Deploying Server-side File System Monitoring at NERSC

Andrew Uselton
NERSC, Lawrence Berkeley National Laboratory Berkeley, CA 94720

May 7, 2009

Abstract

The Franklin Cray XT4 at the NERSC center was equipped with the server-side I/O monitoring infrastructure Cerebro/LMT, which is described here in detail. Insights gained from the data produced include a better understanding of instantaneous data rates during file system testing, file system behavior during regular production time, and long-term average behaviors. Information and insights gleaned from this monitoring support efforts to proactively manage the I/O infrastructure on Franklin. A simple model for I/O transactions is introduced and compared with the 250 million observations sent to the LMT database from August 2008 to February 2009.

1 Introduction

With the advent of petascale High Performance Computing (HPC) systems the gap between an HPC system's compute power and aggregate memory, on the one hand, and the best-case data rates of mass storage devices, on the other, grows wider than ever [10]. In order to maintain a balance between compute and storage, the storage side has become ever larger. This increased parallelism in the storage and I/O subsystem places an ever increasing burden on the file system software supporting it.

A parallel file system is a complicated combination of hardware and software. There are many ways for it perform below optimum and it can be difficult to determine what the optimum performance should be in the first place [7]. Measuring, analyzing, and understanding file system performance is challenging given the complex interplay of I/O system components.

The most common technique for measuring parallel file system performance is to run a benchmark application, such as IOR [8, 11], across many or all of the nodes of an HPC system. Simplified somewhat, an IOR test opens the file or files to be targeted in the parallel file system, writes a large amount of data from each task on the HPC system, reads all that data back in, and closes the file(s). IOR measures the time from the beginning of the first open to the end of the last close. That time, combined with the aggregate data transferred, gives a figure for the performance of the file system - often reported as a write and read performance numbers (measured in bytes per second). A large number of such tests at varying concurrencies and with varying parameters for transfer size, and such, can give an overall characterization of the performance of the file system, and is often summarized with *peak* write and read rates observed during testing.

It is not uncommon for such numbers to be published and for users of the HPC system to subsequently complain to system staff when their favorite application does not achieve those published peak rates. In 2007 the National Energy Research Scientific Computing (NERSC) center took initial delivery of the Franklin Cray XT4 supercomputer (described in Section 2) which uses Lustre [4] for its */scratch* parallel file system. NERSC staff have conducted performance tests on Franklin [3] and observed a peak write rate in the vicinity of $11GB/s$ and a peak read rate around $8GB/s$ for */scratch* []. NERSC sought to better understand the peak rates observed, as well as the departures from peak under some loads, and the apparent variability [2] of performance results under repeated testing. To that end, NERSC deployed Cerebro (Section 3) and the Lustre Monitoring Tool (LMT - Section 4). Cerebro [6] is a data transport infrastructure, and LMT [12] is a mechanism for observing and recording server-side performance statistics.

During a benchmark test it can be valuable to see a moment-by-moment summary of server data rates, and LMT provides mechanisms and tools for doing so (Section 5). Database query tools in LMT can retrieve past I/O patterns to produce graphs of data rate versus time (Section 6), which can assist with the analysis of I/O trouble reports. In addition to monitoring dedicated time tests and

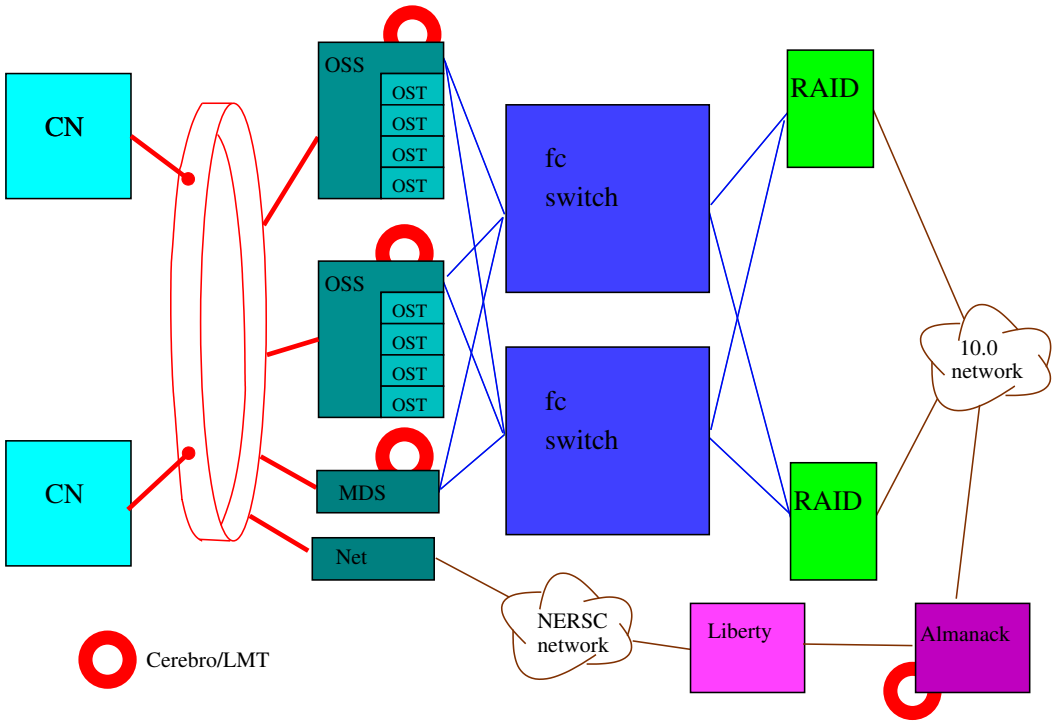


Figure 1: The Franklin Cray XT4 has a SeaStar2 interconnect linking compute nodes (CN) and service nodes (OSS, MDS, Net) in a torus (red links). I/O support is provided by RAID devices connected over $4Gb/s$ *fibrechannel* (blue). External connections to the wider NERSC network are over *ethernet* (brown). The Cerebro/LMT service resides on the I/O servers (MDS, OSS) and the database server (Almanack).

investigating reported problems, it is useful to review the I/O data rates observed over the course of a day (Section 7). It is possible to gain insights from the data even without being prompted by tests or trouble. Keeping track of the average behavior (Section 8) over time can reveal changes in the system work load. Finally, a statistical analysis of the observations collected over an extended period can provide insights into the I/O patterns on the system (Section 9), which can be compared against a simple model characterizing those patterns as governed by a Poisson distribution. This paper touches on all of the foregoing themes and concludes with a selection of additional activities that would be of interest to NERSC.

The effort to monitor I/O performance and behavior contributes significantly to the ongoing effort to manage the Franklin HPC resource in a proactive fashion.

2 Franklin

The Franklin Cray XT4 supercomputer (Figure 1) at the National Energy Research Scientific Computing (NERSC) center is a 9660 node system employing a quad-core 2.1 GHz AMD Opteron processor in each compute node (CN). The nodes communicate via a 3-D torus over the Cray SeaStar2 interconnect, which is a 6.4 GB/s bidirectional Hypertransport interface. In the Autumn of 2008 Franklin was upgraded to the quad-core architecture from a dual-core architecture, and the memory was doubled from $4MB$/node to 8. The data summarized in this report were gathered both before and after the upgrade.

