

Efficient Data-Movement for Lightweight I/O

Ron A. Oldfield, Sandia National Laboratories*

Patrick Widener, University of New Mexico[†]

Arthur B. Maccabe, University of New Mexico[†]

Lee Ward, Sandia National Laboratories*

Todd Kordenbrock, Hewlett-Packard[‡]

Abstract

Efficient data movement is an important part of any high-performance I/O system, but it is especially critical for the current and next-generation of massively parallel processing (MPP) systems. In this paper, we discuss how the scale, architecture, and organization of current and proposed MPP systems impact the design of the data-movement scheme for the I/O system. We also describe and analyze the approach used by the Lightweight File Systems (LWFS) project, and we compare that approach to more conventional data-movement protocols used by small and mid-range clusters. Our results indicate that the data-movement strategy used by LWFS clearly outperforms conventional data-movement protocols, particularly as data sizes increase.

1 Introduction

Efficient data movement is an important part of any high-performance I/O system, but it is especially critical for the current and next-generation of massively parallel processing (MPP) systems. The massive scale, the systems architecture, and the organization of the components in an MPP each play an important role in the design of the I/O systems.

1.1 Impact of system size on data-movement design

The scale of current and next-generation systems is immense. For example, “Red Storm”, the Cray XT3 machine at Sandia National Laboratories [CT03] has over ten

*P.O. Box 5800, Albuquerque, NM 87185-1110; {raoldfi, lee}@sandia.gov. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

[†]Department of Computer Science, University of New Mexico, Albuquerque, NM 87131; {widener, maccabe}@cs.unm.edu.

[‡]P.O. Box 5800, Albuquerque, NM 87185-1110; thkorde@sandia.gov.

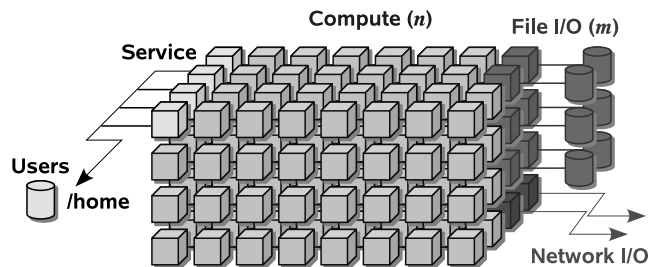


Figure 1. The compute nodes in a partitioned architecture use a “lightweight” operating system with no support for threading, multi-tasking, or memory management. I/O and service nodes use a more “heavyweight” operating system (e.g., Linux) to provide shared services.

thousand compute nodes, and the IBM BlueGene/L [Tea02] 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.

In MPP systems such as Red Storm, any I/O node may receive requests from any compute node. This implies that an efficient data-movement protocol should be connectionless; an I/O node cannot manage tens-to-hundreds of thousands of compute-node connections and still perform its I/O responsibilities in an efficient way. We need to maintain a “stateless” scheme in which the data-movement interface generates all information required to move a piece of data from the source address to its destination address.

1.2 Impact of system architecture on data-movement design

In addition to the massive scale of these systems and applications, the system architecture and organization also has a significant impact on the design of the data-movement scheme. Both Red Storm and BlueGene/L have adopted a “partitioned architecture” [GBF⁺97] (illustrated in Figure 1). The compute nodes in a partitioned architecture use a “lightweight kernel” [MW93, Tea02] 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 lightweight kernel OS (LWK) used by compute nodes in a partitioned architecture heavily influences the design of the data-movement protocols and interfaces used by the I/O system. For example, an LWK commonly does not include support for multiprogramming, preventing the use of threading for asynchronous data transfer as is done in clusters. Traditional connection-based transfer protocols like TCP/IP are also considered too heavyweight for an LWK and so are not included. To minimize overheads on the client, LWKs should take advantage of optimizations in the network such as OS-bypass and remote DMA. Their primary role in communication is to efficiently manage the available bandwidth in the network.

