SANDIA REPORT

SAND2006-3057
Unlimited Release
Printed May 2006

Lightweight I/O for Scientific Applications

Ron A. Oldfield, Arthur B. Maccabe, Sarala Arunagiri, Todd Kordenbrock, Rolf Riesen, Lee Ward, and Patrick Widener

Prepared by

Sandia National Laboratories

Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/ordering.htm

SAND2006-3057
Unlimited Release
Printed May 2006

Lightweight I/O for Scientific Applications

Ron A. Oldfield [*] Arthur B. Maccabe [†] Sarala Arunagiri [†]
Todd Kordenbrock [‡] Rolf Riesen [*] Lee Ward [*] Patrick Widener [†]

Abstract

Today's high-end massively parallel processing (MPP) machines have thousands to tens of thousands of processors, with next-generation systems planned to have in excess of one hundred thousand processors. For systems of such scale, efficient I/O is a significant challenge that cannot be solved using traditional approaches. In particular, general purpose parallel file systems that limit applications to standard interfaces and access policies do not scale and will likely be a performance bottleneck for many scientific applications.

In this paper, we investigate the use of a "lightweight" approach to I/O that requires the application or I/O-library developer to extend a core set of critical I/O functionality with the minimum set of features and services required by its target applications. We argue that this approach allows the development of I/O libraries that are both scalable and secure. We support our claims with preliminary results for a lightweight checkpoint operation on a development cluster at Sandia.

[*] Sandia National Laboratories
[†] University of New Mexico
[‡] Hewlett Packard

Contents

Figures

Tables

Lightweight I/O for Scientific Applications

1 Introduction

Efficient I/O is sometimes referred to as the "Achilles' heel" of massively-parallel processing (MPP) computing [5]. While part of the blame can be placed on the inability of the hardware advances for I/O systems to keep pace with advances in CPU, memory, and networks [18], we believe the real problem is in the I/O system software. In particular, today's parallel file systems are unable to meet the specific needs of many data-intensive MPP applications. Current parallel file systems and I/O libraries limit applications to a standard, predefined, set of access interfaces and policies; however, data-intensive applications have a wide variety of needs and often do not perform well using general-purpose solutions. In addition, data-intensive applications show significant performance benefits when using application-specific interfaces that enable advanced parallel-I/O techniques. Examples include collective I/O, prefetching, and data sieving [25, 26, 27, 33]; tailoring prefetching and caching policies to match an application's access patterns, reducing latency and avoiding unnecessary data requests [20, 29]; intelligent application-control of data consistency and synchronization virtually eliminating the need for file locking [11]; and matching data-distribution policies to the application's access patterns in order to optimize parallel access to distributed disks [40].

This paper describes the *Lightweight File System* (LWFS) project, a collaboration between Sandia National Laboratories and the University of New Mexico investigating the applicability of "lightweight" approaches for I/O on MPP systems. Lightweight designs identify the essential functionality needed to meet basic operation requirements. The design of Catamount (the lightweight OS for Sandia's Red Storm machine) focused on the need to support MPI style programs on a space-shared system, i.e., a system in which nodes in the compute partition are allocated to different applications. Because compute nodes are the unit of allocation, the lightweight kernel needs to insure that applications running on different nodes cannot interfere with one another, but does not need to address issues related to competition for resources within a single compute node. Once this essential functionality has been defined and implemented, additional functionality is relegated to the libraries and the application itself. The Compute Node Kernel (CNK) [35] developed for BlueGene/L follows a similar strategy. The advantages of the lightweight approach are that underlying services do not implement functionality that might degrade the scalability of an application and applications are free to implement the functionality they need in a way that is optimal for the application. The clear disadvantage is that many needed services must be implemented either in libraries or in some cases within an application itself.

While the benefits of the lightweight approach have been demonstrated in the context of operating systems for MPP architectures, this approach has not been applied to the design

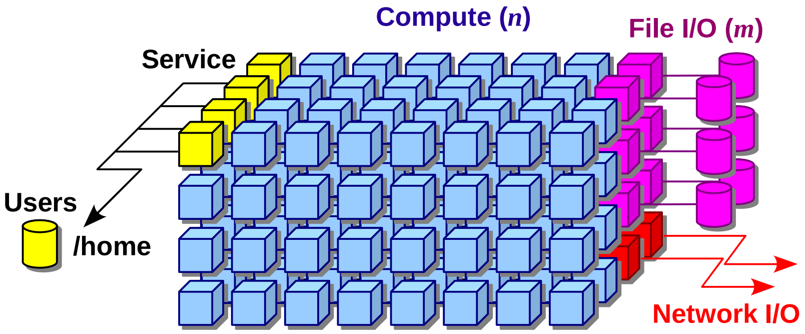


Figure 1. The compute nodes in a partitioned architecture use a "lightweight" operating system with no support for threading, multi-tasking, or memory management. I/O and service nodes use a more "heavyweight" operating system (e.g., Linux) to provide shared services.

of other system services. LWFS represents a lightweight approach to I/O in which the core system consists of a small set of critical functionality that the I/O library or file system developer extends to provide custom services, features, and optimizations required by the target applications.

2 Background and Requirements

Today's high-end MPP machines have tens of thousands of nodes. For example, "Red Storm", the Cray XT3 machine at Sandia National Laboratories [8] has over ten thousand nodes, and the IBM BlueGene/L [35] installed at Lawrence Livermore National Laboratory, has over sixty-four thousand compute nodes. Both machines are expected to be used for large scale applications. For example, 80% of the node-hours of Red Storm are allocated to applications that use a minimum of 40% of the nodes.

The scale of current and next-generation MPP machines and their supported applications presents significant challenges for designers of their system software. In this section, we discuss the accepted solution for their system architecture, and we present the general design requirements for I/O systems on such architectures.

2.1 System Architecture

To address scaling issues, both Red Storm and BlueGene/L have adopted a "partitioned architecture" [16] (illustrated in Figure 1). The compute nodes in a partitioned architec-

Table 1. Compute and I/O nodes for MPPs at the DOE laboratories.

Computer	Compute Nodes	I/O Nodes	Ratio
SNL Intel Paragon (1990s)	1840	32	58:1
ASCI Red (1990s)	4510	73	62:1
Cray Red Storm (2004)	10,368	256	41:1
BlueGene/L (2005)	65,536	1024	64:1

ture use a "lightweight kernel" [24, 35] operating system with no support for threading, multi-tasking, or memory management. I/O and service nodes use a more "heavyweight" operating system (e.g., Linux) to provide shared services.

The number of nodes used for computation in an MPP is typically one to two orders of magnitude greater than the number of nodes used for I/O. For example, Table 1 shows the compute- and I/O-node configurations for four Massively Parallel Processing (MPP) systems. Unlike most clusters, compute nodes in MPPs are diskless. This means that all I/O traffic must traverse the communication network, competing with non-I/O traffic for the available bandwidth.

