RC22602 (W0210-130) October 23, 2002
Computer Science

IBM Research Report

Revisiting Link-Layer Storage Networking

Eric Van Hensbergen, Freeman L. Rawson
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Revisiting Link-Layer Storage Networking

Eric Van Hensbergen   Freeman Rawson

evanhensbergen@us.ibm.com   frawson@us.ibm.com

Austin Research Laboratory
International Business Machines
Austin, TX 78758

October 23, 2002

Abstract

This paper revisits the historical concept of link-layer, block-oriented storage networking in light of commodity-based, high-bandwidth local area networks. It examines how advances in data communication technology such as virtual LANs and switched gigabit Ethernet make it relatively simple and inexpensive to re-centralize the storage resources used by clusters of workstations and servers. In particular, the work is motivated by an interest in how to support scalable and efficient access to read-only and private read-write data such as root file systems, swap partitions, log files and static web pages. These techniques complement, but do not replace, higher level distributed file systems whose primary goal is to provide coherent access to shared read/write data.

This paper describes the design and implementation of a very simple, link-layer protocol, the Ethernet Block Device (EBD), for accessing remote block devices. It compares the EBD prototype to a locally attached disk and to similar, network-based techniques that use TCP/IP such as the Linux Network Block Device (NBD), as well as higher level distributed file systems such as NFS. Functionally, the implementation is compared with a local disk to determine what restrictions and visible differences exist. The performance evaluation is based on a series of standard disk and file access benchmarks run on commodity servers and standard networking equipment. The performance results show that for large sequential reads and random reads and writes EBD generally outperforms comparative network storage technologies by 15% to 30%, and performance is best when the data set fits into the server's memory. On a benchmark that is very metadata-intensive and does small sequential operations, EBD and NBD perform similarly. On certain tests, EBD shows a maximum request latency is that is about four (4) times that of NFS, but an average latency that is approximately the same. This suggests that work toward eliminating the occasional very slow response to EBD requests should yield a significant improvement in overall performance.

1 Introduction

The desire for increased density and maintainability along with recent advances in networking technology and bandwidth have led to a growing academic and commercial interest in storage networking. Consolidating storage resources for a data center, compute node cluster, or set of workstations eases the administrative tasks of software installation, data backup, and hardware replacement. It produces economies of scale for storage equipment, such as RAID arrays, by sharing it among several systems. Remote storage resources also provide an easy medium for sharing common information and applications across many client machines. The delivery of commercial, blade-based servers adds additional impetus for the introduction of storage networking, often as the only form of disk access for a cluster of blade servers. Removing the local hard disks from blade servers reduces space and power needs, simplifies chassis design, reduces the amount of heat produced and increases overall product density.

There are a number of commercial examples of these technologies, and their use is becoming more widespread as organizations attempt to control and manage their data, provide proper levels of access control, improve performance and eliminate unnecessary redundancy. For example, in data centers, organizations are increasingly separating storage resources from computational ones due to their differing failure rates, the need to avoid data loss by performing back-ups and the economic value of controlled, but shared access to the data resources of an organization. In addition, by centralizing storage, the installation can simplify system administration and reduce the cost of repair-and-replacement actions.

Modern storage networking technologies naturally divide into two categories – distributed file systems, referred to as network-attached storage (NAS) and remote block device access mechanisms, called storage-area networks (SANs). The difference between the two is in the level of the interface that they offer to remote data. NAS offers access to files using a standard distributed file system protocol, generally the Network File System (NFS) [4] or the Common Internet File System (CIFS) [18] while SAN provides block-level access to remote disk resources. Today, these storage networking technologies generally are implemented using standard, layered protocols based on the TCP/IP protocol stack. In particular, with the emergence of SCSI over IP (iSCSI [34]), IP over Fibre Channel [24] and Fibre Channel over IP [30], the SAN environment is increasingly using TCP/IP, and the distributed file system protocols used by NAS are almost universally based on it. This use of TCP/IP offers the strong combination of being a good, conservative extension to previously existing offerings and making use of the transport, control and routing features that have made TCP/IP essentially the only data networking protocol suite in active use today. However, historically, other approaches to storage access over the network were tried, and it may be worth reconsidering some of them in light of the current generation of local-area network technology.