1.3 Impact of system organization on data-movement design

System organization also plays a role in the design of the data-movement protocols and APIs. 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. In addition, 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.

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. In addition, because many I/O operations for scientific applications are naturally “bursty”, an I/O node may receive tens of thousands of near-simultaneous I/O requests. To allow I/O nodes to better respond to such surges in load, bulk data-movement for I/O requests should be controlled by the server [Kot01]. That is, the server should “pull” data from the client for writes and “push” data to the client for reads.

2 Background

The context of our discussion on efficient data-movement comprises two other projects: the Lightweight File System and the Portals message passing interface.

2.1 The Lightweight File System

The Lightweight File System (LWFS) project investigates the applicability of lightweight solutions in storage systems. This work is patterned after research into lightweight kernels for MPP systems, in which kernels are customized for individual applications by including only services that are necessary (the canonical example being the irrelevance of the line-printer daemon *lpd* to a MPP compute node). In the case of LWFS, traditional filesystem semantics such as atomicity and naming are not provided by the storage system. Instead, LWFS emphasizes secure and direct access to storage devices. The LWFS architecture is extensible through the use of libraries, which may be combined according to application needs to provide exactly the services required. We are using this approach to extend LWFS with naming and metadata services as well as transactional semantics for applications that need them. Section 3 describes the data movement architecture of LWFS.

2.2 Portals

Portals [RBP⁺06] is a message passing abstraction that was designed as a response to the near-parity (achieved in the late 1990s) in bandwidth between memory-to-memory and network bandwidths. As this rough parity was approached, the overheads imposed by the intermediate memory-to-memory copies performed by commonly-used message passing solutions grew to unacceptable proportions of the available network bandwidth. Portals eliminated these intermediate copies by providing users with the means to describe which data should be placed where by the kernel as messages were received. The *which* was solved by referring to user-defined criteria (*match-lists*), and the *where* was accomplished by providing the kernel with reserved user-space buffers to be used as DMA targets.

A *portal* is an “opening” in the address space of a process. Each portal has a region of memory associated with it, and other processes can use the Portals API to read, write, or atomically swap the contents of that memory.

The Portals interface provides the following beneficial properties for a server-directed I/O architecture:

- Communication is connectionless, avoiding the necessity to construct and destroy per-connection resources. This has very positive implications for large scale parallel systems in which any node can communicate with any other node.

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

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

- Receivers manage communication, not senders. Under sender-managed communication, persistent blocks of information must remain available for every process, implying that memory usage will increase linearly with job size.
- Although receivers manage communication spatially (by providing and describing the destinations for received messages), they are not required to manage data transfer temporally. That is, they are not required to poll or wait for data to arrive, but instead may proceed with their processing after indicating their readiness to receive a message or set of messages. Receivers are notified asynchronously only after messages have been completely received and transferred into user space.
- In addition to managing messages which they expect, receivers can define what should be done with unexpected messages.

3 Design of the LWFS RPC mechanism

LWFS relies on an asynchronous remote procedure call (RPC) interface based on the Sun RPC interface [Sun87, SM86]. This interface is responsible for both access to remote services and efficient transport of bulk data.

In contrast to the Sun RPC interface, we designed the LWFS RPC interface to be asynchronous. This allows clients to overlap computation and I/O — a feature particularly important given that I/O operations are remote for most MPP architectures (see Section 1 for a description of MPP architectures).

Like Sun RPC, LWFS relies on client and server stub functions to encode/decode (i.e., *marshal*) procedure call parameters to/from a machine-independent format. This approach is portable because it allows access to services on heterogeneous systems, but it is not efficient for I/O requests that contain raw buffers that do not need encoding. It also employs a “push” model for data transport that puts tremendous stress on servers when the requests are large and unexpected, as is the case for most I/O requests.

