Unifier: Unifying Cache Management and Communication Buffer Management for PVFS over InfiniBand *

Jiesheng Wu[1] † Pete Wyckoff[2] Dhabaleswar Panda[1] Rob Ross[3]

[1]Computer and Information Science
The Ohio State University
Columbus, OH 43210
{wuj, panda}@cis.ohio-state.edu

[2]Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
pw@osc.edu

[3]Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
rross@mcs.anl.gov

Abstract

The advent of networking technologies and high performance transport protocols facilitates the service of storage over networks. However, they pose challenges in integration and interaction among storage server application components and system components. In this paper, we put forward a component, called Unifier*, to provide more efficient integration and better interaction among these components. Unifier has three notable features. (1) Unifier integrates cache management and communication buffer management. It offers a single copy data sharing among all components in a server application safely and concurrently. (2) It reduces memory registration and deregistration costs to enable applications to take full advantage of RDMA operations. (3) It provides means to achieve adaptation, application-specific optimization, and better cooperation among different components.*

This paper presents the design and implementation of Unifier. This component has been deployed and evaluated in a version of PVFS1 implementation over InfiniBand. Experimental results show performance improvements between 30% and 70% over other approaches. Better scalability is also achieved by the PVFS I/O servers.

*This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506, and National Science Foundation's grants #EIA-9986052, #CCR-0204429, and #CCR-0311542.

†Work done while visiting Mathematics and Computer Science Division, Argonne National Laboratory during Summer 2003.

1 Introduction

Network storage systems are increasingly becoming a mainstream solution for I/O intensive applications in various domains, such as data-centers, high performance computing systems, and the corporate computing environments. Network storage systems provide potentials to achieve high performance, scalability, reliability, and manageability. However, performance of network storage systems is often limited by the low performance of network subsystem [1, 2, 22, 28, 29].

The advent of networking technologies and high performance transport protocols facilitates the service of storage over networks. Emerging network architectures such as Virtual Interface (VI) Architecture [15] and InfiniBand Architecture [21] (IBA) provide two key features, namely *user-level networking* and *remote direct memory access* (RDMA), to offer low latency, high throughput, and low CPU overhead communication in network storage systems. These enabling technologies eliminate or reduce costs of memory copy, network access, interrupt, and protocol processing in the network subsystem. However, there are a number of challenges to be addressed [39, 23, 16, 36, 37]. One of the most significant issues is efficient communication buffer management to reduce memory registration and deregistration costs.

Another source of performance limitation in network storage systems is the lack of integration among various system components (the file cache, the file system, and the network subsystem) and the storage server applications in the general-purpose operating system [18, 25, 4]. This often results in redundant data copying, multiple buffering, and other performance degradation [25]. Redundant mem-

ory copying leads to high CPU overhead and limited server throughput. In networks such as IBA that provides comparable performance to the memory system, this can be the primary performance bottleneck. Multiple buffering of data wastes memory. Consequently, the effective size of cache space is reduced, increasing cache miss rates and disk accesses. In addition, the narrow interface [18, 4, 3, 19, 31] between system components and applications becomes a barrier to achieving efficient cooperation.

In this paper, we present the design, implementation, and evaluation of *Unifier*. Unifier is a component in server applications such as network storage system servers and other I/O serving applications (e.g., Web servers). It attempts to enable efficient interaction and integration among all components of the server application. Unifier is designed to improve the performance of server applications. In particular, Unifier has three main goals. First, Unifier eliminates redundant data copying in the I/O path. Each data object can have only one copy in the whole system which is shared by all application components and system subsystems safely and concurrently. Unifier also eliminates multiple buffering of data, thus the cache size is effectively increased. Second, Unifier serves as a buffer manager to provide buffers to RDMA operations in the emerging network technologies. Unifier tries to manage these communication buffers in a manner to reduce memory registration and deregistration costs as much as possible. Therefore, the server application can take full advantage of RDMA-capable networks such as InfiniBand. Third, Unifier provides an application-level cache to achieve cache adaptivity and application-specific cache optimization. It provides expressive interfaces to achieve better cooperation among components.

A prototype of Unifier was implemented as a stand-alone component. It has well-defined interfaces. It also allows flexible accesses to the underlying file and storage systems via various interfaces. This component can be deployed in a wide range of server applications as both an application-level cache manager and a communication buffer manager for RDMA operations. In this paper, we focus on the design of Unifier over InfiniBand network and its deployment in an implementation of PVFS1 over InfiniBand [36, 37]. Our central performance results are the performance of the PVFS1 implementation with Unifier, in addition to other micro-benchmarks to measure the cache performance itself.

Experimental results show that the Unifier can offer a factor of improvement between 1.3 and 2.7 over the existing approaches in a simple client/server architecture. The Unifier method also increases the effective cache size due to the integration of communication buffers and the cache buffers, leading to increased performance. Performance results of PVFS1 with Unifier show performance improvements between 30% and 70% over two other methods often used in the PVFS I/O server implementation. Better scalability is also achieved by the PVFS I/O servers.

The rest of the paper is organized as follows. We first introduce PVFS and InfiniBand in Section 2. Section 3 describes our motivation. Section 4 presents the design of Unifier, including its architecture, API, potential benefits, and design issues. Sections 5 gives an overview of the prototype implementation of Unifier. The performance results are presented in Section 6. We examine related work in Section 7 and draw our conclusions and discuss future work in Section 8.

