Large Scale System Monitoring and Analysis on Blue Waters using OVIS

Michael Showerman, Jeremy Enos, Joseph Fullop*,
Paul Cassella†,
Nichamon Naksinehaboon, Narate Taerat, Thomas Tucker‡,
James Brandt, Ann Gentile, and Benjamin Allan§
*National Center for Supercomputing Applications (NCSA)
University of Illinois at Urbana-Champaign, 1205 W. Clark St., MC-257, Room 1008, Urbana, IL 61801
Email: (mshow|jenos|jfullop)@ncsa.illinois.edu
†Cray, Inc.
901 5th Ave #1000, Seattle, WA
Email: cassella@cray.com
‡Open Grid Computing
4030 West Braker Lane STE 130, Austin, TX 78759
Email: (nichamon|narate|tom)@opengridcomputing.com
§Sandia National Laboratories
Albuquerque NM, 87185
Email: (brandt|gentile|baallan)@sandia.gov

***Abstract*—Understanding the complex interplay between applications competing for shared platform resources can be key to maximizing both platform and application performance. At the same time, use of monitoring tools on platforms designed to support extreme scale applications presents a number of challenges with respect to scaling and impact on applications due to increased noise and jitter. In this paper, we present our approach to high fidelity whole system monitoring of resource utilization including High Speed Network link data on NCSA's Cray XE/XK platform *Blue Waters* utilizing the OVIS monitoring framework. We then describe architectural implementation details that make this monitoring system suitable for scalable monitoring within the Cray hardware and software environment. Finally we present our methodologies for measuring impact and the results.**

I. INTRODUCTION

On large scale High Performance Computing (HPC) platforms there can be significant variation in application run times due to other concurrently running applications and application placement. This variation on the Cray XE/XK platform stems largely from applications' contention for shared High Speed Network (HSN) resources for inter-process communication and storage subsystem access. Therefore, understanding how applications utilize the HSN can be key to maximizing both platform and application performance. Such understanding can only come from taking a whole system view of metrics affecting performance through the analysis of hardware performance counters and collected measurements for differing network traffic types. This requires an application independent monitoring system which can provide high fidelity snapshots of performance related metrics of interest including HSN traffic/contention. The major challenges to such a monitoring system are scalability, access to pertinent data, and the ability to minimize adverse impact on running applications resulting from increased system and network noise.

NCSA collaborated with Cray and Sandia National Laboratories (SNL) to evaluate the suitability of SNL's OVIS [1], [2] monitoring system for performing run-time whole system monitoring of performance related metrics of interest on their 27,648 node Cray XE/XK platform Blue Waters [3] with specific attention to previously unavailable HSN metrics. As part of this evaluation: NCSA defined a set of performance related metrics to be collected as well as collection intervals; SNL and Open Grid Computing (OGC) implemented several enhancements to OVIS's Lightweight Distributed Metric Service (LDMS) framework to enable efficient collection, transport, and storage of these metrics; and Cray enhanced access to its HSN performance counters pertinent to the NCSA defined metric set. In this paper we present details and evaluation of this work. We discuss the choice of the performance metrics collected, including details of attribution of application vs. OS traffic, and intended use cases for the data. We also discuss impediments to gathering HSN performance metrics in a lightweight fashion and how those impediments were overcome. We provide a description of the architectural details of the

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number ACI 1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

OVIS monitoring framework that address scalability and application impact. Additionally, we discuss details of a number of enhancements to the monitoring framework that were motivated by NCSA's requirements for production deployment of large scale data analysis. Further, we present our evaluation criteria and methodologies for assessing the impact of OVIS's monitoring on running applications and benchmarks. In particular, we evaluate both additional OS noise introduced and changes in the run times of various applications over their baseline values for several different collection intervals. Finally we present the results of these evaluations along with our conclusions on the viability of this approach.

II. DATA SET

NCSA needed to better understand the current state of the Cray HSN at any given time. As we began to see more impact from network contention, it became apparent that we needed a method to gather more detailed information regarding the usage of individual network links at the administrative level for analysis. Cray suggested that OVIS was the preferred method for gathering HSN metrics for a long term solution. Given the broader scope of OVIS beyond merely link data, we assembled a team of systems engineers and application specialists to compile a list of metrics that could help us best analyze system and user behaviors.

After finalizing a metric set covering processor, storage, and network subsystems, we needed to balance the value of the data against both the performance impact on applications and data volume that would need to be stored. We determined that we could achieve most goals with a one minute collection rate of data, but incorporated tests of one second intervals to measure the potential impact of higher rates.

A. High Speed Network Performance Metrics

1) Summary: We report the following HSN metrics for each XE/XK node:

- Total Output Bandwidth
- Total Input Bandwidth
- Total Fast Memory Access (FMA) Output Bandwidth
- Total Block Transfer Engine (BTE) Output Bandwidth
- Total Non-Application Output Bandwidth
- Total Non-Application Input Bandwidth

and the following metrics for each network link, or HSN torus dimension:

- Bandwidth
- Average Packet Size
- Time Stalled
- Channel Status

We derive HSN performance metrics from two sources: performance counter registers on the Gemini devices [4] and Linux kernel modules software implementing the Gemini device driver and the LNet driver.

2) Metrics Acquisition: The performance counters were already available via the `gpcd` kernel module [5]. This module provides an interface for applications to sample the counters to measure their own use of the HSN. This interface has several drawbacks for this monitoring effort. Notably, it requires revalidating the set of registers requested each time the registers are sampled.

Additionally, not all nodes on the system combine the registers in the same manner to produce the metrics. Each Gemini has 40 network channels. Certain of those channels may be used for one torus dimension on one node, and for a different dimension on another, depending on a given system's topology, and upon each node's physical location in that system.

As part of this work Cray has provided a new kernel module, `gpcdr`, providing a more suitable interface to the performance counter registers for this effort and userspace tools to configure it appropriately for the node.

A single configuration file defines what metrics to report. For each metric, it defines the combination of registers to use to generate them. For the metrics that are defined per-NIC, the configuration file defines the metric in terms of specific registers. For the metrics defined per-dimension, the metric is defined using partial register names that are later combined with channel numbers to fully specify the registers.

Cray also enhanced the RCA subsystem to provide each node with the mapping of links to dimensions for its Gemini. This mapping will not change while a node is booted. When each node boots, an init script combines the configuration file with the link-dimension mapping, and configures the `gpcdr` module to compute and report the desired metrics.

The `gpcdr` module reports the metrics via files in the `/sys` file system, which are read by the LDMS sampler. The `gpcdr` module reads the registers and calculates the metrics when the files are read, allowing the daemon to control the poll rate.

3) Metrics details: The `gpcdr` module calculates the per-link metrics, except for Channel Status, from per-channel performance counters. It calculates the per-node metrics from Gemini NIC performance counters. It calculates the Channel Status metric from registers that are not performance counters. It reports the number of functioning lanes in each Link at the time of each sampling.

In addition to the HSN metrics acquired through `gpcdr` and reported directly, the LDMS sampler produces several derived metrics based on these counters, collection time interval, network media speed, and others as follows:

- Percent of bandwidth used in each torus dimension is calculated from the change in traffic in that dimension over the sampling interval divided by the actual time since the previous sample was collected. This is converted to a percentage of the maximum using knowledge of the network media type maximum bandwidth in the following way: $100 *$

(Δtraffic/Δtime/max bandwidth) with traffic in bytes, Δtime in sec, and bandwidth in bytes/sec.
- Average packet size for each torus dimension is calculated as the quotient of the difference in traffic divided by the difference in packets over the last sampling interval.
- Percent of time stalled for each torus dimension is calculated from stall count, number of links, and the time delta between current and previous samples as: $100 * (\text{time stalled}/\text{num lanes})/\Delta\text{time}$ with time and time stalled in nanoseconds.

