Efficient Object Storage Journaling in a Distributed Parallel File System

Sarp Oral, Feiyi Wang, David Dillow, Galen Shipman, Ross Miller
National Center for Computational Sciences
Oak Ridge National Laboratory
{*oralhs,fwang2,gshipman,dillowda,rgmiller*}*@ornl.gov*

Oleg Drokin
Lustre Center of Excellence at ORNL
Sun Microsystem Inc.
oleg.drokin@sun.com

Abstract

Journaling is a widely used technique to increase file system robustness against metadata and/or data corruptions. While the overhead of journaling can be masked by the page cache for small-scale, local file systems, we found that Lustre's use of journaling for the object store significantly impacted the overall performance of our large-scale center-wide parallel file system. By requiring that each write request wait for a journal transaction to commit, Lustre introduced serialization to the client request stream and imposed additional latency due to disk head movement (seeks) for each request.

In this paper, we present the challenges we faced while deploying a very large scale production storage system. Our work provides a head-to-head comparison of two significantly different approaches to increasing the overall efficiency of the Lustre file system. First, we present a hardware solution using external journaling devices to eliminate the latencies incurred by the extra disk head seeks due to journaling. Second, we introduce a software-based optimization to remove the synchronous commit for each write request, side-stepping additional latency and amortizing the journal seeks across a much larger number of requests.

Both solutions have been implemented and experimentally tested on our Spider storage system, a very large scale Lustre deployment. Our tests show both methods considerably improve the write performance, in some cases up to 93%. Testing with a real-world scientific application showed a 37% decrease in the number journal updates, each with an associated seek – which translated into an average I/O bandwidth improvement of 56.3%.

1 Introduction

Large-scale HPC systems target a balance of file I/O performance with computational capability. Traditionally, the standard was 2 bytes per second of I/O bandwidth for each 1,000 FLOPs of computational capacity [18]. Maintaining that balance for a 1 Petaflops (PFLOPs) supercomputer would require the deployment a storage subsystem capable of delivering 2 TB/sec of I/O bandwidth at a minimum. Building such a system with current or near-term storage technology would require on the order of 100,000 magnetic disks. This would be cost prohibitive not only due to the raw material costs of the disks themselves, but also to the magnitude of the design, installation, and ongoing management and electrical costs for the entire system, including the RAID controllers, network links, and switches. At this scale, reliability metrics for each component would virtually guarantee that such a system would continuously operate in a degraded mode due to ongoing simultaneous reconstruction operations [22].

The National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL) hosts the world's fastest supercomputer, Jaguar [8] with over 300 TB of total system memory. Rather than rely on a traditional I/O performance metric such as 2 byte/sec of I/O throughput for each 1000 FLOP of computational capacity a survey of application requirements was conducted prior to the design of the parallel I/O environment for Jaguar. This resulted in a requirement of delivered bandwidth of over 166 GB/sec based on the ability to checkpoint 20% of total system memory, once per hour, using no more than 10% of total compute time. Based on application I/O profiles and available resources, the Jaguar upgrade targeted 240 GB/s of storage bandwidth. Achieving this target on Jaguar has required a careful attention to detail and optimization of the system at multiple levels, including the storage hardware, network topology, OS, I/O middleware, and application I/O architecture.

There are many studies on user-level file system performance of different Cray XT platforms and their respective storage subsystems. These provide important information for scientific application developers and system engineers such as peak system throughput and the impact of Lustre file striping patterns [33, 1, 32]. However – to the best of our knowledge – there has been no work done to analyze the efficiency of the object storage system's journaling and its impact of overall I/O throughput in a large-scale parallel file system such as Lustre.

Journaling is widely used by modern file systems to increase file system robustness against metadata corruptions and to minimize file system recovery times after a system crash. Aside from journaling, there are several other techniques for preventing metadata corruption. Soft updates handle the metadata update problem by guaranteeing that blocks are written to disk in their required order without using synchronous disk I/O [10, 23]. Vendors such as Network Appliance [3], have addressed the issue with a hardware assisted approach (non-volatile RAM) resulting in performance superior to both journaling and soft updates at the expense of extra hardware. NFS version 3 [20] introduced asynchronous writes to overcome the bottleneck of synchronous writes. The server is permitted to reply to the client before the data is on stable storage, which is similar to our Lustre asynchronous solution. The Log-based file system [17] took a departure from the conventional update-in-place approach by writing modified data and metadata in a log. More recently, ZFS [13] has been coupled with flash-based devices for intent logging so that synchronous writes are directed to these log devices with very low latency, improving overall performance.

While the overhead of journaling can be masked by using the page cache for local file systems, our experiments show that on a large-scale parallel Lustre file system it can substantially degrade overall performance.

In this paper, we present our experiences and the challenges we faced towards deploying a very large scale production storage system. Our findings suggest that sub-optimal object storage file system journaling performance significantly hurts the overall parallel file system performance. Our work provides a head-to-head comparison of two significantly different approaches to increasing overall efficiency of the Lustre file system. First, we present a hardware solution using external journaling devices to eliminate the latencies incurred by the extra disk head seeks for the journal traffic. Second, we introduce a software-based optimization to remove the synchronous commit for each write request, side-stepping additional latency and amortizing the journal seeks across a much larger number of requests.

Major contributions of our work include measurements and performance characterization of a very large storage system unique in its scale; The identification and elimination of serial bottlenecks in a large-scale parallel system; A cost-effective and novel solution to file system journaling overheads in a large scale system.

The remainder of this paper is organized as follows: Section 2 introduces Jaguar and its large-scale parallel I/O subsystem, while Section 3 provides a quick overview of the Lustre parallel file system and presents our initial findings on the performance problems Lustre file system journaling. Section 4 introduces our hardware solution to the problem and Section 5 presents the software solution. Section 6 summarizes and provides a discussion on results of our hardware and software solutions and presents results of real science application using our software-based solution. Section 7 presents our conclusions.

2 System Architecture

Jaguar is the main simulation platform deployed at ORNL. Jaguar entered service in 2005 and has undergone several upgrades and additions since that time. Detailed descriptions and performance evaluations of earlier Jaguar iterations can be found in the literature [1].

2.1 Overview of Jaguar

In late 2008, Jaguar was expanded with the addition of a 1.4 PFLOPs Cray XT5 in addition to the existing Cray XT4 segment[1]. Resulting in a system with over 181,000 processing cores connected internally via Cray's SeaStar2+ [4] network. The XT4 and XT5 segments of Jaguar are connected via a DDR InfiniBand network that also provides a link to our center-wide file system, Spider. More information about the Cray XT5 architecture and Jaguar can be found in [5, 19].

