GNBD/VIA: A Network Block Device over Virtual Interface Architecture on Linux

Kangho Kim Jin-Soo Kim Sung-In Jung
Computer & Software Technology Laboratory
Electronics and Telecommunications Research Institute (ETRI)
Daejeon 305-350, Republic of Korea
{khk,jinsookim,sijung}@etri.re.kr

Abstract

This paper describes a design and implementation of GNBD/VIA, a Network Block Device (NBD) over Virtual Interface Architecture (VIA), and evaluates its performance on Linux-based cluster of PCs. VIA is a user-level memory-mapped communication model which provides zero-copy communication by removing the operating system from the critical communication path. Typically, an NBD layer offers the abstraction of a storage media across the network. GNBD/VIA attempts to improve the performance of the NBD layer by employing the lightweight VIA communication mechanisms between NBD servers and clients. To our best knowledge, GNBD/VIA is the first implementation of NBD on VIA.

GNBD/VIA outperforms the normal NBD placed on top of TCP/IP protocol stacks, and achieves the performance comparable to local disk devices, showing the read (write) bandwidth of 30.6MB/s (25.9MB/s) on the evaluation platform with UDMA100 hard disks and Emulex cLAN adapters.

1. Introduction

The System Area Network (SAN) is a key element in building scalable cluster systems by providing low latency and high bandwidth communication. However, the traditional communication models were unable to fully exploit the raw performance of the recent SANs operating at gigabit speeds, due to the high overhead added by software layers [8].

The Virtual Interface Architecture (VIA) [1] is an industry standard on user-level memory-mapped communication model, whose main objective is to reduce the communication overhead further for high-speed SANs. The basic idea in user-level communication is to factor out protection from the critical path of communication operations. To provide protected communication, two conditions must be satisfied. First, the kernel must grant the permission for a process to communicate with another process by providing a communication channel. Second, the network interface must multiplex user-level DMA performed through these channels. This support eliminates the need to trap into the kernel each time a send is executed, and makes the send operation lightweight. At the same time, no copy is necessary by sending data from the user space to a remote receive buffer and the end-to-end communication bandwidth approaches to the raw bandwidth provided by the network hardware. There are several hardware and software implementations of VIA today. Emulex (Giganet before) has a hardware VIA implementation called cLAN with drivers for Linux and Windows NT. VIA implementations at the firmware level are available for ServerNet (Tandem) and Myrinet (Myricom) interconnects. M-VIA [6] provides Linux software VIA drivers for various fast ethernet and gigabit ethernet adapters.

In this paper, we describe a design and implementation of a network block device (NBD) over VIA for a Linux-based cluster of PCs. The NBD is a software layer which offers the abstraction of a storage media across the network, where a remote server provides the real physical storage. NBD clients can access the server's disk device as if it were a local one through a virtual device created at the client side. The virtual device acts exactly like a traditional block device to client applications and it is even possible to make a file system on it using the UNIX `mkfs` command.

As each disk read/write request to the virtual device is delivered to the NBD server over the communication network, the performance of NBD heavily depends on the underlying communication performance. Our goal is to design and implement a highly efficient NBD layer which takes advantage of the low latency and high bandwidth characteristics of VIA, in order to minimize the performance gap

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 IEEE

between local disks and NBDs. The existing NBD layers utilize TCP/IP protocols to communicate between NBD servers and clients. However, if NBD servers and clients are interconnected through VIA-enabled SAN within a cluster, we can accelerate the performance of NBD by replacing the TCP/IP protocols with the lightweight VIA communication mechanisms. Among many NBD implementations, we have modified GNBD (GFS NBD), which is used to build GFS (Global File System) on IP-based networks [9].

The rest of the paper is organized as follows. Section 2 overviews the VIA and NBD. Section 3 discusses issues for designing GNBD over VIA. In section 4, we present the evaluation methodology and the performance results of our GNBD implementation over VIA. Finally, we conclude in section 5.

2. Background

2.1. Virtual Interface Architecture (VIA)

In the traditional network architecture, the operating system (OS) virtualizes the network hardware into a set of logical communication endpoints available to network consumers. The OS multiplexes access to the hardware among these endpoints. In most cases, the OS also implements protocols that make communications between connected endpoints reliable. This model permits the interface between the network hardware and the OS to be very simple. The drawback of this organization is, however, that all communication operations require a call or trap into the OS kernel, which can be quite expensive to execute. The demultiplexing process and reliability protocols also tend to be computationally expensive.