Note that the time deltas are taken between `gpcdr` timestamps and not sampler timestamps as the former reflect when the `gpcdr` metrics were collected by the module. Where percentage values are calculated they are multiplied by 10^6 because we report integer values and this provides more dynamic range. Since each Gemini is shared by two nodes we could collect all HSN related information using half of the nodes. We decided, however, that in the interest of simplicity through uniformity, we would collect the same counters on both nodes connected to each Gemini. This also provides redundant Gemini performance counter information in case one of the nodes sharing a Gemini is down.

4) HSN Metrics Limitations: Due to limitations of the Gemini performance counters, several of the metrics can only be approximated. These are the per-node Total Output, FMA Output, and BTE Output metrics. Two approximations of the Total Output and BTE Output metrics are provided, distinguished by the optA and optB suffixes.

B. Non HSN Metrics

In addition to the HSN performance metrics described above, NCSA identified other information to collect for understanding system performance. Of note are:

- Lustre file system counters
- CPU load averages
- Current free memory
- LNet traffic counters
- ipogif counters

The ipogif component of the Gemini device driver reports how much HSN traffic is IP traffic. The LNet driver reports how much is LNet traffic (e.g., the Lustre and DVS file systems). All IP and LNet traffic is deemed to be system traffic. All other traffic is deemed to be application traffic.

III. OVIS

OVIS is a suite of HPC monitoring tools, under active development at SNL and OGC, that collectively provide: 1) lightweight collection of metrics from HPC platform components, 2) a variety of analysis and visualization tools to operate on stored data, 3) the ability to evaluate data against some criteria as it is being collected, and 4) the ability to provide notification to system administrators, feedback to platform components (e.g. resource manager), and performance data to running applications. This section focuses on use of the Lightweight Distributed Metric Service (LDMS) component of the OVIS suite and its application to Blue Waters. In particular we discuss details of the architecture that make it viable for providing collection, transport, and storage of the data metrics to the NCSA system administration team on relevant time scales with minimal adverse impact on running applications. In this section we also describe enhancements that we have made to the infrastructure in order to meet NCSA's needs with respect to size, time attribution, ease of storage, and timely whole system analysis.

A. LDMS: The OVIS Data Collection, Transport, and Storage Framework

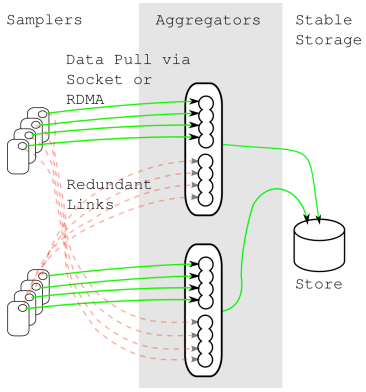

Figure 1. Simple LDMS use case

The Lightweight Distributed Metric Service consists of: 1) a daemon (`ldmsd`) that allocates data set memory, supports data sampler, storage and filter plug-ins, requests data sets from other `ldmsd`'s, and listens for connection requests, 2) data sampler plug-ins that periodically sample data on-node 3) storage plug-ins that write data to a variety of storage formats, and 4) multiple transport methods that define the media of data transmission and whether a socket connection or Remote Direct Memory Access (RDMA) will be used. In this section we describe each of these components, their functional characteristics, and the techniques used to minimize monitoring overhead.

Figure 1 illustrates the single tiered topology including redundant fail-over connections. Because service nodes where the aggregators run are a scarce resource, the ratio of monitored nodes to aggregators must be reasonably large. Currently upper limit of this ratio is about 15,000:1.

1) Metric Sets: Data is stored as collections of data values called metric sets. The data contained in a particular metric set can be raw or derived with all values taken from a particular data source (e.g. `/proc/meminfo`) or with values coming from a variety of sources.

Metric sets also contain metadata, including name, size, and generation numbers (GN). Whenever a metric set's

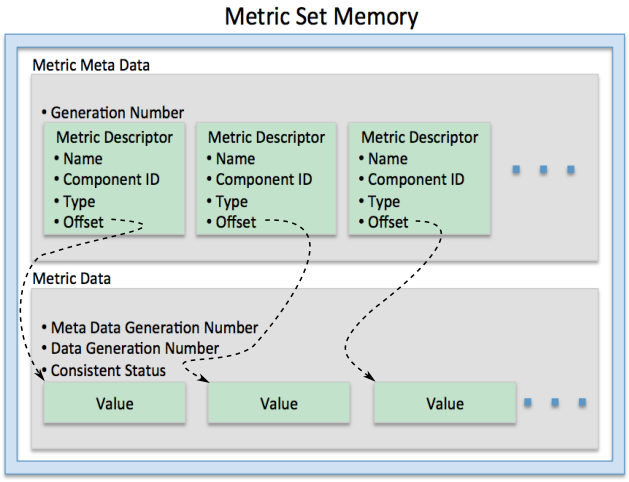

Figure 2. Diagram of LDMS Metric Set Memory.

definition changes or when a metric value within a set has been updated, the GN of the set's metadata or data, respectively, changes. Generation numbers enable a consumer to discriminate between current and previous instantiations of data. The metric set is laid out in memory such that the metadata of the metrics are in one contiguous memory region and the data values are in another (Figure 2); this allows the data values to be efficiently acquired as a memory chunk independent of the associated metadata thus minimizing network perturbation.

Information about metric sets, metrics, and metadata can be obtained through the `ldms_ls` command line utility. Parameters to `ldms_ls` include the *host* being queried, *transport*, and *port*. Selected lines for a metric set using the optional `-l` flag (long listing) and `-v` flag (verbose listing) are shown in Figure 3.

Note that a metric set in this case is associated with a single component identified by a unique `component_id` which is the node identification (NID) number from which the data was gathered (e.g. the component_id for nid00044 is 44).

2) Sampler Plug-ins: Samplers are plug-ins to the LDMS framework.

Because we target minimal processing on the node, raw data values rather than functions of values, typically constitute the metrics. However, if there is the desire to directly utilize derived data (e.g. quickly search data for outliers without having to do differences as a post processing step) then this processing can be performed in the sampler and may even decrease the amount of data needing to be moved over the network.