Jaguar has 200 Cray XT5 cabinets. Each cabinet has 24 compute blades. Each blade has 4 compute nodes and each compute node has two AMD Opteron 2356 Barcelona quad-core processors. Figure 1 shows the high-level Cray XT5 node architecture. The configuration tested, has 16 GB of DDR2-800 MHz memory per compute node (2 GB per core), for a total of 300 TB of system memory. Each processor is linked with dual HyperTransport connections. The HyperTransport interface enables direct high-bandwidth connections between the processor, memory and the SeaStar2+ chip. The result is a dual-socket, eight-core node with a peak processing performance of 73.6 GFLOPS.

[1] A more recent Jaguar XT5 upgrade swapped the quad-core AMD Opteron 2356 CPUs (Barcelona) with hex-core AMD Opteron 2435 CPUs (Istanbul), increasing the installed peak performance of Jaguar XT5 to 2.33 PFLOP and total number of cores to 224,256.

The XT5 segment has 214 service and I/O nodes, of which 192 provide connectivity to the Spider center-wide file system with 240 GB/s of demonstrated file system bandwidth over the scalable I/O network (SION). SION is deployed as a multi-stage InfiniBand network [25], and provides a backplane for the integration of multiple NCCS systems such as Jaguar (the simulation and analysis platform), Spider (the NCCS-wide Lustre file system), Smoky (the development platform), and various other compute resources. SION allows capabilities such as streaming data from the simulation platforms to the visualization center at extremely high rates.

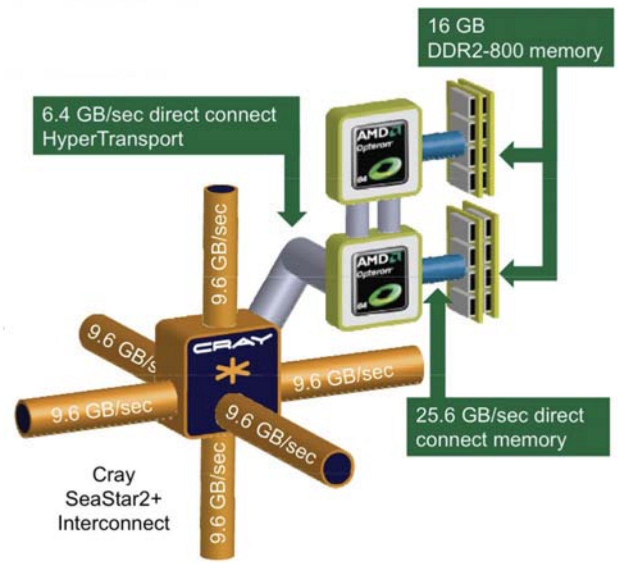


Figure 1. Cray XT5 node (*courtesy of Cray*)

2.2 Spider I/O subsystem

The Spider I/O subsystem consists of Data Direct Networks' (DDN) S2A9900 storage devices interconnected via SION. A pair of S2A9900 RAID controllers is called a couplet. Each controller in a couplet works as an active-active fail-over unit. There are 48 DDN S2A9900 couplets [6] in the Spider I/O subsystem. Each couplet is configured with five ultra-high density 4U, 60-bay disk drive enclosures (56 drives populated), giving a total of 280 1TB hard drives per S2A9900. The system as whole has 13,440 TB or over 10.7 PB of formatted capacity. Fig. 2 illustrates the internal architecture of a DDN S2A9900 couplet. Two parity drives are dedicated in the case of an 8+2 parity group or RAID 6. A parity group is also known as a **Tier**.

Spider, the center-wide Lustre [28] file system, is built upon this I/O subsystem. Spider is the world's fastest and largest production Lustre file system and is one of the world's largest POSIX-compliant file systems. It is designed to work with both Jaguar and other computing resources such as the visualization and end-to-end analysis clusters. Spider has 192 Dell PowerEdge 1950 servers [7] configured as Lustre servers presenting a global file system name space. Each server has 16 GB of memory and dual socket, quad core Intel Xeon E5410 CPUs running at 2.3 GHz. Each server is connected to SION and the DDN arrays via independent 4x DDR InfiniBand links. In aggregate, Spider delivers up to 240 GB/s of file system level throughput and provides 10.7 PB of formatted disk capacity to it users. Fig. 3 shows the overall Spider architecture. More details on Spider can be found in [26].

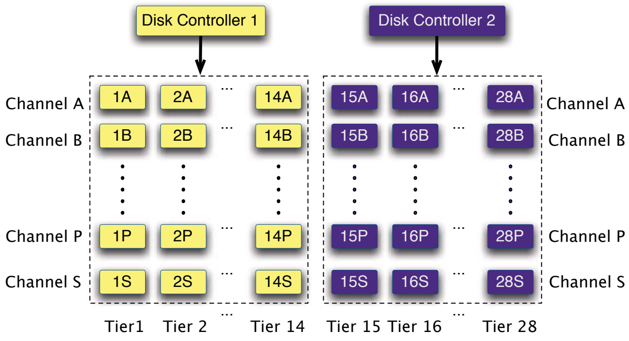


Figure 2. Architecture of a S2A9900 couplet

3 Lustre and file system journaling

Lustre is an open-source distributed parallel file system developed and maintained by Sun Microsystems and licensed under the GNU General Public License (GPL). Due to the extremely scalable architecture of Lustre, deployments are popular in both scientific supercomputing and industry. As of June 2009, 70% of the Top 10 systems, 70% of the Top 20 and 62% of the Top 50 fastest supercomputers systems in the world used Lustre for high-performance scratch space [9], including Jaguar[2].

3.1 Lustre parallel file system

Lustre is an object-based file system and is composed of three components: Metadata storage, object storage, and clients. There is a single metadata target (MDT) per file system. A metadata server (MDS) is responsible for managing one or more MDTs. Each MDT stores file metadata, such as file names, directory structures, and access permissions. Each object storage server (OSS) manages one or more object storage targets (OSTs) and OSTs store file data objects.

[2] As of November 2009, 60% of the Top 10 fastest supercomputers systems in the world used Lustre file system for high-performance scratch space, including Jaguar.