2.2 I/O System Scalability

The disparity in the number of I/O and compute nodes, coupled with the fact that compute nodes are diskless, puts a significant burden on the communication network between the compute nodes and the I/O nodes. To reduce this burden, the I/O system should minimize the number of system-imposed communications and allow the clients direct access to the storage devices.

I/O for scientific applications is often "bursty" in nature. Since there are many more compute nodes than I/O nodes, an I/O node may receive tens of thousands of near-simultaneous I/O requests. To handle such surges in load, bulk data-movement for I/O requests should be controlled by the server [19]: the server should "pull" data from the client for writes and "push" data to the client for reads. We describe further our approach to this issue in Section 3.2.

2.3 Application Scalability

Perhaps the most important requirement for an MPP I/O system is that it does not hinder the scalability of applications. That is, it should not impose unnecessary functionality that adds overhead on compute nodes. This is a fundamental motivation behind using a lightweight approach for I/O. To address this concern we designed the core architecture of the lightweight file system based on the following rules (where n is the number of compute nodes and m is the number of I/O nodes):

1. Prohibit system-imposed operations that require $O(n)$ operations.
2. Prohibit system-imposed data structures of size $O(n)$. This implies that the I/O system may not use connection-based mechanisms for communications or security.
3. Make operations with $O(m)$ messages between I/O nodes as rare as possible.

2.4 Access Control

Security is a critical concern for I/O systems in general. However, the DOE Laboratories have particular requirements that impose a significant challenge on I/O system design. In particular, we need scalable mechanisms for authentication and authorization as well as "immediate" revocation of access permissions when access policies change.

A critical design requirement for developing scalable authentication and authorization mechanisms is to minimize the number of required communications to centralized control points like a metadata server. In traditional file systems, the file system controls access by forcing every access request to go through a centralized metadata server that authenticates the user and authorizes the request before allowing the request to pass through to the storage system (i.e., the I/O nodes). As applications scale to use thousands of nodes, the metadata server becomes a severe bottleneck for data access. In a partitioned architecture, we need an authorization model that allows for centralized definitions of access-control policies, but distributed enforcement of those policies. In the ideal case, every access request could be independently authenticated at an I/O node without communicating with a centralized "authorization server".

We consider it necessary and beneficial to integrate authentication and authorization into the I/O system architecture. However, the controlled environment of a DOE laboratory allows us to make a different choice with respect to network security (privacy of the information carried over the wire). For our purposes, the I/O system can assume a trusted transport mechanism that does not allow "replay" attacks, "man-in-the-middle" attacks, or eavesdropping. From the application-interface level, it is safe for the application and other system components to transmit private data in clear text. The assumption of a secure transport allows for a more efficient design of the security infrastructure because the I/O system does not need to encrypt data on the wire, a potentially costly operation. For environments that already have a secure and reliable network, adding these features to the I/O system is redundant and adds unnecessary overheads. For environments that are not secure, the I/O system should use a transport mechanism that provides encryption internally. In either case, provision of a secure and reliable transport is not an issue for the I/O system.

To provide the level of access control required by our security model, the system must allow for the "immediate" revocation of access privileges should the access-control policies change. Because of the distributed nature of our target I/O system, and the need for distributed enforcement of access-control policies, immediate revocation presents a scalability challenge that is not easily solved. We discuss our proposed solution in Section 3.1.4.

3 The LWFS-core

The primary challenge associated with designing the fixed core of a lightweight file system (called the *LWFS-core*) is choosing which functionality is required (i.e. will be provided by the LWFS-core) and which is optional (allowing applications to implement it in different ways). General design guidelines for the LWFS-core are:

1. The LWFS-core should provide the infrastructure needed to provide controlled access to data distributed across multiple storage servers.
2. The LWFS-core should be a thin layer above the hardware that presents an accurate reflection of costs associated with resource usage.
3. The LWFS-core should expose the parallelism of the storage servers to clients to allow for efficient data access and control over data distribution.
4. The LWFS-core should provide an "open architecture" for optional functionality that allows the client implementation to accept, reject, replace, or create additional functionality.

In short, the LWFS-core consists of the minimal set of functionality required by all I/O systems. Based on our guidelines and the requirements expressed in Section 2, we defined the LWFS-core to include mechanisms for security (i.e., authentication and authorization), efficient data movement, direct access to data, and support for distributed transactions.

We chose not to include policies for data distribution, caching, prefetching, and others in the LWFS-core because there are no general solutions that work well for all applications. Instead, we take an "open architecture" approach that allows the I/O-system developer to either choose from existing libraries, or implement desired functionality directly. For example, Figure 2 illustrates potential software stacks that an application may use to access data. The layers above the LWFS-core provide application-specific functionality in the form of libraries or file system implementations. Note that since the LWFS-core contains all required mechanisms for access control, each layer (including the application) may access the LWFS-core directly.

Figure 3 shows the components that make up the core functionality of a lightweight I/O system. These components represent the minimum functionality needed to support the LWFS security and data-access models. In particular, the LWFS-core consists of an authentication server, an authorization server, and a collection of storage servers. The

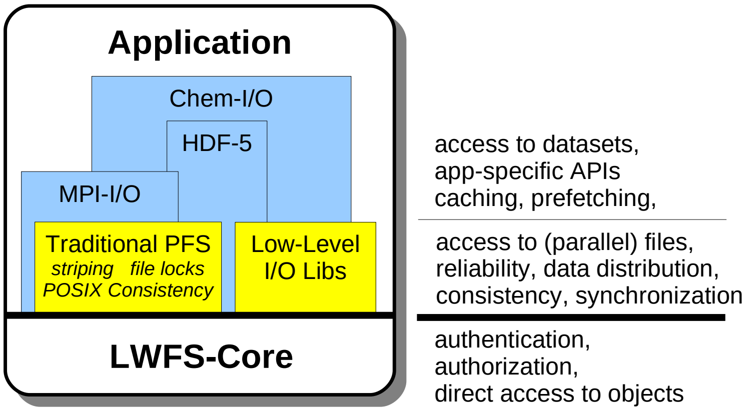


Figure 2. The LWFS-core provides object-based access, user authentication, and authorization. Layers above provide application-specific functionality in the form of libraries or file system implementations.