1.1 A Short History of Storage Networking

The origins of storage networking can be traced back to two sources in the 1980s – the support needs of the disk-less workstations and the development of PC file servers. Sun's Network Disk [36] provided access to remote storage media using only the Ethernet data link layer. It conveyed information about the disk's physical blocks, leaving higher level abstractions to file systems running locally on the client. As late as 1990, disk-less machines were in heavy use at places such as the University of California at Berkeley, where a study by Gusella [10] revealed a large percentage of the Ethernet traffic to be the Network Disk protocol. It is interesting in the context of this report to note that the original NFS paper [33], while making the point that the desire for coherent, shared read-write access drove the Sun development team to abandon Network Disk in favor of NFS, makes the point that Network Disk offered better performance. Similarly, the major use of PC servers in the 1980s was as file and print servers using a pre-cursor of CIFS [18] whose Server Message Block (SMB) protocol originally ran on a very network-specific, LAN-oriented version of the NetBIOS protocol that today is used on top of TCP/IP.

In the research efforts of the 1980s, there was a significant amount of interest in the use of remote disk storage to support not only the long-term storage of user data but also program binaries, libraries and configuration data. For example, the V System [6] created a complete computing system using multiple nodes, some disk-less workstations and some disk-based servers using a novel interconnection protocol, VMTP [5]. More recently a somewhat different system, Plan 9 [28], attached disk-less workstations and disk-less compute servers to a specialized central file server.

The commoditization of storage technology such as hard drives that resulted from the personal computer revolution prompted a movement away from network storage. Local drives offered far superior performance to a remote file service over a shared 10 Mbit Ethernet. NFS and related technologies continued to thrive as a mechanism to access shared data, but what is deemed to be local data including the root file systems, which contain the bulk of the executable binaries, libraries, configuration data and management information, any personal, unshared data and the system's swap space all became almost exclusively the domain of local disk systems.

The decade since disk-less computing fell out of favor was one of dramatic progress in network technologies. Today's data centers use high-speed switched networks, and emerging technology provides channelized, gigabit networking with security and reliability built into the networking hardware, leading in part to the rise of SANs. However, most current SAN implementations use one of a number of high-end connection technologies, the most successful of which is fibre channel. The fibre channel standard offers storage access without many of the unnecessary features of the TCP/IP stack. However, fibre channel is evolving to provide both encapsulation of fibre channel packets within TCP/IP for transport over IP networks and acting as a transport mechanism for routing TCP/IP packets between fibre channel devices.

1.2 Approach and Goal

This report revisits the idea of a link-layer, block-oriented network disk protocol to provide targeted storage services within system area networks, but it does so in the context of commodity hardware, commodity switched ethernet networks and commodity server operating systems, in contrast to the specialized hardware and software used in other environments such as fibre channel SANs. Here *link layer* refers to the interface to the network adapter hardware or the data link layer, which is the way that the OSI protocol stack [8] definition uses the term. An alternative term is *layer-2 protocols*. In TCP/IP layering terminology this is the network interface or bottom-most layer of the protocol stack. The link-layer protocol defined and evaluated here is called the *Ethernet Block Device* and its prototype, the *Ethernet Block Driver*. (This report uses the acronym *EBD* for both.) Despite the name, it is easy to modify both the protocol and the implementation to work with any other suitable data link layers, and, unlike iSCSI, they are independent of the underlying disk technology and interfaces.

The goal of EBD is to increase performance and scalability by reducing complexity and removing any unnecessary overhead associated with both the file and block storage networking protocols based on standard TCP/IP. The solution presented is not intended to provide an all-encompassing network storage solution, but rather to complement both coherent distributed file systems (or NAS) and high-end SANs. In the case of distributed file systems it provides scalable high-performance access to static or private data. The result offers advantages in performance, scalability and support for traditionally difficult-to-implement features such as swapping over the network. Compared to high-end SANs and emerging SAN-over-TCP/IP environments such as iSCSI, it provides a lower-cost and lower overhead alternative that offers all of the functional characteristics of a local disk. The ultimate criteria for success in this effort is whether the resulting software makes the switched ethernet ”just like” a local I/O bus and the remote disk ”just like” a locally attached drive.