An HPC-oriented Storage Area Network (SAN) file system needs the high throughput achieved from simultaneous access to a large number of disks. The throughput requirement must be balanced against ease of use, and a parallel file system is tasked with managing the data across all disks while presenting a single POSIX-complaint name space to the application programmer. Franklin employs the Lustre parallel file system for its temporary file space, and mounts it as the */scratch* file system. */scratch* was configured with:

20 Object Storage Servers (OSSs) These are Franklin service nodes with two external $4Gb/s$ *fibrechannel* links each and are responsible for managing bulk data objects for the file system.

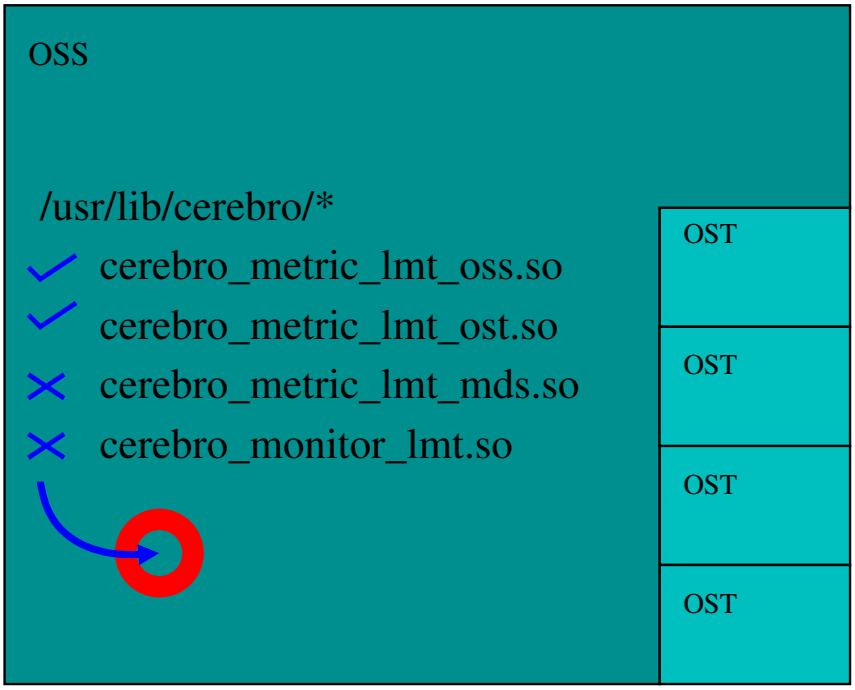


Figure 2: Cerebro is a lightweight, extensible daemon for data gathering. On the OSS it will attempt to load any library in the *<u>/usr/lib/cerebro</u>* directory. Since the OSS is not an MDS nor is it configured to be a *monitor*, two of the libraries exit immediately. The other two begin sending data via UDP packets managed by Cerebro.

4 Object Storage Targets (OSTs) per OSS These provide kernel services mediating access to the SCSI disk resources.

A $4TB$ Logical Unit (LUN) for each OST The RAID-based disk resource is presented to the node as a SCSI disk device.

1 Meta Data Server (MDS) This is another service node and provides object location and name space services for the file system.

Additional resources provide a Lustre-based "home" file system. The Franklin I/O resources were substantially expanded in April of 2009, but the results reported here are all from before the update.

Figure 1 depicts the Franklin Supercomputer with two of the 9660 CNs on the left connecting to the torus along with two of the 20 OSSs and the MDS. The Lustre servers (OSSs, MDS) run a data collection daemon called Cerebro and use a set of Cerebro plug-in modules collectively named the Lustre Monitoring Tool (LMT). Those daemons communicate via a *network* service node, which has a $10Gb/s$ link to the wider NERSC network. A firewall called Liberty controls access to a private "10.0.x.y" network that is dedicated to monitoring and instrumentation data collection. The Almanack database server on the private network also runs a Cerebro daemon and saves the LMT data in a database for the */scratch* file system. The next sections describe Cerebro and LMT in more detail.

3 Cerebro

Cerebro (Figure 2) is a lightweight user-space application written in C. It resembles Ganglia [1] in that it is primarily a data communication facility with a few rudimentary functions intrinsic to its own operation. One core function is a *heart beat* mechanism that the data collector (the daemon running on Almanack) can use to determine if a node is up or down. Some plug-in service activities are also invoked at each heart beat interval.

Most of the functionality available through Cerebro is provided by plug-in modules. A module can be (*i*) a *metric* module - gathering data to be forwarded, (*ii*) a *monitor* module - accepting gathered data and doing something useful with it, or (*iii*) an *event* module - recognizing and acting upon some condition. The modules can themselves be very lightweight. Cerebro recognizes and loads modules because they are present in a specific modules directory, for example */usr/lib/cerebro*. Any *.so* file in that directory is loaded at Cerebro start-up. The LMT modules

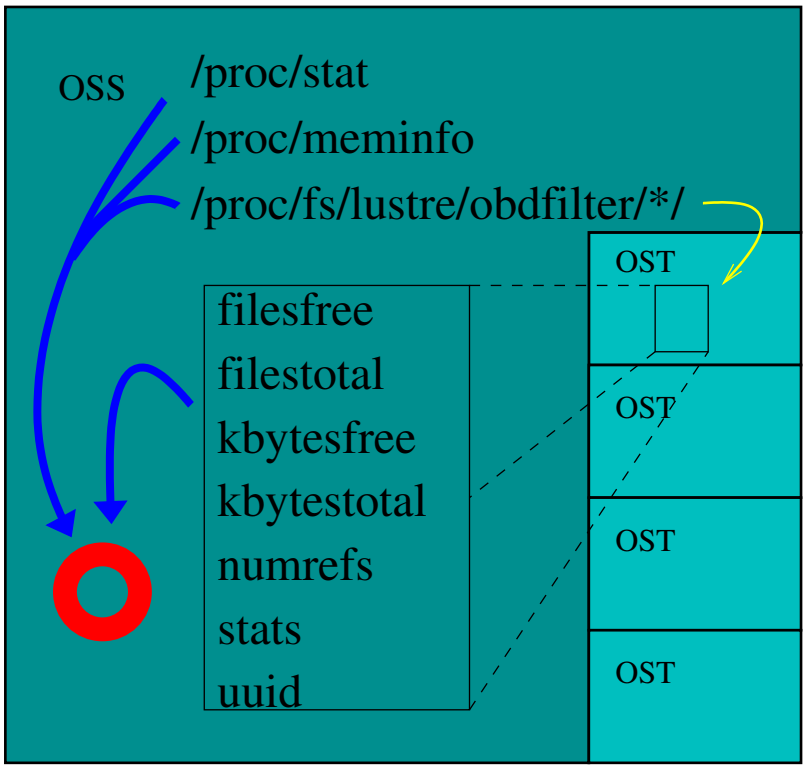


Figure 3: The LMT metric modules gather Lustre statistics from *proc*.The OSS module gets data from *stat* and *meminfo*, and the OST module gets data from the files under the *obdfilter* directory, which has a separate sub-directory for each OST.

are just such libraries and are all written in C. This can be a significant saving compared to the overhead required to run a data collection mechanism based on a scripting language. The downside is that programming to the Cerebro module API is a more complex development task.

A configuration file (*/etc/cerebro.conf*) determines if a particular instance will only run metric and event modules (a *speaker*), only run monitor modules (a *listener*), or both. The configuration file determines the destination for any outbound messages a speaker will send, and the source for any inbound messages a listener will accept. There are generally many more speakers than listeners. A given speaker can send to more than one listener and vice-versa. A Cerebro daemon that is both a speaker and a listener acts as a relay for Cerebro messages. Sources and destinations can be specified as multicast.

