A Scalable Observation System for Introspection and In Situ Analytics

Chad Wood*, Sudhanshu Sane*, Daniel Ellsworth*, Alfredo Gimenez†, Kevin Huck*, Todd Gamblin†, Allen Malony*

* Department of Computer and Information Science
University of Oregon
Eugene, OR United States
Email: {cdw,ssane,dellswor,khuck,malony}@cs.uoregon.edu

† Lawrence Livermore National Laboratory
Livermore, CA United States
Email: {gimenez1,gamblin2}@llnl.gov

***Abstract*—SOS is a new model for the online in situ characterization and analysis of complex high-performance computing applications. SOS employs a data framework with distributed information management and structured query and access capabilities. The primary design objectives of SOS are flexibility, scalability, and programmability. SOS provides a complete framework that can be configured with and used directly by an application, allowing for a detailed workflow analysis of scientific applications. This paper describes the model of SOS and the experiments used to validate and explore the performance characteristics of its implementation in SOSflow. Experimental results demonstrate that SOS is capable of observation, introspection, feedback and control of complex high-performance applications, and that it has desirable scaling properties.**

***Index Terms*—hpc; exascale; in situ; performance; monitoring; introspection; monalytics; scientific workflow; sos; sosflow;**

I. Introduction

Modern clusters for parallel computing are complex environments. High-performance applications that run on modern clusters do so often with little insight about their or the system's behavior. This is not to say that information is unavailable. After all, sophisticated parallel measurement systems can capture performance and power data for characterization, analysis, and tuning purposes, but the infrastructure for observation of these systems is not intended for general use. Rather, it is specialized for certain types of performance information and typically does not allow online processing. Other information sources of interest might include the operating system (OS), network hardware, runtime services, or the parallel application itself. Our general interest is in parallel application monitoring: the observation, introspection, and possible adaptation of an application during its execution.

Application monitoring has several requirements. It is important to have a flexible means to gather information from different sources on each node — primarily the application and system environment. Additionally, for the gathered information to be processed online, analysis will need to be enabled in situ with the application [3]. Query and control interfaces are required to facilitate an active application feedback process. The analysis performed can be used to give feedback to both the application, the operating environment, and performance tools. There exists no general purpose infrastructure that can be programmed, configured, and launched with the application to provide the integrated observation, introspection, and adaptation support required.

This paper presents the *Scalable Observation System (SOS)* for integrated application monitoring. A working implementation of SOS is contributed as a part of this research effort, *SOSflow*. The SOSflow platform demonstrates all of the essential characteristics of the SOS model, showing the scalability and flexibility inherent to SOS with its support for observation, introspection, feedback, and control of scientific workflows. The SOS design employs a data model with distributed information management and structured query and access. A dynamic database architecture is used in SOS to support aggregation of streaming observations from multiple sources. Interfaces are provided for in situ analytics to acquire information and then send back results to application actuators and performance tools. SOS launches with the application, runs along side it, and can acquire its own resources for scalable data collection and processing.

A. Scientific Workflows

Scientific workflows feature two or more components that are coupled together, operating over shared information to produce a cumulative result. These components can be instantiated as lightweight threads belonging to a single process, or they may execute concurrently as independent processes. Components of workflows can be functionally isolated from each other or synchronously coupled and co-dependent. Some workflows can be run on a single node, while others are typically distributed across thousands of nodes. Additionally, parts of workflows may even be dynamically instantiated and terminated. The computational profile of a workflow can change between invocations or even during the course of one execution.

B. Multiple Perspectives

Application state and events can be sent to SOS from within the application at any point during its execution. Developers can instrument their programs to be efficiently self-reporting

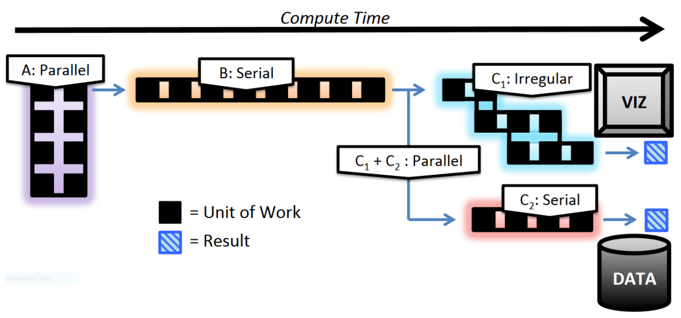


Fig. 1. Applications Coupled Together Into a Workflow

the data that is relevant to their overall performance, such as progress through specific phases of a simulation.

Application performance can be dramatically impacted by changes in *the state of the operating environment* that is hosting it. The effects of contention for shared resources by multiple concurrent tasks can be discovered when the events of concurrent tasks are fixed into a common context for reasoning about their individual and combined performance. SOS's distributed in situ design is well-suited for capturing perspective of the global state of a machine. By co-locating the observation system with the workflow components that are observed, SOS improves the fidelity of system performance data without requiring the costly delays of synchronization or congesting the shared network and filesystem resources in use by applications.