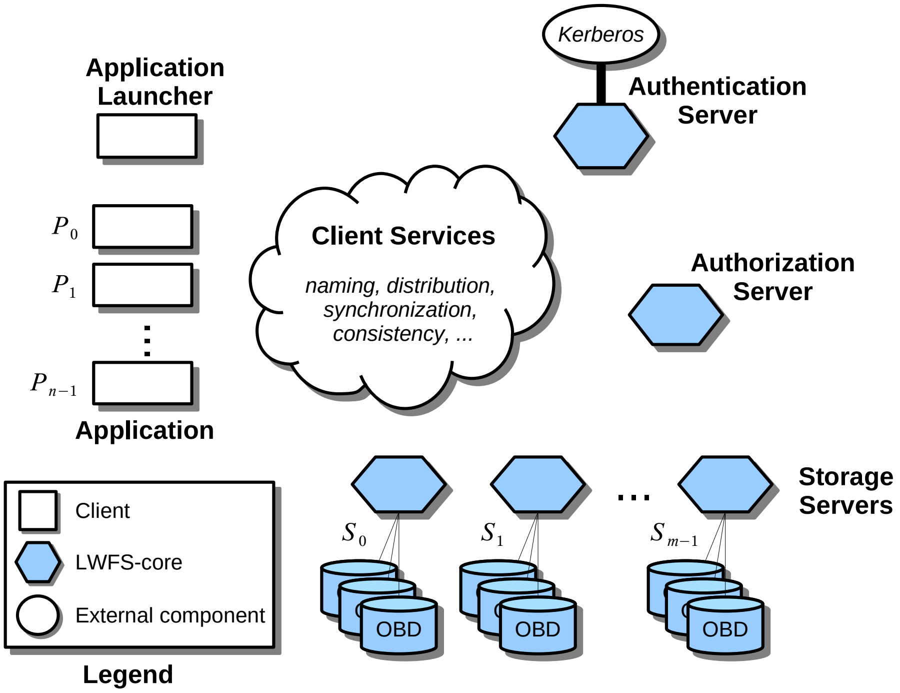


Figure 3. The components in a lightweight I/O system. The gray portion represents the LWFS-core.

authentication server interfaces with an external authentication mechanism (e.g., Kerberos) to manage and verify identities of users. The authorization server manages and verifies access-control policies for authorized users. The storage servers enforce the access-control policies of the authorization server and export an API to access data stored on the attached device. We discuss details of these services in Sections 3.1 and 3.2.

The remaining components include the client application, the application launcher (part of the client application), the authentication mechanism, and additional services required by the client.

3.1 Security

Our security design builds on traditional capability-based systems to provide scalable mechanisms for authentication and access control, with near-immediate revocation when access policies change.

3.1.1 Coarse-grained access control

Unlike many file systems that provide fine-grained access control at the byte level, the LWFS-core provides coarse-grained access control to *containers* of objects. Every object belongs to a unique container, and all objects in the same container are subject to the same access control policy. LWFS knows nothing about the organization of objects in a container; higher-level libraries are responsible for implementing and interpreting container organization. Since LWFS does not constrain object organization, library programmers may experiment with data distribution and redistribution schemes that efficiently match the access patterns of different applications.

3.1.2 Credentials and capabilities

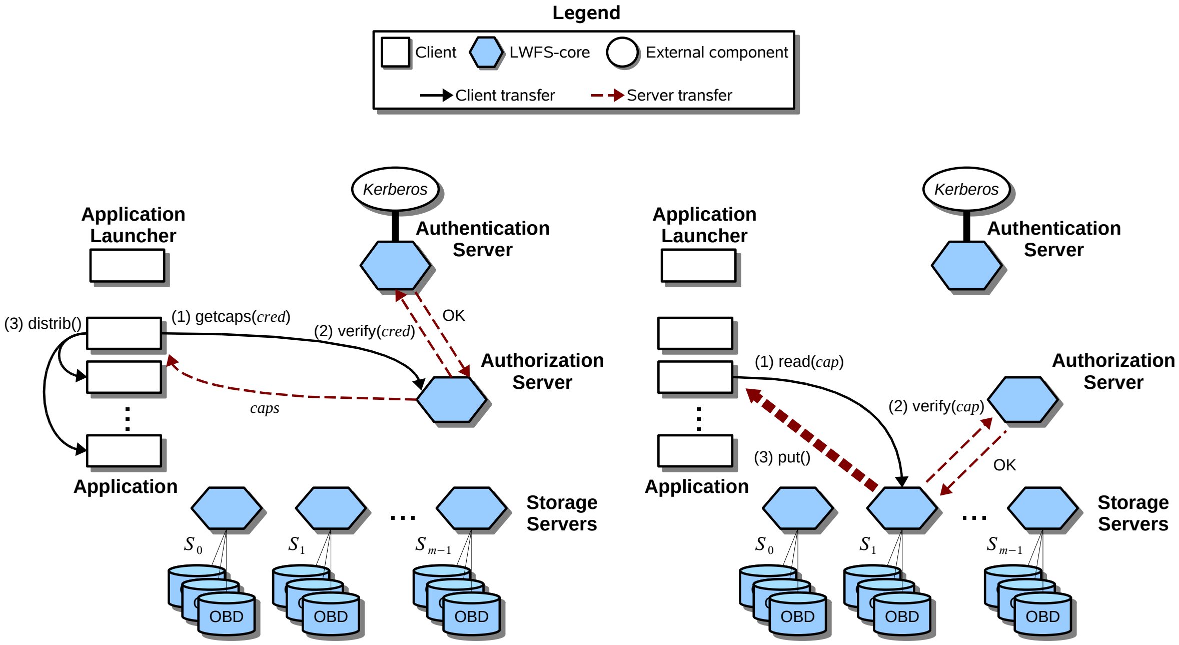
The LWFS-core uses capability-like [23] data structures for authentication and authorization. For authentication, a *credential* provides the LWFS system components with proof of user authentication from a trusted external mechanism (e.g., Kerberos, GSS-API, SASL). Credentials are fully transferable. Once obtained, the application may distribute the credential to other processes that act on behalf of the principal. Such functionality is useful, for example, in distributed applications that want each process composing the distributed application to share a single identity. The contents of a credential are opaque to the user and contain a random string of bits that is sufficiently difficult to guess, so as to minimize the likelihood of unknown users correctly forging valid credentials. Associated with the credential is the identity of the authenticated user and a lifetime modifier that limits how long the credential remains valid. Depending on the implementation, these values may be cryptographically hidden in the credential object or managed by the LWFS-core system.

In the same way that credentials provide proof of authentication, a *capability* provides proof of authorization. A capability is a data structure that entitles the holder to perform a specific operation on a container of objects. For example, a capability could allow the holder to read from the objects belonging to the container "foo". Like credentials, capabilities are transient — limited in life to the current, issuing instance of the authorization service as well as bounded by the authentication service in use. Capabilities are also fully-transferable. Once acquired, an application may transfer a capability to any process, including processes in other applications–allowing the delegation of access rights. Also like credentials, capabilities are opaque to the user and contain a cryptographically secure random number (a signature) generated by the authorization service. This random number is difficult to guess and can only be verified by the authorization service, thus reducing the vulnerability of unauthorized users forging the capabilities.

Having fully-transferable credentials and capabilities limits the number of wire calls to the authentication or authorization server and makes the distribution of credentials or capabilities the responsibility of the client. Figure 4-a shows the protocol for acquiring capabilities from the authorization server. A single client processor first requests a capability from the authorization server and passes the credential as proof of identification. If this is the first authorization request from the client, the authorization server asks the authentication server to verify the credential. Once the initiating client has the capability, it can use a logarithmic "scatter" routine to distribute capabilities to other client processors.