The Virtual Interface Architecture (VIA) eliminates the system processing overhead of the traditional model by providing each consumer process with a protected, directly accessible interface to the network hardware - a Virtual Interface (VI). Each VI represents a communication endpoint. A pair of VI endpoints can be logically connected to support bi-directional, point-to-point data transfers. A process may own multiple VIs exported by one or more network adapters. A network adapter performs the endpoint virtualization directly and subsumes the tasks of multiplexing, de-multiplexing, and data transfer scheduling normally performed by an OS kernel and device driver. An adapter may completely ensure the reliability of communication between connected VIs. Alternately, this task may be shared with transport protocol software loaded into the application process, at the discretion of the hardware vendor [1].

The efficiency of memory-mapped communication provided by VIA does not come for free. As various projects have started to use VIA or other memory-mapped communication libraries, it becomes obvious that the lack of

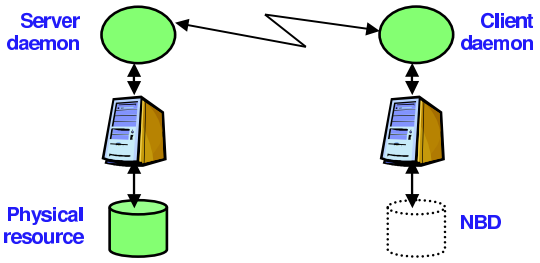


Figure 1. The concept of Network Block Device

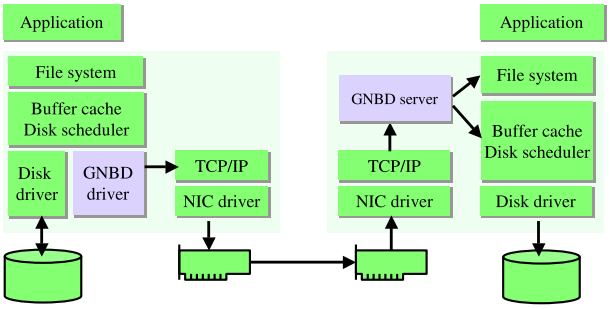


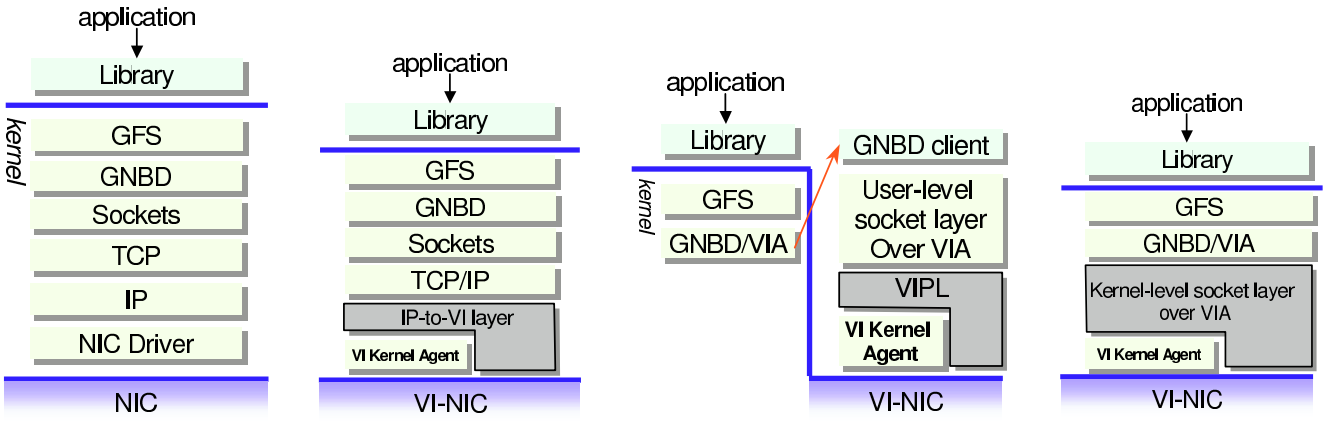
Figure 2. The structure of GNBD (GFS Network Block Device)

buffer management, flow control, automatic incoming message handling, etc. can make communication programming more complicated. One solution to this problem is to build high-level communication abstractions on top of VIA, while preserving its performance benefits. Recently, several communication libraries over VIA have been announced [5, 4].