2 Overview of PVFS and InfiniBand

In this section, we give a brief overview of both PVFS and InfiniBand.

2.1 Overview of PVFS

PVFS is a leading parallel file system for Linux cluster systems. It was designed to meet increasing I/O demands of parallel applications in cluster systems. As of this writing, *PVFS Version 2* (PVFS2) [27] has just been released. The PVFS overview in this section is about PVFS1, though some basic concepts may be applied to PVFS2 as well.

As shown in Figure 1, a number of nodes in a cluster system can be configured as I/O servers and one of them is also configured to be the metadata manager. It is possible for a node to host computations while serving as an I/O node.

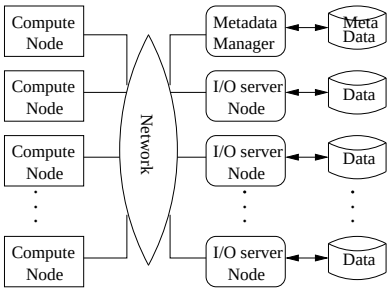

Figure 1. Typical PVFS setup.

PVFS achieves high performance by striping files across a set of I/O server nodes to achieve parallel accesses and aggregate performance. An I/O daemon runs on each I/O node and services requests from compute nodes, particularly read and write requests. Thus, data is transferred directly between I/O servers and compute nodes. PVFS uses the native file system on the I/O servers to store individual file stripes. A manager daemon runs on a metadata manager node. It handles metadata operations involving file permissions, truncation, file stripe characteristics, and so on. Metadata is also stored in the local file system. The metadata manager provides a clusterwide consistent name space to applications. In PVFS, the metadata manager does not participate in read/write operations.

2.2 Overview of InfiniBand

The InfiniBand Architecture [21] defines a System Area Network for interconnecting both processing nodes and I/O nodes. It provides a communication and management infrastructure for inter-processor communication and I/O.

Both channel and memory semantics are available for transferring data. In channel semantics, send/receive oper-

ations are used for communication. In memory semantics, Remote Direct Memory Access (RDMA) write and read operations are used. A basic requirement in the current SDK of InifniBand is that data buffers should be registered before any communication.

3 Motivation

In this section, we first discuss three data transfer methods in PVFS over TCP/IP. Then, we analyze issues with these methods when we design and implement PVFS over InfiniBand. This analysis serves as our motivation.

3.1 PVFS Data Transfer over TCP/IP

The I/O path in a PVFS I/O server combines both network I/O operations and file I/O operations. Therefore, the efficiency of PVFS I/O servers relies on performance of both operations, as well as the interaction between their associated subsystems: the network subsystem and the file system. In the implementation of PVFS over TCP/IP, three data transfer methods can be provided, reflecting different interactions.

Normal: In the *Normal* method, a PVFS server translates a PVFS read request into two separate calls: a file read call and a network write call. Similarly for a PVFS write request, it is translated to a network read call and a file write call. As analyzed in [32], there are usually four context switches. There are at least two data copies: copy between the user buffer and the file cache, and copy between the user buffer and the network buffer.

Mmap: The *Mmap* method maps the requested part of a file into the application user space using the system call `mmap(2)`. Then an application read or write on the mapped buffers results in a file read or write. This avoids data copy between the user buffer and the file cache. But the context switches remain same. The improvement comes at the cost of several constraints, complicated memory management, and error-prone pitfalls [32].

Sendfile: `sendfile(2)` is a system call providing direct data transfer between two file descriptors, including a TCP socket descriptor. Using sendfile, a PVFS server can do the file read and the network write together in one call. This reduces not only context switches, but also two data copies as mentioned in the Normal method. Over networks with Zero-copy TCP/IP implementation, the *Sendfile* method enables Zero-copy I/O path for transmitting data from the file to the network [32]. However, there is no support for *recvfile-like semantics*. That is, to serve a PVFS write request, the I/O server should follow either the Normal or the Mmap method.

Note that in the PVFS implementation, PVFS write uses the Normal method. PVFS read uses the Mmap method or the Sendfile method. Users can choose one of them when they compile PVFS.

3.2 Data Transfer Issues in PVFS over InfiniBand

In [36], we designed and implemented a version of PVFS (PVFS 1.5.6) over InfiniBand. Our results show that redesigning PVFS over the InfiniBand native transport layer is worthy with up to 3 times improvement over TCP/IP on the same IBA network when performance of the local file system is well balanced compared to the network system. The Normal and Mmap methods can be applied to PVFS over InfiniBand when we use the InfiniBand native transport layer, while we cannot use the Sendfile method directly. In addition, there are several issues to be addressed to further improve PVFS performance.

Data copying between different components: I/O data is copied between the file cache and PVFS server communication buffers. This happens when the Normal method is used. It also happens when we want to avoid dynamic memory registration and deregistration in the Mmap method. Data copying incurs high per-byte overhead for PVFS read and write operations.

Explicit communication buffer pool: To avoid expensive dynamic memory registration and deregistration, an often used solution is to pre-register a list of buffers and to keep using them for all communication. To serve a large number of requests concurrently, a significant amount of memory space should be allocated. Since these buffers are not swappable, they actually reduce the effective size of main memory, and thus the size and hit rate of the server's file cache.

Data duplication in communication buffers: When we use an explicit communication buffer pool, same data object may be in multiple communication buffers to serve different requests which access the same data object. This duplication reduces the efficiency of the communication buffers, leading to a possible increase of the communication buffers and service stalls.