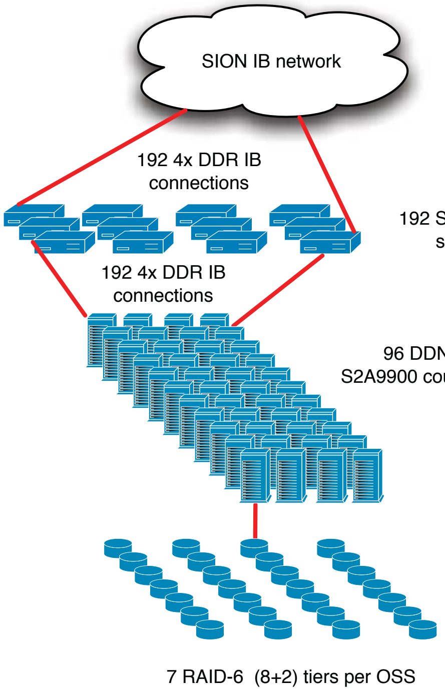

SION IB network
192 4x DDR IB connections
192 Spider OSS servers
192 4x DDR IB connections
96 DDN S2A9900 couplets
7 RAID-6 (8+2) tiers per OSS

Figure 3. Overall Spider architecture

For file data read/write access, the MDS is not on the critical path, as clients send requests directly to the OSSes. Lustre uses block devices for file data and metadata storage and each block device can only be managed by one Lustre service (such as an MDT or an OST). The total data capacity of a Lustre file system is the sum of all individual OST capacities. Lustre clients concurrently access and use data through the standard POSIX I/O system calls. More details on the inner workings of Lustre can be found in [31].

Currently, Lustre version 1.6 employs a heavily patched and enhanced version of the Linux *ext3* file system, known as *ldiskfs*, as the back-end local file system for the MDT and all OSTs. Among the enhancements, improvements over the regular *ext3* file system journaling are of particular interest for this paper and will be covered in the next sections.

3.2 File system journaling in Lustre

A journaling file system, such as *ext3*, keeps a log of metadata and/or file data updates and changes so that in case of a system crash, file system consistency can be restored quickly and easily [30]. The file system can journal only the metadata updates or both metadata and data updates, depending on the implementation. The design choice is to balance file system consistency requirements against performance penalties due to extra journaling write operations and delays. In Linux *ext3*, there are three different modes of journaling: *write-back mode*, *ordered mode*, and *data journaling mode*. In *write-back mode*, updated metadata blocks are written to the journal device while file data blocks are written directly to the block device. When the transaction is committed, journaled metadata blocks are flushed to the block device without any ordering between the two events. *Write-back mode* thus provides metadata consistency but does not provide any file data consistency. In *ordered mode*, file data is guaranteed to be written to their fixed locations on disk before committing the metadata updates to the journal. This ordering protects the metadata and prevents stale data from appearing in a file in the event of a crash. *Data journaling mode* journals both the metadata and the file data. More details on *ext3* journaling modes and their performance characteristics can be found in [21].

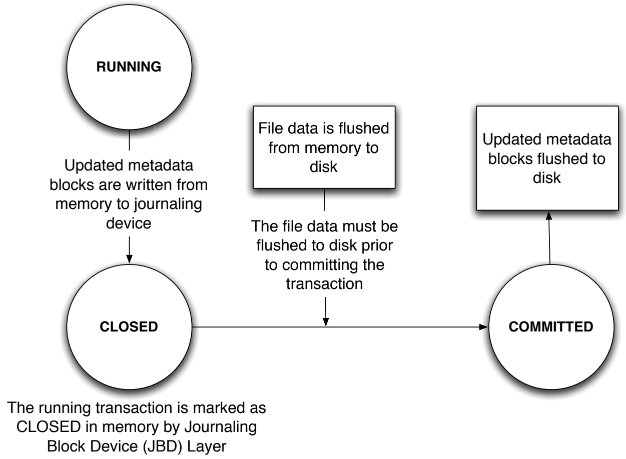

RUNNING
Updated metadata blocks are written from memory to journaling device
File data is flushed from memory to disk
The file data must be flushed to disk prior to committing the transaction
Updated metadata blocks flushed to disk
CLOSED
COMMITTED
The running transaction is marked as CLOSED in memory by Journaling Block Device (JBD) Layer

Figure 4. Flow diagram for the ordered mode journaling.

Although in the latest Linux kernels the default journaling mode for *ext3* file system is a build-time kernel configuration switch (between *ordered mode* and *write-back mode*), *ordered mode* is the default journaling mode for the *ldiskfs* file system used as the object store in Lustre.

Journaling in *ext3* is organized such that at any given time there are two transactions in memory (not written to the journaling device yet): the *currently running transaction* and the *currently closed transaction* (that is being committed to the disk). The currently running transaction is open and accepting new threads to join in and has all its data still in memory. The currently closed transaction is not accepting any new threads to join in and has started flushing its updated metadata blocks from memory to the journaling device. After the flush operation is complete and all transactions are on stable storage, the transaction state will be changed to "*committed*." The currently running transaction can not be closed and committed until the closed transaction fully commits to the journaling device, which for slow disk subsystems can be a point of serialization. Also, even when the disk subsystem is relatively fast, there is another potential point of serialization due to the size of the journaling device. The largest transaction size that can be journaled is limited to 25% of the size of the journal. When a transaction reaches the limit, it is locked and will not accept any new threads or data.

The following list summarizes the steps taken by *ldiskfs* for a Lustre file update in the default ordered journaling mode. The sequence of events is triggered by a Lustre client sending a write request to an OST.

1. Server gets the destination object id and offset for this write operation.

2. Server allocates necessary number of pages in memory and fetches the data from the remote client into the these pages via an Remote Memory Access (RMA) GET operation.

3. Server opens a transaction on its back-end file system.

4. Server updates file metadata in memory, allocates blocks and extends the file size.

5. Server closes transaction handle and obtains a wait handle, but does **not** commit to journaling device.

6. Server writes pages with file data to disk synchronously.

7. After current running transaction is closed, server flushes updated metadata blocks to the journal device and then marks the transaction as committed.

8. Once transaction is committed, server can send a reply to client that the operation was completed successfully and client marks the request as completed.

   Also, the updated metadata blocks, which have been committed to journal device by now will be written to disk, without particular ordering requirement. Fig. 4 shows the generic outline of ordered mode journaling.

There is a minor difference between how this sequence of events happen on an *ext3* file system and the Lustre *ldiskfs* file system. In an *ext3* file system the sequence of steps 6 and 7 are strictly preserved. However, in Lustre *ldiskfs*, the metadata commit can happen before all data from Step 6 is on disk, Step 7 (flushing of updated metadata blocks to the journaling device) can partially happen before Step 6.