To address the issue of efficient transport for bulk data, the LWFS uses separate communication channels for control and data messages. In our model, a *control message*

is typically small. It identifies the operation to perform, where to get arguments, the structure of the arguments, and so forth. In contrast, a *data message* is typically large and consists of “raw” bytes that, in most cases, do not need to be encoded/decoded by the server. The LWFS client uses the RPC-like interface to push control messages to the servers, but the LWFS server uses a different, one-sided API (similar to the Portals Message Passing Interface [BHMR99]), to push or pull data to/from the client. This combined approach (illustrated in Figure 2) allows interactions with heterogeneous servers, but also benefits from allowing the server to control the transport of bulk data [Kot01, SCJ⁺95]. The server can thus manage large volumes of requests with minimal resource requirements. Furthermore, since we expect servers to be a critical bottleneck in the system (recall the high proportion of compute nodes to I/O nodes in MPPs), a server-directed approach allows the server to optimize the processing of requests for efficient use of underlying network and storage devices — for example, re-ordering requests to a storage device [Kot01].

4 Implementation

As mentioned in Section 3, the LWFS RPC APIs consist of a RPC-like interface that allows the client to send requests to a server, and a one-sided interface that allows the server to control the transport of bulk data to/from the client. To enable the implementation of these APIs on MPPs that use a partitioned architecture, we chose to layer our communication APIs and protocols on top of the Portals Message Passing Interface [BHMR99]. Portals was a logical choice for several reasons. First, Portals is the standard protocol used on Sandia MPP machines. It uses a completely connectionless architecture. It has one-sided communication APIs that enable the exploitation of remote DMA and operating-system bypass to avoid memory copies in the kernel-managed protocol stack. Finally, it runs on lightweight operating systems such as the Catamount OS for the Cray XT3 [CT03].

Figure 3 illustrates the required Portals data structures and describes the basic protocol for calling and executing a remote `READ()` function using the LWFS RPC library. In the remainder of this section, we describe each of these steps in detail.