Dynamic memory registration and deregistration: This happens when we use the Mmap method. As shown in [36], up to 35% performance can be degraded due to the costs of memory registration and deregistration.

These issues have a root in the lack of integration and interaction among the PVFS transport layer over InfiniBand, the file/storage component, and the underlying I/O subsystem. To solve these issues, we propose a component to unify the communication buffer space and the cache space. We deploy an application-level cache in this component. The cache space is directly used for communication. We call this component *Unifier*, working as both a cache manager and a communication buffer manager. It provides pre-registered communication buffers without reducing the effective cache size. It also offers other features to enable better cooperation with related components. We describe the detailed design of Unifier in Section 4 and a prototype implementation in Section 5.

4 The Design of Unifier

In this section, we present the design of Unifier. We start with its basic software architecture and its application programming interface (API), followed by its potential benefits and design issues.

4.1 Basic Software Architecture

Unifier is designed to provide efficient interaction between components in PVFS I/O servers. The basic architecture and its interaction with other components are shown in Figure 2.

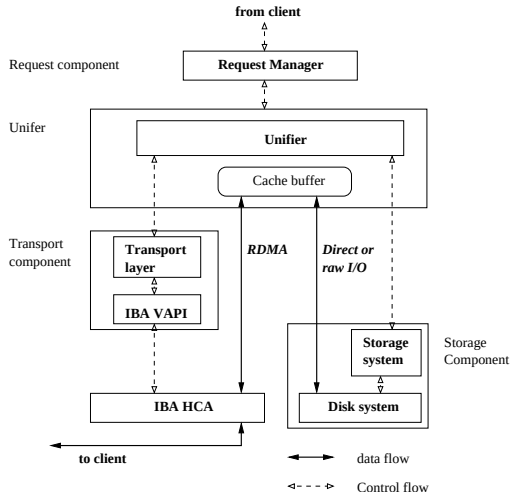

Figure 2. Basic software architecture of Unifier.

The control flow is shown by the dotted lines in Figure 2. Unifier, as a central hub, interacts with the request manager, the transport component, and the storage component. First, it receives requests from the request manger. Second, it provides cache buffers to serve these requests. Lastly, for a read request, it first talks to the storage component to read the requested data into its cache buffer if data is not cached. Then, it provides the same buffer to the transport component to transmit data to the client. For a write request, it first asks the transport component to receive data into its cache buffer. These data then is cached in the Unifier's cache buffer and flushed to the storage component at appropriate time.

The data flow is shown by the solid line. The data flow is simple. All data is placed in the Unifier's cache buffers. The cache buffers are also used by the transport component for communication, as well as the storage component for file and storage I/O operation. Given a data object, there is only one copy in the Unifier's cache buffers shared by all components safely and concurrently.

Unifier provides two main functionality. First, it acts as a cache manager, maintaining an application-level cache. It also hides the details of the storage component. Second, it acts as a buffer manager, providing buffers to the transport component. The cache buffer pool is managed in a way to enable efficient RDMA operations. Further, it intends to optimize cache management for better network performance, such as buffer coalescing and variable cache units.

4.2 Unifier Interface

The underlying observation that shapes our design of the Unifier API is that a high-performance API should adopt the lessons learned from the design of the high-performance server architectures. As a result, we provide the following features in the Unifier API.

Supporting structured data access: Structured data access is a common access pattern in many applications. Native structured data access support in each component is a key for high performance [33, 37, 13]. The Unifier API should cater to this requirement and enable possible optimizations for structured data access.

Supporting asynchronous operations: Asynchronous operations provide opportunities to overlap I/O operations with other processing. Network I/O operations in IBA are asynchronous. File and storage systems have been evolving to provide asynchronous I/O support [6]. Unifier API should provide an interface to support asynchronous operations and to take advantage of the advances in both network and storage I/O.

A more expressive interface: Significant research work has pointed out that narrow interfaces in the existing systems have become a barrier for different subsystems to exchange their semantic information to improve system performance [19, 31, 4]. A more expressive interface is expected, which allows more cross-subsystem optimizations and more flexible extended services.

Recognizing the importance of these features, we define a simple yet powerful Unifier's interface. This section briefly describes its interface. A complete discussion of the whole interface can be found in the PVFS2 document [27]. Currently, the interface includes five types of calls: 1) Post a request; 2) Check the request completion; 3) Query cache information; 4) Completion notification; 5) Release resources. As an example, we use `Unifier_post_read` to show how we achieve the aforementioned features in the Unifier API.

```
Unifier_post_read(int fd,
    ACCESS_Agg *    access_info ,
    BUFFER_Agg *    buffer_info,
    INFO_Agg *      semantic_info,
    COMP_Info *     comp_info)
```

In `Unifier_post_read`, *ACCESS_Agg* aggregates information of a structured access. This aggregate structure can be easily represented by an MPI Datatype if other components accept Datatype directly [13], or a representation of structured access. *INFO_Agg* contains semantic information the caller wants to pass to the Unifier. Currently, we only support cache policy selection and the cache unit size. We intend to extend this to convey more information to Unifier for optimization and for differential services.

COMP_Info guides Unifier to set up the completion notification. The Unifier_post_read operation returns buffers which hold the requested data. We use *BUFFER_Agg* to aggregate a list of buffers. These buffers will be provided to the transport component for communication.

4.3 Potential Benefits

The primary goal of Unifier is to improve the performance of PVFS I/O servers. It offers the following potential benefits.