2.2. Network Block Device (NBD)

Figure 1 illustrates the basic concept of Network Block Device (NBD). The NBD offers an access model that simulates a block device, such as a hard disk or a hard-disk partition, on the local client, but connects across the network to a remote server that provides the real physical storage. For clients, the device looks like a local disk partition, but it is only an entrance for the remote. Even though the actual access requests and data blocks are communicated on the network, the NBD layer hides all the details and the client simply uses the virtual device as if it were a local disk device. This is a little lower level and more basic than network file systems such as NFS or Samba, which require more kernel interaction to properly handle the file-level access requests from the remote host.

There are several implementations of the generic NBD: Linux/NBD, DRBD, ENBD, ODR, and GNBD. Linux/NBD is the basic NBD driver that is included in the Lin-



(a) The traditional GNBD (b) GNBD over LANEVI (c) GNBD over user-level socket layer (d) GNBD over kernel-level socket layer

Figure 3. Design alternatives for GNBD client

ux kernel. DRBD (Distributed Replicated Block Device) is used for mirroring file systems across the LAN, with a quick sync option for bringing the mirror up to date quickly. ENBD (Enhanced Network Block Device) is an extension of Linux/NBD. ODR (Online Disk Replicator) is similar to DRBD, but it supports more than two nodes and has a journaling feature. Finally, GNBD is the NBD included in GFS (Global File System), which is a modified version of the Linux/NBD mentioned above. The biggest difference between Linux/NBD and GNBD is that GNBD allows multiple clients to access the same block device concurrently while Linux/NBD driver only allows a single client at a time [2].

Among these NBD implementations, we have chosen GNBD as our target NBD layer, because GNBD running over VIA allows us to construct a GFS, a cluster file system, on VIA-enabled system area networks. The structure of GNBD is depicted in figure 2. Currently, GNBD is implemented as a block device driver in client side and as a kernel thread in server side, which are connected by TCP/IP networks.

2.3. Motivation

Generally, the NBD layer is built on top of the TCP/IP layer, which means the location of NBD server and client can be separated across the LAN or WAN. There are, however, many application domains of the NBD layer where the NBD server and client are closely linked within a cluster. For example, combined with software RAID driver, GNBD can provide a shared storage between two servers configured with a fail-over support. Another example is the GFS: it aggressively uses a set of GNBD servers and clients to implement a cluster file system on IP-based networks, even in the absence of Fibre Channel switch and storages. In these environments, communication within a cluster is a major factor which limits the overall performance. As the performance of NBD heavily depends on the underlying communication performance, it is important to extract the raw performance of network hardware as much as possible in order to minimize the performance gap between local disks and NBDs.

The TCP/IP protocol stack is not required to transfer data between two endpoints on the same cluster if the physical interconnect is reliable and provides transport-level functionality, as is done in VIA. Therefore, it is desirable for the NBD layer to run directly over the VIA bypassing the TCP/IP protocols. In this way, we can implement a highly efficient NBD layer which takes advantage of low latency and high bandwidth characteristics of VIA.

3. Design Issues

3.1. Design Alternatives

The original GNBD works on Sockets interface on top of the TCP/IP stacks as can be seen in figure 3(a). The GNBD can be implemented on the VIA-enabled system area networks in several ways: (1) by emulating IP layer over VIA, (2) by using a user-level Sockets layer over VIA, and (3) by using a kernel-level Sockets layer over VIA. We briefly examine the characteristics of each alternative and discuss which is the desirable choice for our goal.

Emulating IP layer over VIA. One solution to support GNBD on VIA is to use an adaptation layer between IP and VI Kernel Agent which emulates the IP layer over VIA, as shown in figure 3(b). This is the approach taken by the LANEVI (LAN Emulation on VI) [3] driver supplied by Emulex for its cLAN adapters. As IP is emulated on

VIA, an IP address is assigned to the VI-NIC and the system becomes fully compatible with any of the existing IP-based network applications including GNBD. However, it is not simple to emulate connectionless IP services on the connection-oriented VIA, and applications still suffer from the overhead of TCP/IP protocols [4]. In addition, our experience shows that the LANEVI module is not so stable as we have expected.