3) LDMS Daemon: The LDMS component that orchestrates sampling, transport, and storage is a daemon called `ldmsd`. Both data and metadata associated with a sampler's metric set are written into local memory. The data portion

```
# ldms_ls -h nid00044 -x ugni -p 412 -l
nid00044/cray_system_sampler_r: consistent,
last update: Wed Apr 09 08:52:40 2014 [726us]
 U64 1                  nettopo_mesh_coord_X
 U64 1                  nettopo_mesh_coord_Y
 U64 6                  nettopo_mesh_coord_Z
 U64 0                  X+_traffic (B)
 U64 3265901109447      X-_traffic (B)
 U64 0                  Y+_traffic (B)
 U64 21509840670687     Y-_traffic (B)
 U64 53884897461291     Z+_traffic (B)
 U64 32970043654509     Z-_traffic (B)
 U64 864430967892       Z-_packets (1) (XYZ+/-)
 U64 30131891442        Z-_inq_stall (ns) (XYZ+/-)
 U64 2089757046         Z-_credit_stall (ns) (XYZ +/-)
 U64 24                 Z-_sendlinkstatus (1) (XYZ +/-)
 U64 24                 Z-_recvlinkstatus (1) (XYZ +/-)
 U64 1955               Z-_SAMPLE_GEMINI_LINK_BW (B/s) (XYZ +/-)
 U64 13                 Z-_SAMPLE_GEMINI_LINK_USED_BW (% x10e6) (XYZ +/-)
 U64 20                 Z-_SAMPLE_GEMINI_LINK_PACKETSIZE_AVE (B) (XYZ +/-)
 U64 0                  Z-_SAMPLE_GEMINI_LINK_INQ_STALL (% x10e6) (XYZ +/-)
 U64 0                  Z-_SAMPLE_GEMINI_LINK_CREDIT_STALL (% x10e6) (XYZ +/-)
 U64 13071017859520     totaloutput_optA
 U64 10299753336224     totalinput
 U64 9838624535520      fmaout
 U64 286137223792       bteout_optA
 U64 283312966954       bteout_optB
 U64 13068237253226     totaloutput_optB
 U64 984                SAMPLE_totaloutput_optA (B/s)
 U64 139                SAMPLE_totalinput (B/s)
 U64 100                SAMPLE_fmaout (B/s)
 U64 35                 SAMPLE_bteout_optA (B/s)
 U64 26                 SAMPLE_bteout_optB (B/s)
 U64 976                SAMPLE_totaloutput_optB (B/s)
 U64 17588842           dirty_pages_hits#stats.snx11024
 U64 27547858           dirty_pages_misses#stats.snx11024
 U64 0                  writeback_from_writepage#stats.snx11024
 U64 0                  writeback_from_pressure#stats.snx11024
 U64 0                  writeback_ok_pages#stats.snx11024
 U64 0                  writeback_failed_pages#stats.snx11024
 U64 1551040415605      read_bytes#stats.snx11024
 U64 111681033094       write_bytes#stats.snx11024
 U64 9506816            brw_read#stats.snx11024
 U64 0                  brw_write#stats.snx11024
 U64 154570             ioctl#stats.snx11024
 U64 33185713           open#stats.snx11024
 U64 33459578           close#stats.snx11024
 U64 0                  mmap#stats.snx11024
 U64 43592577           seek#stats.snx11024
 U64 1                  fsync#stats.snx11024
 U64 190714             setattr#stats.snx11024
 U64 190625             truncate#stats.snx11024
 U64 0                  lockless_truncate#stats.snx11024
 U64 0                  flock#stats.snx11024
 U64 1213918            getattr#stats.snx11024
 U64 153847             statfs#stats.snx11024
 U64 1718087            alloc_inode#stats.snx11024
 U64 0                  setxattr#stats.snx11024
 U64 0                  getxattr#stats.snx11024
 U64 0                  listxattr#stats.snx11024
 U64 0                  removexattr#stats.snx11024
 U64 135021880          inode_permission#stats.snx11024
 U64 0                  direct_read#stats.snx11024
 U64 0                  direct_write#stats.snx11024
 U64 0                  lockless_read_bytes#stats.snx11024
 U64 0                  lockless_write_bytes#stats.snx11024
 U64 0                  nr_dirty
 U64 0                  nr_writeback
 U64 200                loadavg_latest(x100)
 U64 203                loadavg_5min(x100)
 U64 2                  loadavg_running_processes
 U64 217                loadavg_total_processes
 U64 32069868           current_freemem
 U64 180128670          SMSG_ntx
 U64 84138092941        SMSG_tx_bytes
 U64 179201767          SMSG_nrx
 U64 62591572089        SMSG_rx_bytes
 U64 2463841            RDMA_ntx
 U64 166910425701       RDMA_tx_bytes
 U64 5995457            RDMA_nrx
 U64 265128956892       RDMA_rx_bytes
 U64 207633071910       ipogif0_rx_bytes
 U64 116299863623       ipogif0_tx_bytes

# ldms_ls -h nid00044 -x ugni -p 412 -v
nid00044/cray_system_sampler_r: consistent,
last update: Wed Apr 09 08:55:20 2014 [727us]
  METADATA --------
              Size : 13560
             Inuse : 7144
      Metric Count : 130
                GN : 131
  DATA ------------
         Timestamp : Wed Apr 09 08:55:20 2014 [727us]
        Consistent : TRUE
              Size : 1088
             Inuse : 1088
                GN : 1735
  -----------------
```

Figure 3. Representative ldms_ls long and verbose listing for the Blue Waters metric set. While link-aggregated metrics are reported separately, here for readability, after the first instance (traffic), only one value (Z-) is shown. Such metrics are indicated by (XYZ +/-). Similarly, Lustre metrics are reported per Lustre mount point (3 for Blue Waters) here only 1 is shown.

is over-written each time a new datum is collected. Thus an `ldmsd` retains no history.

LDMS daemons run multiple worker threads to perform the following functions asynchronously: The sampling thread takes care of scheduling sampler plug-ins to run according to their configured period. The period for each sampler plug-in is independent of all others being run on a given `ldmsd` and is user configurable during run time.

A connection thread listens for a socket connection request from an aggregator or `ldms_ls` query. A persistent connection is then established over the requested transport type. Connections are maintained for the mutual life of the `ldmsd` pair and an `ldmsd` may maintain multiple simultaneous pair-wise connections (e.g. for fail-over standby).

A configuration thread handles the task of external configuration of `ldmsd` via a Unix Domain Socket connection. This configuration consists of defining sampler plug-ins, sampling periods, connections to other `ldmsd`'s and their collection periods, and storage plug-ins to load along with file system information, output format, and data-sets to be written.

`ldmsd`'s operate asynchronously in a pairwise fashion e.g., a request for data results in transfer of data from sampler to aggregator only and does not result in additional queries to other `ldmsd`'s.

4) Writing to a Store: The LDMS daemon also supports a variety of storage plug-ins (CSV, MySQL, Flat File). Because LDMS uses an asynchronous pull model it is possible for an aggregator to read a sampler's data before or during an update. We elect to drop rather than store redundant or inconsistent data. Thus, before values are written, both the GN of the new set must have changed and the *consistent* flag must be *TRUE*. If either or both of these conditions is not met collection is rescheduled for the next configured interval. Thus, there can be data gaps in the store due to either samplers not updating or the unlikely event that the aggregator happens to perform a collection during a sampler update. Such losses are unlikely, however, when utilizing the synchronous mode of operation and scheduling the aggregator to have a time offset greater than expected clock skew plus sampling time.

B. Enhancements to LDMS for Blue Waters

Synchronization: Because LDMS daemons on different nodes all perform sampling and aggregation asynchronously, the data sampled in a given recording cycle on one host could be up to two sampling periods removed from that of another host. Since the original target was a sampling period of 10 minutes, the resulting data could at best be used to gain a vague understanding of trends but certainly not for the kind of understanding NCSA needed (Section II). Thus we implemented a *synchronous* feature which enables all nodes to sample within a small time window of approximately 5ms. Figure 4 (top) shows the time at which samples were taken, according to each node's clock, across 10,000 nodes and over a 25 minute time interval where the sampling and aggregation intervals were each 30 seconds. Figure 4 (bottom) zooms in on the interval around the last sample and shows the tight grouping of samples. Note that this does not take into account clock skew and is not related to the time at which the aggregator collects the samples. Using this feature we configure the aggregators to start their collection from the compute nodes at an offset from the nodes sample time that is sufficient to ensure all collected metric sets are from the same sample time.

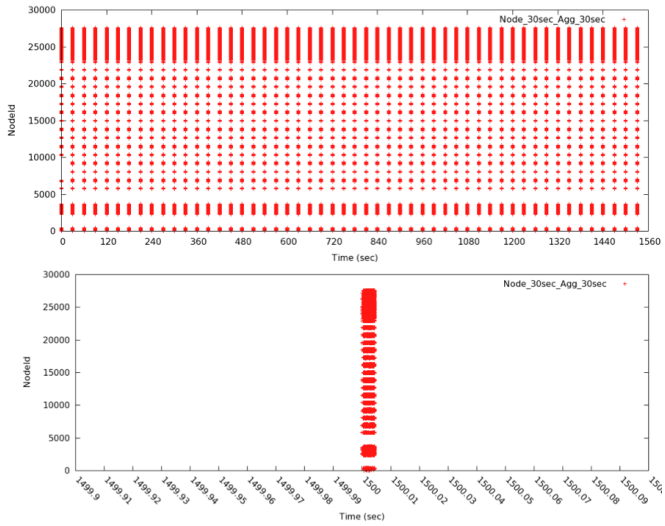

Figure 4. Synchronized sampling and collection over 10,000 *Blue Waters* compute nodes. Y-axis is nid number. The top plot shows 30 second collection intervals over a 1500 second time window. The bottom plot zooms in on a particular collection interval to show the tight grouping.