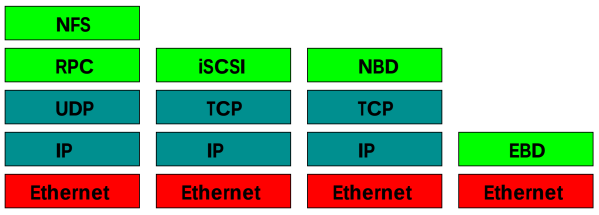


Figure 1: Layers Comparison

In a larger context, the motivation for this effort is related projects on disk-less server blades and centralized system management for clusters. The overall goal of the server blade project is to reduce power and improve density, and one technique used is to remove the local disk on each blade and replace it with storage resources accessed over the network. The management project uses diskless operation as a way of eliminating redundant copies of standard system files and configuration information, thus also eliminating the installation and update distribution problem common in clusters and reducing the amount of management code required on each server system. In addition, the separation of the disk from the compute resource increases the mean time before failure of the compute resources, and enables hot-swapping of storage resources without interruptions in compute service. Centralizing storage also makes it easier to increase storage density (that is, the number of drives per unit volume) and enables easy use of RAID hardware and more tightly coupled backup systems.

Given these motivations and the ability of the diskless operating system environment being used to cleanly separate such information into globally shared read-only data and per-target-system read/write information, it is natural to consider a block-oriented, rather than a file-oriented, solution since the block-oriented approach allows each target system to cache and manage file system meta-data. The need for limited system dependencies and a desire to control the network and system resources required for remote access has motivated the notion of link-layer protocols. Specialized hardware solutions such as fiber channel, Myrinet, InfiniBand, or hardware based iSCSI solutions were undesirable due to the density and power constraints of the target systems.

1.3 Environment

The environment studied here, which typifies many modern data center configurations, has the following networking characteristics:

- All clients and servers are connected on a single switched network so that all hosts are MAC-addressable.
- The network has a very low error-rate.
- The network switch is able to switch layer-2, MAC-addressed packets at link speed eliminating the possibility of on-link or in-switch network congestion.
- The network switch is futher capable of configuring secure point-to-point channels between hosts and filtering unauthorized packets, as is provided in the VLAN [11] specification.

The prototype described here runs on the Linux operating system, commodity servers and a switched gigabit Ethernet although there is no intrinsic reason why the protocol and much of the implementation cannot be carried over to other operating systems and hardware platforms. It has been evaluated to determine how well it provides the services of a local block device driver and what its comparative performance is.

2 Protocol and Message Formats

Conceptually, EBD is a client/server protocol and implementation for remote disk access where the client and server are both attached to the same physical subnet. The protocol itself consists of the message headers that are prepended to any data payload being transmitted and that are subsequently framed by the network device driver as well as the message exchanges between client and server. It also includes the congestion and flow control mechanisms that are required, and its design is driven primarily by a desire for simplicity and low overhead, as well as the need to match the characteristics of disk access with those of the network hardware. Many of these same considerations also affect the implementation described later.

The two protocol descriptions given here are for two closely related versions of EBD. The first is an idealized version and represents the target protocol for the research effort while the second, which differs from the first only in detail, represents the current state of the prototype implementation. These two protocols share common error recovery, flow control and congestion avoidance mechanisms. Since EBD is a layer-2 protocol, it is sometimes helpful to say that the EBD packets are enclosed in ethernet frames, but this report uses the terms *packet* and *frame* interchangibly.

2.1 Disk and Network Hardware and I/O Characteristics

In order to understand the design of the EBD protocol and control mechanisms, one must first consider their environment including the nature of network and disk I/O and the characteristics of the networks that are capable of supporting it.

2.1.1 Solicited and Unsolicited I/O

There is a significant difference between disk and network I/O that has a profound effect on the implementation of the EBD client and kernel-level server. I/O done to the disk is typically the result of some direct request to the disk hardware. Both read (incoming from the perspective of the operating system) write (outgoing) data transfers are specifically requested through the device driver: this type of I/O is called *solicited* in this report. On the other hand, while network transmission is solicited, packets simply arrive from the network and only after arrival may be associated by the system with a request made by some program. This type of I/O is termed *unsolicited*. This difference is fundamental and affects the protocol, the control mechanisms and the buffering strategies used in the implementations. For example, with solicited I/O, the caller provides the required buffers when the request is made, and no pre-allocation or asynchronous replacement is needed. However, with unsolicited I/O, the operating system must have a set of buffers ready for the network hardware to fill, usually allocated from a system-owned pool, and it has to replace them with new buffers when the network device indicates that the current ones are full. As a result, operating systems generally have very different buffering schemes for network and disk.