1. **Zero-copy I/O serving:** Unifier eliminates data copying between PVFS server components in the I/O path. Further, it maintains an application-level cache which enables the storage component to bypass the operating system file/storage cache without losing performance. Therefore, Unifier can achieve the minimal number of data copies to the extent permitted by the hardware. Zero-copy I/O serving path is easily achieved in a typical I/O server hardware setup over InfiniBand, as shown in Figure 2.

2. **Increased cache size:** Unifier eliminates all multiple buffering. Each object can have only one single copy in the Unifier's cache buffer. This actually increases the effective cache size, and thus the cache hit rate. Considering the increasing gap between the memory system and the disk system and the increasing gap between the network system and the disk system, a small increase in the cache hit rate can improve the performance of I/O intensive applications significantly.

3. **Reduced memory registration and deregistration costs:** A part if not all of the cache buffers in Unifier can be pre-registered for communication without any memory registration or deregistration cost on these buffers.

4. **Native structured data access support:** We kept the structured data access support in mind from the beginning when we designed Unifier. This support not only fits application common access patterns well, but also provides tremendous optimization potential in both Unifier and other components. For example, the storage component can perform optimizations such as active sieving on a structured data access [38].

In this paper, we focus on the above benefits. Many other potential benefits, such as providing cache information to the request scheduler for cache-aware scheduling, application-controlled caching policies, and moving hot data into the memory of the IBA Channel Adapter, are not discussed.

4.4 Design Issues

Unifier and the Unifier-based I/O server software architecture show very attractive potential benefits, however, several issues need to be addressed for this architecture to be used in real systems to achieve high performance. We consider the following three important issues, namely adaptive PVFS I/O server cache, buffer sharing, and the size of registered cache buffers.

4.4.1 Adaptive PVFS I/O server Cache

Application-level cache has been popularly used in many server applications, such as database management applications, web server applications [34], and Grid data servers [5], We could borrow these designs into the design of PVFS I/O server cache. We could also reuse the design of the system cache for general-purpose systems. However, the reason why we consider the design of PVFS I/O server cache is an issue is that applications using PVFS have different I/O workload characteristics and I/O requirements from that on other systems [35]. Compared to database applications, PVFS applications may have more diversified access patterns. On the other hand, compared to applications on general-purpose systems, PVFS applications may have less variation in access patterns. Therefore, the design of PVFS I/O server cache should reflect these differences and provide high performance in general. An adaptive cache to cater to various requirements is expected.

There is no "one size fits all" solution for a cache with fixed policies [30]. In our design, we attempt to increase the cache adaptivity from two aspects. First, we explicitly expose cache information to other components. Research work in [8, 4] has shown that applications can adapt their own behavior to that of the OS for improved performance with cache information. Unifier provides explicit cache information queries to enable adaptation. Second, we allow applications to specify their cache requirements. These requirements are passed down to Unifier. Consequently, different cache policies can be applied, different cache units can be used. Note that Unifier only provides best-effort services to these requirements. It is possible that some of them may be overruled [10].

4.4.2 Buffer Sharing

In Unifier, network read and write and file/storage read and write all share a single copy of a given data object. This results in problems of synchronization and consistency in buffer sharing. Techniques such as *immutable buffers* used in *IO-Lite* [25] can be used to solve these problems. Immutable buffers provide read-only buffer sharing to eliminate synchronization and consistency problems. However, it comes with a price that that data can not generally be modified in place. As also mentioned in IO-Lite, immutable buffers are not suitable for scientific applications where in-place modification is a must.

Because scientific applications are the main target of PVFS, we propose other means to solve the buffer sharing problems. We use an *allocate-release* model to manage and control sharing on the cache buffers. The main design points are as follows:

Single owner: The only owner of all cache buffers is Unifier. This implies that Unifier has control on all buffer sharing. This method reduces the design complexity significantly.

Allocate: Unifier allocates the cache buffers to each operation. When a conflict sharing occurs, the allocation will be deferred. When there is no conflict sharing, the same cache buffers may be allocated to several concurrent operations. This enables safe and concurrent sharing.

Release: When an operation is granted with the cache buffers, it should release these buffers to Unifier when it completes.

With this design, Unifier supports both read-only sharing as well as write sharing. I/O data can be modified in place if it is not currently shared. Therefore, Unifier provides the sendfile semantics over InfiniBand transport protocols, which transmits data in the cache buffers directly to the network without any copy. It also provides a *recvfile*-like support that data received by the network is placed directly into the cache buffers which are associated with a data object in file/storage systems.

There are three reasons why we support the recvfile-like semantics which is not supported by the operating system on the traditional network protocols. First, the IBA network performance is comparable to the system memory system. Second, RDMA operations provide a "shared-memory illusion". To some extent, a process on a remote machine could be equally considered as a local process running on the same machine. Third, write sharing is very little in parallel applications [35]. A PVFS write can be done without affecting others. Therefore, providing recvfile-like support over InfiniBand can improve performance of PVFS writes without costs in common cases. Even when write sharing does occur, since the network performance is high, the cost to maintain writing sharing is low.

4.4.3 The Size of Registered Cache Buffers

Another main goal of Unifier is to reduce memory registration and deregistration cost imposed by RDMA operations. Ideally, a part if not all cache buffers can be registered and be always ready for RDMA operations. However, there are several tradeoffs to be addressed to achieve this objective. First, the size of Unifier's cache should be as large as possible. Unifier should use all free memory as cache to increase cache hit rate. Due to dynamic memory demands, a static size may cause virtual memory penalties. Second, as many cache buffers as possible should be registered during the cache initialization. However, the size of registered cache buffers should be limited not to degrade the system performance. Because registered buffers are pinned and not swappable, the effective size of physical memory used for other purposes is reduced.