Minimal Image Footprint: On the Cray XE/XK platform, compute nodes boot an image served by the boot node. Larger images take longer to boot. Since the image resides in main memory, the image size also reduces memory available to applications. Starting monitoring at boot time on all nodes (including service nodes) requires inclusion of a minimum set of LDMS libraries and binaries in the system image. In order to minimize the image footprint we made some components of the `ldmsctl` interface (readline and ncurses) compile time options thus saving 1MB of space over the original size. The current size of LDMS required libraries and binaries is 2.5MB. In addition to LDMS component, an interconnect text file (obtained on the system management workstation using the `rtr --interconnect` command) containing link *type* information is required in order to calculate percent of bandwidth used. For *Blue Waters* this file is 41MB (16x the size of the LDMS image and 10% of the original compute node image). We algorithmically consolidated this to 31kB while still retaining the per node direction and *media type* information thus making it viable

for inclusion in the image.

Ease of system wide start-up: A problem faced by all HPC systems is that not all components serve the same function (e.g. compute nodes vs. service nodes) and thus there are usually differences in what file systems are mounted, number of cores, amount of memory, and connectivity. In keeping with ease of data handling, we decided to have all metric sets contain the same data whether or not the data is actually available on any particular node. Thus the resulting common metric set contains a superset of the desired data across both compute and service nodes. In the event the data does not exist, the sampler records zeroes. This enables use of a simple start script that is the same for all nodes and the ability to bulk load data from all nodes into the same database without preprocessing.

Single time attribution: In order to facilitate analysis, data sampled concurrently should all have a single time stamp. While the canonical LDMS implementation supports multiple concurrent samplers which typically are based on a single data source each, for this application we created a single multi-source sampler for the data of interest. Thus all desired data, including some derived data, is included in a single metric set. Since the time to sample, derive, and write all of the data for a metric set is approximately 400 microseconds and the timescales of interest are seconds to minutes, this gives a reasonably representative single time stamp for all data.

Ease of Large Scale Post Processing of Data: As of the start of this project the LDMS infrastructure supported multiple storage formats but stored each data type separately (e.g. X+ stalls stored to own file with time stamp per data). Because we wanted to be able to bulk load data we wanted data to be stored at the time stamped metric set level. This necessitated a re-write of the interface to the storage plug-ins as well as data structures for data movement within `ldmsd`. The result is that we are now storing each metric set as a set of comma separated values (CSV) to a single flat file which can be bulk loaded into a MySQL database for post processing.

C. Implementation on Blue Waters

Figure 5 presents a high level overview of the topology and configuration of the LDMS monitoring software on *Blue Waters*. The associated components and configurations are described below.

Sampler Configuration: As described in Sections II and III-B, all sampler `ldmsd`'s present identical metric sets for collection by their aggregators independent of node type and availability of particular data. The metric sets are comprised of 194 metrics and represent 1,552 Bytes of data to be transmitted over the network for each compute node at each collection interval. Representative data from this metric set is shown in Figure 3. We defined two collection frequencies *normal* and *high* to be once per minute and once per second respectively. The impact of collection of this metric set at these frequencies is described below in Section IV. Whether collecting at sixty second or one second intervals, all samplers are scheduled for the same wall clock time (Figure 4).

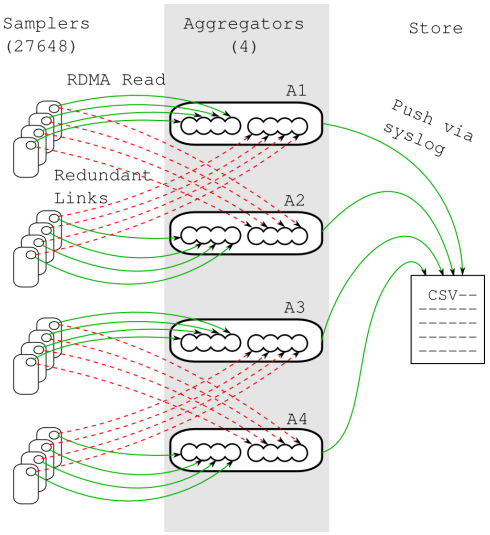

Figure 5. LDMS configuration on *Blue Waters*. Note that this includes redundant connections to each sampler `ldmsd` for fast fail-over capability

Aggregator Configuration: Aggregators pull metric set data from sampler *ldmsd*s using the Remote Direct Memory Access (RDMA) protocol. RDMA was chosen in this deployment to eliminate collection related impact on compute node CPU utilization. We configured four aggregators to each be a primary aggregator of metric sets for a quarter of the nodes (6,912) and a fail-over aggregator for another quarter of the nodes. As can be seen in Figure 5 aggregators A1 and A2 are paired for fail-over as are aggregators A3 and A4. The sampler to aggregator pairings were chosen to minimize metric set network impact. The aggregators are configured to collect on the same time intervals as the samplers but with sufficient offset from the samplers to allow all samples to have been taken before collection by the aggregator. Because of possible clock skew we configure the aggregators collection offset to be that of the samplers plus 0.4 seconds.

Store Configuration: Aggregators are configured to store to flat file with a CSV format. This means that at each aggregation period each metric set from each node is written as a comma separated set of data to a flat file which in this case is a named pipe that is forwarded by *syslog-ng* to a host where it is bulk inserted into a MySQL database. A header file is written that describes what particular data is in each location. The common metric set guarantees the data and order is the same for every node's metric set. Additionally, though each piece of data has an associated component identifier, we use a single identifier for all data for a given

node. To save on storage bandwidth and make it possible to do bulk inserts we only write the component identifier once for each metric set storage event along with the time stamp. The time stamp itself is broken into two comma separated values, seconds since the epoch and microseconds past the second. Splitting time this way enables faster database loading and faster searches than storing it as a single float value.

IV. Performance

Test Selection: The Cray Linux Environment is a streamlined Linux implementation focused on minimizing the operating system impact on the scalability of applications. We must balance collection cost against the potential impact to application performance. We compiled a suite of benchmarks and applications to test the various impacts of both the collection of the metrics on each compute node and the transfer of the data off of the compute resource. We measured the operating system noise impact of LDMS costs using the PAL System Noise Activity (PSNAP) benchmark. We used the Intel MPI Benchmark (IMB) and Cray's internal tool LinkTest to measure network performance variations at full scale for collective operations and individual link performance respectively. Finally, we selected the MIMD Lattice Computation (MILC) QCD application and SNL's Minighost to measure the holistic impact on applications.

Test Configuration: Five monitoring configurations were used to gather performance data:

- Baseline with no data collection or aggregation (novis)
- Once every 60 seconds collection with no aggregation (c60noa)
- Once every 60 seconds collection with aggregation (c60a60)
- Every second collection with no aggregation (c1noa)
- Every second collection with 1 second aggregation (c1a1)

These sample scenarios were chosen a) to measure the network transfer separately from the local impact on the compute nodes, i.e., no-aggregation conditions would potentially impact the node performance, but not give rise to any network traffic, and b) demonstrate potential impacts at higher than targeted data sampling rates. All tests, including partial scale, were executed with no other applications utilizing the HSN.