2.1.2 Burst Management

Another important difference between disk and network I/O is in the management of bursts. On the one hand, the physical characteristics of the disk arm seeking across the surface and the presence of rotational delay leads to disk driver designs and implementations that attempt to create properly sized and sorted bursts of disk activity. This is the goal, for example, of the disk elevator algorithm [35]. In contrast, even though network traffic often exhibits bursty characteristics, the aim of much of the network protocol stack is to smooth bursts out into more uniform request streams to reduce the maximum system and network resource requirements and to provide even response. Ordering remains important, however, especially with the trend toward transmission packet scheduling.

2.1.3 Network Environment

Since the EBD protocol assumes that the packets flow on a single physical network that is capable of transmitting packets with a very low error-rate, there is no need for the checksums used in the internet and transport layers of the TCP/IP protocol suite. Since the link-level CRC flows endpoint-to-endpoint, the packets do not need any additional error checking. In addition, the type of flow control and congestion avoidance required by TCP/IP is unnecessary since the network is able to switch packets at line rate. Using ethernet hardware as an underlying transport offers no guarantees regarding in-order delivery or the framing of multi-packet payloads. However, since EBD is targeted for high-bandwidth, tightly coupled, switched networks, the environment has imperceptible error rates with almost no packets dropped by the network. In addition, modern switches offer facilities that allow them to monitor their traffic and to report packet error-rates which allows management code to monitor the behavior of the environment and provides an additional way of tracking and reporting the incidence of errors should it be necessary. Moreover, the client and the server use uniform data representation systems which allows us to avoid the neutral encoding schemes used by TCP/IP and protocols built on it like NFS that must deal with heterogeneous data representation.

As the protocol operates only within a single physical network, all addressing is done using hardware Media Access Control (MAC) addresses, a very uniform and simple addressing scheme. There is no need for routing or time-to-live information. Finally, the design relies on switch-configured VLAN [11] channels to provide security and the ability to bridge the storage network between LAN segments. By using VLANs, traffic security is done underneath rather than within or on top of EBD, avoiding many of the complexities associated with encryption, authentication and digesting for security. The result of this reliance on the underlying hardware is a very simple protocol with low per-packet overhead.

With minimal changes, the EBD protocol should be suitable for other SAN environments such as VIA or InfiniBand. However, these environments offer other mechanisms for doing remote disk access such as SCSI using RDMA that may well be superior to EBD.

2.2 Idealized Protocol

The EBD protocol is based on the simple abstraction of a fixed-size linear array of sectors. All protocol parameters, including the data length and the disk address of the data, are in terms of sectors. Just as with standard disk I/O interfaces, request and response payload data are clustered as blocks of contiguous sectors. To keep the protocol simple and to reduce the processing necessary, there is no re-blocking of disk blocks into network packets as is generally done with network storage protocols. Thus, the maximum transmission unit (MTU) size of the physical network (or data link layer) bounds the block size that can be used on an EBD disk. This avoids the problems of framing, fragmentation, reassembly and out-of-order delivery. In addition, the block size is constrained to be a multiple of the system sector size, typically 512 bytes, used on the remote disk. Finally, the size of the hardware sector places a lower bound on the MTU of the network: it cannot be smaller than the sum of the ethernet frame overhead, the EBD request/response header and the size of a single remote disk sector. In summary, each packet carrying data contains precisely one block.

The protocol uses a very simple request/response mechanism where the response to a request also serves as acknowledgment. With the exception of two forms of control message, requests are sent from the client to the server, and responses are sent from the server to the client. Disk read requests may be clustered so that a single packet can initiate the transfer of up to 255 sectors. Clustered read requests are not fully acknowledged until responses for all requested sectors are received. On the other hand, write requests contain the data inline and cannot be clustered.

Figure 2 shows the format of an EBD frame. Each link-layer packet contains a header identifying the operation and its parameters. The protocol header has eight fields: an 8-bit operation type, an 8-bit identifier, an 8-bit target-device major number, an 8-bit target-device minor number, an 8-bit flow-control field, an 8-bit length, representing the length of the request or response in sectors, a 32-bit sector identifier and, finally, the payload section which is only populated during *HANDSHAKE* and *WRITE* requests and *READ* responses. The link-layer header is assumed to contain source and destination address information. For ethernet the total header size, including the ethernet framing and the EBD request header, is 28 octets, somewhat smaller than the header size for a similar protocol running over TCP (54 octets), UDP (42 octets) or even IP (34 octets).