Using a user-level Sockets layer over VIA. As we have mentioned in section 2.3, the TCP/IP layer plays an unnecessary role in the communication within cluster systems connected by VIA-enabled networks. One way to bypass the TCP/IP protocol stack is to use a client daemon living in user-land which talks to a GNBD server also running in user-land. Both GNBD server and client communicate each other through an intermediate layer which emulates the Sockets interface over VIA at user-level. Such a user-level Sockets layer makes it easy to port the existing Sockets-based applications over VIA [4].

However, we still need to modify the GNBD code to fit into the new structure, and the context switching and data copying overhead between the kernel and user space become significant. The disk I/O requests arrived at the GNBD driver in kernel should be queued by the driver method and then pumped into the client daemon using a local socket or a special device. It means the client daemon enters repeatedly into the kernel in this design to copy data or the requests.

Using a kernel-level Sockets layer over VIA. The two approaches explained earlier do not effectively exploit the VIA's advantages even though they are built on the NIC that supports VIA mechanisms in hardware. Instead, we implement GNBD/VIA based on the kernel-level Sockets layer over VIA, as depicted in figure 3(d). We design and implement a slim layer which provides a subset of Sockets-like interfaces over VIA inside the kernel. We believe this approach is the simplest way to port GNBD over VIA with minimal efforts while producing the maximum performance. Although VIA enables user-level communication, the previous observation in figure 3(c) suggests it is not necessary to move the disk I/O requests and data to the user-level because such requests and data are already present in the kernel.

We show the detailed organization of our GNBD/VIA implementation on cLAN adapters in figure 4. In figure 4, `clan1k` is a driver module for cLAN hardware and `clanmgr` is a user-space daemon which is responsible for establishing and closing VIA connections. We describe the role of other layers, specifically KVIPL and VCONN, in the following subsections in detail.

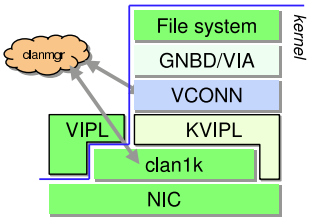


Figure 4. The layering of GNBD/VIA implementation

3.2. KVIPL (Kernel-level VIPL) layer

For user-level applications, the VIA specification defines a set of standardized API called VIPL (VI Provider Library). However, as the VIPL is provided in the form of a user-level library, it can not be used inside the kernel. Fortunately, the cLAN driver from Emulex has a set of kernel-level VI Provider Library or KVIPL, as part of its LANEVI layer.

Even though KVIPL is included in the cLAN driver, KVIPL does not provide a complete set of APIs for VI programming, because it is an unofficial submodule for implementing the LANEVI layer. For example, since it relies on `clanmgr` when a node connects or disconnects with a remote node, it does not provide APIs related to the connection management such as `VipConnectRequest()`, `VipConnectAccept()`, `VipConnectWait()`, `VipConnectReject()`, and `VipDisconnectVi()`. It also lacks blocking send / receive APIs such as `VipSendWait()` and `VipRecvWait()`, and error handling ones.

To get around this problem, we have extended the KVIPL so that it supports the same set of APIs as VIPL. In fact, our modified KVIPL is not so different as VIPL except handling of a receive descriptor. As both KVIPL and VIPL are consumers of `clan1k` driver and cLAN adapter, they show nearly the same structure and behavior.

Using the KVIPL layer offers a number of advantages. First, kernel codes access the VIA hardware essentially in the same way as user-level applications using the familiar VIPL interfaces. Without the KVIPL layer, kernel codes would have needed to access the `clan1k` driver and cLAN adapter directly. Second, the communication functions are well-modularized and the KVIPL layer alone can be used for other purposes later. Finally, it is easy to migrate user-level VIA applications into the kernel and vice versa.