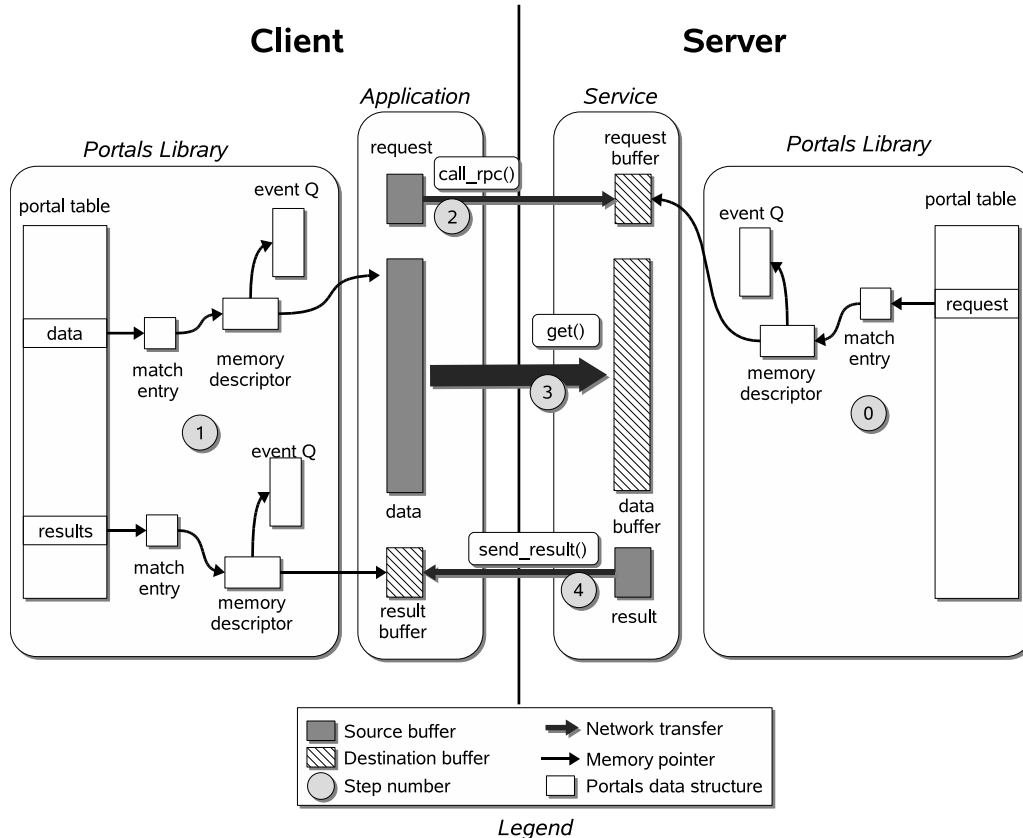


Figure 3. The figure illustrates the required data structures and protocol for calling and executing a remote `READ()` function using the LWFS RPC library. The protocol is initialized at the Server (0) and Client (1); the Server allocates Portals structures for incoming requests and the Client allocates Portals structures for data and result buffers. A request is initiated by the Client (2) by placing a short read request in the Server’s request buffer. The Server gets data from the client’s data buffer while processing the read request (3). Finally (4), the Server puts the result in the client’s result buffer after processing the request.

When a remote service starts, before it can accept remote requests, it allocates a buffer for incoming requests and creates all the necessary Portals data structures required to direct an incoming request to the right location in the buffer. We illustrate this process as *step 0* in Figure 3. Although the details are not important in the context of this paper, the required Portals data structures include a memory descriptor to describe the buffer for incoming requests, a match list used to identify appropriate messages, a Portal table to index match lists in the Portals library, and an event queue that contains a log of successfully matched messages. Once the server creates these data structures, it registers a new service description in a known service registry and spawns a thread to processes incoming messages (i.e., requests) that get logged to the event queue. We show the type definitions

for the service descriptor and remote memory addresses in Figures 4 and 5.

After a client looks up the service description from the service registry, it can send a `READ()` request to that service with the `LWFS_CALL_RPC()` function. The parameters for the `LWFS_CALL_RPC()` function include a service descriptor for the remote service, an opcode to identify the function, and buffers for the arguments, data, and the result. The final parameter is a request structure that identifies a pending asynchronous request.

The first task for the `LWFS_CALL_RPC()` function is to allocate the Portals data structures for the data buffer and the result buffer (*step 1* from Figure 3). This process results in two remote memory addresses that are marshalled, along with the opcode and arguments, into a single control mes-

```

struct lwfs_service {
    lwfs_rma req_addr; /* Where to put requests */
    lwfs_rpc_encode rpc_encode; /* How to encode reqs */
};

```

Figure 4. The service descriptor includes a remote memory address that identifies where to PUT() incoming requests, and a variable that identifies the encoding scheme used to marshal request parameters (the current implementation only supports XDR).

```

struct lwfs_rma {
    lwfs_buffer_id buffer_id; /* Portals index */
    lwfs_size len; /* Length of the buffer */
    lwfs_match_bits match_bits; /* Match bits */
    lwfs_remote_pid match_pid; /* Processor ID */
    lwfs_size offset; /* Buffer offset */
};

```

Figure 5. The remote memory address contains values required by Portals to access a memory buffer on a remote node.

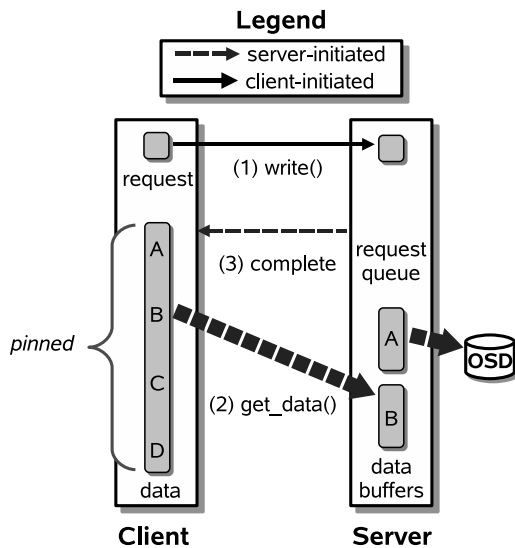


Figure 2. An LWFS I/O server controls data movement by pulling data from the client for writes or pushing data to the clients for reads.

sage. In *step 2*, the client puts the control message into the request buffer on the remote service and returns control to the client.