Typical configurations use 512-byte sectors, a 1024-byte block size and a nominal ethernet MTU of 1500 octets, which allows the protocol to address up to 65536 target devices per server, each up to 2 terabytes in size. Since these

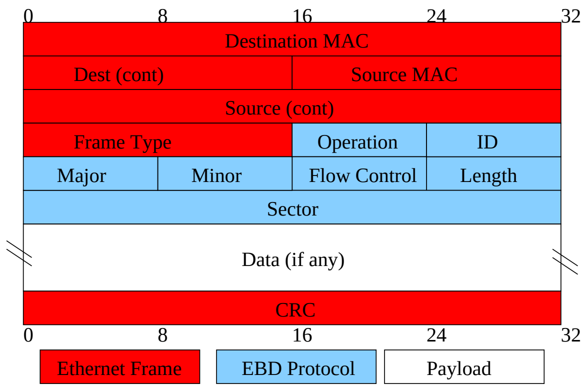


Figure 2: EBD Frame Format

target devices may equate to disk partitions, this gives the initial protocol a substantial amount of addressability while avoiding 64-bit data types and operations. The jumbo-frame gigabit ethernet allows nominal MTUs of up to 9000 octets, providing payloads of up to 8 KB per packet. From a protocol perspective, large payloads improve network link and switch hardware utilization as well as reducing per-payload overhead on both client and server. The similar reasons have led to the use of larger on-disk block sizes by file systems.

While all operations reference a major and minor device number used to identify the target device on the server, the server itself is identified by the ethernet hardware address provided by the ethernet header. Server discovery and naming are outside the scope of the protocol and are provided independently, as is access control and naming of the EBD devices on the client and its relationship to disk partitioning.

Client and server negotiate the maximum request size, block size, partition capacity, and initial flow control values using the *HANDSHAKE* request. This allows the client to query the server for certain information about the target block device that EBD needs to have for use by the client's operating system and request that a certain block size be used. The block size must be consistent with the network device MTU. The *HANDSHAKE* request contains only the type, major number, and minor number with the requested blocksize in the payload. The response maintains these values and fills in the device size (in sectors) in the sector field along with the maximum sectors per request in the length field and the initial request credit value in the flow-control field. The may adjust the requested blocksize if necessary and places the negotiated value in the payload of the response.

READ requests are acknowledged by *READ* responses carrying the payload requested. As mentioned previously, a single *READ* request can ask for several blocks triggering multiple *READ* responses. *WRITE* requests are acknowledged by *WRITE* responses with the same header information but an empty payload. No clustering of acknowledgments is done, so a *WRITE* response will be generated for every request received.

A special operation, *SYNC-WRITE* is available if a synchronous write is desired. Syncronous writes are not acknowledged until the block is physically committed to the underlying disk or medium. This is particularly useful for journaling file systems and other storage requiring high-reliability. Finally a *FLUSH* operation allows the client to request all outstanding blocks on the server be written to disk and all cached blocks be invalidated. The client may not send any further requests to the server until he receives a *FLUSH* response.

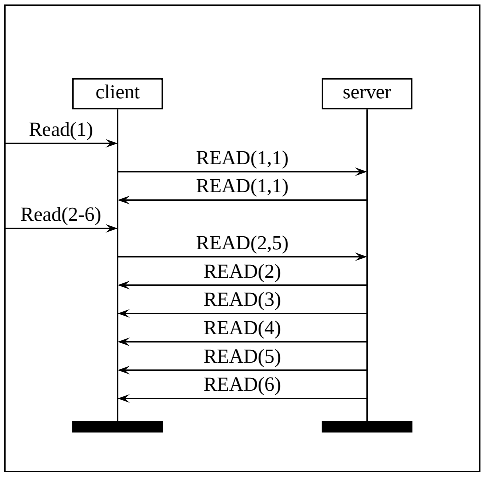


Figure 3: Read Requests

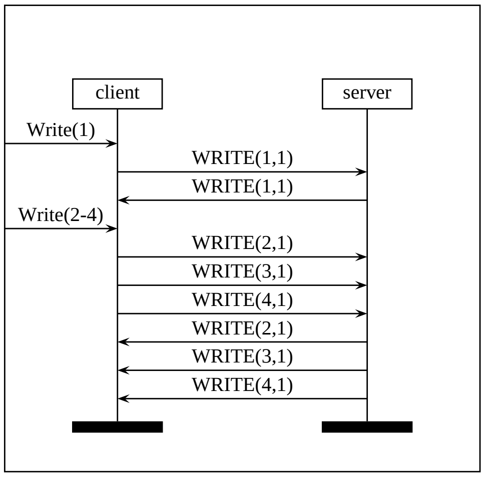


Figure 4: Write Requests

2.3 Implemented Protocol

The protocol described in the previous section is idealized, and there are some differences in detail between it and the protocol as currently implemented. The near-term goal is to converge the implemented protocol to the idealized one described above. Figure 5 shows the request and response frame format used by the current implementation during ordinary data transfer operations while Figure 6 depicts that used for the *HANDSHAKE* request and response.