Cerebro comes with many other modules besides those for LMT data collection. There are modules for monitoring network interfaces, memory utilization, swap, and even a module for automating the configuration of the *genders* cluster management software. The rest of this report will confine itself to LMT data collection.

4 The Lustre Monitoring Tool (LMT)

On the OSSs and MDS, Lustre is implemented as a set of kernel services, and those services maintain an extensive array of statistics in the */proc* file system*. The LMT modules (Figure 3) know how to harvest this data and pass it along to Cerebro. At start-up, Cerebro loads all the modules it finds including the LMT modules. Part of the Cerebro module API includes a start-up function for the module. On each server only the relevant modules end up loaded. As an example, consider an OSS node as the daemon starts: Cerebro will load every module in */usr/lib/cerebro*, including the LMT module for MDS data acquisition, but as part of the MDS module start-up process the module recognises that this is not an MDS and that module exits immediately without any error.

Cerebro will interact with a metric module in one of two ways. Either it invokes the module *get_metric_value* method with each heartbeat interval, or it invokes the

*I was recently told [9] that the */proc* interface may be deprecated in favor of a Lustre internals API.

get_metric_thread method which allows the module to start up a persistent activity of its own. The metric module for the OSS employs the former technique, and the OST module uses the latter.

LMT only collects a few statistics from an OSS. The metric module for OSS statistics is *cerebro_metric_lmt_oss.so*. It opens */proc/stat* and gathers values for cpu utilization - summing the values for *usr*, *nice*, *sys*, *iowait*, *irq*, and *softirq* - leaving out only *idle*. It keeps around the previously sampled value of these and uses the difference to calculate a *percent CPU utilization* value.

$$\text{usage} = \text{usr} + \text{nice} + \text{sys} + \text{iowait} + \text{irq} + \text{softirq}$$

$$\text{total} = \text{usage} + \text{idle}$$

$$\text{CPU utilization} = 100 \times \Delta(\text{usage})/\Delta(\text{total})$$

A similar calculation with the contents of */proc/meminfo* gives a *percent memory utilization* value. The OSS metric module passes a message to Cerebro with the foregoing values, along with the host name, and the Cerebro protocol version, as a tuple: (*ver*;*host*;*cpu*;*mem*). The result resembles the following:

```
1.0;nid04187;4.990020;39.303989
```

Cerebro transports this message to the daemon running on Almanack which passes it, in turn, to the LMT monitor module *cerebro_monitor_lmt.so*. The monitor module connects to the MySQL server at start-up and reads a configuration file *lmtrc* to determine the names of the databases to connect to. Each file system gets a separate database, though a given OSS can participate in more than one file system. At start-up the module queries the databases for their specific details. In particular, a table called *OSS_INFO* lists the host names of the OSSs for that file system. Those names are gathered in a hash table (in Cerebro) and associated with a link back to each file system (database) a given host participates in.

When an OSS-related message arrives: it is parsed, the hash for the host is identified, and the new data is added to the *OSS_DATA* table in each appropriate database. The LMT database keeps all entries indefinitely, which is different from the *round-robin database* strategy employed in Ganglia and Cacti [5]. In the eight months since beginning operation the LMT database has collected approaching 250 million OST samples and a corresponding number of MDS samples. It is on track to gather about $100GB$ of LMT data a year and is provisioned for three years.

The metric module for OST statistics is *cerebro_metric_lmt_ost.so*. At module load time this library spawns a thread for each OST on the OSS. The thread for an OST collects data and forms a message to pass to Cerebro every five seconds. Note that this is not part of the heart beat process. The five second interval is "hard coded" into the module. An OST message carries the following values:

Protocol version The version string for the module.

Host name The name of the service node on which the OST data acquisition thread runs. There will be more than one OST on the node, in general.

UUID The universally unique identifier for the OST.

Bytes read This is a 64 bit quantity for the total number of bytes read from the OST since the last reset. The counters are usually only reset at boot time. This value and the next come from the *stats* file in the directory */proc/fs/lustre/obdfilter/***ost**, where there is a separate directory for each **ost**.

Bytes written A 64 bit value for total bytes written to that OST since reset.

Kbytes free The free space on the device managed by the OST, measured in KB.

Kbytes used The amount of space in use on the device.

Inodes free The number of *inode* objects available in the file system on the device managed by the OST.

Inodes used The number of *inode* objects in use.

The monitor module running on the Almanack database server queries each file system's *OST_INFO* table to get the list of *UUID* values for that file system. In this case they had better be unique, since the UUID is used as the hash value for the OST[†]. When a message arrives, the hash for that UUID is retrieved identifying the file system that gets the new information. An entry with the new values goes into the *OST_DATA* table.

The metric module for the MDS statistics (Table 1) is *cerebro_metric_lmt_mds.so*. That module operates in the same mode as the OSS: At each heart beat it harvests its assigned data from */proc* and passes a message to Cerebro, which puts the new values into the appropriate database based on the UUID for the MDS. The *MDS_INFO* table identifies the MDS for the database. In the case of MDS data the statistics gathered are quite extensive. Some data ends up in the *MDS_DATA* table, but only the tuple (*kbytes free*, *kbytes used*, *inodes free*, *inodes used*). There are many other values that arrive with an MDS message. The *OPERATION_INFO* table lists all the kinds of operational data the MDS provides. For each value, a tuple (*timestamp*, *op*, *value*) gets put in the *MDS_OPS_DATA* table. Table 1

[†]An early configuration error on Franklin had the same names - *ost01*, *ost02*, ... - on every file system. The result was that data from multiple file systems accumulated in each database. Worse, since each UUID is unique in the hash table, the values from the multiple file systems were in a "race" to see which value would be recorded in any particular interval.

mysql> select * from OPERATION_INFO;

OPERATION_ID	OPERATION_NAME	UNITS	OPERATION_ID	OPERATION_NAME	UNITS
1	req_waittime	usec	26	mds_getattr_lock	usec
2	req_qdepth	reqs	27	mds_close	usec
3	req_active	reqs	28	mds_reint	usec
4	reqbuf_avail	bufs	29	mds_readpage	usec
5	ost_reply	usec	30	mds_connect	usec
6	ost_getattr	usec	31	mds_disconnect	usec
7	ost_setattr	usec	32	mds_getstatus	usec
8	ost_read	bytes	33	mds_statfs	usec
9	ost_write	bytes	34	mds_pin	usec
10	ost_create	usec	35	mds_unpin	usec
11	ost_destroy	usec	36	mds_sync	usec
12	ost_get_info	usec	37	mds_done_writing	usec
13	ost_connect	usec	38	mds_set_info	usec
14	ost_disconnect	usec	39	mds_quotacheck	usec
15	ost_punch	usec	40	mds_quotactl	usec
16	ost_open	usec	41	mds_getxattr	usec
17	ost_close	usec	42	mds_setxattr	usec
18	ost_statfs	usec	43	ldlm_enqueue	usec
19	ost_san_read	usec	44	ldlm_convert	usec
20	ost_san_write	usec	45	ldlm_cancel	usec
21	ost_sync	usec	46	ldlm_bl_callback	usec
22	ost_set_info	usec	47	ldlm_cp_callback	usec
23	ost_quotacheck	usec	48	ldlm_gl_callback	usec
24	ost_quotactl	usec	49	obd_ping	usec
25	mds_getattr	usec	50	llog_origin_handle_cancel	usec

50 rows in set (0.03 sec)

Table 1: The OPERATION_INFO table lists all the kinds of operational data that LMT monitors on the MDS. Every MDS message received on Almanack contains a value for each of these. The tuple (*timestamp*, *op*, *value*) for each is entered into the MDS_OPS_DATA table.

shows the names of the MDS operation values collected. It is beyond the scope of this report to give a detailed semantics for each MDS "op", but the names correspond closely to the */proc* entries they were harvested from. Consult the Lustre documentation [**?**] for more details.