A. PSNAP

PSNAP [6] performs multiple iterations of a loop calibrated to run for a given amount of time. On an unloaded system, variation from the ideal amount of time can be attributed to system noise. We ran PSNAP with and without the OVIS monitoring in order to determine the additional impact of the monitoring. PSNAP was run without the barrier mode, making the effects on each node independent. 32 tasks per node were executed with a 100 microsecond loop.

Figures 6 and 7 show the baseline and monitoring results with 1 minute sample interval and the 1 sec sample interval, respectively.

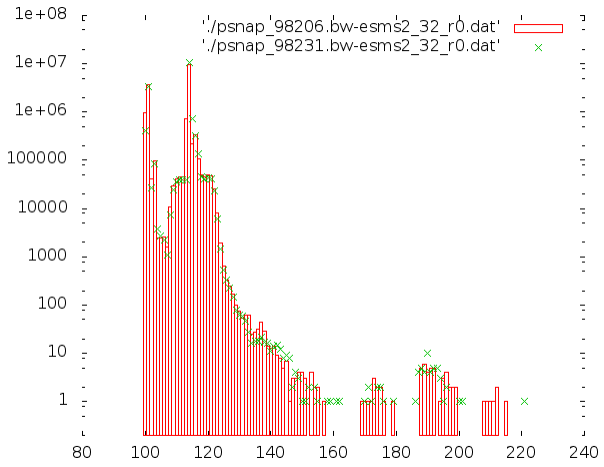

Figure 6. PSNAP results: Histogram of occurrences vs. loop time (us) with 1 minute sampling data (X) compared to none (red boxes).

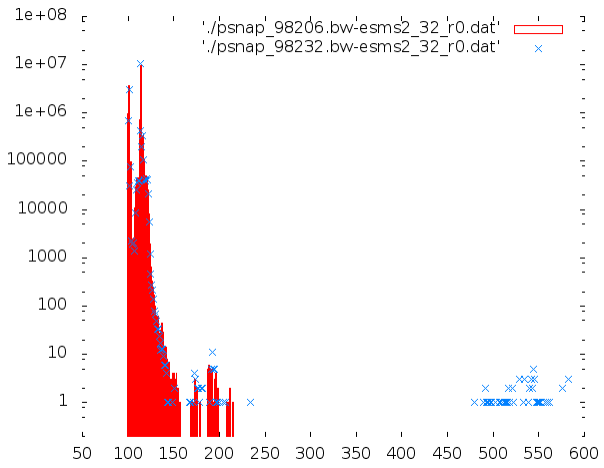

Figure 7. PSNAP results: Histogram of occurrences vs. loop time (us) with 1 second sampling data (X) compared to none (red boxes).

The one minute sampling interval results show no significant impact. This is because the run time for the test was slightly less than a minute and a sampling event did not occur during the test. The one second sampling interval shows an additional $50-60$ events, out of 16 million total, out in the tail with an additional delay of $375-475$ microseconds. This is in line with the expected delay caused by the known sampling execution time of order 400 microseconds and the expected number of occurrences given the execution time of around a minute and the sampling period of 1 second.

While an application process running on the node would only be impacted during the sampling time, an MPI application might wait upon processes on other nodes and thus random sampling across nodes might result in greater impact. The synchronized sampling feature (Section III-B) has the additional benefit that the occurrences of sampling across the nodes can be coordinated in time and thus bound how many of an application's iterations would be affected.

B. LinkTest

Cray has developed an MPI program that measures the individual link performance within a job. For this test we used the the more extreme test configurations of novis and c1a1. We used 10,000 iterations of 8kB messages. This gives us multiple collections of data per link test. The results showed a baseline of 1.74278 milliseconds per packet with novis, and 1.74276 milliseconds with the data collection active. While the c1a1 time is slightly shorter, the difference is statistically insignificant.

C. Intel MPI Benchmark

The Intel MPI benchmark for MPI AllReduce was initially tested at small scale. This test was run on the same 2744 nodes as the MILC runs to validate it against MILC's internal AllReduce performance measurements. Also, this allowed us to validate on a consistent node set that was topology optimized for maximum network performance. This test used a 64B payload and one task per compute node. The results are shown in Figure 8. While the average time rose for the c60a60 configuration, the maximum time (e.g., worst case performance outlying performance data point) for three of the configurations decreased. Overall, there was not a correlating impact factor with the OVIS configuration.

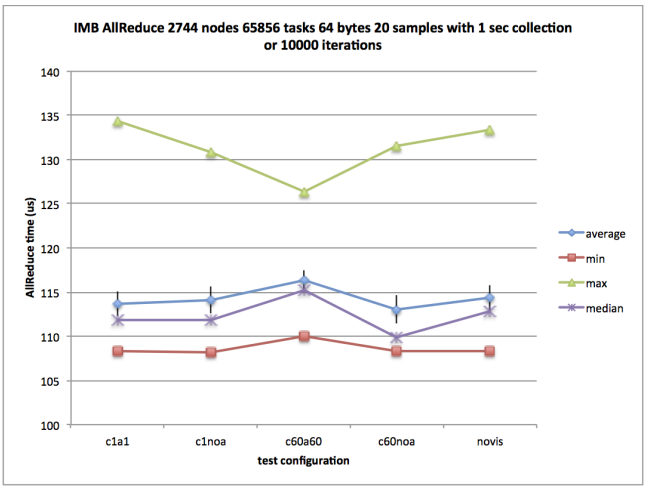

Figure 8. Intel MPI Benchmark results at small scale.

Table I
MILC TIMING RESULTS FOR A 2744 NODE RUN (50 STEPS)

Test	seconds	novis	c60noa	c60a60	c1noa	c1a1
Llfat	Ave	0.502	0.514	0.503	0.508	0.510
	Min	0.473	0.472	0.474	0.472	0.472
	Max	0.624	0.819	0.653	0.734	0.691
	StdDev	0.035	0.061	0.046	0.053	0.051
Lllong	Ave	0.014	0.015	0.015	0.016	0.015
	Min	0.010	0.010	0.011	0.011	0.011
	Max	0.043	0.048	0.048	0.048	0.050
	StdDev	0.007	0.009	0.006	0.008	0.008
CG	Ave	5.20e-3	5.21e-3	5.20e-3	5.20e-3	5.19e-3
(per iter)	Min	5.00e-3	5.20e-3	5.00e-3	5.01e-3	5.00e-3
	Max	5.43e-3	5.44e-3	5.44e-3	5.45e-3	5.41e-3
	StdDev	0.14e-3	0.14e-3	0.14e-3	0.12e-3	0.13e-3
GF	Ave	0.485	0.494	0.497	0.493	0.498
	Min	0.450	0.456	0.451	0.451	0.449
	Max	0.597	0.734	0.635	0.834	0.803
	StdDev	0.040	0.060	0.046	0.067	0.066
FF	Ave	0.739	0.741	0.750	0.731	0.756
	Min	0.664	0.665	0.664	0.665	0.667
	Max	0.967	1.030	1.034	1.045	1.005
	StdDev	0.078	0.072	0.085	0.082	0.091
Steptime	Ave	11.151	11.195	11.179	11.164	11.179
	Min	10.678	10.658	10.664	10.669	10.659
	Max	11.694	11.631	11.783	11.597	11.710
	StdDev	0.299	0.316	0.312	0.294	0.293

D. MILC

MILC [7], [8] is a large scale numerical simulation to study quantum chromodynamics (QCD) and is used extensively across a wide variety of platforms, and is known to have sensitivity to interconnect performance variation. In this test, the application was run using 2744 XE nodes with a topology aware job submission to minimize congestion. It utilizes a 64B Allreduce payload in the Conjugate Gradient (CG) phase with a local lattice size of 6^4. Performance variations for each of the phases of the calculation are shown in Table I. Overall application performance is most dependent on the CG phase which has many iterations per step. No statistically significant impact was observed.

