SAND2018-6408C

Integrating Low-latency Analysis into HPC System Monitoring

Ramin Izadpanah
University of Central Florida
Department of Computer Science
Orlando, Florida
izadpana@cs.ucf.edu

Nichamon Naksinehaboon
Open Grid Computing
Austin, Texas
nichamon@opengridcomputing.com

Jim Brandt
Sandia National Laboratories
Albuquerque, New Mexico
brandt@sandia.gov

Ann Gentile
Sandia National Laboratories
Albuquerque, New Mexico
gentile@sandia.gov

Damian Dechev
University of Central Florida
Department of Computer Science
Orlando, Florida
dechev@cs.ucf.edu

ABSTRACT

The growth of High Performance Computer (HPC) systems increases the complexity with respect to understanding resource utilization, system management, and performance issues. While raw performance data is increasingly exposed at the component level, the usefulness of the data is dependent on the ability to do meaningful analysis on actionable timescales. However, current system monitoring infrastructures largely focus on data collection, with analysis performed off-system in post-processing mode. This increases the time required to provide analysis and feedback to a variety of consumers.

In this work, we enhance the architecture of a monitoring system used on large-scale computational platforms, to integrate streaming analysis capabilities at arbitrary locations within its data collection, transport, and aggregation facilities. We leverage the flexible communication topology of the monitoring system to enable placement of transformations based on overhead concerns, while still enabling low-latency exposure on node. Our design internally supports and exposes the raw and transformed data uniformly for both node level and off-system consumers. We show the viability of our implementation for a case with production-relevance: run-time determination of the relative per-node files system demands.

CCS CONCEPTS

• **General and reference** → **Measurement**; **Performance**; *Evaluation*; *Experimentation*; Empirical studies; Metrics; • **Computing methodologies** → *Massively parallel and high-performance simulations*; *Simulation evaluation*; Modeling methodologies; • **Software and its engineering** → *Software performance*; *Software maintenance tools*;

KEYWORDS

Application and System Monitoring, Performance Data Processing, Low-latency Analysis

ACM Reference Format:
Ramin Izadpanah, Nichamon Naksinehaboon, Jim Brandt, Ann Gentile, and Damian Dechev. 2018. Integrating Low-latency Analysis into HPC System Monitoring. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225086

1 INTRODUCTION

Comprehensive understanding of resource utilization and system state is required to understand and mitigate issues in HPC application and system performance. 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, this data can be many TB/day [7, 12].

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

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225086

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.

Here, 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) [2]. LDMS is used in monitoring large-scale HPC systems such as NCSA's Blue Waters [23] Cray XE/XK system with 27,648 nodes. Data collection intervals are order 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 [18], 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.

This paper makes the following contributions:

- We present an approach to do chains of streaming analysis within the monitoring architecture, as opposed to post-processing.
- We incorporate this capability into a scalable monitoring system.
- We leverage the arbitrary communication topologies of the monitoring system, including bi-directional communications, to minimize the impact of the computations while supporting access to the transformed data.
- We provide a uniform interface to both the raw and transformed data to increase flexibility and facilitate the data utilization by consumers.

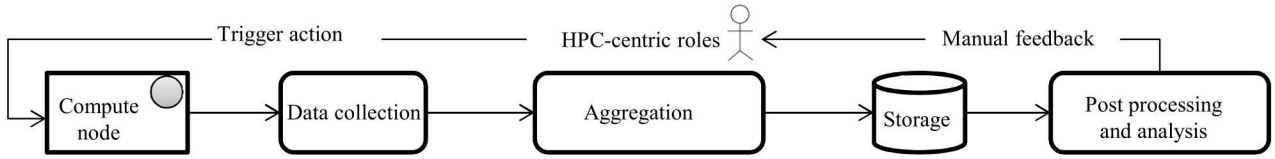
The rest of this paper proceeds as follows. We provide some use cases and motivations for this work in Section 2. In Section 3, we present the background on LDMS required for understanding the implementation here. In Section 4, we describe the design and challenges introduced. In Section 5, we present the experimental evaluations and application of our work to a production-relevant example. In Section 6, we discuss related work. Finally, we conclude in Section 7.