Although Step 5 minimizes the time a transaction is kept open, the above sequence of events may be sub-optimal. For example:

- An extra disk head seek is needed for the journal transaction commit after flushing file data on a different sector of the disk if the journaling device is located on the same device as the block file data.

- The write I/O operation for a new thread is blocked on the currently closed transaction which is committing on Step 7.

- The new running journal transaction has to wait for the previous transaction to be closed.

- New I/O RPCs are not formed until the completion replies of the previous RPCs have been received by the client creating yet another point of serialization.

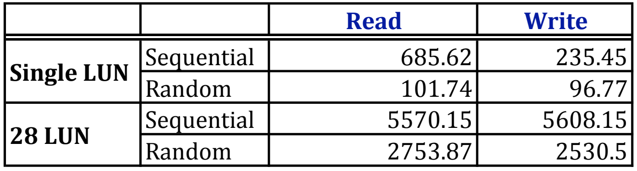
The *ldiskfs* file system by default performs journaling in *ordered mode* by first writing the data blocks to disk followed by metadata blocks to the journal. The journal is then written to disk and marked as committed. In the worst case, such as appending to a file, this can result in one 16 KB write (on average – for bitmap, inode block map, inode, and super block data) and another 4 KB write for the journal commit record for every 1 MB write. These extra small writes cause at least two extra disk head seeks. Due to the poor IOP performance of SATA disks, these additional head seeks and small writes can substantially degrade the aggregate block I/O performance.

A potential optimization (and perhaps the most obvious one) for *ordered mode* to improve the journaling efficiency is to minimize the extra disk head seeks. This can be achieved by either a software or hardware optimization (or both). Section 4 describes our hardware based optimization while Section 5 discusses our software based optimization.

Using journaling methods other than *ordered mode* (or no journaling at all) in the *ldiskfs* file system is not considered in this study, as the OST handler waits for the data writes to hit the disk before returning, and only the metadata is updated in an asynchronous manner. Therefore, *write-back mode* would not help in our case – Lustre would not use the write-back functionality. *Data journaling mode* provides increased consistency and satisfies the Lustre requirements, but we would expect it to result in a reduction of performance from our pre-optimization baseline due to doubling the amount of bulk data written. Of course, running without any journaling is a possibility for obtaining better performance, but the cost of possible file system inconsistencies in a production environment is a price that we could ill afford.

To better understand the performance characteristics of each implementation we have performed a series of tests to obtain a baseline performance of our configuration. In order to obtain this baseline on the DDN S2A9900, the XDD benchmark [11] utility was used. XDD allows multiple clients to exercise a parallel write or read operation synchronously. XDD can be run in sequential or random read or write mode. Our baseline tests focused on aggregate performance for sequential read or write workloads. Performance results using XDD from 4 hosts connected to the DDN via DDR IB are summarized in Fig. 1. The results presented are a summary of our testing and show performance of sequential read, sequential write, random read, and random write using 1MB transfers. These tests were run using a single host for the single LUN tests, and 4 hosts each with 7 LUNs for the 28 LUN test. Performance results presented are the best of 5 runs in each configuration.

Table 1. XDD baseline performance



		Read	Write
Single LUN	Sequential	685.62	235.45
	Random	101.74	96.77
28 LUN	Sequential	5570.15	5608.15
	Random	2753.87	2530.5

After establishing a baseline of performance using XDD, we examined Lustre level performance using the IOR benchmark [24]. Testing was conducted using 4 OSSs each with 7 OSTs on the DDN S2A9900. Our initial results showed very poor write performance of only 1,398.99 MB/sec using 28 clients where each client was writing to different OST. Lustre level write performance was a mere 24.9% of our baseline performance metric of XDD sequential writes with a 1MB transfer size. Profiling the I/O stream of the IOR benchmark using the DDN S2A9900 utilities revealed a large number of 4 KB writes in addition to the expected 1 MB writes. These small writes were traced to *ldiskfs* journal updates.

4 The Hardware Solution

To separate small-sized metadata journal updates from larger (1 MB) block I/O requests and thus enhance our aggregate block I/O performance, we evaluated two hardware-based solutions. Our first option was to use SAS drives as external journal devices. SAS drives are proven to have higher IOP performance compared to SATA drives. For this purpose we used two tiers of SAS drives in a DDN S2A9900, and each tier was split into 14 LUNs. Our second option was to use an external solid state device as the external journaling device. Although the best solution is to provide a separate disk for journaling for each file block device (or even a tier of disks as a single journaling device for each file block device tier), this is highly cost prohibitive at the scale of Spider.

Unlike rotating magnetic disks, solid state disks (SSD) have a negligible seek penalty. This makes SSDs an attractive solution for latency-sensitive storage workloads. SSDs can be flash memory based or DRAM or SRAM based. Furthermore, in recent years, solid state disks have become much more reasonable in terms of cost per GB [14]. The nearly zero seek latency of SSDs make them a logical choice to alleviate our Lustre journaling performance bottleneck.

We have evaluated Texas Memory Systems' RamSan-400 device [29] (on loan from the ViON Corp.) to assess the efficiency of an SSD based Lustre journaling solution for the Spider parallel file system. The RamSan is a 3U rackable solution and has been optimized for high transactional aggregate performance (400,000 small I/O operations per second). The RamSan-400 is a non-volatile SSD with backup hard drives configured as a RAID-3 set. The front end non-volatile solid state disks are a proprietary implementation of Texas Memory Systems' using highly redundant DDR RAM chips. The RamSan-400's block I/O performance is advertised by the vendor at an aggregate of 3 GB/sec. It is equipped with four 4x DDR InfiniBand host ports and supports the SCSI RDMA protocol (SRP).

For our testing purposes, we have connected the RamSan device to our SION network via four 4x DDR IB links directly to the Core 1 switch. This configuration allowed the Lustre servers (MDS and OSSes) to have direct connections to the LUNs on the RamSan device. We configured 28 LUNs (one for each Lustre OST, 7 per each IB host port) on the RamSan device. Fig. 5 presents our experiment layout.

Each LUN on the RamSan was formatted as an external *ldiskfs* journal device and we established a one-to-one mapping between the external journal devices and the 28 OST block devices on one DDN S2A9900 RAID controller. The *obdfilter-survey* benchmark [27] was used for testing both the SAS disk-based and the RamSan-based solutions. *Obdfilter-survey* is part of the Lustre I/O kit and it allows one to exercise the underlying Lustre file system with sequential I/O with varying numbers of threads and objects (files). *Obdfilter* can be used to characterize the performance of the Lustre network, individual OSTs, and the striped file system performance (including multiple OSTs and the Lustre network components). For more details on *obdfilter* readers are encouraged to read the Lustre User Manual [28]. Fig. 6 presents our results for these tests.