Many existing *performance tools can provide useful observations* at runtime of applications, libraries, and the system context. The low-level timers, counters, and machine-level data points provided by specialized performance tools can be a valuable addition to the higher-level application and system data.

C. Motivation

Observing and reasoning about the performance of workflows on *exascale computational platforms* presents new challenges. Exascale systems will be capable of more than a billion billion calculations per second, a factor of between 50 to 100 times faster than present day machines. The physical scale and complexity of exascale machines is expected to grow by similar factors as its computational speed, motivating a model that can scale to the same extent.

II. Related Work

Traditionally, HPC research into enhancing performance has been focused on the low-level efficiency of one application, library, or a particular machine.

Tools like TAU [6] are able to bring HPC developers a closer look into to their codes and hardware, gathering low-level performance data and aggregating it for integrated analysis after an application concludes. Low-level metrics can help identify performance bottlenecks, and are naturally suited for non-production or offline episodic performance analysis of individual workflow components. Such deep instrumentation is necessarily invasive and can dictate rather than capture the observed performance of the instrumented application when the application is running at scale or required to engage in significant amounts of interactivity. SOS provides a model that can accept low-level information such as what TAU collects, while also operating over light-weight higher-level information suitable for online operation during production runs.

Focused on the needs of large scale data centers, Monalytics [8] demonstrated the utility of combining monitoring and analytics to rapidly detect and respond to complex events. SOS takes a similar approach but adopts a general purpose data model, runtime adaptivity, application configurability, and support for the integration of heterogenous components for such purposes as analytics or visualization.

Falcon [5] proposed a model for online monitoring and steering of large-scale parallel programs. Where Falcon depended on an application-specific monitoring system that was tightly integrated with application steering logic and data visualizations, SOS proposes a loosely-coupled infrastructure that does not limit the nature or purpose of the information it processes.

WOWMON [11] presented a solution for online monitoring and analytics of scientific workflows, but imposed several limitations and lacked generality, particularly with respect to how it interfaced with workflow components, types of data it could collect and use, and its server for data management and analytics.

Online distributed monitoring and aggregation of information is provided by the DIMVHCM [10] model, but it principally services performance understanding through visualization tools rather than the holistic workflow applications and runtime environment. DIMVHCM provides only limited support for in situ query of information.

Cluster monitoring systems like Ganglia [9] or Nagios [7] collect and process data about the performance and health of cluster-wide resources, but do not provide sufficient fidelity to capture the complex interplay between applications competing for shared resources. In contrast, the Lightweight Distributed Metric Service [2] (LDMS) captures system data continuously to obtain insight into behavioral characteristics of individual applications with respect to their resource utilization. However, neither of these frameworks can be configured with and used directly by an application. Additionally, they do not allow for richly-annotated information to be placed into the system from multiple concurrent data sources per node.

LDMS uses a pull-based interaction model, where a daemon running on nodes will observe and store a set of values at a regular interval. SOS has a hybrid push-pull model that puts users in control of the frequency and amount of information exchanged with the runtime. Further, LDMS is currently limited to working with double-precision floating point values, while SOS allows for the collection of many kinds of information including JSON objects and ”binary large object” (BLOB) data.

TACC Stats [4] facilitates high-level datacenter-wide logging, historical tracking, and exploration of execution statistics for applications. It offers only minimal runtime interactivity and programmability.

The related work mentioned here, and many other performance monitoring tools, are well-implemented, tested, maintained, and regularly used in production and for performance research studies. However, each have deficiencies that render them unsuitable for a scalable, general-purpose, online performance analysis framework.

III. SOS Architectural Model

Multi-component complex scientific workflows provide a focus for the general challenge of distributed online monitoring. Information from a wide variety of sources is relevent to the characterization and optimization of a workflow.

In order to gather run-time information and operate on it, SOS needs to be active in the same environment as the workflow components. This online operation is capable of collecting data from multiple sources and efficiently servicing requests for it. Information captured is distinct and tagged with metadata to enable classification and automated reasoning. SOS aggregates necessary information together online to enable high-level reasoning over the entire monitored workflow.

A. Components of the SOS Model

The SOS Model consists of the following components:

- **Information Producers** : SOS APIs for getting information from different sources to SOS.
- **Information Management** : SOS online information databases/repositiories.
- **Introspection Support** : Online access to the information databases.
- **In Situ Analytics** : Components to perform the online analysis of the information.
- **Feedback System** : SOS APIs for sending feedback information to non-SOS entities.

B. Core Features of SOS

- **Online** : It is necessary to obtain observations at run time to capture features of workflows that emerge from the interactions of the workflow as a whole. Relevant features will emerge given a program's interactions with its problem set, configuration parameters, and execution platform.
- **Scalable** : SOS targets running at exascale on the next generation of HPC hardware. SOS is a distributed runtime platform, with an agent present on each node, using a small fraction of the node's resources. Observation and introspection work is distributed across the observed application's resources proportionally. Performance data aggegation can run concurrently with the workflow.