In our design, the cache buffers are divided into two groups: *Ready* and *Raw*. Ready buffers are registered and resident in the system during the Unifier's life time. Raw buffers are allocated during the cache initialization, but not registered. Communication on these buffers needs on-the-fly registration and deregistration. The size of Ready buffers is projected conservatively according to the estimate of memory needed by a PVFS server application with its maximum support of outstanding requests. The size of raw buffers is the total physical memory size subtracted by the size of Ready buffers and the size of memory needed by a PVFS server application with a light load. With this design, we can achieve a good tradeoff between the cost of memory registration and deregistration and the cost of potential virtual memory activities.

5 Implementation

This section gives an overview of the implementation of the Unifier component and its deployment in PVFS over InfiniBand.

Unifier is implemented as a user-level component in PVFS software architecture [27, 36]. As a prototype implementation, the cache implementation is mostly based on the file cache implementation in Linux 2.6. Our implementation supports variable cache unit sizes from 4 KBytes to 64 KBytes. Applications can advise Unifier to choose a cache unit size for a file when the file is first opened. Unifier uses the *O_DIRECT* support to read and write file data with bypass of the system cache. Unifier provides both polling and callback completion notification. The callback completion notification depends on the support of callback completion notification provided by the underlying storage component. To support structured data access, our current implementation uses a list of <*offset, length*> pairs to represent a structured data access and cache buffers. This is compliant with both PVFS1 and PVFS2 implementations where the request manager interprets the high-level abstraction (e.g. MPI Datatype) of structured data access.

The deployment of Unifier in PVFS is straightforward, as shown in Figure 2. In the current implementation, Unifier provides explicit information queries to the request manager. However, how to make use of the cache information is under study. We are also working on the adaptive cache management.

6 Experimental Results

In this section, we provide three sets of results. First we show the basic results of the network, the file system, and the memory system. Next, we compare the micro-benchmark level performance of Unifier with the Normal and Mmap methods. Lastly, we analyze the performance of a PVFS implementation over InfiniBand with the deployment of Unifier. The PVFS implementation over InfiniBand is based on the PVFS 1.5.6 release. Details can be found in [36].

All our experiments used the following experimental testbed. A cluster system consisting of 8 nodes built around SuperMicro SUPER P4DL6 motherboards and GC chipsets

which include 64-bit 133 MHz PCI-X interfaces. Each node has two Intel Xeon 2.4 GHz processors with a 512 kB L2 cache and a 400 MHz front side bus. The machines are connected with Mellanox InfiniHost MT23108 Dual-Port 4x HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The Mellanox InfiniHost HCA SDK version is thca-x86-0.2.0-build-001. The adapter firmware version is fw-23108-rel-1_18_0000. Each node has a Seagate ST340016A, ATA 100 40 GB disk. We used the Linux 2.4.7-10 kernel. Unless stated otherwise, the unit megabytes (MB) in this paper is an abbreviation for 2^{20} bytes.

6.1 Basic System Performance Results

Performance realized by PVFS applications depends on the performance of three main subsystems: the network, the memory, and the file system. Table 1 compares the throughputs of IBA VAPI Send/Recv, RDMA Write, RDMA Read, memory copy, file read and write with and without cache. In the IBA throughput tests, memory registration and deregistration costs are not included. In the memory copying test, the amount of data copied is 20 MBytes, much larger than L1 and L2 caches to eliminate cache effect. The *bonnie* [20] file-system benchmark is used to test the file system performance .

Table 1. Throughput of different subsystems

Subsystem	Throughput (MB/s)
VAPI Send/Recv	830
VAPI RDMA Write	830
VAPI RDMA Read	826
Memory Copying	596
File Read w/o cache	20
File Write w/o cache	25
File Read w/i cache	590
File Write w/i cache	476

Memory registration and deregistration costs are crucial for us to leverage InfiniBand features. Figure 3 shows these costs with different buffer sizes using Mellanox fast memory registration extension in VAPI [24]. Note that much higher costs should be paid if we use VAPI regular memory registration facilities. We show two types of deregistration. One is single deregistration, labeled by *Dereg*. Another is batched deregistration. Multiple deregistration operations are done in one call. The batched number is 60. The average cost of each operation is reported by *Batched Dereg*. We can see that the total registration and deregistration costs are significantly high. This is the reason why we make great effort in Unifier to reduce these costs.

It can be seen that there is a large difference in bandwidth realizable over the network and the memory system compared to that which can be obtained to a disk-based file system without cache effect. However, applications can still benefit from fast networks for many reasons in spite

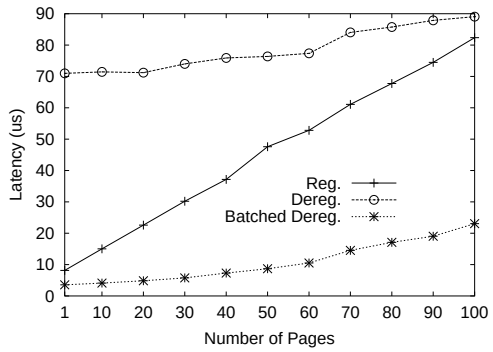

Figure 3. Costs of Memory Registration and Deregistration.

of this disparity. Data is frequently in server memory due to file caching and read-ahead when a request arrives. Also, in large disk array systems, the aggregate performance of many disks can approach network speeds. Caches on disk arrays and on individual disks also serve to speed up transfers. Therefore, the following experiments are designed to stress the network data transfer and independent of any disk activities. We consider data is cached. The results are representative of workloads with sequential I/O on large disk arrays or random-access loads on servers which are capable of delivering data at network speeds from a well-balanced storage system.