E. Minighost

MiniGhost [9] is a Sandia code which is used for studying the communications, not the computations, relevant to mesh-based codes. An instrumented version [10] of MiniGhost was run which reports total run time, time spent in communication time, and time spent in a phase which includes time spent waiting at the barrier (GRIDSUM). Parameters were chosen to obtain an approximately 1.5 minute run time on $8,192$ nodes in order to determine effects on timing. Because of the short runtime, three repetitions of the code were run under the novis and c1a1 conditions only. In order to ensure that the rank-to-core (and node) mapping was consistent for all repetitions, each repetition was launched on the same nodes and an internally computed ordering for determining based on the known communication pattern of the application was used.

Table II
TIMING RESULTS FOR MINIGHOST WITH AND WITHOUT MONITORING. TOTAL RUN TIME AND AVE/MAX/MIN TIMES PER RANK FOR COMMUNICATION AND GRIDSUM ARE REPORTED.

	novis				c1a1			
	Tot	Ave	Max	Min	Tot	Ave	Max	Min
Run	98.5	-	-	-	92.3	-	-	-
Time	95.3	-	-	-	90.2	-	-	-
	91.8	-	-	-	90.8	-	-	-
Comm.	-	9.2	15.8	2.6	-	9.0	15.0	2.5
Time	-	9.0	15.6	2.7	-	9.0	14.9	2.5
	-	9.1	15.2	2.7	-	9.0	14.4	2.6
GRIDSUM	-	60.4	67.2	54.4	-	53.6	60.4	48.4
Time	-	56.2	63.8	50.2	-	52.4	59.1	47.1
	-	53.6	60.4	48.3	-	53.0	59.3	47.8

Timing results for three runs MiniGhost are shown in Table II. There was no statistically significant impact in any measure when using OVIS at the one second collection rate vs. no collection. Average, maximum, and minimum timings are computed over the set of iterations within each run.

V. USE CASES FOR DATA

On the Cray XE/XK platform, applications contend for HSN resources for inter-process communication and storage subsystem access. In order to gain insight into an application's performance, an understanding of the entire system mesh is necessary. The size of the Blue Waters system and the number of metrics make it non-trivial to mine the information gathered and to develop meaningful representations. However, initial explorations of the data and representations have been promising and are discussed here.

In this section we consider data from a 24 hour period on Blue Waters over the entire system. This is about 40 million data points per metric or 7.7 billion data points total. A high-level understanding of the system and identification of hot spots can be gained by even a simplistic visualization of data through time. In the plots in this section, data points are larger than the divisions between points (e.g., points for consecutive NIDs will overlap) in order to aid in visualization. Data values are plotted from smallest to largest in order to ensure that the highest value in any group in points is not obscured. Values less than 1 for any quantity are not included in the plots. We utilized the (negative) cubehelix [11] palette, to take advantage of continuous perceived intensity.

1) Lustre: The Lustre parallel file system is a shared resource that is critical to the operation of *Blue Waters*. Too many open/close operations in a given time interval can overwhelm the Lustre meta-data server, as too much write/read traffic can overwhelm the storage server infrastructure. Both of these conditions, if left unchecked, can negatively impact the system as a whole. Thus when the Lustre file system becomes slow or unresponsive, identification of which nodes and, by extension, which applications are causing the problems can enable system administrators to take mitigating action (e.g. kill an offending job) rather than having to do a system reboot. Even if mitigating action could not be taken fast enough, such identification can still enable the system administrators to work with the user(s) to identify the problem and ensure it isn't repeated. Using monitored data about Lustre file system activity enables easy identification of hot spots. Figures 9 and 10 show Lustre reads and writes and opens and closes, respectively, on `snx11001` which hosts the home file system. Note that it is easy to visually identify components with a lot of Lustre file system activity as well as time frames over which it occurred. Together with the Moab job scheduler logs it is then simple to identify associated users and applications.

2) HSN Link Stalls: The High Speed Network (HSN) is also a shared resource whose health is critical to the healthy operation of the system as a whole. Deveci et al. [10] have shown a direct correlation between output credit stalls on HSN links and application performance degradation. Output credit stalls are a good indicator of network hot spots. Identification of where and when hotspots occur, amount of traffic, maximum link speeds, and applications' behavioral characteristics with respect to network traffic can enable better understanding of application performance variation as well as better job scheduling and placement. As in the case of Lustre file system usage, monitoring these characteristics about the HSN in a whole system context can enable easy identification of all of the above. Figures 11, 12, and 13 show, for each direction, the derived metric percent of time spent in output stalls (Section II-A3). This is the percentage of time that a logical link (comprised of multiple physical links) is unable to transmit data due to lack of credits on one or more of its physical links. This can be used to give an idea of regions and times of HSN congestion. Note that the stall rate can vary by more than an order of magnitude over the different links of a particular node over the 60 sec sampling interval.

VI. CONCLUSIONS AND FUTURE WORK

A. Conclusions

From the application impact testing presented in Section IV it is clear that the OVIS data collection, transport, and storage infrastructure provides scalable access to whole system data, in a snapshot fashion, with no statistically significant adverse impact to running applications at the tested sampling rates of 1 and 60 seconds.

We have shown that whole system snapshots of shared system resource utilization data can provide valuable insights to system performance which in turn impacts application performance. Additional analysis and visualization tools need to be developed/applied in order to fully utilize the new volume of data that we are collecting.

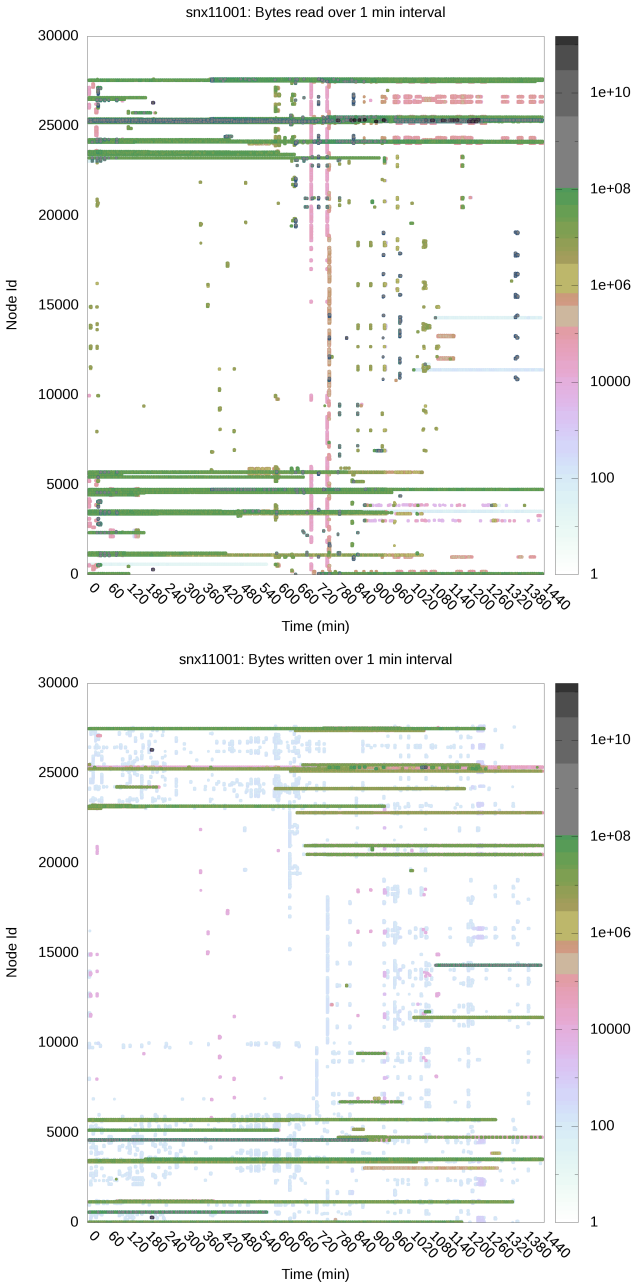

Figure 9. Lustre reads (top) and writes (bottom) on snx11001 which hosts the home file system.