  Node-level SOS agents transfer information off-node using the high-performance communication infrastructure of the host cluster. SOS supports scalable numbers and topologies of physical aggregation points in order to provide timely runtime query access to the global information space.
- **Global Information Space** : Information gathered from applications, tools, and the operating system is captured and stored into a common context, both on-node and across the entire allocation of nodes. Information in this global space is characterized by —
  - **Multiple Perspectives** - The different perspectives into the performance space of the workflow can be queried to include parts of multiple perspectives, helping to contextualize what is seen from one perspective with what was happening in another.
  - **Time Alignment** - All values captured in SOS are time-stamped, so that events which occured in the same chonological sequence in different parts of the system can be aligned and correlated.
  - **Reusable Collection** - Information gathered into SOS can be used for multiple purposes and be correlated in various ways without having to be gathered multiple times.
  - **Unilateral Publish** - Sources of information need not coordinate with other workflow or SOS components about what to publish, they can submit information and rely on the SOS runtime to decide how best to utilize it. The SOS framework will automatically migrate information where it is needed for analysis while managing the retention of unused information efficiently.

IV. Implementation

The SOSflow library and daemon codes are programmed in C99 and have minimal external dependencies:

- Message Passing Interface (MPI)
- pthreads
- SQLite

SOSflow's core routines allow it to:

- Facilitate online capture of data from many sources.
- Annotate the gathered data with context and meaning.
- Store the captured data on-node in a way that can be searched with dynamic queries in real-time as well as being suitable for aggregation and long-term archival.

SOSflow is divided into several components, central among them:

- **libsos** - Library of common routines for interacting with sosd daemons and SOS data structures.
- **sosd_listener** - Daemon process running on each node.
- **sosd_db** - Daemon process running on dedicated resources that stores data aggregated from one or more in situ daemons.
- **sosa** - Analytics framework for online query of SOS data.

A. Architecture Overview

Data in SOSflow is stored in a "publication handle" (pub) object. This object organizes all of the application context information and value-specific metadata, as well as managing the history of updates to a value pending transmission to a sosd_listener, called *value snapshots*. Every value that is passed through the SOSflow API is preserved and eventually stored in a searchable database, along with any updated metadata such as its timestamp tuples. Prior value snapshots are queued and transmitted along with the most recent update to that value.

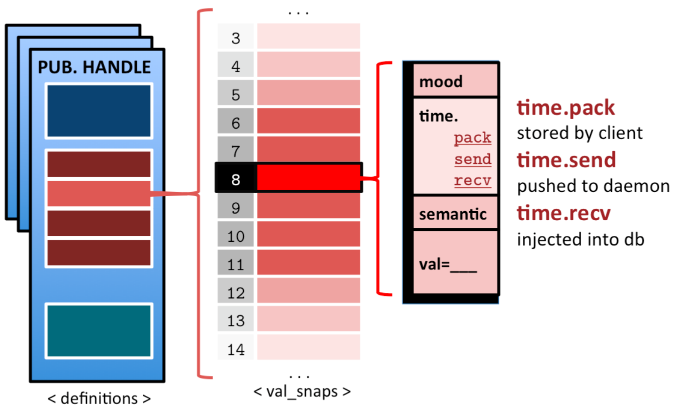


Fig. 2. Complete History of Changing Values is Kept, Including Metadata

SOSflow utilizes different information transport methods and communication patterns where appropriate [1]. Communication between client applications and their on-node daemon takes place over a TCP socket connection. Messages read

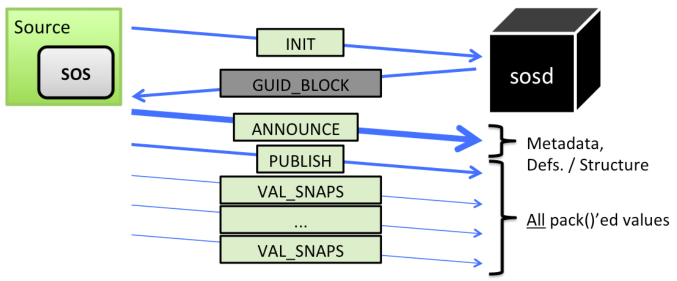


Fig. 3. Client/Daemon Socket Communication Protocol

from the socket are immediately placed in the daemon's asynchronous queues to be processed by a worker thread. The socket is then ready for the next queued message to be received. Messages are first enqueued for storage into an on-node database. The same message is re-enqueued for transmission to an off-node data aggregation target. The SOSflow runtime uses MPI and the high-speed interconnect network of the HPC machine when transmitting information off-node. SOSflow does not participate in the MPI communicator[s] of the applications that it is monitoring, so no special integration programming is required of application developers who already use MPI or sockets in their code.