5 Real-time Data Observation

One use for the Cerebro/LMT data collection infrastructure is to monitor a file system in *real time*. The LMT code base includes three Java-based programs for such monitoring: *lstat*, *ltop*, and *lwatch*. The two command line utilities *lstat* and *ltop* function analogously to the UNIX utilities they were named for. Both sample the current latest values in the LMT database and present the results, optionally updated periodically. The GUI-based *lwatch* "Lustre Dashboard" (Figure 4) is a comprehensive interface for displaying real time data on multiple file systems. It has a separate panel for the data on each of the MDS, the OSSs, and the OSTs. To compose the OST panel, for instance, the dashboard queries the database periodically for the current *read_bytes* and *write_bytes* values for each OST. It takes the difference from one sample to the next to present a data rate for each OST and sums all those values to present a file system wide data rate.

This can be useful for dedicated time testing, where immediate feedback can help guide the testing process. For example, an IOR test may report upon completion that the write rate was $2GB/s$ on a file system that ought to do $10GB/s$. By watching the dashboard one may distinguish between a test that shows uniform $2GB/s$ for the length of the test and a test that shows $10GB/s$ at first and then falls off to near zero as one or a few OSTs straggle in.

During regular production time this can also help guide application development and testing. An application that seems to be performing poorly may just be suffering from contention with other jobs on the system. If the dashboard shows the file system at or near peak, then the application probably isn't going to do much better until other I/O activity abates. If the dashboard shows little I/O activity, on

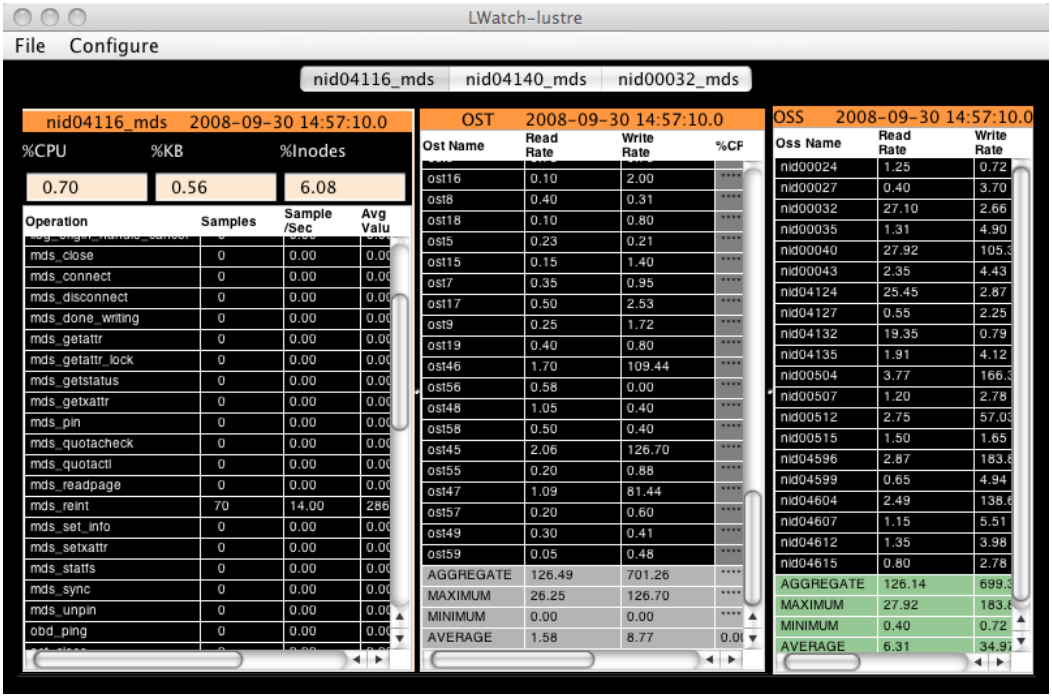


Figure 4: The *lwatch* "Lustre Dashboard" is a Java-based GUI program for monitoring file systems in real time.

the other hand, it may indicate the application itself is not presenting its I/O in the most efficient fashion.

Even when no specific test is going on it can be useful to watch the file system and keep an eye out for apparent problems before they rise to the level of a system wide outage. The OSS panel will turn a particular OSS row yellow if its CPU utilization goes above a threshold value. If an OSS flashes yellow occasionally that is probably indicative of normal operation, but if one OSS stays yellow that may indicate a problem - perhaps a stuck thread.

There is a wealth of data gathered for and preserved in the LMT database. After gathering data for nine months there is value in mining the data for additional insights about the file systems being monitored. The next sections exploit the OST *read_bytes* and *write_bytes* values to analyze recent experience on the Franklin */scratch* file system.

6 Bulk I/O Transport

The data rates that LMT reports can be a valuable insight into the performance of the file system, on the one hand, and the performance of particular applications, on the other. Some caution is necessary when interpreting the data rates reported, though. Data sent to the LMT database on Almanack from the OSTs arrives asynchronously, and without any specific effort to coordinate across all the data sources. The monitor module periodically gathers whatever data it has on hand and puts it in the database. When the dashboard or other tools aggregate such data over all the OSTs there is a tacit assumption that all the updates marked with the same *timestamp* value are actually simultaneous on the system. In practice, the values may not be coordinated more closely than one or two sample intervals, which would correspond to five or ten seconds in the Franklin */scratch* file system database. One should keep this uncertainty in mind when looking at such very short time scales.

Another difficulty with the way the data is gathered is that there can be missing updates. The data packets are sent via UDP, which does not guarantee delivery. In practice, there is usually less than one percent loss, but one must still consider carefully how to treat those gaps. The *read_bytes* and *write_bytes* values from the OSTs are monotonically increasing counters, so a lost packet does not lead to missed OST activity, but only the loss of time resolution for when that activity occurred. When calculating a data rate in the presence of such lost packets it is necessary to accommodate the gap, though. The tools developed by the author assume that any change in a counter happened uniformly over the entire interval from the last received update. This also introduces a slight uncertainty to the timing of the reported values.