For comparative analysis, we ran the same *obdfilter-survey* benchmark on three different target configurations. The first target had external journals on a tier of SAS drives in the DDN S2A900, the second target had external journals on the RamSan-400 device, and third target had internal journals on a tier of SATA drives on our DDN S2A900. We varied the number of threads for each target while measuring the observed block I/O bandwidth. Both solutions with external journals provided good performance improvements. Internal journals on the SATA drives performed the

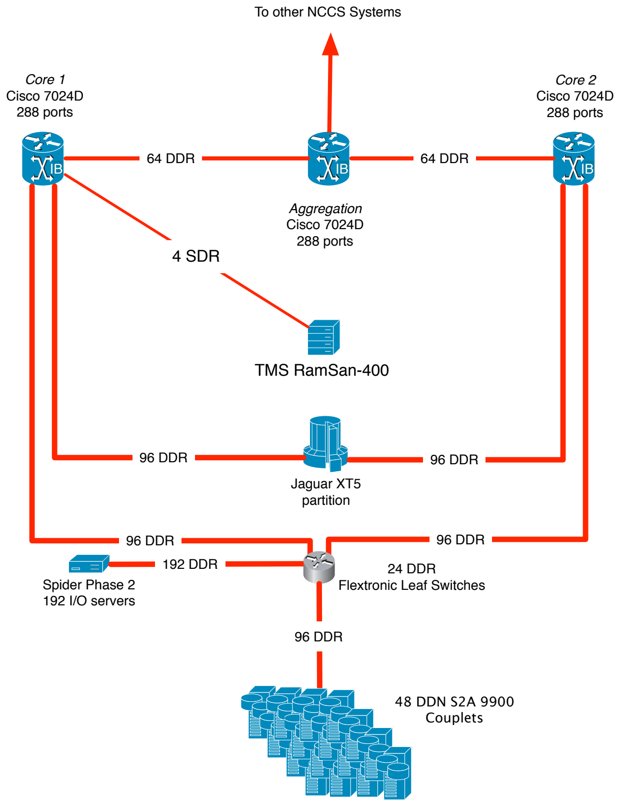


Figure 5. Layout for Lustre external journaling experiment with a RamSan-400 solid state device. The RamSan-400 was connected to the SION network via 4 DDR links and each link exported 7 LUNs.

worst for almost all cases. External journals on a tier of SAS disks showed a gradual performance decrease for more than 256 I/O threads. External journals on the RamSan-400 device gave the best performance for all cases and this solution provided sustained performance with an increasing number of I/O threads. Overall, RamSan-based external journals achieved 3,292.6 MB/sec or 58.7% of our raw baseline performance. The performance dip for the RamSan-400 device at 16 threads was unexpected and is believed to be caused by queue starvation as a result of memory fragmentation pushing the SCSI commands beyond the scatter-gather limit. Unfortunately, we were unable to fully investigate this data point prior to losing access to the test platform and it should be noted that the 16 threads data point is outside of our normal operational envelope.

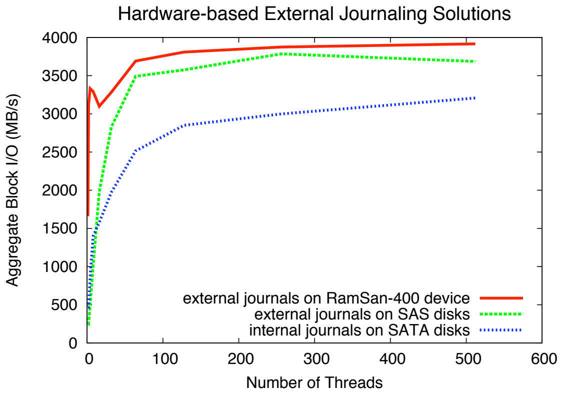


Figure 6. SAS disk, solid state disk external Lustre journaling and SATA disk internal journaling performances.

5 The Software Solution

As explained in Section3.2, Lustre's use of journals guarantees that when a client receives an RPC reply for a write request, the data is on stable storage and would survive a server crash. Although this implementation ensures data reliability, it serializes concurrent client write requests, as the currently running transaction cannot be closed and committed until the prior transaction fully commits to disk. With multiple RPCs in flight from the same client the overall operation flow would appear as if several concurrent write I/O RPCs arrive at the OST at the same time. In this case the serialization in the algorithm still exists, but with more requests coming in from different sources, the OST pipeline is more efficiently utilized. The OST will start its processing and then all these requests will block on waiting for their commits. Then, after each commit, replies for respective completed operations will be sent to the requesting client and then the client will send its next chunk of I/O requests. This algorithm works reasonably well from the aggregate bandwidth point of view as long as there are multiple writers that can keep the data flowing at all times. If there is only one client requesting service from a particular OST the inherent serialization in this algorithm is more pronounced; waiting for each transaction to commit introduces significant delay.

An obvious solution to this problem would be to send replies to clients immediately after the file data portion of a RPC is committed to disk. We have named this algorithm "*asynchronous journal commits*" and have implemented and tested this on our configuration.

Lustre's existing mechanism for metadata transactions allows it to send replies to clients about operation completion without waiting for data to be safe on disk. Every RPC reply from a server has a special field indicating the "id of the last transaction on stable storage" from that particular server's point of view. The client uses this information to keep a list of completed, but not committed operations, so that in case of a server crash these operations could be resent (replayed) to the server once the server resumes operations.

Our implementation extended this concept to write I/O RPCs on OSTs. In our implementation, dirty and flushed data pages are pinned in the client memory once they are submitted to the network. The client releases these pages only after it receives a confirmation from the OST indicating that the data was committed to stable storage.

In order to avoid using up all client memory with pinned data pages waiting for a confirmation for extended periods of time, upon receiving a reply with an uncommitted transaction id, a special "ping" RPC is scheduled on the client 7 seconds into the future (*ext3* flushes transactions to disk every 5 seconds). This "ping" RPC is pushed further in time if there are other RPCs scheduled by the client. This approach limits the impact to the client's memory footprint by bounding the time that uncommitted pages can remain outstanding. While the "ping" RPC is similar in nature to NFSv3's *commit* operation, Lustre optimizes this away in many cases by piggy-backing commit information on other RPCs destined for the same client-server pair.