B. Library: libsos

Applications that make direct use of SOSflow through its API are called *clients*. Clients must link in the libsos library which provides them with all of the data structures and routines essential for interacting with the SOSflow runtime platform. The library routines are thread-safe, and no process-wide state is maintained within the library, allowing application components to interact with SOSflow independent of each other.

The primary interaction between a client and SOSflow is through the pub. When a client initializes its SOSflow instance, it communicates with the daemon and obtains a set of global unique ID (GUID) tags. Clients pack values into a pub and they are automatically assigned a GUID. When the client publishes that handle, all values are transmitted to the SOSflow on-node daemon, including the complete history of each value's updates from the last to the present publish call.

All communication functions in the SOSflow client library are handled transparently. Users need only interact with a simple API to define and store values that they can then publish to the daemon as appropriate. The protocols and the codes of the client library are designed to be fast and minimize resource usage, though they will buffer values for the user if they choose to hold them and only transmit to the daemon at intervals.

Communications with the sosd_listener are always initiated by the clients, such as when they explicitly publish their pub. SOSflow clients can voluntarily spawn a light-weight background thread that periodically checks with their local daemon to see if any feedback has been sent for them. This loosely-coupled interactivity allows for run-time feedback to happen independent of an application's schedule for transmitting its information to SOSflow.

C. Daemon: sosd_listener

The sosd daemon is itself an MPI application, and it is launched as a background process in the user space at the start of a job script, before the scientific workflow begins. The daemons first go through a coordination phase where they each participate in an MPI_Allreduce() with all other daemon ranks in order to share their role (DAEMON, DB, or ANALYTICS) and the name of the host they are running on. During the coordination phase, listener daemons select the sosd_db aggregate database that they will target for automatic asynchronous transfer of the data they capture. After initialization, SOSflow does not perform any further collective communications.

D. Database: sosd_db

The open-source SQLite database engine is used by sosd_db for the on-node database. SQLite databases are persistent, lightweight, fast, and flexible, suitable to receive streams of tuple data with very low overhead. SOSflow provides a simple API for interacting with its database to streamline access both on and off-node.

At the time of this writing, SQLite technology is also used for the aggregate databases, though work is ongoing to provide alternatives for aggregation, starting with an interface to the Cassandra database.

E. Analytics: sosa

SOSflow analytics modules are independent programs that are launched and operate alongside the SOSflow run-time. The primary role of the analytics modules is to query the database and produce functional output such as real-time visualizations of performance metrics, feedback to facilitate optimizations, or global resource bound calculation and policy enforcement. The modules can be deployed in a distributed fashion to run on the nodes where the applications are executing, or they can be deployed on dedicated resources and coupled with the aggregate databases for fast queries of the global state. Analytics modules have the ability to make use of the high-speed interconnect of the HPC machine in order to share data amongst themselves.

SOSflow provides an API for client applications to register a callback function with a named trigger handle. Those triggers can be fired off by analytics modules, and arbitrary data structures can be passed to the triggered functions. Triggers may be fired for a specific single process on one node, or for an entire node, or an entire scientific workflow. This capability facilitates the use of SOSflow as a general-purpose observation, introspection, feedback, and control platform.

V. Results

A. Evaluation Platform

All results were obtained by either interrogating the daemon[s] directly to inspect their state or by running queries against the SOSflow databases.

B. Experiment Setup

The experiments performed had the following purposes:

- **Validation** : Demonstrate that the SOSflow model works for a general case.
- **Exploration** : Study the latency and overhead of SOSflow's current research implementation.

The SOSflow implementation is general-purpose and we did not need to tailor it to the deployment environment. The same SOSflow code base was used for each of the experiments. The study was conducted on three machines, the details of which are given below —

1) **ACISS** : The University of Oregon's 128-node compute cluster. Each node has 72 GB of memory and 2x Intel X5650 2.66 GHz 6-core CPUs, providing 12 cores per node. Each node is connected together with a 10GigE ethernet switch.
2) **Cori** : A Cray XC40 supercomputer at the National Energy Research Scientific Computing Center (NERSC). Nodes are equipped with 128 GB of memory and 2x Intel Xeon E5-2698v3 2.30 GHz 16-core CPUs. Cori nodes are connected by a Cray Aries network with Dragonfly topology, that has 5.625 TB/s global bandwidth.
3) **Catalyst** : A Cray CS300 supercomputer at Lawrence Livermore National Laboratory (LLNL). Each of the 324 nodes is outfitted with 128 GB of memory and 2x Intel Xeon E5-2695v2 2.40 GHz 12-core CPUs. Catalyst nodes transport data to each other using a QLogic InfiniBand QDR interconnect.