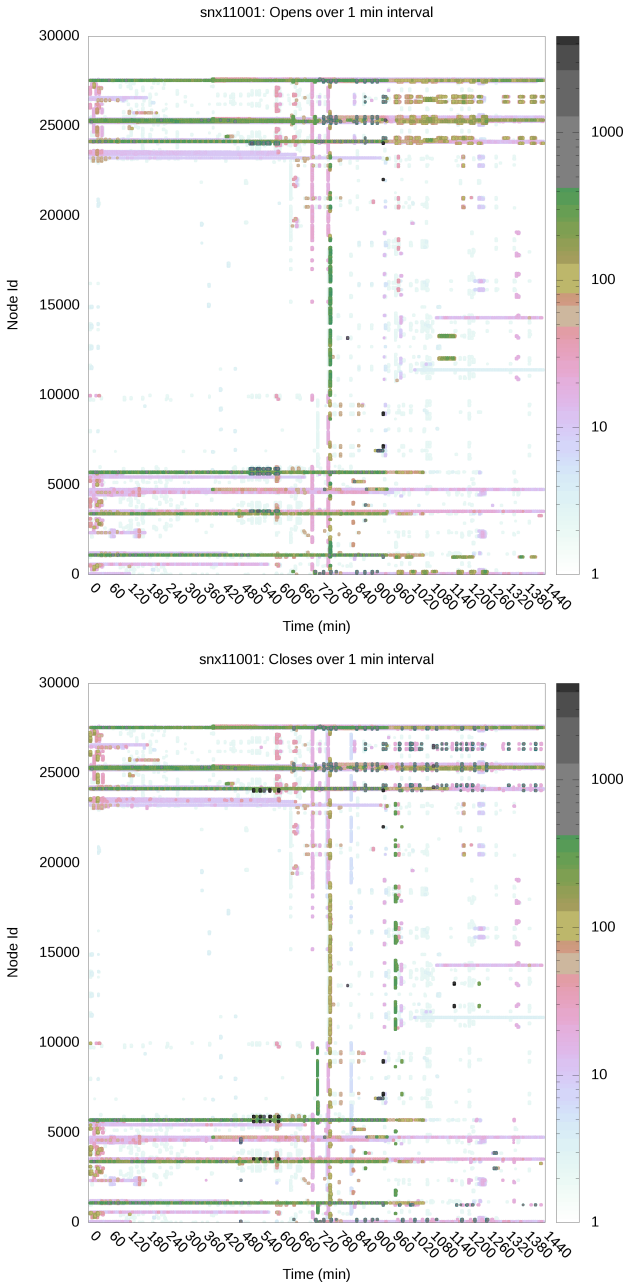

Figure 10. Lustre opens(top) and closes (bottom) on snx11001 which hosts the home file system.

B. Future Work

While the use of OVIS's LDMS framework has worked out well overall, the future work planned and described here will improve on several aspects of LDMS and the available HSN data.

Separate thread pool for connect: While the number of worker threads available for the various tasks of a `ldmsd` can be user defined at the time the `ldmsd` is started, this thread pool is shared for connection, collection, and storage. Thus if there are a significant number of nodes that are in an unreachable state, an aggregator `ldmsd`'s collection operation can be starved for threads. This happens when all threads in the pool are blocked waiting on connection timeouts while trying to set up connections to unreachable nodes. Thus we will be modifying `ldmsd` to enable creation of a separate (number will be user defined) thread pool for connection setup to mitigate the collector thread starvation problem.

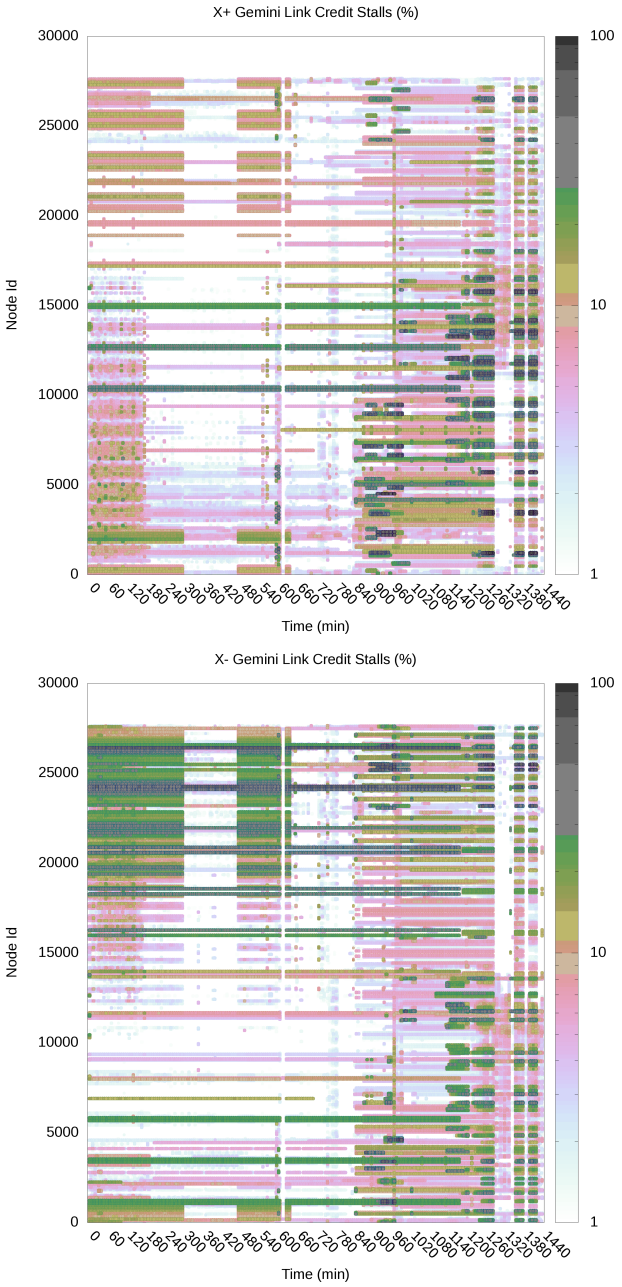

Figure 11. Link aggregated percent of time spent in credit stalls in X+ (top) and X- (bottom).