2 MOTIVATION

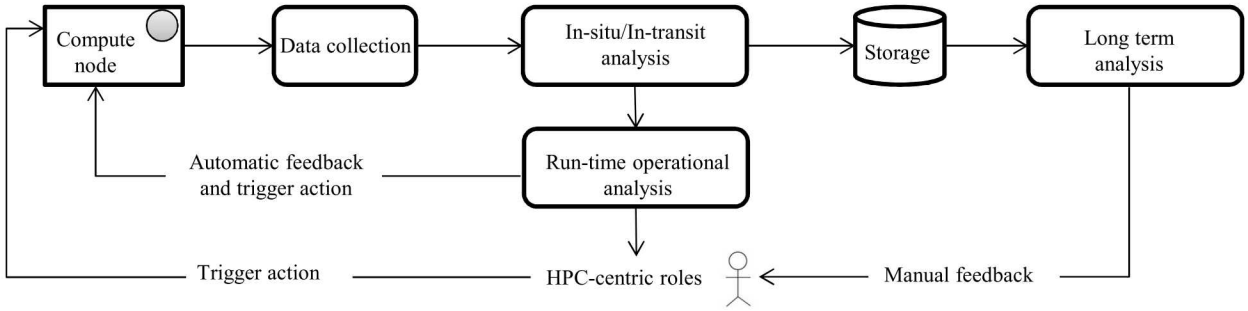
Large-scale HPC systems utilize a variety of resources (e.g., network and file systems) that are shared by both processes of a parallel application and those of other concurrently running applications. Contention for these resources can create congestion that can severely impact application performance 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 1 provides a comparison between typical resource utilization and performance analysis in HPC monitoring systems based on post-processing (Figure 1a) versus our approach based on integrating in-situ processing on the node or in-transit processing at data aggregation points (Figure 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 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 adminstrators.

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



(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 1a.

Figure 1: Differences in the processes of performance analysis: post-processing vs integrated analysis.

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 [27], and thread contention analysis has helped to tune non-blocking algorithms [20]. 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 [3].

In the remainder of this paper, 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 LDMS MONITORING FRAMEWORK BACKGROUND

Our approach leverages and builds upon LDMS's data collection, transport, and aggregation capabilities. 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 aggregate data from other LDMS daemons over various transports, including Infiniband, iWarp, and Ethernet, in arbitrary communication topologies. Thus, multiple aggregation points can be configured to pull data from disjoint and overlapping sources, including other aggregators, as shown in Figure 2. 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.

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, (e.g., a named pipe to syslog [2]) 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.

[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.