We simulated workflows using the following —

1) **LULESH with TAU** : An SOSflow-enabled branch of the Tuning and Analysis Utilities program (TAUflow) was created as a part of the SOSflow development work. On Cori, TAUflow was used to instrument the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) code. During the execution of LULESH, a thread in TAUflow would periodically awaken and submit all of TAU's observed performance metrics into the SOSflow system.
2) **Synthetic Workflow** : Synthetic parallel MPI applications were developed that create example workloads for the SOSflow system by publishing values through the API at configurable sizes and rates of injection.

C. Evaluation of SOS Model

This experiment was performed to validate the SOS Model and demonstrate its applicability for the general case of workflow observation. The Cori supercomputer was used to execute a LULESH + TAUflow simulation. Power and memory usage metrics were collected and stored in SOSflow for each node. During the execution of the workflow, a visualization application was launched from outside of the job allocation which connected to SOSflow's online database and was able to query and display graphs of the metrics that SOSflow had gathered.

The LULESH job was run both with and without the presence of SOSflow (all other settings being equal) in order to validate the ability of SOSflow to meet its design goals while being minimally invasive.

D. Evaluation of Latency

Experiments were performed to study the latency of data moving through SOSflow. When a value is published from a client into SOSflow, it enters an asynchronous queue scheme for both database injection and off-node transport to an aggregation target. Latency in this context refers to the amount of time that a value spends in these queues before becoming available for query by analytics modules. To study latency we ran experiments on both ACISS and Catalyst.

Tests run on ACISS were deployed with the Torque job scheduler as MPICH2 MPI applications at scales ranging from 3 to 24 nodes, serving 10 Synthetic Workflow processes per node in all cases. The ACISS battery of runs were tuned as stress tests to ensure that the sosd daemons could operate under reasonably heavy loads. In the 24-node ACISS experiment (Figure 9), SOSflow clients published 72,000,000 double-precision floats with associated metadata during a 90 second window containing three rounds of extremely dense API calls.

Latency tests were performed on LLNL's Catalyst machine at various scales up to 128 nodes, with 8 data sources contributing concurrently on each node in each case. Catalyst's tests measured the latency introduced by sweeping across three different parameters:

- Count of unique values per publish
- Number of publish operations per iteration
- Delay between calls to the publish API

Unlike the ACISS experiments, the Catalyst tests did not attempt to flood the system with data, but rather aimed to observe how slight adjustments in size and rates of value injection would impact the latency of those values.

E. Results

1) SOS Model Validation: SOSflow was able to efficiently process detailed performance information from multiple sources on each node. During the LULESH run, SOSflow's online database successfully serviced queries on-line, and the results were plotted as an animated live view of the performance of the workflow. The cost of using SOSflow was calculated simply as the increase in walltime for LULESH + SOSflow, expressed as a percentage of the walltime of LULESH by itself. The results of these runs are shown in Figure 5.

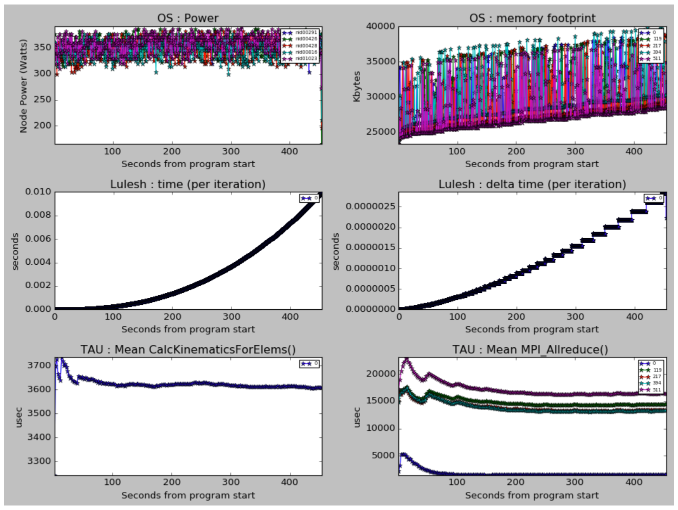
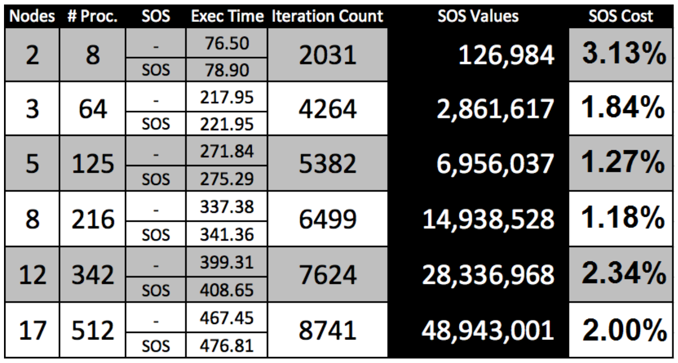


Fig. 4. On-line Workflow Performance Visualization Using SOSflow on Cori. Live View of 512 Processes From Three Perspectives: OS, LULESH, TAU