Step 3 shows what happens when the server receives a request from the client. The server first decodes the message to identify the opcode, procedure arguments, and remote addresses it needs to perform the operation. In this example in Figure 3, the operation is a READ() that causes the server to *get* data from the client's data buffer using the Portals one-sided messaging API. The request contained the remote memory address that identifies the client-side memory location of the buffer.

The final step, *step 4*, is to put the result in the result buffer of the client. In the case of a READ() operation, the result is an integer representing the number of bytes transferred and written to disk by the server.

It is important to note that LWFS_CALL_RPC() is an asynchronous operation. The client does not have to sit idle waiting for the remote operation to complete. When the client is ready for the result, it calls the LWFS_WAIT() function that blocks the client until the specified remote request is either complete (possibly with an error) or has timed out. When the LWFS_WAIT() function returns, the client may release or re-use all buffers reserved for the remote operation.

```

int LWFS_CALL_RPC (
    lwfs_service *svc; /* Remote service descriptor */
    lwfs_opcode opcode; /* Identifies the operation */
    void *args; /* Args buffer */
    void *data; /* Data buffer */
    lwfs_size data_len; /* Length of data buffer */
    void *result, /* Result buffer */
    lwfs_request *req); /* The request structure */
};

```

Figure 6. The `LWFS_CALL_RPC()` function has parameters for the remote service, the opcode, and buffers for the arguments, data, and result. The request structure identifies the pending request so the client can wait for the result using the `LWFS_WAIT()` function.

5 Analysis

To investigate the performance of the LWFS data-movement protocols, we ran experiments that compare the throughput (measured in MB/sec) of the LWFS RPC mechanism to three other approaches: TCP/IP, Sun’s RPC, and Sandia’s reference implementation of Portals 3.3. Each experiment consists of two processes: a client that sends an integer followed by an array of 16-byte data structures to the server, and a server that receives the array and returns a single data structure to the client. Each 16-byte data structure (defined in Figure 7) consists of a 4-byte integer, a 4-byte float, and a 8-byte double.

We ran all experiments on a cluster of 32 IA32 compute nodes, each with 2-way SMP Dell PE1550 processors (1.1 GHz PIII Xeons) with a Myrinet 2000 interconnect. For the LWFS experiments, we used the reference implementation of Portals 3.3 that is not optimized to use the Myrinet GM library. In all cases, the lowest-level transport was TCP/IP over Myrinet.

For the raw TCP/IP version, the client makes a socket connection to the server, writes the length of the buffer, writes the buffer, then reads the result. When the server receives the size of the buffer, it allocates space for the incoming array, reads the buffer, then writes back the first data structure in the buffer.

For the raw Portals version, the client and server perform the same steps as the TCP version, except that a send from the client involves a `PTLPUT()` operation from the client buffer into the remote buffer on the server. Like the TCP/IP version, when the server receives the length of the incoming buffer, it allocates space for the array, waits for the buffer to arrive (an event gets logged to the receive buffer’s event queue), then sends the first value of the array back to the client.

In contrast to raw TCP and Portals, the Sun RPC version encodes and transfers the entire array in one transac-

tion. The decoding mechanism on the server side allocates the necessary space for the incoming array and reads and decodes the data as it is transferred to the server.

The LWFS version works much like the Portals version except it encodes/decodes a small request that includes the size of the incoming array. The server then fetches the data from the client-side array using the one-sided `PTLGET()` function.