The "*asynchronous journal commits*" algorithm results in a new set of steps taken by an OST processing a file update in the ordered journaling mode as detailed below. The following sequence of events is triggered by a Lustre client sending a write I/O request to an OST.

1. Server gets the destination object id and offset for this write operation.
2. Server allocates the necessary number of pages in memory and fetched the data from remote client into the pages via an RMA GET operation.
3. Server opens a transaction on the back-end file system.
4. Server updates file metadata, allocates blocks and extends the file size.
5. Server closes the transaction handle (not the JBD transaction) and if the RPC does NOT have the "async" flag set, then it obtains the wait handle.
6. Server writes pages with file data to disk synchronously.
7. If the "async" flag is set in the RPC, then Server completes the operation asynchronously.

   7a Server sends a reply to client.

   7b JBD then flushes the updated metadata blocks to the journaling device and writes the commit record.

8. If the "async" flag is NOT set in the RPC, then Server completes the operation synchronously.

   8a JBD flushes transaction closed in Step 5.

   8b Server sends a reply to the client that the operation was completed successfully.

The *obdfilter* benchmark was used for testing the asynchronous journal commit performance. Fig. 7 presents our results. The *ldiskfs* journal devices were created internally as part of each OST's block device. A single DDN S2A9900 couplet was used for this test. This approach resulted in dramatically fewer 4 KB updates (and associated head seeks) which substantially improved the aggregate performance to over 5,222.95 MB/s or 93% of our baseline performance. The dip at 16 threads is believed to be caused by the same mechanism as explained in the previous section and is outside of normal operational window.

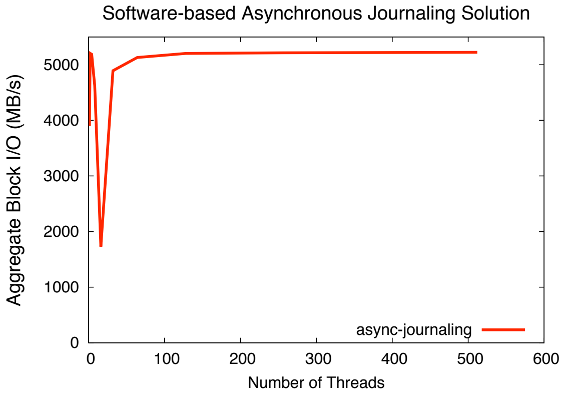


Figure 7. Asynchronous journaling performance

6 Results and Discussion

A comparative analysis of the hardware-based and software-based journaling methods is presented in Fig. 8. Please note that, the data presented in this figure is based on the data provided in figures 6 and 7. As can be seen, the software-based asynchronous journaling method clearly outperforms the hardware-based solutions, providing virtually full baseline performance from the DDN S2A9900 couplet. One potential reason for the software-based solution outperforming the RamSan-based external journals may be the elimination of a network round-trip latency for each journal update operation as the journal resides on an SRP target separate from that of the block device in this configuration. Also, the performance of external journals on solid-state disks suggests that there may be other performance issues in the external journal code path which is encouraged by the lack of a performance improvement when asynchronous commits are used in combination with the RamSan-based external journal. The performance dip at 16 threads, present in both external journal and asynchronous journal methods, requires additional analysis.

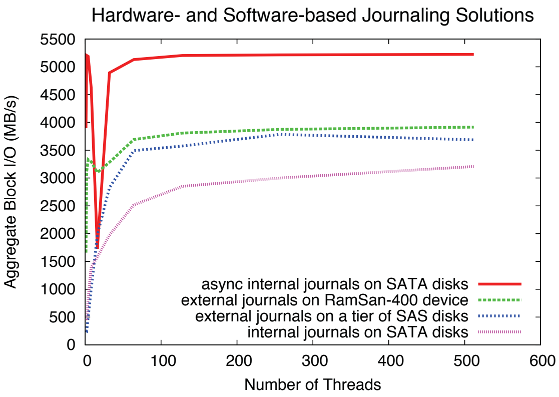


Figure 8. Aggregate Lustre performance with hardware- and software-based journaling methods.

The software-based asynchronous journaling method provides the best performance of the presented solutions, and does so at minimal cost. Therefore, we deployed this solution on Spider. We then analyzed the performance of Spider with the asynchronous journaling method on a real scientific application. For this purpose we used the Gyrokinetic Toroidal Code (GTC) application [15]. GTC is the most I/O intensive code running at scale (at the time of writing, the largest scale runs were at 120,000 cores on Jaguar) and is a 3D gyrokinetic Particle-In-Cell (PIC) code with toroidal geometry. It was developed at the Princeton Plasma Physics Laboratory (PPPL) and was designed to study turbulent transport of particles and energy in burning plasma. GTC is part of the US Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) program. GTC is coded in standard Fortran 90/95 and MPI.

We used a version of GTC that has been modified to use the Adaptable IO System (ADIOS) I/O middleware [16] rather than standard Fortran I/O directives. ADIOS is developed by Georgia Tech and ORNL to manage I/O with a simple API and a supplemental, external configuration file. ADIOS has been implemented in several scientific production codes, including GTC. Earlier GTC tests with ADIOS on Jaguar XT4 showed increased scalability and higher performance when compared to the GTC runs with Fortran I/O. On the Jaguar XT4 segment, GTC with ADIOS achieved 75% of the maximum I/O performance measured with IOR [12].

Fig. 9 shows the GTC run times for 64 and 1,344 cores on Jaguar with and without asynchronous journals on Lustre file system. Both runs were configured with the same problem, and the difference in runtime can be attributed to the compute load of each core. During these runs, the observed I/O bandwidth by the application was increased by 56.3% on average and 64.8% when considering only the median values.

Translating the I/O bandwidth improvements to shorter runtimes will depend heavily on the I/O profile of the application and domain problem being investigated. In the 64 core case for GTC, the cores have a much larger compute load, and the percentage of runtime spent performing I/O drops from 6% to 2.6% when turning asynchronous journals on, with a 3.3% reduction in overall runtime. The 1,344 core test has much lighter compute load, and the runtime is dominated by I/O time – 70% of the runtime is I/O with synchronous journals, and 36% with asynchronous journals. This is reflected in the 49.5% reduction in overall runtime.