The capabilities (and credentials) used in the LWFS-core are different from traditional capabilities because LWFS capabilities can only be verified by the entity that generated them. In a true capability system [23], any entity can verify the authenticity and integrity of the capability. We provide the benefits of independently verifiable capabilities by caching the result of the "verify" request sent from an LWFS component (e.g., storage server) to the authorization service. For example, Figure 4-b shows the protocol for reading data from an LWFS storage server. The process starts when a client sends an access request along with a capability (labeled *cap*) to a storage server. If the storage server does not have the *cap* in its cache of valid capabilities, it sends a "verify" request to the authorization server. The authorization server then verifies the request and sends a response back to the storage server. To support revocation (see Section 3.1.4), the authorization server keeps track of clients that are caching valid capabilities. If the *cap* is valid, the storage server saves the capability in its cache and initiates the transport of data between the client and the storage server along a high-throughput data channel.

Our approach of caching valid capabilities on the storage server diverges from other approaches. The most common method used to implement capability-based storage architectures is a symmetric-encryption scheme that shares a secret key between the authorization service and the storage service. This is the approach taken by NASD [14] and Panasas [28] and it is the recommended approach given by the T10 standards document for object-based storage devices [38]. In their scheme, the authorization service uses the key to sign new capabilities and the storage service uses the key to verify the capability. The problem with this approach is that the authorization server has to trust the storage server to only use that



(a) Acquire capabilities.

(b) Get data.

Figure 4. Figure (a) illustrates the protocol for acquiring capabilities in LWFS. Once the client processors have a capability, they access data by sending request directly to the storage servers, as illustrated in Figure (b).

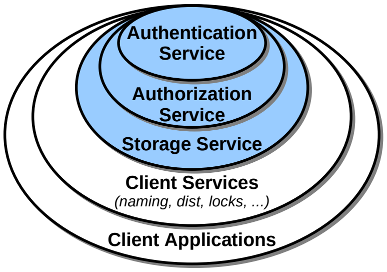


Figure 5. Trust relationships between the LWFS components. A component trusts everything within its circle, but trusts nothing outside of the circle. The gray portion represents the LWFS-core services.

key to verify existing capabilities (not generate new ones). Our caching scheme only allows the storage server to verify previously authorized capabilities, thus eliminating the need for the authorization server to trust the storage server. Our scheme, however, requires explicit communication between the storage server and the authorization server. An amortized analysis of this approach proves that given the computing environment for MPPs, the amortized impact of this additional communication is minimal; however, space restrictions do not allow a complete explanation of our analysis.

3.1.3 Trust relationships

Figure 5 illustrates the trust relationships between the different LWFS components. Each circle represents a single component and encompasses all of the components it trusts. Applications are not trusted by any components, but applications trust the storage service to allow access to entities with proper authorization (i.e., capabilities). The storage service trusts the authorization service to grant capabilities to authorized users, and the authorization service trusts the authentication service to properly identify users. These trust relationships are not reciprocal.

3.1.4 Revocation of capabilities and credentials

In order to provide the level of access control required by our security model, credentials and capabilities may be revoked by the authentication or authorization service at any time. We need “immediate” revocation of credentials when an application terminates or for security-related reasons (e.g., system compromise). Revocation of capabilities is needed, for instance, when an application changes the access-policy of a previously authorized operation.

Revocation is a challenge for true capability-based systems because capabilities need to be independently verifiable and fully transferable. These requirements make it difficult for the system to track down and invalidate capabilities in a scalable way.

The LWFS scheme uses a combination of the two commonly used methods for capability revocation: secure keys and back pointers. LWFS credentials and capabilities contain a secure hash key, but, the hash can can only be verified by the entity that generated the hash (i.e., the authorization service). We added an optimization to allow a trusted entity (e.g., a storage server) to cache results from the authorization service so that subsequent requests using previously verified capabilities do not require additional communication with the authorization service. These optimizations require back pointers (method 2) so that when the authorization service revokes a capability, the system can invalidate the cached entries on each of the storage servers.

One of the nice features of the LWFS capability model is that the system can revoke partial access to a container of objects. Consider an application that has two capabilities on a container: one that enables writing, and another to enable reading. Our authorization service can revoke one capability without revoking the other. For example, if a user decides to remove write access to the container (via a “chmod”), the storage servers (after being contacted by the authorization service) can invalidate the capability that allows writing without invalidating the capability that allows reading.

3.2 Data movement

One of the principal challenges for parallel file systems on MPP systems is dealing with device contention created by having tens of thousands of compute nodes competing for the I/O resources of hundreds of I/O servers. At any point in time, hundreds, or even thousands, of compute nodes may be competing for the same I/O server. Without control of the movement of data to the I/O server, a “burst” of large I/O requests can quickly overwhelm the resources of an I/O server causing bottlenecks that affect the performance and reliability of every competing application and the system as a whole.

To illustrate the problem, consider the hardware configuration of the Cray Red Storm system at Sandia, generally considered a “well-balanced” system. Based on the specifications presented in Table 2, an I/O node can receive 6 GB/s from the network, but only output 400 MB/s to the RAID storage. Requests that arrive but cannot be processed are either buffered on the I/O node or rejected if the I/O node buffer is full. Rejecting buffers causes the compute nodes to actively re-send the data at some later time based on the flow-control mechanism implemented by the I/O system or the network transport layer. The re-sending of I/O requests creates overhead on the compute nodes that hinders the scalability of the application and consumes valuable network resources.

Well-designed applications avoid resource conflicts by coordinating access among application processors either explicitly [12] or by using collective parallel I/O interfaces [37];

Table 2. Red Storm Communication and I/O Performance [5]

I/O Performance	
I/O node topology (per end)	8 × 16 mesh
Aggregate I/O B/W (per end)	50 GB/s
I/O node B/W (to RAID)	400 MB/s
Interconnect Performance	
MPI Latency	2.0 μs 1 hop, 5.0 μs max
Bi-Directional Link B/W	6.0 GB/s
Minimum Bi-Section B/W	2.3 TB/s

however, their solutions do not solve the problem of multiple applications competing for I/O servers.

We address this problem by using a server-directed approach [19, 32], illustrated in Figure 6, in which the server controls the transfer of bulk data to/from the client. In our scheme, the server receives a small request that identifies the operation to perform and where to "put" or "get" data for reads or writes. The bulk-data transfer operation uses Portals [6], a zero-copy, one-sided, messaging API that allows the server to make efficient use of remote direct-memory access (DMA) [41], operating-system bypass, and other optimization features that may be available in the underlying network. The server can also re-order independent requests to improve access to the storage device [36].

We are not unique in this approach. The Lustre Parallel File System uses a sophisticated request-processing layer on top of Portals to provide many of the same features we desire [4]. Another effort by Wu, Wykoff, and Panda modified the Parallel Virtual File System (PVFS) to take advantage of the remote DMA capabilities of the InfiniBand network [41], to reduce OS involvement in the data-transfer, and to have the server control the movement of data. The main distinction between LWFS and those efforts is that our security model allows us to expose the data-movement interface to the application (rather than "hiding" it inside the file-system interface), thus allowing a client access to individual storage devices.