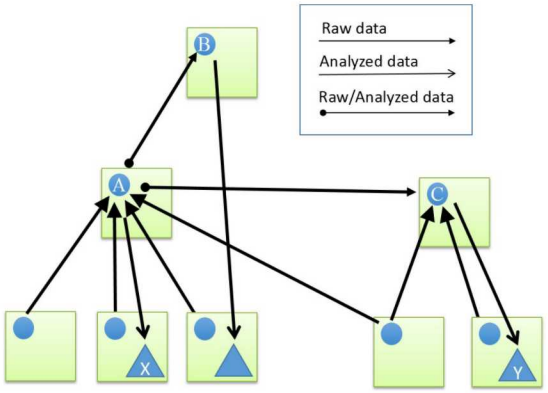


Figure 2: LDMS supports arbitrary communication topologies. Green squares indicate nodes; blue circles indicate LDMS daemons; blue triangles indicate applications. Arrows indicate the direction of aggregation and data accessing by applications. For example, A can aggregate metric sets from 4 of LDMS 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.

```
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 3: Output from the `ldms_ls` command, showing part of a metric set produced by a sampler plugin written to collect a variety of metrics from a Cray XE/XK system. In Section 5.2, we utilize transform plugins to perform some analyses using the metrics related to the Lustre file system.

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 3, 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 the work presented in this paper, in LDMS, some limited streaming computations and data transformations have been performed using store plugins called *function store* plugins. These plugins [6] 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 this work, we overcome this limitation by designing a flexible approach to support streaming analysis. 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.

4 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.

4.1 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 5. 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.

4.1.1 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 5.2, 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 4.2.

4.1.2 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 `ldms_ls` output. For example, in Figure 6, 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.

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

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

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

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

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

- **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.

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

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

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

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

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

- **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 2. In Section 5.2, we show how we utilize transform plugins for performing an analysis on the Lustre file system, based on the raw data displayed in Figure 3.

4.2 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:

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.

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 4.1.2, 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.

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.

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 3 includes Lustre, network, GPU, and CPU data. This design choice was motivated by the desire to have a single timestamp associated with all metrics [7], 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.

5 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.

5.1 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 [2, 12, 13, 23]. 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 `/proc/meminfo` and `/proc/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 4. Each experiment runs for 30 minutes with a sampling interval of 1 second. Baseline experiments were intended

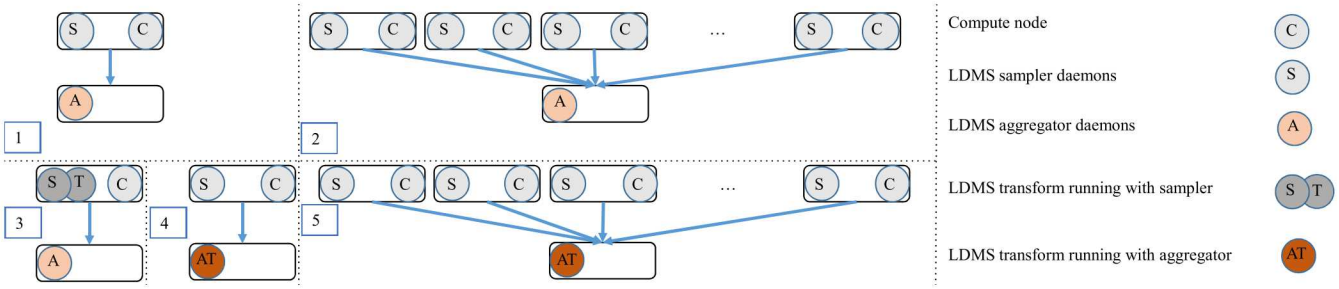


Figure 4: Experimental setup for measuring overhead. Samplers are collecting two metrics set on each compute node. Aggregators are pulling data from either one sampler, specified by experiments 1,3, and 4, or ten samplers, specified by experiments 2 and 5. The experiments specified by sections 1 and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

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.

Table 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.

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

Table 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.

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.

5.2 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 5 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 3.

The first layer of transform plugins shown in Figure 5 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

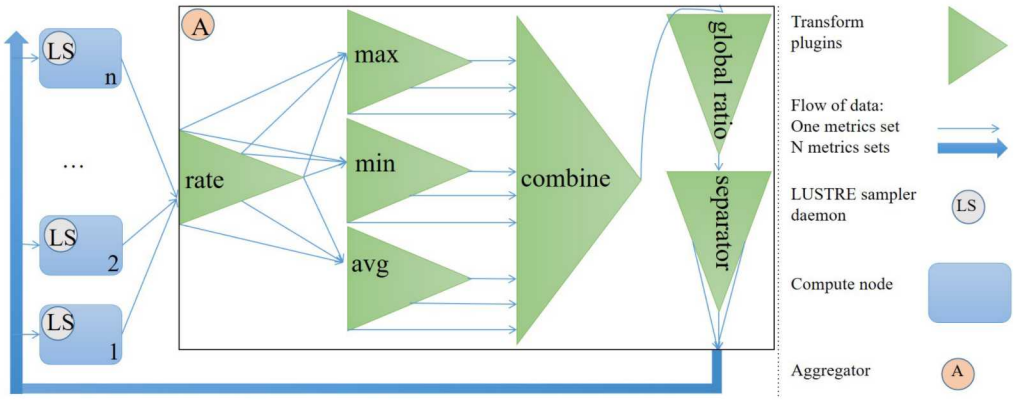


Figure 5: Transform sequence and positions in the computation of the transformed Lustre metrics.

extracts metrics from it to output multiple metric sets, one for each compute node. Finally, compute nodes receive this information.

Figure 6 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 [14]. 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.

6 RELATED WORK