Nodes	# Proc.	SOS	Exec Time	Iteration Count	SOS Values	SOS Cost
2	8	-	76.50	2031	126,984	3.13%
		SOS	78.90			
3	64	-	217.95	4264	2,861,617	1.84%
		SOS	221.95			
5	125	-	271.84	5382	6,956,037	1.27%
		SOS	275.29			
8	216	-	337.38	6499	14,938,528	1.18%
		SOS	341.36			
12	342	-	399.31	7624	28,336,968	2.34%
		SOS	408.65			
17	512	-	467.45	8741	48,943,001	2.00%
		SOS	476.81			

Fig. 5. Percent Increase in LULESH Running Time When SOSflow is Used

2) Evaluation of Latency: The on-node (Figure 6) and aggregate (Figure 7) results from the largest 128-node runs are presented here. Results from smaller runs are omitted for space, as they show nothing new: "Time in flight" queue latency at smaller scales linearly approached the injection latency figures for a single (on-node) database.

In the 128-node runs, across all configurations, the mean latency observed was 0.3 seconds (and a maximum of 0.7 seconds) for a value, and its full complement of metadata and timestamps, to migrate from one of 1,024 processes to the off-node aggregate data store, passing through multiple asynchronous queues and messaging systems on 128 nodes.

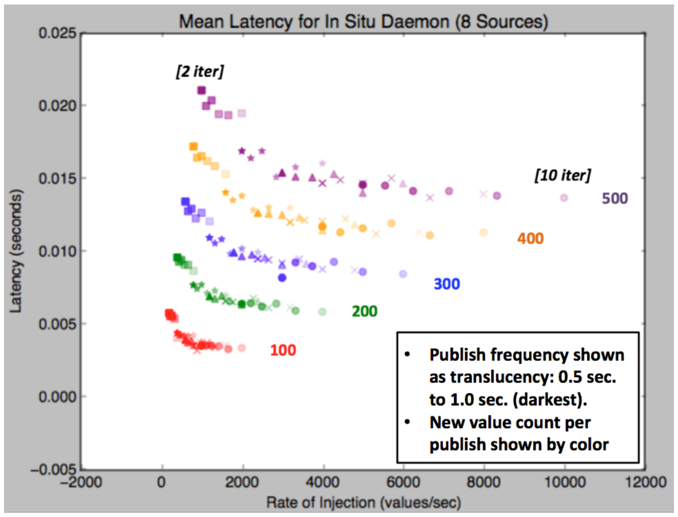


Fig. 6. Average Latency for In Situ Database (128 nodes on Catalyst)

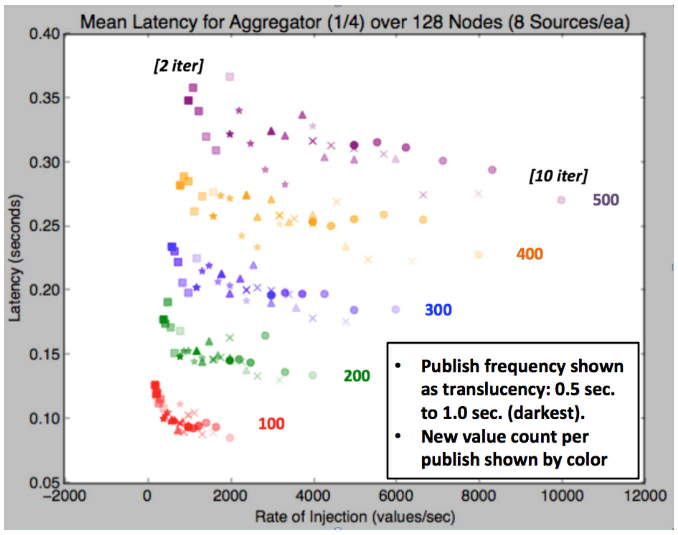


Fig. 7. Average Latency for Aggregate Database (128 nodes on Catalyst)

The in situ and aggregate results in Figures 6 and Figure 7 are promising, given the research version of SOSflow being profiled is not optimized. Exploring the optimal configuration and utilization of SOSflow is left to future research effort.

F. Discussion

Many of the behavioral characteristics of SOSflow are the product of its internal parameters and the configuration of its runtime deployment, rather than products of its data model and algorithms. For now, the effort was made to select reasonable default SOSflow configuration parameters and typical/non-priviledged cluster queues and topologies. Because of the general novelty of the architecture, the results presented here could be considered the *performance baseline* for SOSflow to improve on as the research matures.

Expanding on the direct experimental results, here are some additional experiences and observations about the behavior of SOSflow: —

1) Aggregation Topology: The current version of SOSflow is configured at launch with a set number of aggregator databases. The validation tests on ACISS used 3 sosd_db instances to divide up the workload, while the TAUflow + LULESH experiments on Cori used a single aggregator. The parameter sweeps run on the LLNL Catalyst machine were done with four sosd_db aggregation targets at 128 nodes. Tests on ACISS and Catalyst were exploring the latency of data movement through SOSflow, and so both configurations featured dedicated nodes for sosd_db aggregators to avoid contention with other on-node work. The Cori runs captured a real application's behavior, and was primarily intended to demonstrate the fitness of SOSflow for capturing the performance of a scientific workflow along with meaningful context. Instances of aggregators can be spawned, as many as needed, in order to support the quantity of data being injected from a global perspective. All data sent to SOSflow is tagged with a GUID. This allows for shards of the global information space to be concatenated after the run concludes without collision of identities wiping out distinct references.