3.3 Object-Based Data Access

The LWFS-core storage service follows a recent trend to utilize intelligent, object-based storage devices. The object-based storage architecture is more scalable than the traditional server-attached disk (SAD) architecture because it separates policy decisions from policy enforcement. Figure 7 illustrates the differences between the server-attached architecture and the object-based storage architecture. In traditional SAD architectures (Figure 7-a) the file server manages the block layout of files and decides on and enforces the access-control policy for every access request. Object-based storage architectures (shown on the right)

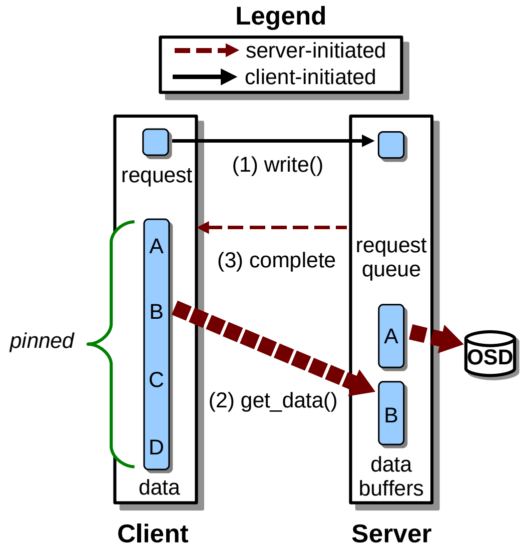


Figure 6. An LWFS I/O server controls data movement by pulling data from the client for writes or pushing data to the clients for reads.

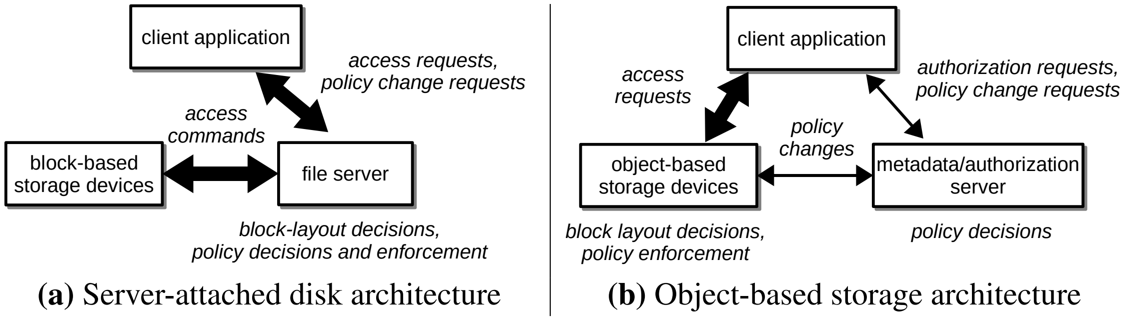
move the block layout decisions and policy enforcement to the storage device, reducing the number of calls to the metadata server and allowing clients direct access to storage devices.

3.4 Transactional semantics

LWFS provides two mechanisms for implementing ACID-compliant transactions: journals and locks. Journals provide a mechanism to ensure atomicity and durability for transactions. A two-phase commit protocol (part of the LWFS API) helps the client to preserve the atomicity property because it requires all participating servers to agree on the final state of the system before changes become permanent. Durability exists because a journal exists as a persistent object on the storage system. Locks enable consistency and isolation for concurrent transactions by allowing the client to synchronize access to portions of the code that require protection or which must complete in a particular order based on the consistency semantics of the application.

4 Case study: checkpointing

Checkpointing application state to stable storage is the most common way for large, long-running applications to avoid loss of work in the event of a system failure. On MPP sys-



(a) Server-attached disk architecture

(b) Object-based storage architecture

Figure 7. In traditional SAD architectures (shown on the left) the file server manages the block layout of files and decides on and enforces the access-control policy for every access request. Object-based storage architectures (shown on the right) move the block layout decisions and policy enforcement to the storage device.

tems, checkpoints are highly I/O intensive and account for nearly 80% of the total I/O usage in some instances [30]. In this section, we describe how to implement a checkpoint operation using the core features of LWFS and we compare the performance of a preliminary implementation to two alternative approaches using traditional parallel file systems.

In order to maximize MPP application throughput, checkpoint processing should proceed as quickly as possible with as little interference as possible from the I/O system. However, checkpointing is an example of a logically simple operation that is made unneccesarily complex by the functionality imposed by traditional file systems. For example, checkpointing requires no synchronization because all writes are non-overlapping. Checkpointing also has minimal requirements for data consistency among the participating servers, and it requires the use of a naming service to reference the checkpoint data when the application needs to reconstruct the process on a restart.

Figure 8 shows pseudocode of the steps required to implement a checkpoint operating using the LWFS core services. The first step is to create a container and acquire the capabilities required to create and write to objects into that container (lines 2 and 3 of the MAIN() function). Since we can create multiple checkpoint files using the same container ID, it is only necessary to perform this step once. At application-defined intervals, the application pauses computation to perform a CHECKPOINT() operation. In our implementation, the client processors independently, in parallel, create and dump process state to individual storage objects. After completing the writes, a single process gathers and creates sufficient metadata to describe the checkpoint objects as a coherent dataset. That process then writes the metadata to a single storage object, creates a name in the naming service, and associates the metadata object with that name. Since the checkpoint operation involves a number of of distributed tasks to different servers, we execute each task inside a distributed transaction.

We illustrate the benefits of the lightweight checkpoint operation by comparing it with

```
Main()
cred ← GetCreds()
cid ← CreateContainer(cred)
caps ← GetCaps(cid)
while not done do
    state ← Compute()
    Checkpoint(state, path, caps)
end while
```

```
Checkpoint(state, path, caps)
txnid ← BeginTxn()
obj ← CreateObj(txnid, caps)
DumpState(txnid, state, obj, caps)
if $rank = 0$ then
    mdobj ← CreateObj(txnid, caps)
end if
GatherMetadata(mdobj, 0)
if $rank = 0$ then
    CreateName(txnid, path, mdobj)
end if
EndTxn(txnid)
```

Figure 8. Pseudocode for checkpointing application state using the LWFS.

two commonly used implementations that access storage through a traditional parallel file system. In the first alternative, the application creates a single parallel file shared by all application processors. The second alternative is for the application to create a single parallel file per process.

In both implementations, limitations inherent in the parallel file system introduce significant performance bottlenecks. These bottlenecks are shown in Figures 9 and 10. The plots show measured throughput and bandwidth of the lightweight checkpoint and the two alternative implementations running on an I/O-development cluster at Sandia. The cluster is comprised of 40 2-way SMP 2.0 GHz Opteron nodes with a Myrinet interconnect. We used 1 node for the metadata/authorization server, 8 as storage servers, and the remaining 31 we used for compute nodes. For the larger runs, some of the compute nodes host multiple client processes.