6.2 Performance of Micro-benchmarks

In this section, we designed several micro-benchmarks to show the performance of Unifier. We put Unifier in a simple client-server environment, which is similar to the PVFS architecture but simpler. In these tests, a client sends one or more read or write requests to a server. The server then serves these requests using three different methods: *Normal*, *Mmap*, and *Unifier*, respectively. Details of Normal and Mmap methods are discussed Section in 3.1.

Cached read performance: We first measured the cached read performance of these three methods. In this test, all data is in the system cache in the Normal and Mmap method. All data is also in the *Ready* cache buffer in the Unifier method. We used this test to show the best case performance of all methods.

Figure 4 shows the results. The Normal method gives a peak bandwidth of 324 MBytes/sec. We see a small drop when the access sizes are larger than 128 KBytes, probably this is because the increase of the memory footprints affects the memory copy performance.

In the Mmap method, the memory registration and deregistration costs have a significant impact, particularly for small access sizes. When the access size increases, the costs of memory registration and deregistration become less than the cost of memory copy, this method performs better than the Normal method.

In the Unifier method, data is cached in the Unifier Ready

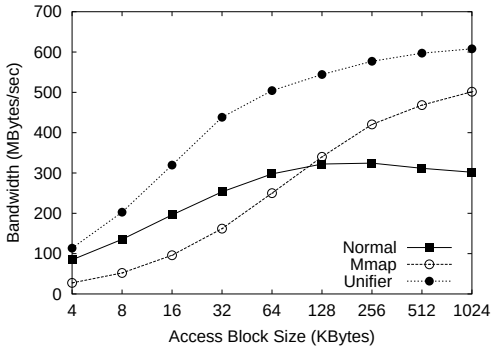

Figure 4. Cached read bandwidth.

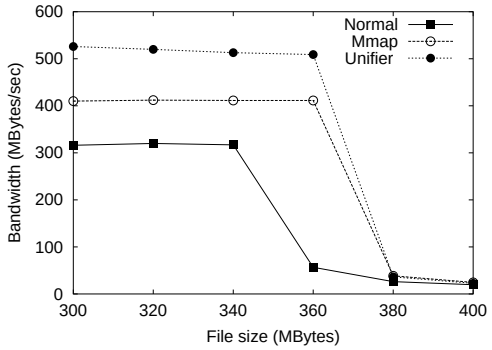

Figure 5. Effects of cache size.

cache buffers. Thus, the server can RDMA write data directly to the client buffer from its Unifier's cache buffers. Unifier achieves an improvement of a factor of 2.1 over the Normal method, a factor of 1.3 over the Mmap method when the access size is large, a factor of up to 2.7 over the Mmap method when the access size is small.

Effects of cache size: As discussed earlier, the effective cache size in each method is different. Given a system with 512 MBytes physical memory, the maximum size of memory which can be used for cache is around 420 MBytes. In our test, the server application consumes around 60 MBytes. Then around 360 MBytes memory can contribute to cache data. The Mmap and Unifier methods can make full use of these 360 MBytes for caching. However, since we need some pre-registered communication buffers in the Normal method, we allocate 20 MBytes for this use, thus, the effective cache size is around 340 MBytes. Note that to allow the server to serve a large number of concurrent requests in a real PVFS configuration, even a larger buffer pool may be needed. In the Unifier method, the maximum size of Ready buffers allowed by the system is around 200 MBytes. So that around 160 MBytes Raw buffers are in the Unifier cache, which requires dynamic registration and deregistration.

We used a *re-read* test to show the effects of cache size. In this test, the client reads a file whose size varies from 300 MBytes to 400 MBytes. This test reads a file sequentially with the block size of 128 KBytes. Then, it reads the same file again sequentially. The bandwidth achieved by the second read is reported in Figure 5. We can see that both the Mmap and the Unifier methods can still hold the entire file in the cache when its size is not larger than 360 MBytes, while the Normal method can not. When the file size increases to 380 MBytes, all methods suffer due to the disk-bound access on a normal IDE disk which can offer a read bandwidth of 20 MBytes/sec. All methods are comparable. This also shows that the Unifier cache can provide comparable performance to the system cache with the sequential workload.

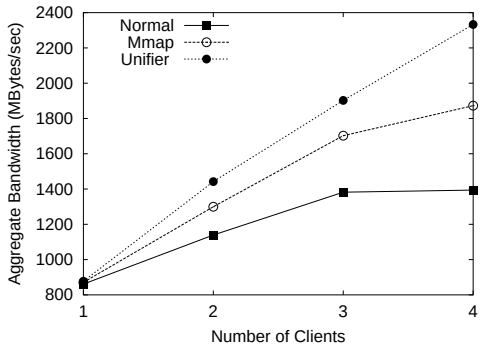

Figure 6. PVFS cached read performance.