Figure 5 shows the LMT data for *read_bytes* and *write_bytes* during a sequence of four IOR tests run dur-

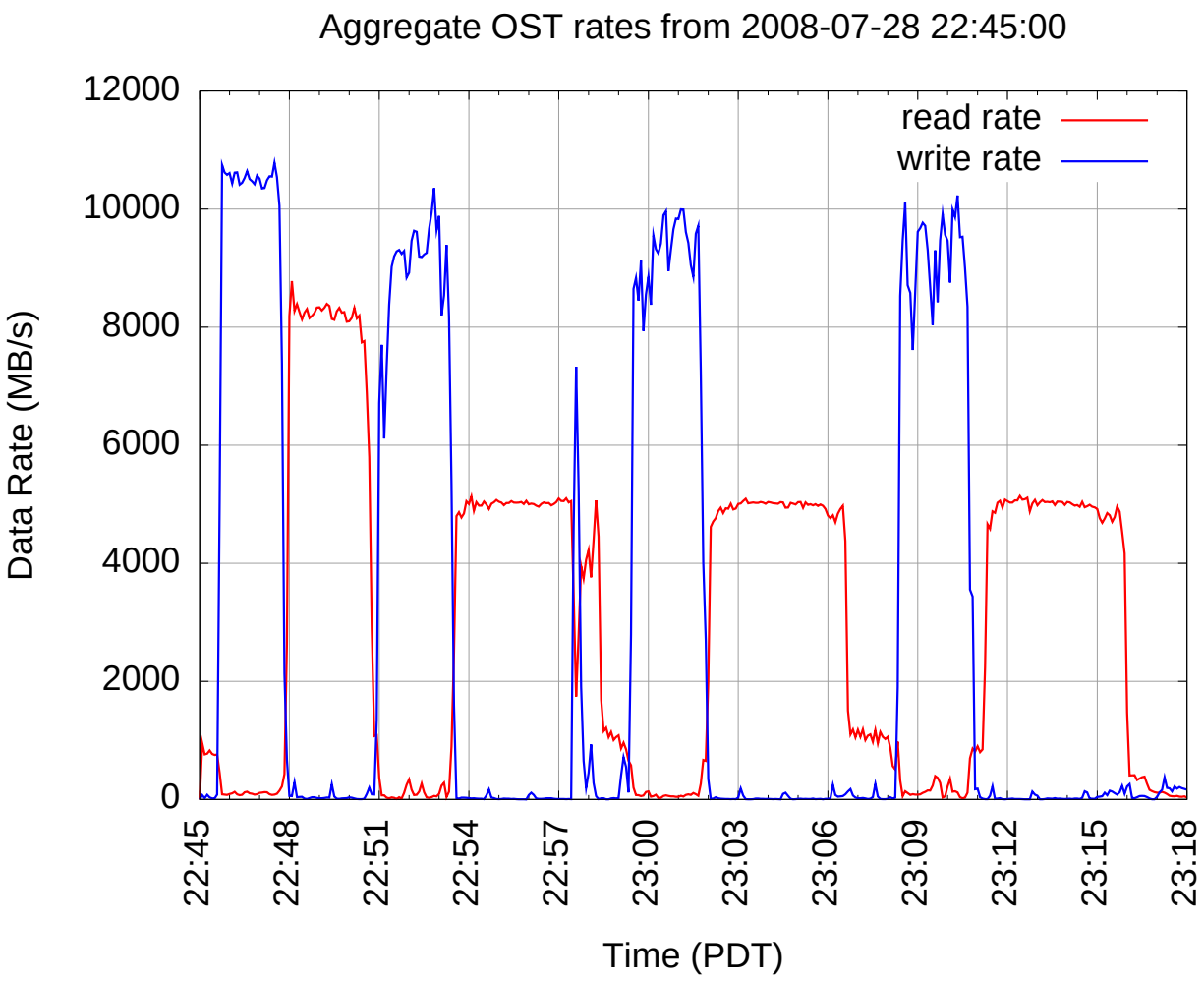


Figure 5: Four IOR tests were run during "dedicated time", i.e. there was no other activity on the system. Each of 1024 tasks wrote and then read $1.27GB$ during each test. The *x*-axis is wall-clock time, and the *y*-axis is the aggregate data rate across all OSTs.

ing dedicated time on Franklin's */scratch* file system. The *x*-axis shows wall-clock time and the *y*-axis gives the read and write rates for the OSTs aggregated into a pair of numbers as in the dashboard in Figure 4. The IOR tests proceed one after the other with fewer than five seconds between the end of one and the start of the next. Each test writes $1.27GB$ from each of 1024 tasks and pauses at a barrier waiting for the last to finish before proceeding. When all the writes are complete, the test reads the $1.27GB$ back in, and again pauses at a barrier waiting for the slowest to catch up. This repeats four times. In this series of tests the I/O libraries were being varied from one test to the next. The first test uses POSIX I/O to a separate file for each task - this is a *file-per-process* test. The second test uses POSIX I/O to a single shared file, where each task has an exclusive range of offsets in the file - this is a *single-file* test. The third and fourth tests are also *single-file* tests and use MPI-I/O library calls and HDF5 I/O calls respectively.

During the write phase of each test there is almost no read traffic and vice versa, and that is because the tests were run on a dedicated machine. The one significant exception is during the read phase of the second test (POSIX *single-file*) when it appears that some other activity on the system wrote for a brief but noticeable interval. An eyeball estimate might suggest about 100GB was written during the interval. It disrupted the read traffic enough to show a noticeable decrease in the delivered I/O read rate. Also note that there is an interval at the end of the second and third tests, during the read phase, when the read rate falls off sharply. It would appear that there were a few slow

OSTs that had difficulty finishing their assigned I/O in a timely fashion. An examination of LMT data during or after the test can reveal problems with I/O. This might lead to an investigation of the choice of I/O model, perhaps the object placement across the OSTs was not uniform, or it might call into question the hardware. The details provided by the LMT data can guide the focus of such an investigation. By contrast, the data rate that IOR reports at the end of its run is not nearly as informative.

The foregoing observation about the read rates is indicative of a wider issue with applications performing this kind of parallel, coordinated I/O. When you look at the I/O performance of a particular job you are seeing the worst case scenario from among the individual I/O tasks in the job. If the I/O completion time for task x among N tasks is t_x, then the job completion time will be $T_N = max_{x \in N} t_x$. If the t_x vary about some mean with a Gaussian shaped distribution then the expected value of T_N will depend on the width of that Gaussian and therefore on the size of the job N. For a weak-scaling series of such tests, with the amount of work growing in proportion to the size of the job, one would expect the delays due to such stragglers to increase with N, and this increase is just exactly the sort of delay that appears to be occurring in the second and third tests in Figure 5.

7 Incident Investigation

Another useful piece of infrastructure packaged with the LMT source code is a Perl module, *LMT.pm*. That module mediates the connection to the LMT database and facilitates queries of various sorts. In particular, a query for all the OST observations during an interval will deliver a set of tuples (*timestamp*, *OST*, *read_bytes*, *write_bytes*). A Perl script, *osts.pl*, written by the author and using the LMT.pm module, gathered and manipulated the OST data and produced the graph in Figure 5. That script is quite general and very handy. A daily summary of all file system activity over a 24 hour period is now a standard part of the file system monitoring at NERSC.

Figure 6 is the graph for a 24 hour period during which the read rate was very high for an extended period. That read activity was unusual enough to prompt NERSC user services personnel to investigate. It turns out that a chemistry application was being used intensively and caused the unusual read activity. In this case the LMT data was not used to monitor a specific controlled test or production time application. Instead, the data itself was unusual enough to warrant further investigation. Needless to say, if LMT weren't in use or weren't being tracked closely the activity would have gone unnoticed.

8 Historical Review

The surprisingly high reads in Figure 6 came to light in September of 2008. Figure 7 gathers the average observed data rates for reads and writes over the course of each day and plots the daily average over the seven months from August to February. The high reads of Figure 6 contributed significantly to the overall average in the late Summer. It is clear that the load on the file system evolved during that time, sometimes dominated by reads, less often by writes.