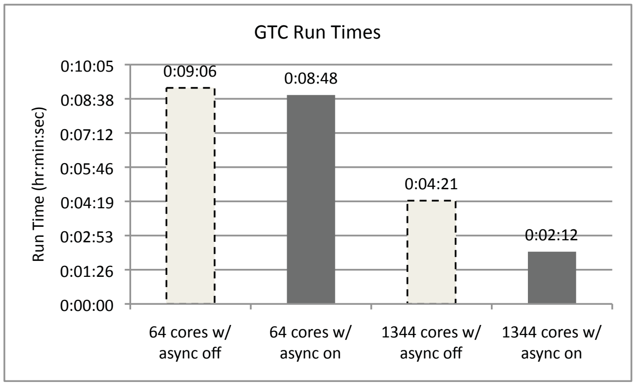


Figure 9. GTC run times for 64 and 1,344 cores on Jaguar with and without asynchronous journals.

Fig. 10 shows the histogram of I/O requests observed by the DDN S2A9900 during our GTC runs as a percent of total I/O requests observed. In this figure, "*Async Journals*" represents I/O requests observed when the asynchronous journals were turned on and "*Sync Journals*" represents when asynchronous journals were turned off. Omitted request sizes from the graph account for less than 2.3% of the total I/O requests for the asynchronous journaling method and 0.76% for the synchronous journaling method. Asynchronous journaling clearly decreased the number of small I/O requests (0 to 127 KB) from 64% to 26.5%. This reduction minimized the disk head seeks, removed the serialization, and increased the overall disk I/O performance. Fig. 11 shows the same I/O request size histogram for 0 to 127 KB sized I/O requests as a percent of total I/O requests observed. Also in this figure "*Async Journals*" represents I/O requests observed when the asynchronous journals were turned on and "Sync Journals" represents when asynchronous journals were turned off. It can be seen that the asynchronous journaling method reduces the number of small I/O requests (0 to 128 KB) sent to the DDN controller (by delaying and aggregating the small journal commit requests into relatively larger but still small I/O requests, as explained in the previous section).

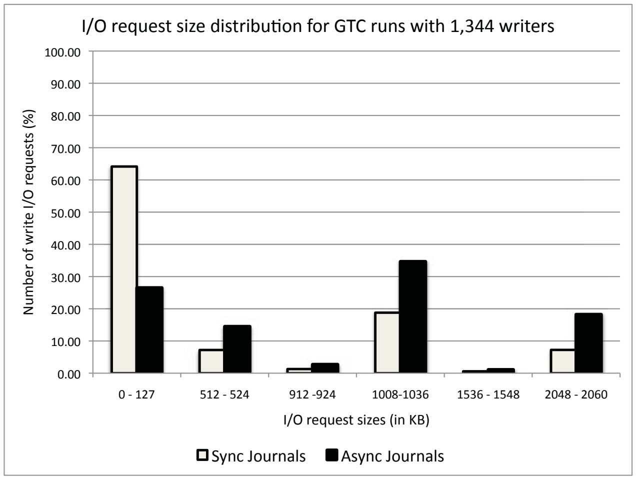


Figure 10. I/O request size histogram observed by the DDN S2A9900 controllers during the GTC runs.

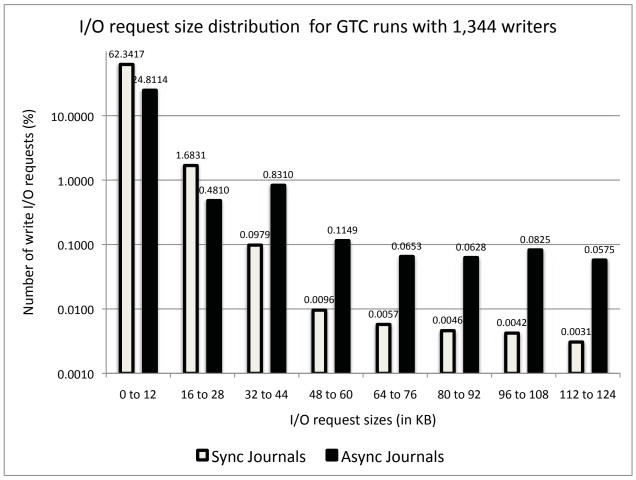


Figure 11. I/O request size histogram for 0 to 127 KB requests observed by the DDN S2A9900 controllers during the GTC runs.

Overall, our findings were motivated by the relatively modest IOPS performance (when compared to the bandwidth performance) of our DDN S2A9900 hardware. The DDN S2A9900 architecture uses "synchronous heads," or a variant of RAID3 that provides dual-failure redundancy. For a given LUN with 10 disks, a seek on the LUN requires a seek by all devices in the LUN. This approach provides highly optimized large I/O bandwidth, but it is not very efficient for small I/O. More traditional RAID5 and RAID6 implementations may not see the same speedup as the DDN hardware with our approach, as the stripes containing active journal data will likely remain resident in the controller cache, minimizing the need to do "read-modify-write" cycles to commit the journal records. Still, there will be head movement for those writes, which will incur a seek-penalty for the drive the stripe chunk that holds that portion of the journal. This will have an affect on the aggregate bandwidth of the RAID array. Some preliminary testing conducted by Sun Microsystems using their own RAID hardware has shown improved performance, but the details of that testing is not currently public. We did not have the chance to test our approach on non-DDN hardware, and are unable to further qualify the impact of our solution on other RAID controllers at this time.

Our approach removed the bottleneck out of the critical write path by providing an asynchronous write/commit mechanism for the Lustre file system. This solution has been previously proposed by NFSv3 and others, and we were able to implement it in an efficient manner to boost our write performance in a very large scale production storage deployment. Our approach comes with a temporary increase in memory consumption on clients, which we believe is a fair price for the performance increases. Our changes are restricted to how Lustre uses the journal, and not the operation of the journal itself. Specifically, we do not wait for the journal commit prior to allowing the client to send more data. As we have not told the client that the data is stable, it will retain it in the event the OSS (OST) dies and the client needs to replay its I/O requests. The guarantees about file system consistency at the local OST remain unchanged. Also, our limited tests with manually injected power failures on the server side with active write/modify I/O client RPCs in flight provided consistent data on the file system, provided the clients successfully completed recovery.

7 Conclusions

Initial IOR testing with Spider's DDN S2A9900s and SATA drives on Jaguar showed that Lustre level write performance was 24.9% of the baseline performance with a 1 MB transfer size. Profiling the I/O stream using the DDN utilities revealed a large number of 4 KB writes in addition to the expected 1 MB writes. These small writes were traced to *ldiskfs* journal updates. This information allowed us to identify bottlenecks in the way Lustre was using the journal – each batch of write requests blocked on the commit of a journal transaction, which added serialization to the request stream and incurred the latency of a disk head seek for each write.