6.3 Performance of PVFS1 with Unifier

The test program used is *pvfs-test*, which is included in the PVFS release package. We followed the same test method as described in [11]. That is, each compute node writes and reads a single contiguous region of size $2N$ MB, where N is the number of I/O nodes in use. The number of I/O nodes was fixed at four, and the number of compute nodes was varied from one to four.

Figure 6 shows the cached read performance with different methods deployed in an implementation of PVFS over InfiniBand VAPI from our group. The aggregate bandwidth realized by all clients is reported. There are two observations. First, PVFS with Unifier scales better than other two methods. This is due to the lower CPU overhead needed to server each request in the Unifier method. In other methods, either memory copying or memory registration and deregistration consumes significant CPU cycles. Second, in terms of the peak bandwidth, the Unifier achieves an improvement of a factor of 1.7 over the Normal method, a factor of 1.3 over the Mmap method.

7 Related Work

There are three main areas which are related to our work, namely *Copy avoidance techniques*, *Information techniques*, and *Networked file and storage systems over*

RDMA. The literature on each area is large and rich, so we only cite a few representative samples.

Copy avoidance techniques: Techniques such as Fbuf [17], Zero-Copy TCP [14], Emulated copy [7], and Page remapping [2] were mainly proposed to eliminate the user-kernel data copy. Our work has its roots in these research aimed at reducing memory copy operations. However, as compared to these research, Unifier is designed to eliminate all the data copies along the data path including the network subsystem, server applications, and the file/storage subsystem. Perhaps the closest work to ours in spirit is IO-Lite [25]. IO-Lite is a unified I/O buffering and caching system for general-purpose operating systems. It allows applications, IPC, the filesystem, the file cache, and the network subsystem to share a single physical copy of the data safely and concurrently. Significant changes are required for all subsystems to leverage the advantages of IO-Lite. In contrast, our work has three important differences. First, our work focuses on specific server applications for I/O serving, such as PVFS, network storage systems, and web servers. Little intrusion is made to existing kernel components. Second, Unifier takes a different approach to provide in-place modification for write-sharing over RDMA-capable networks. Third, Unifier deploys an application-level cache. Another work close to ours is network-centric buffer cache organization [26]. Network-centric buffer cache organization avoids redundant data copying in a pass-through server which acts as a data conduits over the network for remotely stored data. It also caches data in a network-ready form for TCP/IP networks. The network-centric buffer cache is a secondary cache to the system cache, which requires a copy between them. Compared to this work, Unifier has its own cache and bypasses the system cache. No data copying is in the data path. Also, we make great effort to deal with issues associated with user-level networking and RDMA operations which are generally different from TCP/IP protocols.

Information techniques: The idea of exposing OS information to enable adaptation has been stated in a rich set of work. Different approaches have been taken. Exokernel [18] eliminates all fixed, high-level abstractions and exposes all information directly. A library operating system sits on the Exokernel to provide standard interfaces. Infokernel [4] explores kernel information by adding extra code into a commodity operating system. Gray-box systems [3] infers OS information by benchmarks and fingerprinting tools without modification of OS itself. The design and the interface of Unifier reflect the same idea. However, we provide expressive interface for interaction among components. We also focus on specific server applications, instead of general-purpose operating systems.

Networked file/storage systems over RDMA: Direct Access File System (DAFS) [23, 16], PVFS over InfiniBand [36], NFS over RDMA [9], iSCSI extension for RDMA [12], and many others have leveraged emerging network technologies to design high performance networked file/storage systems. These work mostly focuses on using RDMA operations to redesign the transport protocols and to make transition from traditional TCP/IP networks to RDMA-capable networks. In contrast, our work centers around integration and interaction among different components in network storage servers over RDMA-capable networks.

8 Conclusions and Future Work

Unifier is designed to improve the performance of network storage server applications. It provides three notable features. First, Unifier eliminates redundant data copying and multiple buffering in the I/O path. It provide a single data sharing among all components in a server application safely and concurrently over high performance networks such as InfiniBand. Second, the integration of communication buffer management and cache management reduces memory registration and deregistration costs as much as possible. This enables applications to take full advantage of RDMA operations. Third, Unifier provides means to achieve adaptation, application-specific optimization, and better cooperation among different components in a server application.

This paper presents the design and implementation of Unifier. We also deployed and evaluated this component in a version of PVFS1 implementation over InfiniBand. Experimental results from a prototype implementation show performance improvements between 30 and 70% over two other methods often used in the PVFS I/O server implementation. Better scalabilty is achieved by the PVFS I/O servers. The Unifier method also increases the effective cache size due to the integration of communication buffers and the cache buffers, leading to increased performance.

Unifier was started as a research component in the design of PVFS2 [27]. The integration of Unifier with other PVFS2 components, testing, and optimization are underway. We are also working on exploring other potential benefits, such as cache-aware request scheduling and variable cache policies and cache page sizes. The architecture as such could be used in other server applications such as DAFS, iSCSI storage servers, and data-center servers. We plan to have these case studies as our future work.

Acknowledgments

We are grateful to Phil Carns from Clemson University for providing us many insights into PVFS2 and many helps during our design and implementation. We would also like to thank Rajeev Thakur, Neill Miller and Robert Latham at Argonne National Laboratory for many discussions with us.

Software Availability

An alpha-test version of the component discussed in this paper is available in the latest PVFS2 “beta” release. We are working on further testing and optimization on it.

References

[1] American National Standard of Accredited Standards Committee ANSI NCITS T11.1 Technical Committee. Information Technology – SCSI on Scheduled Transfer Protocol (SST) Work Draft. July 2001.