Figure 8 shows the throughput in MB/Sec of the various data-transport schemes as we increase the size of the data array. The worst performer is Sun RPC because it has to encode/decode all messages sent across the network. This is particularly inefficient because the XDR encoding scheme used by Sun RPC has to visit every data-structure in the array, adding a substantial processing burden on both the client and the server. The plot flattens out around 32KB—at which point the transfer becomes bound by the CPU processing required for the XDR encoding.

The LWFS implementation also uses XDR to encode requests, but it transfers the array (the data portion) in “raw” binary format (the assumption is that the server knows the format of the data and does not need to do a conversion), allowing for larger buffer transfers and server-control of the transport. The LWFS scheme achieves almost exactly the same throughput as the raw Portals version, notable in light of the fact that LWFS uses Portals as a middleware layer, and demonstrating that the overhead of encoding/decoding the request structure is minimal for large requests.

Just below the performance of the raw Portals and LWFS, but following the same general shape, is the raw TCP/IP implementation. This may seem a bit strange since our Portals (and thus the LWFS) implementation is layered on top of TCP/IP. We believe that the raw TCP (that uses default settings) performs worse due to TCP/IP optimizations implemented in the Portals library. Such optimizations are common among HPC data-transport protocols that use TCP/IP [DMT02].

```

struct data_t {
    int int_val;      /* 4 bytes */
    float float_val; /* 4 bytes */
    double double_val; /* 8 bytes */
};

```

Figure 7. The 16-byte data structure used for each of the experiments.

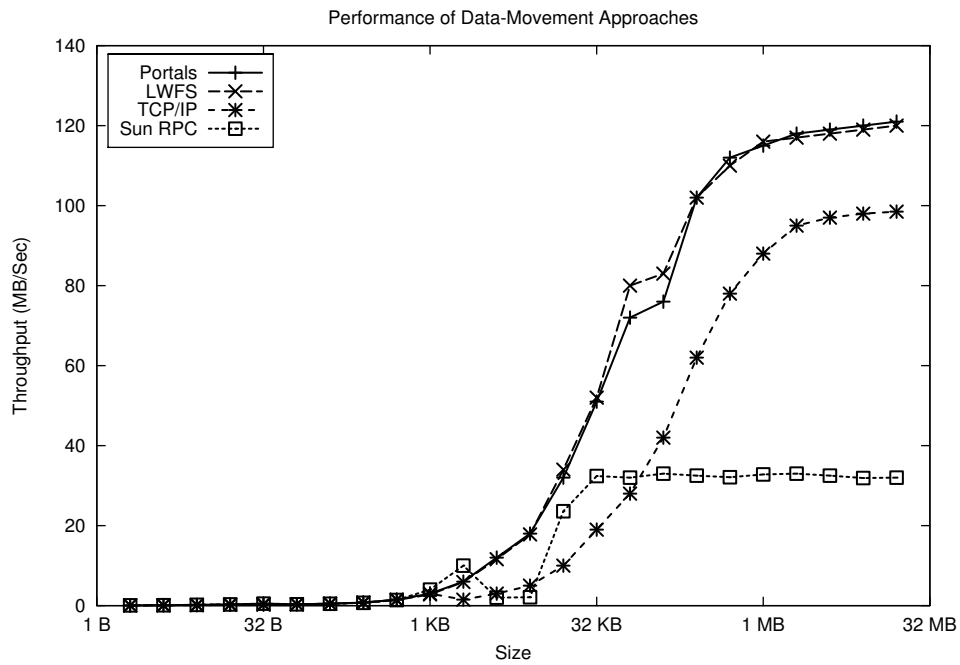


Figure 8. Comparison of LWFS RPC to various other mechanisms.