For the two implementations that uses a traditional PFS, each storage-server node hosted two Lustre object-storage targets (OSTs), each mounted to an `ext3` file system using an LSI MetaStor 4400 fibre channel RAID with 1GB/s fibre channel links. For the LWFS implementation, we disabled the Lustre OSTs on each storage node and configured two LWFS storage servers to use the same RAIDs. In every experiment, each node writes 512 MB of data and measures the time to open, write, sync, and close the file (or object). The application reports the maximum time over all participating processes. All plots show the average and standard deviation over a minimum of 5 trials.

In the shared-file case, even though the processors write their process state to a non-overlapping regions, the file system's consistency and synchronization semantics get in the way, severely limiting the throughput of the checkpoint operation. In fact, as shown in Figure 9, the throughput of the shared-file case is roughly half that of the file-per-process

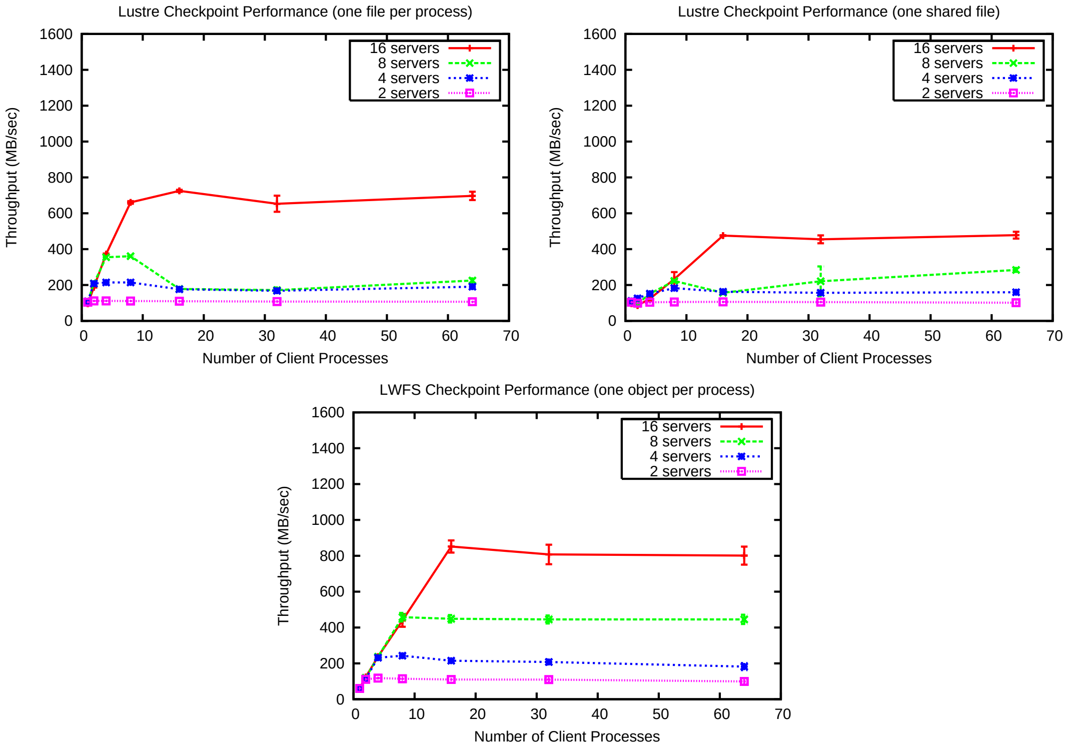
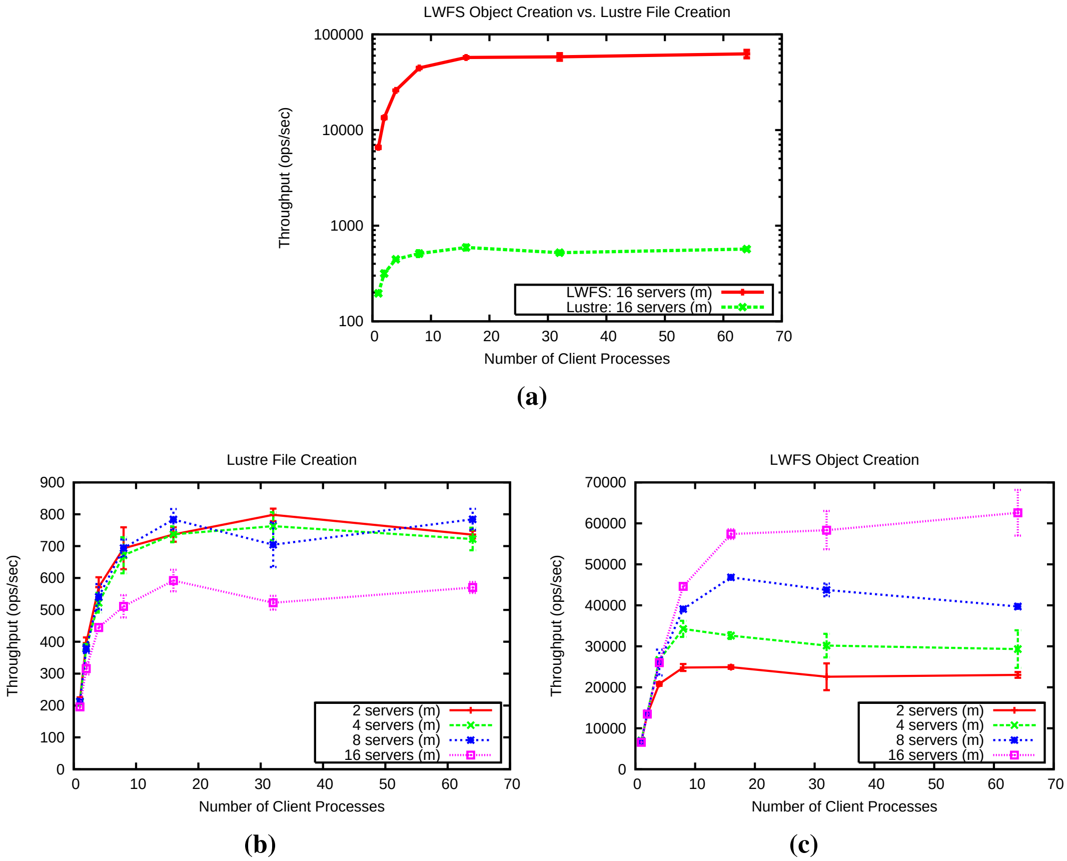


Figure 9. These figures show the throughput in MB/sec as a function of the number of processors of the Lustre file-per-process, Lustre shared file, and LWFS object-per-process implementations of the checkpoint operation.



(a)

(b) **(c)**

Figure 10. Figure (a) shows a logplot comparison of the throughput of creating Lustre files for the file-per-process implementation compared to the throughput creating objects for the LWFS implementation. Figures (b) and (c) show more detail for the individual implementations.

and the lightweight checkpoint implementations.