9 A Poisson-based Model for I/O Transactions

The LMT database on Almanack has a value for the amount read and written at each five second interval for the entire period from August of 2008 to the present. This has now amounted to nearly 250 million samples. It is instructive to look at the statistics governing those samples, and to compare the observations to a simple model. Imagine that the data being sent to each OST occurs as a sequence of independent transactions of M bytes each, and assume further that such a events are random and governed by a probability distribution such that the average expected number of such transactions arriving at any particular OST in any particular 5 second sampling interval is λ. The scenario described is that for a Poisson distribution, where the probability of k such arrivals in a given interval is:

$$f_\lambda(k) = \frac{\lambda^k e^{-\lambda}}{k!}$$

If the data arriving at OSTs fit this simple model, and if an OST could always actually handle all the data arriving, then the N samples in the Almanack database should form a distribution $C(m)$ that counts the number of occurrences of m bytes moved for the possible values of $m = k \times M$:

$$C(m) = N \times f_\lambda(\text{int}(m/M))$$

Figure 8 shows two Poisson distributions with possible values for λ and M given $N = 250M$. Note that the figures are drawn on a semi-log plot. For values of k much larger than λ the probability goes to zero quickly, so a semi-log plot can show more of the structure of the distribution. Both sets of values were chosen so the distribution has some samples out near $2GB$. In Figure 8(a) $\lambda = 2$ - one would expect two transfers in any five second interval. The transfers are $M = 125MB$, and the amount delivered is quantized in units of M, since the Poisson model only contemplates an integer number of arrivals. With a small λ the distribution has a peak near the *y*-axis (the mode is always λ), and the curve for $m > \lambda M$ looks almost like a straight line in the semi-log plot.

Figure 8(b) has a much larger λ and correspondingly

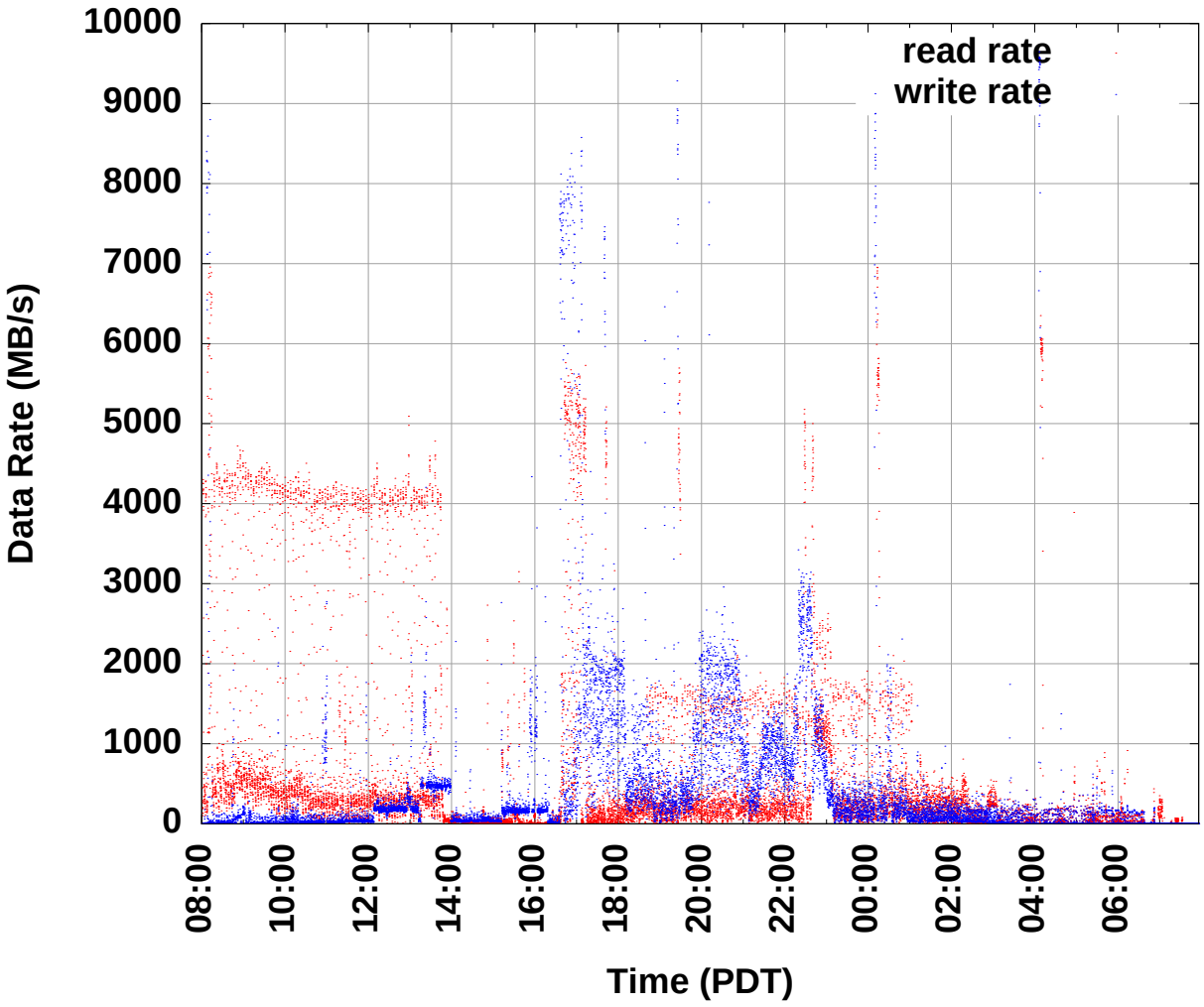


Figure 6: LMT data for a 24 hour period shows high read activity. The *x*-axis is wall-clock time, and the *y*-axis is the aggregate read (red) and write (blue) data rate across all OSTs for the */scratch* file system. Each dot represent the data transferred in a five second interval.

smaller M. This models transactions that are smaller and more frequent. Recall that the amount of data is quantized in M in this model, so with a smaller M there are many more "steps" in the distribution.

Figure 9 shows the distribution of read and write activity for the OSTs for seven months of observations - nearly $250M$ observations. The distributions do not perfectly match either of the distributions from Figure 8. This is not surprising considering the simplicity of our model. The read distribution, in particular, has a slight increase in counts as m approaches $2GB$, then the distribution goes quickly to zero. The model assumed that an OST could always accommodate whatever data was arriving, and we know that is not true. There is indeed a limit of around $400MB/s$ for data going to or coming from an individual OST. In the five second interval this limit gives a maximum of about $2GB$ moved. If more than $2GB$ arrived in that interval then the remainder of the transfer will occur in the next interval, thus the probability for $2GB$ transfers is the cumulative probability of any arrival greater than $2GB$ resulting in a slight increase in probability near the cut off. Call this the "ski jump" effect.