3.3. VCONN layer

Modifying GNBD directly over the KVIPL layer requires a significant change in GNBD internals, because the

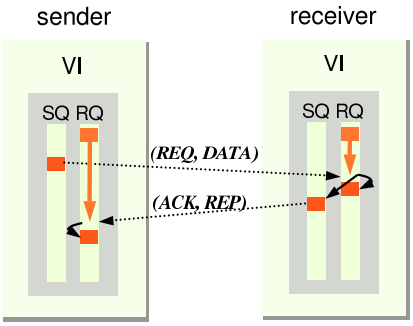


Figure 5. Sender/receiver synchronization

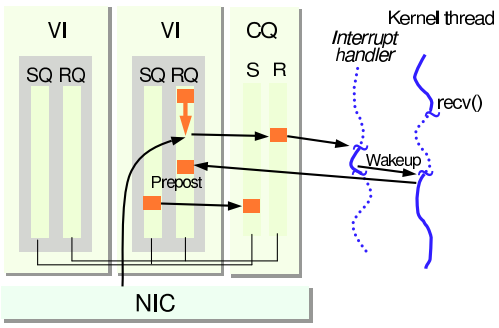


Figure 6. Message handling

interface provided by the KVIPL layer is very primitive compared to the Sockets used by GNBD. Instead, we introduce an intermediate layer called VCONN (VIA CONNection), which emulates Sockets interface over the KVIPL layer. As a normal kernel-level Sockets-like interface is provided by the VCONN layer, we can minimize the code modification in GNBD.

We discuss several issues in implementing the VCONN layer in this subsection. Our goal is to make the VCONN layer as efficient as possible so that the performance of native VIA can be delivered to the upper kernel-level applications such as GNBD/VIA.

Synchronization between sender and receiver. The VIA requires that the receiver should prepost a descriptor to the receive queue (RQ) before the sender requests a data transfer. To satisfy this *preposting constraint*, there should be a synchronization mechanism between sender and receiver, with which the sender guarantees that at least one descriptor is available on the RQ of the destination VI.

VCONN uses a two-way handshaking illustrated in figure 5, where (REQ, DATA) messages are immediately sent to the receiver. Initially, the receiver preposts more than one descriptor in advance and waits an incoming message. The sender is allowed to transmit a message at a time and it should receive an ACK from the receiver to send another message. When the message the sender transmits arrives at the receiver, the receiver extracts the descriptor, and reposts a descriptor for the next message. The receiver sends a (ACK, REP) message to the sender after posting a descriptor. The ACK message informs the sender that the receiver is ready to receive the next message.

In this scheme, the (REQ, DATA) message may arrive before the application calls `recv()` on the destination node. Therefore, the receiver is required to buffer the incoming data temporarily.

Message handling. When a message arrives at a node, the corresponding descriptor should be extracted from a queue and an appropriate action needs to be taken. Normally, the arrival of an asynchronous message is not automatically notified to the application in VIA. As cLAN's implementation of VIA does not support asynchronous notification either, we should invent a scheme to handle the asynchronous messages efficiently.

In the kernel-level VCONN layer, it is possible to run a specific code upon the completion of a descriptor by registering an interrupt handler. When a message arrives, the cLAN hardware issues an interrupt and the corresponding interrupt handler is activated. The handler checks the Completion Queue (CQ), dequeues the entry in CQ and Work Queue, and then wakes up a kernel thread which is waiting for a new message. The kernel thread preposts a descriptor for receiving another message, as depicted in figure 6.

We experience that the task of the interrupt handler is so long that sometimes it fails to serve all the interrupts. The Linux kernel solves this problem by splitting the interrupt handler into two halves: the so-called "top-half" and the "bottom-half". Therefore, we implement our interrupt handler in a way that most work is done by the bottom-half following a very simple top-half.

Flow control. When we analyze the GNBD, we find that the communication pattern is very simple and deterministic enough to avoid a complex flow control in the transport layer. In all the cases, a client and server follow the simple communication pattern: the client initiates connection, sends a request and waits for a response; the server waits for a request and sends a response.

Therefore, we design and develop a very simple but decent flow control method tuned to GNBD communications. VCONN supports a flow control mechanism similar to the TCP's sliding window protocol [10] by extending the two-way handshaking shown in figure 5. Our implementation of VCONN also has the notion of window size w, which denotes the maximum number of messages the sender is allowed to transmit without waiting for an acknowledgment. Initially, the receiver preposts w descriptors to RQ. Whenever the sender transmits a (REQ, DATA), it decreases w, which means that one of the preposted descriptors on the receiving end has been consumed. If w reaches zero, there

is no available descriptors on the receiver and the subsequent transmission is put on hold until *w* becomes positive number [4]. The window size *w* is increased by one when an (ACK, REP) is delivered to the sender. Normally, the window size *w* is fixed in both client and server side with a constant value.