In the file-per-process implementation, the bandwidth scales well, but the limiting factor is the time to create the checkpoint files. Since every file-create request goes through the centralized metadata server, the performance is always limited to the throughput in operations/second of the metadata server. In contrast, the lightweight checkpoint operation creates the checkpoint objects in parallel. The performance comparison in Figure 10 reflects these differences.

For small systems, the overhead of file creation may be small relative to the time it takes to actually dump the file; however, operations to a centralized metadata server are inherently unscalable and as the system grows, this “file creation” overhead becomes a serious problem. For example, if we make conservative approximations to scale the results from our development cluster to a theoretical petaflop system with 100,000 compute nodes and 2000 I/O nodes, creating the files will require multiple minutes to complete–roughly 10% of the total time for the checkpoint operation.

5 Related Work

Our lightweight approach to I/O-system design is motivated by the success of microkernel architectures [1, 3], especially for MPPs [7, 39], and is a direct extension of previous work on “stackable” file systems [17, 21, 42]; however, because of space limitations, we focus this section on other efforts to develop scalable I/O systems.

There are several existing parallel file systems designed for large-scale clusters or MPPs. Of these, Lustre [10], PVFS2 [22, 34], and NASD [13, 14] (and the commercial version Panasas [28]) are the most widely used. LWFS distinguishes itself from these other file systems in two areas: how services are partitioned, and the trust relationship between components.

Lustre, NASD, and PVFS all use a similar architecture that consists of client processors, metadata servers, and storage servers. For each of these systems, the metadata server provides namespace management (including metadata consistency), access-control policy, and some control of data distribution for parallel files. Although they may provide some flexibility with respect to data-distribution policies, the client may not extend those policies or create new ones. In contrast, LWFS separates the functionality of traditional metadata servers to allow for a variety of schemes and implementations.

Unlike LWFS, Lustre and PVFS extend the trust domain all the way to the client. In Lustre, the client-side services exist entirely in a trusted kernel. The PVFS client code runs in user space, but trusts the client to perform operations that were authorized when the client opened the file. While trusting the client eliminates the need to authenticate every access operation, it complicates the development process by tying development of the operating system to the file system. The file system must support each version of the operating-system kernel. Systems like PVFS that trust a client running outside a trusted kernel are inherently insecure because they allow potentially unauthorized operations to access data.

Of the three file systems, NASD is most similar to LWFS. Both LWFS and NASD use capabilities that the system verifies before allowing object access; however, NASD capabilities are different in several ways. In contrast to LWFS capabilities that provide coarse-grained access control to containers, Panasas capabilities enable “fine-grained” access control to objects. While there are some benefits with respect to data consistency and security associated with fine-grained access-control, a NASD client may have to acquire more capabilities to access a file. NASD does have “indirection objects” [15] that group objects into the same access-control domain, but the client still has the ability to change the access-control policy of the sub-objects, invalidating the usefulness of the indirection object. NASD and LWFS also differ in how they invalidate capabilities (i.e., revocation). NASD updates a version attribute on an object, which causes subsequent capability-verification attempts to fail–forcing the client to re-acquire all capabilities for that object. In contrast, LWFS can revoke a subset of capabilities for a container by only removing cache entries (see Section 3.1.4) for a particular operation. For example, LWFS can revoke write capabilities without revoking read capabilities.

There are other differences between LWFS and NASD. NASD (designed primarily for clusters) assumes an untrusted network. For the reasons expressed in Section 2.4, we chose to trust the network. Also, NASD does not automatically refresh expired capabilities. After a capability expires, the client has to re-acquire capabilities (possibly an $O(n)$ operation). NASD staggers expiry times in an effort to reduce the impact of expiring capabilities, but for operations like a checkpoint, with large gaps between file accesses, the cost of re-acquiring expired capabilities is still a problem.

There is also an effort to standardize the interface to object-based storage devices (OSD) [38]. We are looking forward to integrating vendor-supplied devices using this interface into LWFS, but as we mentioned in Section 3.1.2, we use a different approach to verify capabilities. It would be helpful if the T10 standard provided some flexibility in this regard.

6 Future Work

Although our experiments provide insight to the scalability of our approach on MPP systems, our development cluster is clearly insufficient for true scalability experiments. The next logical step is to acquire more compelling evidence by running experiments on Sandia’s large production machines. This effort is underway and we expect to have opportunities for exclusive access to these machines in the near term.

LWFS has potential as both a vehicle for I/O research and a framework for developing production-ready file systems. In the short term, we plan to implement two traditional parallel file systems: one that provides POSIX semantics and standard distribution policies, and another (like the PVFS [9]) with relaxed synchronization semantics that make the client responsible for data consistency.

We are also interested in implementing commonly used I/O libraries like MPI-I/O, HDF-5, and PnetCDF directly on top of the LWFS core. In current implementations, these libraries are layered on top of low-level libraries, which are in-turn layered on top of a general-purpose parallel file system. We believe that commonly used high-level libraries can make better use of the underlying hardware and take advantage of application-specific synchronization and consistency policies if they bypass the intermediate layers and interact directly with the LWFS core components.

We are also investigating how to apply the lightweight file system approach to numerous other research areas including scalable namespace management, application-specific distribution policies, client-coordinated synchronization and data consistency, I/O libraries that incorporate remote processing (e.g., remote filtering) [2, 31], and many others.

7 Acknowledgment

Thanks to James Schutt for quickly fixing bugs in the reference implementation of Portals. Thanks also to Ruth Klundt for helping to set up the Lustre and LWFS experiments on the I/O development machine.

8 Summary

In this paper, we present a lightweight approach to I/O for MPP computing that allows data-intensive operations to bypass features of traditional parallel file systems that hinder the scalability of the application. In addition to being scalable, our design is both secure and extensible, allowing library, I/O systems, and applications to implement functionality specific to their needs.

Our implementation of a lightweight checkpoint operation provides an example that illustrates the simplicity and performance benefits of a lightweight approach, but we believe there are number of other areas that will also benefit. For example, lightweight implementations of common I/O libraries like MPI-IO, HDF-5, netCDF, and others, can avoid the overheads and loss of dataset-specific semantics caused by the I/O abstraction layers that typically sit between the high-level library and the I/O-system hardware. In addition, application-specific I/O libraries can benefit from control over data distribution and a flexible data consistency and synchronization model that allows client processors to coordinate access to shared devices.

LWFS is still in a relatively early stage of development. While performance results from experiments on our development machine are encouraging and provide insight as to how well LWFS will scale to larger machines, we look forward to demonstrating the benefits of the lightweight approach in larger scale application scenarios. This effort is already underway and we expect to have significantly more compelling results in the near future.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Conference*, pages 93–112, 1986.

[2] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 34, pages 499–512. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[3] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemysław Pardyak, Stefan Savage, and Emin Gün Sirer. SPIN: An extensible microkernel for application-specific operating system services. *ACM Operating Systems Review*, 29(1):74–77, January 1995.

[4] Peter J. Braam. The lustre storage architecture. Cluster File Systems Inc. Architecture, design, and manual for Lustre, November 2002. http://www.lustre.org/docs/lustre.pdf.