We developed and implemented both a hardware based solution as well as a software solution to these issues. We used external journals on solid state devices to eliminate head seeks for the journal, which allowed us to achieve 3,292.6 MB/sec or 58.7% of our baseline performance per DDN S2A9900. By removing the requirement for a synchronous journal commit for each batch of writes, we observed dramatically fewer 4 KB journal updates (up to 37%) and associated head seeks. This substantially improved our block I/O performance to over 5,222.95 MB/s or 93% of our baseline performance per DDN S2A9900 couplet.

Tests with a real-world scientific application such as GTC have shown an average I/O bandwidth improvement of 56.3%. Overall, asynchronous journaling has proven to be a highly efficient solution to our performance problem in terms of performance as well as cost-effectiveness.

Our approach removed a bottleneck from the critical write path by providing an asynchronous write/commit mechanism for the Lustre file system. This solution has been previously proposed for NFSv3 and other file systems, and we were able to implement it in an efficient manner to significantly boost our write performance in a very large scale production storage deployment.

Our current understanding and testing show that our approach does not change the guarantees of file system consistency at the local OST level, as the modifications only affect how Lustre uses the journal, and not the operation of the journal itself. However, this approach comes with a temporary increase of memory consumption on clients while waiting for the server to commit the transactions. We find this a fair exchange for the substantial performance enhancement it provides on our very large scale production parallel file system.

Our approach and findings are likely not specific to our DDN hardware, and are of interest to developers and large-scale HPC vendors and integrators in our community. Future work will include verifying broad applicability as test hardware becomes available. Other potential future work includes an analysis of how other scalable parallel file systems, such as IBM's GPFS, approach the synchronous write performance penalties.

8 Acknowledgements

The authors would like to thank our colleagues at the National Center for Computational Sciences at Oak Ridge National Laboratory for their support of our work, with special thanks to Scott Klasky for his help with the GTC code and Youngjae Kim and Douglas Fuller for corrections and suggestions.

The research was sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

References

[1] S. R. Alam, R. F. Barrett, M. R. Fahey, J. A. Kuehn, J. M. Larkin, R. Sankaran, and P. H. Worley. Cray XT4: An early evaluation for petascale scientific simulation. In *Proceedings of the ACM/IEEE conference on High Performance Networking and Computing (SC07)*, Reno, NV, 2007.

[2] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. Scalable i/o forwarding framework for high-performance computing systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, Aug, 2009.

[3] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of the 5th ASPLOS*, pages 10–22, 1992.

[4] R. Brightwell, K. Pedretti, and K. D. Underwood. Initial performance evaluation of the cray seastar interconnect. In *HOTI '05: Proceedings of the 13th Symposium on High Performance Interconnects*, pages 51–57, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Cray Inc. Cray XT5. `http://cray.com/Products/XT/Systems/XT5.aspx`.

[6] Data Direct Networks. DDN S2A9900. `http://www.ddn.com/9900`.

[7] Dell. Dell PowerEdge 1950 Server. `http://www.dell.com/downloads/global/products/pedge/en/1950_specs.pdf`.

[8] J. Dongarra, H. Meuer, and E. Strohmaier. Top500 November 2009 List. `http://www.top500.org/lists/2009/11`, 2008.

[9] J. Dongarra, H. Meuer, and E. Strohmaier. Top500 supercomputing sites. `http://www.top500.org`, 2009.

[10] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 5, Berkeley, CA, USA, 1994. USENIX Association.

[11] ioperformance.com. xdd performance benchmark, version 6.5. `http://www.ioperformance.com`, 2008.

[12] S. Klasky. private communication, Sept. 2009.

[13] A. Leventhal. Hybrid storage pools in the 7410. http://blogs.sun.com/ahl/entry/fishworks_launch.

[14] A. Leventhal. Flash storage memory. *Communications of the ACM*, 51(7):47–51, 2008.

[15] Z. Lin, T. S. Hahm, W. W. Lee, W. M. Tang, , and R. B. White. Turbulent transport reduction by zonal flows: Massively parallel simulations. *Science*, 18:1835–1837, 1988.

[16] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich io methods for portable high performance io. In *In Proceedings of IPDPS'09, May 25-29, Rome, Italy*, 2009.

[17] R. Mendel and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[18] D. A. Nowak and M. Seagar. ASCI terascale simulation: Requirements and deployments. http://www.ornl.gov/sci/optical/docs/Tutorial19991108Nowak.pdf.

[19] Oak Ridge National Laboratory, National Center for Computational Sciences. Jaguar. http://www.nccs.gov/jaguar/.

[20] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 - Design and Implementation. *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 137–152, 1994.

[21] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the Annual USENIX Technical Conference*, May 2005.

[22] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics Conference Series*, 78(1):012022–+, July 2007.

[23] M. Seltzer, G. Ganger, K. McKusick, K. Smith, C. Soules, and C. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the USENIX Technical Conference*, pages 71–84, June 2000.

[24] H. Shan and J. Shalf. Using IOR to analyze the I/O performance of XT3. In *Proceedings of the 49th Cray User Group (CUG) Conference 2007*, Seattle, WA, 2007.

[25] G. Shipman. Spider and SION: Supporting the I/O Demands of a Peta-scale Environment. In *Cray User Group Meeting*, 2008.

[26] G. Shipman, D. Dillow, S. Oral, and F. Wang. The spider center wide file system: From concept to reality. In *Proceedings,Cray User Group (CUG) Conference, Atlanta, GA*, May 2009.

[27] Sun Microsystems. Lustre i/o kit, obdfilter-survey. http://manual.lustre.org/manual/LustreManual16_HTML/LustreIOKit.html.

[28] Sun Microsystems Inc. Luste wiki. http://wiki.lustre.org, 2009.

[29] Texas Memory Systems Inc. Ramsan-400. http://www.ramsan.com/products/ramsan-400.htm.

[30] S. C. Tweedie. Journaling the Linux ext2fs Filesystem. In *Proceedings of the fourth annual Linux expo*, 1998.

[31] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang. Understanding lustre filesystem internals. Technical Report ORNL/TM-2009/117, Oak Ridge National Lab., National Center for Computational Sciences, 2009.

[32] W. Yu, S. Oral, S. Canon, J. Vetter, and R. Sankaran. Empirical analysis of a large-scale hierarchical storage system. In *14th European Conference on Parallel and Distributed Computing (Euro-Par 2008)*, 2008.

[33] W. Yu, J. Vetter, and S. Oral. Performance characterization and optimization of parallel I/O on the Cray XT. In *Proceedings of 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, Miami, FL, 2008.