4. Evaluation

4.1. Evaluation Methodology

The hardware platform used for performance evaluation is two Linux servers running Linux kernel 2.2.18 patched with UDMA-enabling and GFS-enabling code. Each server consists of Pentium III-1GHz microprocessor with 256KB of L2 cache, 512MB of main memory, UDMA100 hard disks and an on-board Intel EtherExpress 10/100 FastEthernet adapter. Additionally, a cLAN1000 adapter has been installed to the 32-bit 33MHz PCI slot of each server. The cLAN1000 adapters are connected in a back-to-back topology without any intermediate switch.

To evaluate GNBD performance, we have measured the file read and write bandwidth after mounting EXT2 and GFS file systems on GNBD, varying the transport layers: LANEVI and VCONN on cLAN, and TCP/IP on 100Mbps FastEthernet (FE for short). We have used `bonnie++` [7] to measure the file system performance.

4.2. Performance

Basic Performance. First, we report the latency of LANEVI and VCONN in table 1. The message sizes, 22 bytes and 4096 bytes, are the most frequently used sizes in GNBD if a block size is 4096 bytes. The 22 byte-long message is used to send a request or receive a response, and 4096 byte-long message to send or receive a block itself. As can be seen in table 1, the latency of VCONN is far better than that of LANEVI, meaning VCONN is able to work faster and more efficiently than LANEVI. Note that although the latency is very important for single block read/write, it can be hidden by pipelining the requests and responses.

EXT2/GNBD performance. Figure 7 shows the performance of EXT2 file system over GNBD. Looking at the performance results, we can see that GNBD/VCONN results in higher read/write bandwidth than GNBD/LANEVI and GNBD/FE. The read bandwidth of EXT2/GNBD/VCONN (30.6MB/s) reaches nearly the local EXT2 read bandwidth (32.5MB/s) in figure 7(a), while the write bandwidth (25.9MB/s) is far lower than the local bandwidth (38.1MB/s). We observe that the followings are responsible for the degradation in the write bandwidth of EXT2 file system over GNBD:

Message size (bytes)	Native VIA (μs)	VCONN (μs)	LANEVI (μs)
22	9	17	31
512	15	26	39
1024	20	32	48
4096	51	68	100

Table 1. Latencies

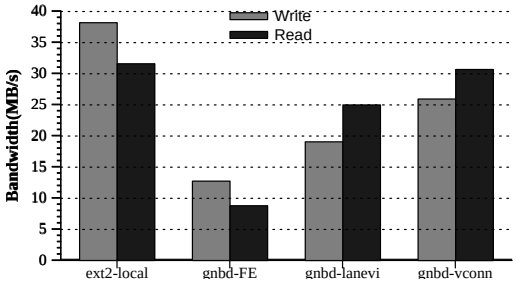
- Synchronous I/O: the GNBD server intentionally opens a file for synchronous I/O and then exports it to the clients. Turning on the synchronous I/O mode reduces the write bandwidth significantly because the buffering system is not utilized. For example, a block device with synchronous I/O mode shows the bandwidth around 30MB/s, while the normal bandwidth is about 41MB/s.

- No block clustering: the kernel combines multiple small I/O operations into a larger single I/O operation in order to decrease the number of read/write requests to disk. However, this block clustering is not enabled for the virtual device created by the GNBD layer on the client side.

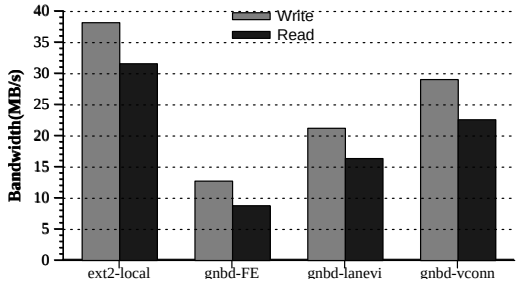
- Dual buffering: a block should be written in the buffer memory of the client and also in the memory of the server before being flushed into a physical disk.

The graphs shown in figure 7(a) are obtained when the server exports a single, large file to the clients. Alternatively, the server can also export a raw partition to clients and figure 7(b) plots the performance of EXT2 file system in this case. First, we see that the write bandwidth is slightly improved when compared to the results of the file-exported case in figure 7(a). The improvement in write bandwidth is caused by the simplicity in writing mechanism of the block device: when a block is written by the client, the server stores nothing but the requested block. When a file is exported, however, the server also needs to touch other meta-data (for example, an i-node block for the exported file) in the server's own file system.