The data handling design trade-offs made for SOSflow do not prioritize the minimization of latency, but focus rather on gracefully handling spikes in traffic by growing (and then shrinking) the space inside the asynchronous message queues. After a value is passed to SOSflow, it is guaranteed to find its way into the queryable data stores, and there are timestamps attached to it that capture the moment it was packaged into a publication handle in the client library, the moment it was published to the daemon, and even the moment it was finally spooled out into the database. Once it is in the database, it is trivial to correlate values together based on the moment of their creation, no matter how long the value was sequestered in the asynchronous queues.

During the ACISS stress-tests, values were injected into the SOSflow system faster than they could be spooled from the queues into the database. While every value will eventually be processed and injected into the data store, some values wound up having to wait longer than others as the queue depth increased. The asynchronous queues have thread-sequential FIFO ordering, but because the MPI messages are queued up based on their arrival time, and a batch is handled completely before the next is processed, there is no real-time interleaving of database value injections, they are injected in batches. Near the bottom of the pile of MPI messages, the latency continually increases until that batch is injected. This accounts for the observed saw-tooth pattern of increasing latency seen in Figure 9, which is not seen in Figure 8.

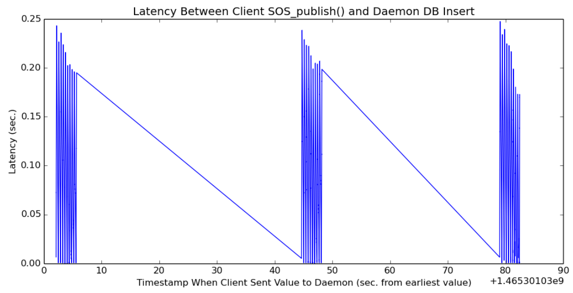


Fig. 8. In Situ Latency (24 nodes on ACISS, 240 Applications)

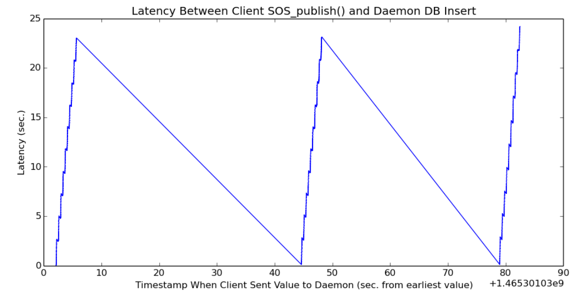


Fig. 9. Aggregate Latency (24 nodes on ACISS, 240 Applications)

2) Time Cost of Publish API: As an accessory to the study of value latency, the length of time that a client application will block inside of an SOSflow API routine was also evaluated. In situ interactions between libsos routines and the daemon are nearly constant time operations regardless of the daemon's workload. Care was taken in the daemon's programming to prioritize rate of message ingestion over immediacy of message processing so that SOSflow API calls would not incur onerous delays for application and tool developers. The constancy of message processing speed is shown in figures 10 and 11, where the round trip time (RTT) of a probe message between a client and the daemon (blue) is projected over a graph of the number of new messages arriving in a sample window (red).

This information was gathered by sending 9000+ probe messages over a 15 minute window, with a single sosd_listener rank processing an average of 724 client messages a second in total, arriving from four different processes on an 8-way Xeon node. The messages from SOS clients contained more than 14.7 GB of data, averaging to 338kB per message. Though there are a few spikes in the probe message RTT visible in Figure 10, they are likely not related to SOSflow at all, as Figure 11 reveals in detail. The RTT holds steady during low and high volume of traffic from the other in situ client processes. The mean RTT for the probe messages was 0.003 seconds, and the maximum RTT was 0.07 seconds.

