked data structures. *ACM Trans. Parallel Comput.*, 5(1), June 2018.

[158] Y. Zuriel, M. Friedman, G. Sheffi, N. Cohen, and E. Petrank. Efficient lock-free durable sets. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.