[5] Ron Brightwell, William Camp, Benjamin Cole, Erik DeBenedictis, Robert Leland, James Tomkins, and Arthur B. Maccabe. Architectural specification for massively parallel computers: an experience and measurement-based approach. *Concurrency and Computation: Practice and Experience*, 17(10):1271–1316, March 2005.

[6] Ron Brightwell, Tramm Hudson, Arthur B. Maccabe, and Rolf Riesen. The Portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratories, November 1999.

[7] Ron Brightwell, Rolf Riesen, Keith Underwood, Trammell Hudson, Patrick Bridges, and Arthur B. Maccabe. A performance comparison of linux and a lightweight kernel. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2003)*, pages 251–258, Hong Kong, December 2003. IEEE Computer Society Press.

[8] William J. Camp and James L. Tomkins. The red storm computer architecture and its implementation. In *The Conference on High-Speed Computing: LANL/LLNL/SNL*, Salishan Lodge, Glenedon Beach, Oregon, April 2003.

[9] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[10] Lustre: A scalable, high-performance file system. Cluster File Systems Inc. white paper, version 1.0, November 2002. http://www.lustre.org/docs/whitepaper.pdf.

[11] Kenin Coloma, Alok Choudhary, Wei keng Liao, Lee Ward, Eric Russell, and Neil Pundit. Scalable high-level caching for parallel I/O. In *Proceedings of the International Parallel and Distributed Processing Symposium*, page 96b, Santa Fe, NM, April 2004. Los Alamitos, CA, USA : IEEE Comput. Soc, 2004.

[12] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, Newport Beach, CA, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[13] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. *Performance Evaluation Review*, 25(1):272 – 84, 1997.

[14] Garth A. Gibson, David P. Nagle, Khalil Amiri, Fay W. Chang, Eugene Feinberg, Howard Gobioff Chen Lee, Berend Ozceri, Erik Riedel, and David Rochberg. A case for network-attached secure disks. Technical Report CMU–CS-96-142, Carnegie-Mellon University, June 1996.

[15] Howard Gobioff. *Security for a High Performance Commodity Storage Subsystem*. PhD thesis, Carnegie Mellon University, July 1999. CMU Technical Report CMU-CS-99-160.

[16] David S. Greenberg, Ron Brightwell, Lee Ann Fisk, Arthur B. Maccabe, and Rolf Riesen. A system software architecture for high-end computing. In *Proceedings of SC97: High Performance Networking and Computing*, pages 1–15, San Jose, California, November 1997. ACM Press.

[17] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[18] John L. Hennessy and David A. Patterson. *Computer architecture (2nd ed.): a quantitative approach*. Morgan Kaufmann Publishers Inc., 1996.

[19] David Kotz. Disk-directed I/O for MIMD multiprocessors. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 35, pages 513–535. IEEE Computer Society Press and John Wiley & Sons, 2001.

[20] David Kotz and Carla Schlatter Ellis. Practical prefetching techniques for multiprocessor file systems. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 17, pages 245–258. IEEE Computer Society Press and John Wiley & Sons, New York, NY, 2001.

[21] Orran Krieger and Michael Stumm. HFS: A performance-oriented flexible file system based on building-block compositions. *ACM Transactions on Computer Systems*, 15(3):286–321, August 1997.

[22] Rob Latham, Neil Miller, Robert Ross, and Phil Carns. A next-generation parallel file system for linux clusters. *LinuxWorld*, 2(1), January 2004.

[23] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984. Available online at http://www.cs.washington.edu/homes/levy/capabook/.

[24] Arthur B. Maccabe and Stephen R. Wheat. Message passing in PUMA. Technical Report SAND-93-0935C, Sandia National Labs, 1993.

[25] David Mackay, G. Mahinthakumar, and Ed D'Azevedo. A study of I/O in a parallel finite element groundwater transport code. *The International Journal of High Performance Computing Applications*, 12(3):307–319, Fall 1998.

[26] Jarek Nieplocha, Ian Foster, and Rick Kendall. ChemIO: High-performance parallel I/O for computational chemistry applications. *The International Journal of High Performance Computing Applications*, 12(3):345–363, Fall 1998.

[27] Ron A. Oldfield, David E. Womble, and Curtis C. Ober. Efficient parallel I/O in seismic imaging. *The International Journal of High Performance Computing Applications*, 12(3):333–344, Fall 1998.

[28] Object-based storage architecture: Defining a new generation of storage systems built on distributed, intelligent storage devices. Panasas Inc. white paper, version 1.0, October 2003. http://www.panasas.com/docs/.

[29] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 16, pages 224–244. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[30] Fabrizio Petrini and Kei Davis. Tutorial: Achieving Usability and Efficiency in Large-Scale Parallel Computing Systems, August 31, 2004. Euro-Par 2004, Pisa, Italy.

[31] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, 34(6):68–74, June 2001.

[32] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[33] E. Smirni and D.A. Reed. Lessons from characterizing the input/output behavior of parallel scientific applications. *Performance Evaluation: An International Journal*, 33(1):27–44, June 1998.

[34] PVFS2 Development Team. *Parallel Virtual File System, Version 2*, September 2003. http://www.pvfs.org/pvfs2/pvfs2-guide.html.

[35] The BlueGene/L Team. An overview of the BlueGene/L supercomputer. In *Proceedings of SC2002: High Performance Networking and Computing*, Baltimore, MD, November 2002.

[36] Rajeev Thakur and Alok Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.

[37] Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, January 2002.

[38] Ralph O. Weber. SCSI object-based storage device commands (OSD). Technical Report ISO/IEC 14776, incits Technical Committee T10, July 2004. Revision 10.

[39] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: An operating system for massively parallel systems. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages II:56–65, Wailea, HI, 1994. IEEE Computer Society Press.

[40] David Womble, David Greenberg, Stephen Wheat, and Rolf Riesen. Beyond core: Making parallel computer I/O practical. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 56–63, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[41] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar Panda. PVFS over InfiniBand: design and performance evaluation. In Chu-Sing Sadayappan, P.; Yang, editor, *Proceedings of the 2003 International Conference on Parallel Processing*, pages 125–132, Kaohsiung, Taiwan, October 2003. Los Alamitos, CA, USA : IEEE Comput. Soc, 2003.

[42] Erez Zadok, Ion Badulescu, and Alex Shender. Extending file systems using stackable templates. In *Proceedings of the 1999 USENIX Technical Conference*, pages 57–70. USENIX Association, June 1999.

DISTRIBUTION:

3 Barney Maccabe
Computer Science Department
MSC01 1130
1 University of New Mexico
Albuquerque, NM 87131

1 MS 1110
Ron A. Oldfield, 1423

1 MS 1110
Rolf Riesen, 1423

1 MS 1110
Lee Ward, 1423

1 MS 1110
Neil Pundit, 1423

2 MS 9960
Central Technical Files, 8945-1

2 MS 0899
Technical Library, 9616