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 [22] and Nagios [28] are invoked periodically on the nodes with the data typically aggregated to a central location. Ganglia is designed to use rrdtool [25] 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 [15] 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 [10] 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 [16] 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 [17] 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 [32] 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 [8] 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 [29] and AMQP [24] could be used for the transport part. However, all the capabilities for data collection, analysis, and exposure of both raw

Figure 6: 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.

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 [5]. 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][11][26][19][9]). 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 [30]. 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 [31] tools are being applied to computations in HPC analysis (e.g., [21]) as are no-SQL databases (e.g., [4]) 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.

7 CONCLUSIONS

In this work, we have designed 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 initial 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. Future work includes hardening of the transform support design. Ultimately we seek to perform run-time global analyses of system and resource state (e.g., network contention measures) and can use that information to invoke more optimal resource usage as a result.

REFERENCES

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

[2] 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. 2014. 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)*. IEEE, 154–165.

[3] G. H. Bauer et al. 2016. Dynamic Model Specific Register (MSR) Data Collection as a System Service. In *Cray Users Group (CUG)*.

[4] S. Benedict, R. Rejitha, and C. Bright. 2014. Energy Consumption Analysis of HPC Applications Using NoSQL Database Feature of EnergyAnalyzer. In *International Conference on Intelligent Cloud Computing*. Springer, 103–118.

[5] S. Bohm, C. Englemann, and S Scott. 2010. 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.

[6] J. Brandt, E. Froese, A. Gentile, L. Kaplan, B. Allan, and E. Walsh. 2016. Network Performance Counter Monitoring and Analysis on the Cray XC Platform. In *Cray Users Group (CUG)*.

[7] J. Brandt, A. Gentile, M. Showerman, J. Enos, J. Fullop, and G. Bauer. 2016. Large-scale Persistent Numerical Data Source Monitoring System Experiences. In *IEEE Int'l Parallel and Distributed Processing Symposium Worksshops (IPDPSW)*. IEEE.

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

[9] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. 2003. Scalable distributed stream processing. In *In CIDR*. Citeseer.

[10] Collectl. 2014. Collectl. (2014). http://collectl.sourceforge.net

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

[12] A. DeConinck et al. 2016. Design and Implementation of a Scalable HPC Monitoring System for Trinity. In *Cray Users Group (CUG)*.

[13] A. DeConinck et al. 2017. Runtime Collection and Analysis of System Metrics for Production Monitoring of Trinity Phase II. In *Cray Users Group (CUG)*.

[14] B. Dong, X. Li, Q. Wu, L. Xiao, and L. Ruan. 2012. A dynamic and adaptive load balancing strategy for parallel file system with large-scale I/O servers. *J. Parallel and Distrib. Comput.* 72, 10 (2012), 1254–1268.

[15] Elasticsearch, BV. 2018. The Elastic Stack. (2018). https://www.elastic.co/webinars/introduction-elk-stack

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

[17] T. Evans, J. Browne, and W. Barth. 2016. Understanding Application and System Performance Through System-Wide Monitoring. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 1702–1710.

[18] S. Feldman, D. Zhang, D. Dechev, and J. Brandt. 2015. Extending LDMS to Enable Performance Monitoring in Multi-core Applications. In *IEEE Int'l Conference on Cluster Computing*. IEEE, 717–720.

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

[20] R. Izadpanah, S. Feldman, and D. Dechev. 2016. A Methodology for Performance Analysis of Non-blocking Algorithms Using Hardware and Software Metrics. In *Int'l Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 43–52.

[21] J. Lothian et al. 2013. Synthetic Graph Generation for Data-Intensive HPC Benchmarking: Background and Framework. (2013).

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

[23] National Center for Supercomputing Applications. 2018. Blue Waters. https://bluewaters.ncsa.illinois.edu. (2018).

[24] OASIS. 2018. AMQP. https://www.amqp.org. (2018).

[25] T Oetiker. 2014. RRDTool. (2014). http:/rrdtool.org

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

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

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

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

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

[31] SciPy. 2018. SciPy - Scientific computing tools for python. (2018). https://www.scipy.org

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