Another deficiency of our model is that during real I/O transactions are not all quantized at a single value for M. On the contrary there will be a separate probability distribution for each value of M. Thus λ changes for different values of M, and the amount arriving in an interval must be summed over all the distributions for the various values of M. Furthermore, the limit on the maximum amount that can be transferred is itself dependent on the size of the

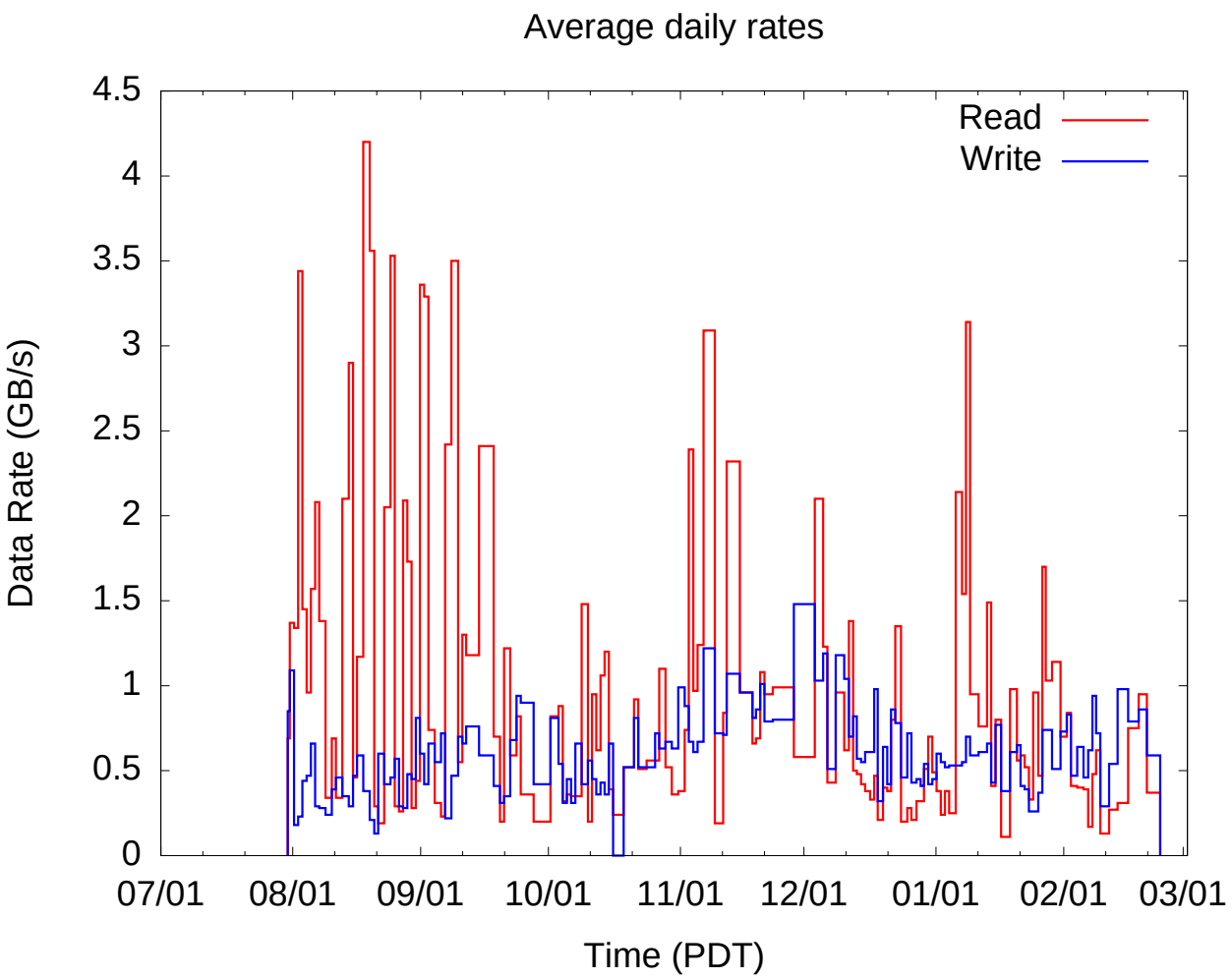
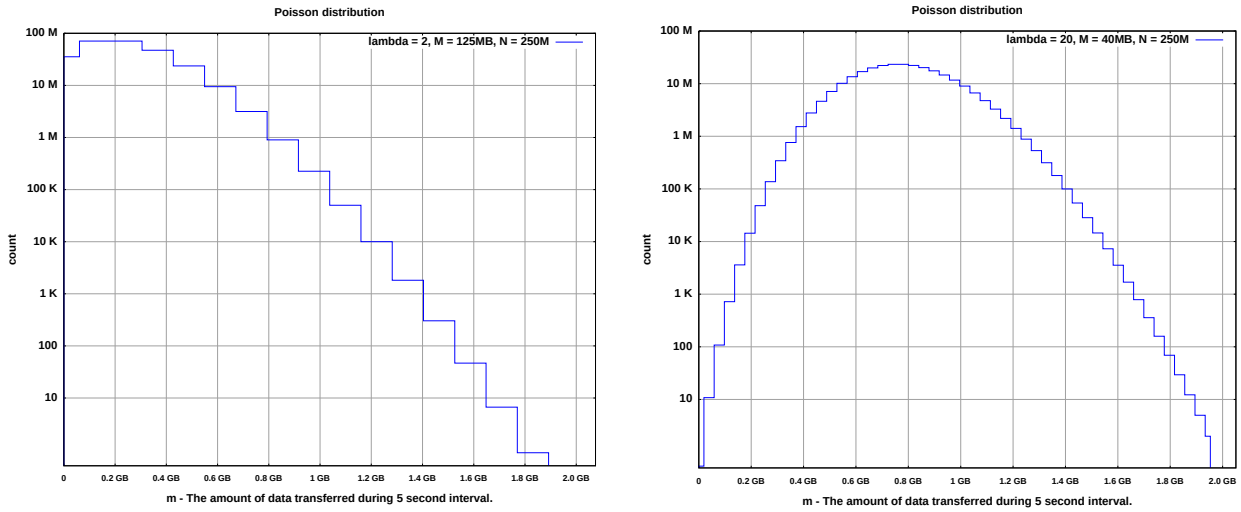


Figure 7: For each day the average read rate and average write rate can be calculated from the LMT data. Reads appear to dominate in the Summer and occasionally after.

transactions. There appears to be a second "ski jump" at around $500MB$, which may indicate that under some circumstances that is the maximum amount that can be transferred. One speculation is that during a journal commit for the underlying file system on the device operated by the OST, the head movement is much larger and the throughput lower.

The observed distributions of Figure 9 resemble the Poisson distribution with small λ in that the mode is close to the y-axis, but resemble the one with large λ in that the amounts transferred are obviously not quantized in large increments. The idea that M varies and has a small λ for each of a range of values of M would seem plausible.

It is apparent from the distribution (and from Figure 7) that more data is being read than written. It is also apparent that the OST was more likely to be unable to handle all the requests for reads as compared with the requests for writes. That is, the "ski jump" is much more pronounced for reads. In the case of the $500MB$ "ski jump" the writes seem more pronounced. Finally, there are actually three levels of quantization in the I/O path. The first is the amount a client seeks to transfer in a given I/O call - this may range from one byte to the all of memory on a node, the second is the size of the individual RPCs used by Lustre to carry out the transfer, and the third is the number of client nodes collectively participating in I/O. Lustre attempts to make RPCs all $1MB$ and there are */proc* entries that report the distribution of RPC sizes. In the case of collective I/O we have a violation of our Poisson assumption that transactions are independent and random. Nevertheless, the simple Poisson model is a useful way to begin characterizing the I/O patterns of OSTs.



(a) $lambda = 2$

(b) $lambda = 20$

Figure 8: A Poisson distribution models the amount of data arriving at an OST. The function is only defined for integral values of $k = m/M$, thus the stair steps. The *x*-axis is the amount transferred m in a five second interval. The *y*-axis gives the proportion of 250M samples that would report m. (a) With a smaller λ and larger M the distribution appears to fall off uniformly in this semi-log plot. (b) With a larger λ the chance of very little data arriving in the given interval is small, and the correspondingly smaller M makes the individual stair steps narrower.