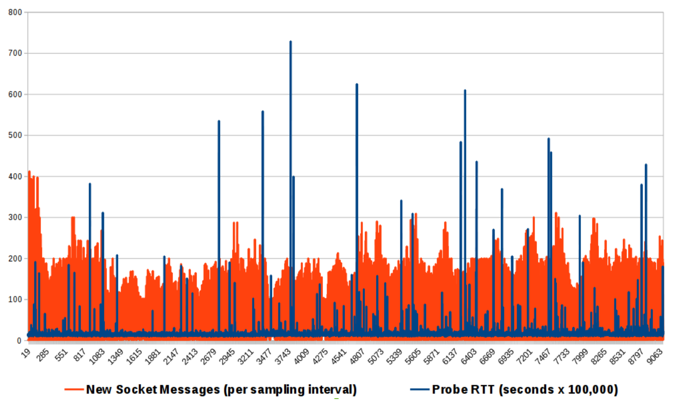


Fig. 10. SOSflow Socket Communication Cost, Projected Over Message Count

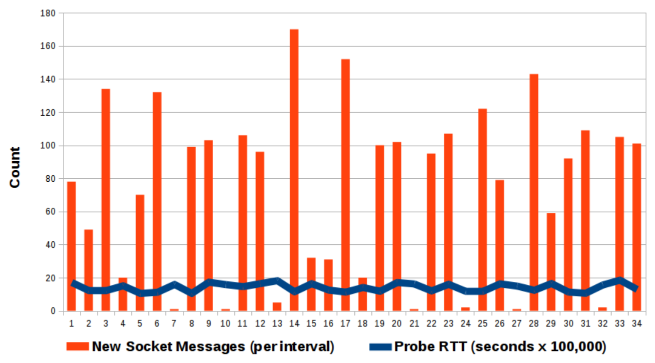


Fig. 11. SOSflow Socket Communication Cost (Detail)

These results show that the cost of making SOSflow API calls is relatively low, and holds constant under changing sosd_listener workload.

VI. Conclusion

The SOS Model presented is online, scalable and supports a global information space. SOS enables online in situ characterization and analysis of complex high-performance computing applications. SOSflow is contributed as an implementation of SOS. SOSflow provides a flexible research platform for investigating the properties of existing and future scientific workflows, supporting both current and future scales of execution. Experimental results demonstrated that SOSflow is capable of observation, introspection, feedback and control of complex scientific workflows, and that it has desirable scaling properties.

As part of future development, we aim to continue refining and expanding the core SOSflow libraries and the SOS model. The SOSflow codes can be optimized for memory use and data latency. Mechanisms can be added for throttling of data flow to increase reliability in resource-constrained cases. Subsequent work will map out best-fit metrics for dedicating in situ resources to monitoring platforms for the major extant and proposed compute clusters. Additionally, we plan on exploring options for deployment and integration with existing HPC monitoring and analytics codes at LLNL and other national laboratories.

VII. Acknowledgments

The research report was supported by a grant (DE-SC0012381) from the Department of Energy, Scientific Data Management, Analytics, and Visualization (SDMAV), for “Performance Understanding and Analysis for Exascale Data Management Workflows.”

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-XXXXXX).

References

[1] Omar Aaziz, Jonathan Cook, and Hadi Sharifi. Push me pull you: Integrating opposing data transport modes for efficient hpc application monitoring. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 674–681. IEEE, 2015.

[2] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, et al. The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 154–165. IEEE Press, 2014.

[3] Guilherme da Cunha Rodrigues, Glederson Lessa dos Santos, Vinicius Tavares Guimaraes, Lisandro Zambenedetti Granville, and Liane Margarida Rockenbach Tarouco. An architecture to evaluate scalability, adaptability and accuracy in cloud monitoring systems. In *Information Networking (ICOIN), 2014 International Conference on*, pages 46–51. IEEE, 2014.

[4] Todd Evans, William L Barth, James C Browne, Robert L DeLeon, Thomas R Furlani, Steven M Gallo, Matthew D Jones, and Abani K Patra. Comprehensive resource use monitoring for hpc systems with tacc stats. In *Proceedings of the First International Workshop on HPC User Support Tools*, pages 13–21. IEEE Press, 2014.

[5] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, Jeffrey Vetter, and Nirupama Mallavarupu. Falcon: On-line monitoring and steering of large-scale parallel programs. In *Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers' 95., Fifth Symposium on the*, pages 422–429. IEEE, 1995.

[6] Kevin A Huck, Allen D Malony, Sameer Shende, and Alan Morris. Taug: Runtime global performance data access using mpi. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 313–321. Springer, 2006.

[7] Gregory Katsaros, Roland Kubert, and Georgina Gallizo. Building a service-oriented monitoring framework with rest and nagios. In *Services Computing (SCC), 2011 IEEE International Conference on*, pages 426–431. IEEE, 2011.

[8] Mahendra Kutare, Greg Eisenhauer, Chengwei Wang, Karsten Schwan, Vanish Talwar, and Matthew Wolf. Monalytics: online monitoring and analytics for managing large scale data centers. In *Proceedings of the 7th international conference on Autonomic computing*, pages 141–150. ACM, 2010.

[9] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[10] Rafael Keller Tesser and Philippe Olivier Alexandre Navaux. Dimvhcm: An on-line distributed monitoring data collection model. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pages 37–41. IEEE, 2012.

[11] Xuechen Zhang, Hasan Abbasi, Kevin Huck, and Allen Malony. Wowmon: A machine learning-based profiler for self-adaptive instrumentation of scientific workflows. *Procedia Computer Science*, 80:1507–1518, 2016.