[2] D. Anderson, J. Chase, S. Gadde, A. Gallatin, K. Yocum, and M. Feeley. Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet. In *Proceedings of the Usenix Technical Conference. New Orleans, LA.*, 1998.

[3] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Symposium on Operating Systems Principles*, pages 43–56, 2001.

[4] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James A. Nugent, and Florentina I. Popovici. Transforming policies into mechanisms with infokernel. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 90–105. ACM Press, 2003.

[5] Azer Bestavros, Robert L. Carter, Mark E. Crovella, Carlos R. Cunha, Abdelsalam Heddaya, and Sulaiman A. Mirdad. Application-level document caching in the Internet. In *Proceedings of the 2nd International Workshop in Distributed and Networked Environments (IEEE SDNE '95)*, Whistler, British Columbia, 1995.

[6] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. Asynchronous IO support in Linux 2.5. In *Ottawa Linux Symposium*, Ottawa, ON, Canada, Jul 2003.

[7] Jose Carlos Brustoloni. Interoperation of copy avoidance in network and file i/o. In *INFOCOM (2)*, pages 534–542, 1999.

[8] Nathan C. Burnett, John Bent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting gray-box knowledge of buffer-cache management. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 29–44, June 2002.

[9] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad. NFS over RDMA. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, pages 196–208. ACM Press, 2003.

[10] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file c aching. In *Proc. First USENIX Symposium on Operating Systems Design an d Implementation*, pages 165–178, November 1994.

[11] 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, 2000. USENIX Association.

[12] Mallikarjun Chadalapaka, Hemal Shah, Uri Elzur, Patricia Thaler, and Michael Ko. A study of iSCSI extensions for RDMA (iSER). In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, pages 209–219. ACM Press, 2003.

[13] Avery Ching, Alok Choudhary, Wei keng Liao, Robert Ross, and William Gropp. Efficient Structured Data Access in Parallel File Systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2003.

[14] H. K. Jerry Chu. Zero-copy TCP in solaris. In *USENIX Annual Technical Conference*, pages 253–264, 1996.

[15] Compaq, Intel, and Microsoft. VI Architecture Specification V1.0, December 1997.

[16] Matt DeBergalis, Peter Corbett, Steve Kleiman, Arthur Lent, Dave Noveck, Tom Talpey, and Mark Wittle. The Direct Access File System. In *Second USENIX Conference on File and Storage Technologies*. USENIX, April 2003.

[17] Peter Druschel and Larry L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Symposium on Operating Systems Principles*, pages 189–202, 1993.

[18] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[19] Gregory R. Ganger. Blurring the Line Between Oses and Storage Devices. CMU SCS Technical Report CMU-CS-01-166, December 2001.

[20] http://www.textuality.com/bonnie/. Bonnie: A File System Benchmark.

[21] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24, 2000.

[22] Chuck Lever and Peter Honeyman. Linux NFS Client Write Performance. In *Proceedings of the Usenix Technical Conference, FREENIX track, Monterey*, June 2001.

[23] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and performance of the direct access file system. In *Proceedings of USENIX 2002 Annual Technical Conference, Monterey, CA*, pages 1–14, June 2002.

[24] Mellanox Technologies. Mellanox IB-Verbs API (VAPI), Rev. 0.95, March 2003.

[25] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[26] Gang Peng, Srikant Sharma, and Tzi cker Chiueh. A Case for Network-Centric Buffer Cache Organization. In *In Hot Interconnect 11*, August 2003.

[27] PVFS2 Developer Team. Parallel Virtual File System, Version 2. http://www.pvfs.org/pvfs2/, Nov. 2003.

[28] M. W. Sachs and A. Varma. Fibre Channel. *IEEE Communications*, pages 40–49, Aug 1996.

[29] Prasenjit Sarkar, Sandeep Uttamchandani, and Kaladhar Voruganti. Storage Over IP: When Does Hardware Support Help? In *Second USENIX Conference on File and Storage Technologies*. USENIX, April 2003.

[30] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Evolving RPC for Active Storage. In *To appear in Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, San Jose, CA, October 2002.

[31] Muthian Sivathanu, Vijayan Prabhakaran, Florentina Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, CA, March 2003.

[32] Dragan Stancevic. Zero copy I: user-mode perspective. *Linux Journal*, 2003(105):3, 2003.

[33] Rajeev Thakur, William Gropp, and Ewing Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.

[34] Brian Tierney, William Johnston, and Jason Lee. A Cache-Based Data Intensive Distributed Computing Architecture For "grid" Applications. In *CERN School of Computing*, Sept. 2000.

[35] Mustafa Uysal, Anurag Acharya, and Joel Saltz. Requirements of I/O Systems for Parallel Machines: An Application-driven Study. Technical Report CS-TR-3802, Dept. of Computer Science, University of Maryland, College Park, 1997.

[36] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *the 2003 International Conference on Parallel Processing (ICPP 03)*, Oct. 2003.

[37] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. Supporting Efficient Noncontiguous Access in PVFS over InfiniBand. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2003.

[38] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. Supporting Efficient Noncontiguous Access in PVFS over InfiniBand. Technical Report, OSU-CISRC-05/03-TR, May 2003.

[39] Yuanyuan Zhou, Angelos Bilas, Suresh Jagannathan, Cezary Dubnicki, James F. Philbin, and Kai Li. Experiences with VI communication for database storage. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 257–268. IEEE Computer Society, 2002.