On the contrary, the read bandwidth goes down largely when compared to the case shown in figure 7(a). The degradation of read bandwidth is due to the inefficiency in sequential block reading from the exported partition. The sequential read access can benefit from prefetching where several adjacent blocks are read in advance before they are actually requested. The default read-ahead size used for prefetching from a partition is only 8 sectors ($8 \times 512 = 4096$ bytes). In EXT2 file system, however, the read-ahead size varies between 3 and 31 pages by considering the read

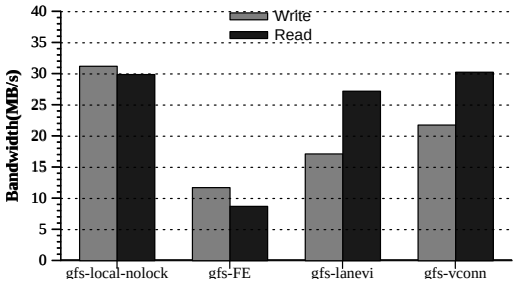


(a) file-exported

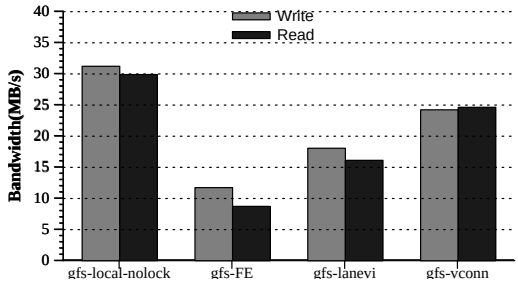


(b) partition-exported

Figure 7. The performance of EXT2 file system over GNBD



(a) file-exported



(b) partition-exported

Figure 8. The performance of GFS file system over GNBD

pattern: the more sequential, the larger read-ahead size, where the size of a page is usually 4096 bytes.

GFS/GNBD performance. Figure 8 shows the performances of GFS over GNBD. Generally, the write performance of GFS/GNBD is slightly lower than that of EXT2/GNBD due to the overhead of GFS itself such as journaling.

5. Conclusion

In this paper, we present a design and implementation of GNBD/VIA, a Network Block Device (NBD) over Virtual Interface Architecture (VIA). First, we have extended the KVIPL layer included in the VIA driver of Emulex cLAN adapters so that it supports the same set of APIs as VIPL in the kernel. And then we have developed an intermediate layer called VCONN which provides a set of kernel-level Sockets-like interfaces over KVIPL. Using the VCONN layer, we can minimize the code modification in GNBD, while maximizing its performance.

Our measurement results show that GNBD/VIA outperforms the normal NBD placed on top of TCP/IP protocol stacks, and realizes the performance comparable to local disk devices, showing the read and write bandwidth of 30.6MB/s and 25.9MB/s on cLAN, respectively.

We plan to extend the VCONN layer to a general kernel-level Sockets layer over VIA in the near future.

References

[1] Compaq Computer Corp., Intel Corp., and Microsoft Corp. Virtual Interface Architecture Specification Draft Revision 1.0. http://www.viarch.org/, Dec. 1997.

[2] K. Duncan. Fibre Channel and Gigabit Ethernet: A Look at Technology for Storage Networking Solutions. Fibre channel group, University of Minnesota, 2001.

[3] Giganet Inc. *cLAN for Linux: Software user's Guide*, 2001.

[4] J.-S. Kim, K. Kim, and S.-I. Jung. SOVIA: A User-level Sockets Layer over Virtual Interface Architecture. In *Proceedings of the 3rd IEEE Int'l Conf. on Cluster Computing*, pages 399–408, Oct. 2001.

[5] National Energy Research Scientific Computing Center. MVICH: MPI for Virtual Interface Architecture. http://www.nersc.gov/research/FTG/mvich/.

[6] National Energy Research Scientific Computing Center. M-VIA: A High Performance Modular VIA for Linux. http://www.nersc.gov/research/FTG/via, 1999.

[7] R. Coker. Bonnie++ Project. http://www.coker.com.au/bonnie++/.

[8] M. Rangarajan and L. Iftode. Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance. Technical report, Department of Computer Science, Rutgers University, 2000.

[9] Sistina Software, Inc. Global File System. http://www.sistina.com.

[10] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.