10 Conclusion

The decision by NERSC to deploy the Cerebro/LMT infrastructure has enabled a wealth of insights into the behavior of Lustre on the Franklin Cray XT4. The server-side data enriches our understanding of the performance results reported by benchmarks like IOR. It provides feedback on the prevailing "weather" during code development or performance analysis. A periodic review of the data can identify interesting or anomalous use patterns on the file system. Detailed statistical analysis of the data collected provides insight into the file system I/O load.

The work reported here is the beginning of a larger project in which NERSC would like to further exploit Cerebro/LMT and further develop its capabilities. A detailed analysis of the metadata server's performance would be valuable. There are interesting file system statistics that LMT does not currently exploit, including RPC statistics and server-side client statistics. Currently, LMT is a network "bad citizen" in that it sends frequent small packets. It would be worthwhile to generalize the five second cycle time to be configurable, and it should be possible to gather several observations into one larger packet sent less frequently.

It would be interesting to develop a detailed model of the I/O transactions that closely matches the observed I/O distribution. If such a model could be uniquely determined from the data it would be a powerful tool for identifying file system use patterns. Extending both the model and the analysis to cover the time since the I/O subsystem upgrade will give a detailed characterization of the effect of the upgrade. We continue to develop data mining, visualization, and data presentation tools both for detailed analysis of the file system and for end-user feedback.

11 Acknowledgments

The author thanks Eric Barton, Andreas Dilger, and Isaac Huang of Sun, Nick Henke and Steve Luzmoor of Cray, and Brian Behlendorf of Lawrence Livermore National Lab for insight into the Lustre file system and many useful discussions. Thanks also go to the authors of Cerebro (Al Chu) and LMT (Herb Wartens), both at Lawrence Livermore National Lab, without whom this work would not have been possible. Many staff at NERSC provided insight, assistance, and encouragement for the work presented here. The author would especially like to thank Katie Antypas, Shane Cannon, Nick Cardo, Jon Carter, Wendy Lin, John Shalf, and David Skinner. This work was supported by the ASCR Office in the DOE Office of Science under contract number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under contract No. DE-AC02-05CH11231.

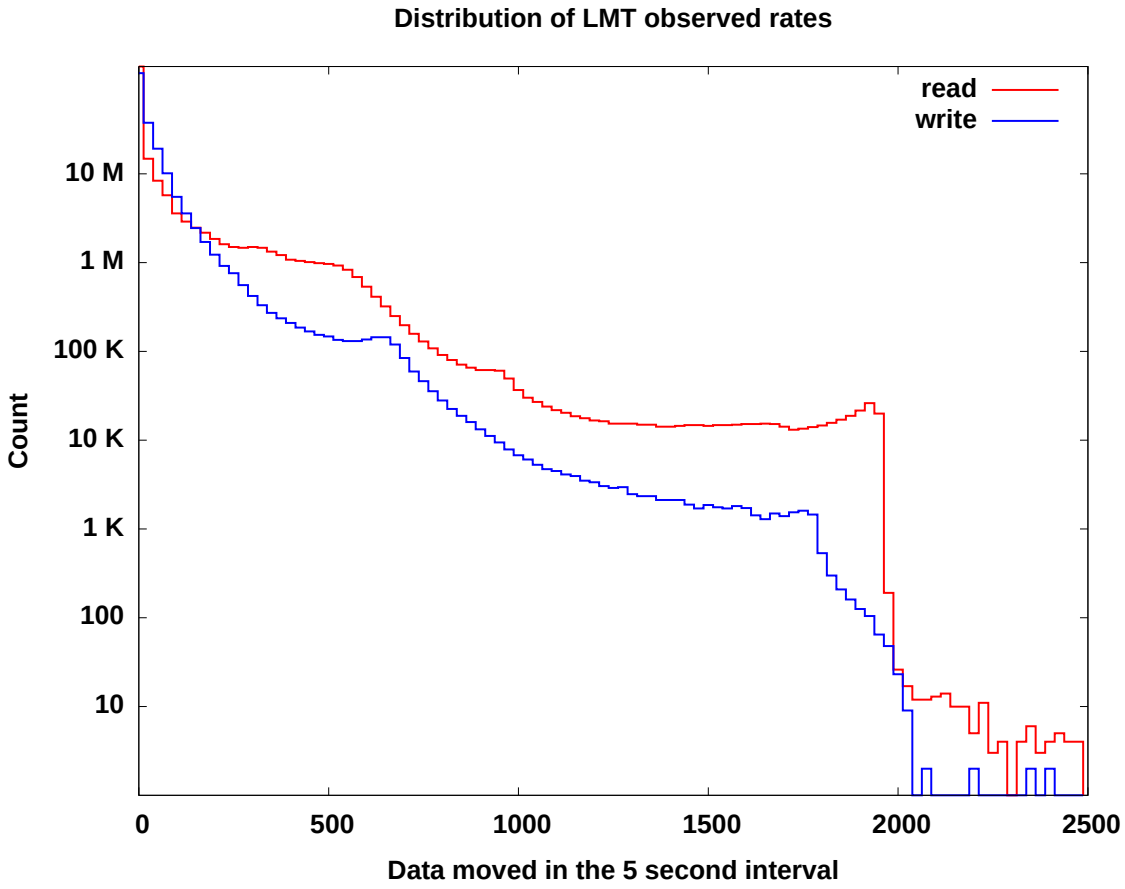


Figure 9: The OSTs report bytes read and bytes written in each five second interval since August. The *x*-axis gives the amount transferred, and the *y* axis gives the number of observation of that amount. The fact that the mode is close to the *y*-axis argues for a Poisson distribution with small λ, but the fact that the distribution varies with small changes in the amount transferred argues for a small M and thus a large λ. It is likely that this distribution represents the cumulative effect of having many values of M each with a small λ. The "ski jump" at $2GB$ is the result of the limit on OST capacity. If CNs attempt transfer more than $2GB$ then the OST still only transfers $2GB$.

References

[1] Ganglia. `http://ganglia.info/`.

[2] K. Antypas. Franklin Performance Monitoring. `https://www.nersc.gov/nusers/systems/franklin/monitor.php`.

[3] K. Antypas, J. Shalf, and H. Wasserman. Nersc-6 workload analysis and benchmark selection process. In *LBNL Tech report 1014E*, 2008.

[4] P. Braam. File systems for clusters from a protocol perspective. In *Proceedings of the Second Extreme Linux Topics Workshop*, Monterey, CA, June 1999.

[5] Cacti, the complete rrd-based graphing solution. `http://www.cacti.net/`.

[6] A. Chu. Cerebro. `http://sourceforge.net/projects/cerebro`. Developed at Lawrence Livermore National Lab.

[7] I/O Tips. `http://www.nccs.gov/computing-resources/jaguar/debugging-optimization/io-tips/`.

[8] IOR: The ASCI I/O stress benchmark. `http://sourceforge.net/projects/ior-sio/`.

[9] Lustre Users Group. Private communication, April 2009.

[10] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *Proc. SC2008: High performance computing, networking, and storage conference*, Austin, TX, Nov 15-21, 2008.

[11] H. Shan and J. Shalf. Using IOR to analyze the I/O performance of HPC platforms. In *Cray Users Group Meeting (CUG) 2007*, Seattle, Washington, May 7-10, 2007.

[12] H. Wartens. LMT - The Lustre Monitoring Tool. `http://sourceforge.net/projects/lmt/`. Developed at Lawrence Livermore National Lab, LMT includes the Cerebro plug-in libraries, the LMT.pm Perl module and the *lwatch* Java script.