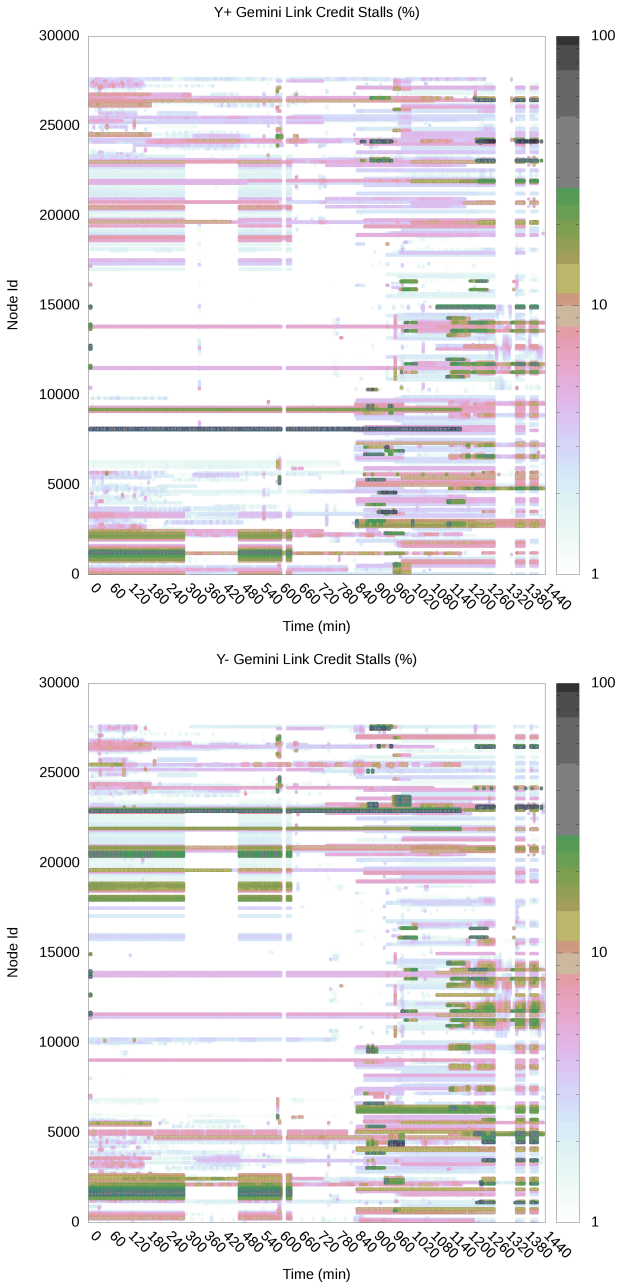

Figure 12. Link aggregated percent of time spent in credit stalls in Y+ (top) and Y- (bottom).

Post processing in storage plug-in: When we defined the original metric set we wanted to minimize the memory and CPU footprint of the sampler `ldmsd` in order to minimize impact on running applications. Thus the metric set is comprised mostly of raw counter values with some derived rates and percentage of total derived for HSN counters. When attempting to utilize the Lustre data, however, it became clear that having rate information for data other than just the HSN would make identification of outliers much faster. We have thus decided to incorporate the ability to produce derived data at the aggregator for storage either with the original metric set or as a separate set.

Export of relevant media type/speed information to nodes: In Section II-A we described how the HSN data is exposed via the *sys* file system. Though all the pertinent dynamic data is being exposed, static data such as what type of media (e.g. cable, mezzanine, backplane) and their associated maximum data rates is not exposed. This lack

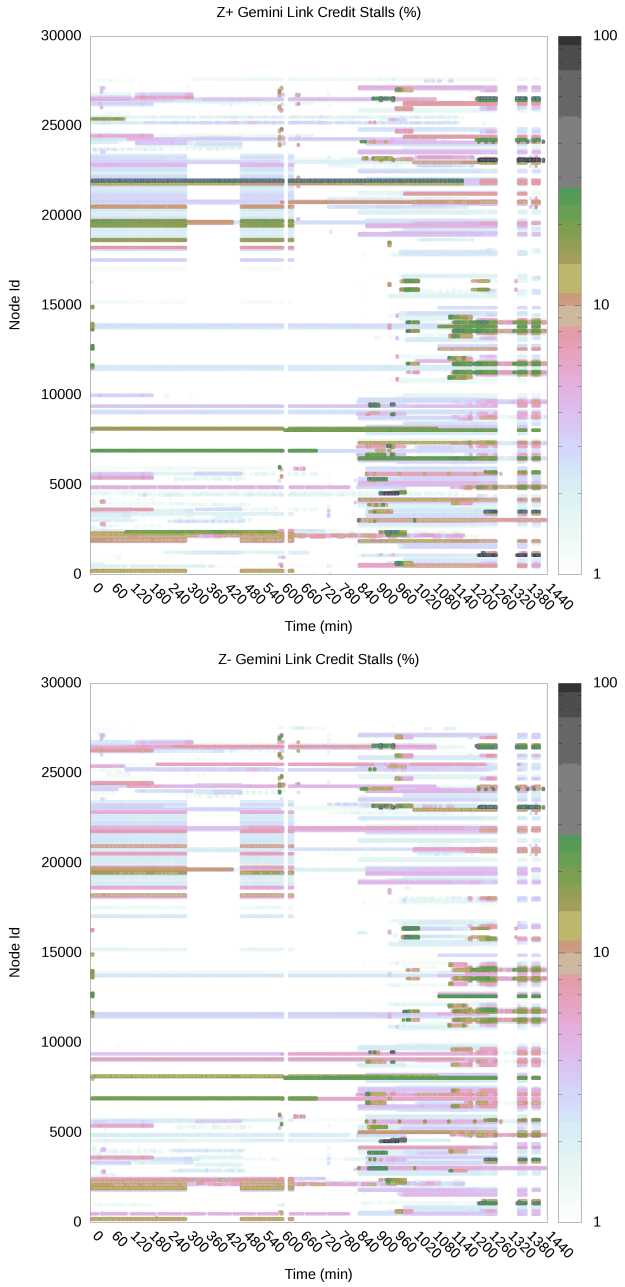

Figure 13. Link aggregated percent of time spent in credit stalls in Z+ (top) and Z- (bottom).

is what necessitated consolidation and inclusion of the *interconnect* data into the image (Section III-B). In order to remove the need to preprocess the *interconnects* file, which is different from system to system, and include it in the image we plan on incorporating this information into the `gpcdr` data or `rca-helper` information or similar.

Acknowledgments

The authors would like to thank: Gary Rogers and Ryan Braby of University of Tennessee at Knoxville for their involvement in installation and testing on the Blue Waters and its development system; and Jason Repik of Cray for assistance in image integration, installing appropriate system software, and access for testing on a Sandia's XK7 development platform.

References

[1] J. Brandt, T. Tucker, A. Gentile, V. Kuhns, and J. Repik, "High Fidelity Data Collection and Transport Service Applied to the Cray XE6/XK6," in *Proc. Cray User's Group*, 2013.

[2] "Ovis," http://ovis.ca.sandia.gov.

[3] "Blue Waters," https://bluewaters.ncsa.illinois.edu.

[4] "MPI for Cray XE/XK7 Systems," Titan Users and Developers Workshop (East Coast) and Users Meeting, January 2013. [Online]. Available: https://www.olcf.ornl.gov/wp-content/uploads/2013/02/MPI_MPT-HP.pdf

[5] Cray Inc., "Cray Linux Environment (CLE) 4.0 Software Release," Cray Doc S-2425-40, 2010.

[6] "PAL System Noise Activity Program," Los Alamos National Laboratory Performance and Architecture Laboratory (PAL), March 2014. [Online]. Available: http://www.c3.lanl.gov/pal/

[7] G. Bauer, S. Gottlieb, and T. Hoefler, "Performance Modeling and Comparative Analysis of the MILC Lattice QCD Application su3_rmd," in *Proc. 12th Int'l. IEEE/ACM Symp. on Cluster, Cloud, and Grid Computing*, 2012.

[8] R. Fiedler and S. Whalen, "Improving Task Placement for Applications with 2D, 3D, and 4D Virtual Cartesian Topologies on 3D Torus Networks with Service Nodes," in *Proc. Cray User's Group*, 2013.

[9] R. F. Barrett, C. T. Vaughan, and M. A. Heroux, "MiniGhost: A Miniapp for Exploring Boundary Exchange Strategies Using Stencil Computations in Scientific Parallel Computing," Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2012-10431, 2012.

[10] M. Deveci, S. Rajamanickam, V. Leung, K. Pedretti, S. Olivier, D. Bunde, U. V. Catalyurek, and K. Devine, "Exploiting Geometric Partitioning in Task Mapping for Parallel Computers," in *Proc. 28th Int'l IEEE Parallel and Distributed Processing Symposium, to appear*, 2014.

[11] D. A. Green, "A Colour Scheme for the Display of Astronomical Intensity Images," *Bulletin of the Astronomical Society of India*, pp. 289–295, 2011.