6 Related Work

There is a large body of work [BALL89, vECGS92, SB90, BF97] that has explored various optimizations to the basic remote procedure call paradigm, from optimizing local calls to taking advantage of shared memory arrangements or special hardware. The LWFS RPC system represents another optimized implementation of basic RPC semantics, using the server-directed approach to improve scalability.

RDMA-based approaches such as described in [LWK⁺03] rely on hardware support from specialized network interfaces such as Infiniband [Ass00] to transfer data between address spaces, and have been used to implement data movement for storage systems [WWP03]. LWFS uses Portals in a similar fashion; however, Portals is an API that can be implemented using various communication technologies including TCP, reliable UDP, and hardware-supported RDMA.

Portals does not specify a particular wire-format to be used. Sun's XDR encoding is used by LWFS and many other systems for this purpose, but more efficient alternatives (especially for the larger data sizes at which Portals is more efficient in transmission) exist. A comparison of wire-formatting approaches is presented in [BESW00]. The LWFS RPC design does not preclude the use of any particular data encoding, and more efficient encoding/marshaling may well result in more efficient data movement.

7 Summary

We have described a novel method of accomplishing efficient data movement in the context of a storage system designed for lightweight I/O, the Lightweight File System. LWFS communicates using a custom RPC layer on top of a mature, efficient data transfer API, Portals. Our experimental results show that the data-movement strategy implemented in LWFS is entirely appropriate in the very large scale environments for which it was designed. Our future work includes investigating more efficient data encoding strategies for LWFS and further profiling on production-scale machines.

References

- [Ass00] InfiniBand Trade Association. Infiniband architecture specification, release 1.0, October 24 2000.
- [BALL89] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 102–113, New York, NY, USA, 1989. ACM Press.
- [BESW00] Fabian E. Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener. Efficient wire formats for high performance computing. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 39, Washington, DC, USA, 2000. IEEE Computer Society.
- [BF97] Angelos Bilas and Edward W. Felten. Fast RPC on the SHRIMP virtual memory mapped network interface. *Journal of Parallel and Distributed Computing*, 40(1):138–146, 1997.
- [BHMR99] 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.
- [CT03] 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, Gleneden Beach, Oregon, April 2003.
- [DMT02] Tom Dunigan, Matt Mathis, and Brian Tierney. A TCP tuning daemon. In *Proc. Supercomputing 2002*, Baltimore, Maryland, November 2002.
- [GBF⁺97] 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.
- [Kot01] 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.
- [LWK⁺03] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High performance RDMA-based MPI implementation over InfiniBand. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 295–304, New York, NY, USA, 2003. ACM Press.
- [MW93] Arthur B. Maccabe and Stephen R. Wheat. Message passing in PUMA. Technical Report SAND-93-0935C, Sandia National Labs, 1993.
- [RBP⁺06] Rolf Riesen, Ron Brightwell, Kevin Pedretti, Arthur B. Maccabe, and Trammell Hudson. The Portals 3.3 Message Passing Interface. Technical Report SAND2006-0420, Sandia National Laboratories, Albuquerque, New Mexico, April 2006.
- [SB90] Michael D. Schroeder and Michael Burrows. Performance of the Firefly RPC. *ACM Trans. Comput. Syst.*, 8(1):1–17, 1990.
- [SCJ⁺95] 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.
- [SM86] Robert J. Souza and Stephen P. Miller. UNIX and remote procedure calls: A peaceful coexistence? In *Proc. 6th International Conference on Distributed Computing Systems*, pages 268–277, Cambridge, MA, June 1986.
- [Sun87] Sun Microsystems. XDR: External data representation standard. IETF RFC 1014, June 1987.
- [Tea02] The BlueGene/L Team. An overview of the BlueGene/L supercomputer. In *Proceedings of SC2002: High Performance Networking and Computing*, Baltimore, MD, November 2002.

- [vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 256–266, New York, NY, USA, 1992. ACM Press.
- [WWP03] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar Panda. PVFS over InfiniBand: design and performance evaluation. In Chusong 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.