The major differences in the frame format from the idealized protocol are in the mutual identification of the client and server, the size of the flow control value and the use of handles or tokens that are meaningful only to the client or the server and are passed back unchanged by the other side. As a result the EBD request and response header is 18 octets rather than 10 octets for a total frame overhead of 34 octets. The first of the two one-octet identification fields is used to designate the client-side minor making the request. The second's meaning depends on whether or not the request is a *HANDSHAKE*. If it is a *HANDSHAKE* request, then on request the field contains the minor number of the target to be associated with the particular client minor, and on response, it contains an index. All subsequent requests by that client minor pass the index back to the server, which, on response, returns it unchanged. This index is used by

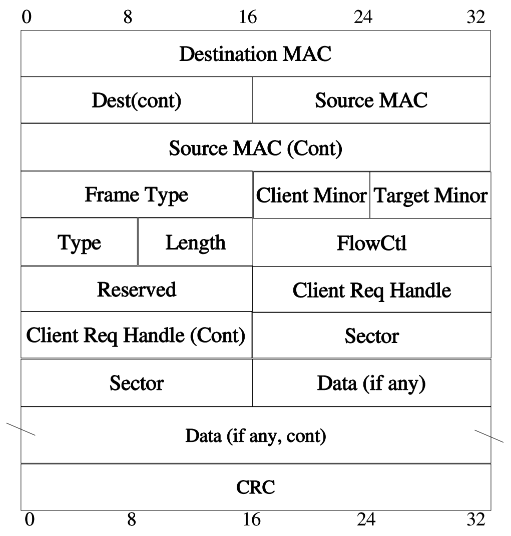


Figure 5: Implemented EBD Frame Format

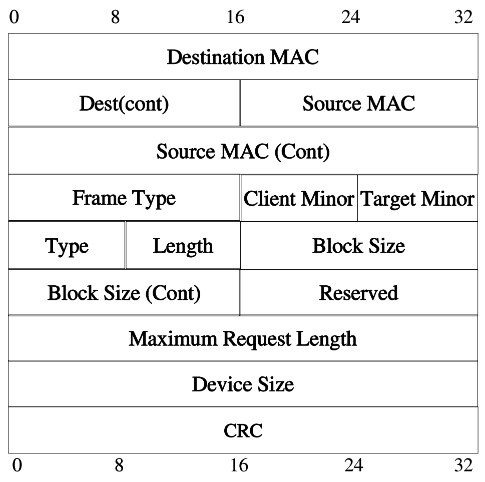


Figure 6: Implemented EBD Handshake Frame Format

the server to identify which client is making the request. Of course, this protocol has the obvious limitation that all of the minors exported by a server must have the same major device number. In practice, this is a serious limitation of the implemented protocol since it prevents a server from exporting partitions on different types of disks such as SCSI and IDE devices, since they use different major numbers. In addition, the response on a *HANDSHAKE* carries the information being returned by the server in a slightly different fashion than specified by the idealized protocol. There is no data payload since the slightly larger implemented header carries all of the information returned by the server. The flow control field is two octets rather than one: this expansion is for experimental purposes. The client request handle is used to identify the disk request that caused the EBD request; it is a pass-through value that is set by the client on the request, unused by the server and then returned unchanged in the corresponding response.

Clearly, there is some room for optimization and convergence toward the idealized protocol's frame format. The reserved field can carry version information and the major number of the remote device while the two-octet flow control field is, in practice, excessively large since reasonable values fit into a single octet. All of the remaining differences help the client or the server locate data structures and, thus, reduce execution time and the serialization between minor devices. However, restructuring the information that is passed by the client inserting a level of indirection allows one to shrink the number of octets used for the client minor and client request id from five (5) to two (2).

In addition to the differences in frame format, the implemented protocol uses some message types and flows that are not present in the idealized protocol. The first and most common of these is *WACK* or weak acknowledgment. Although the protocol necessarily declusters clustered *WRITE* requests, it marks the very first data block request that it sends with a bit indicating that a *WACK* is required. All *READ* requests are also marked to indicate that a *WACK* is required. Upon receiving a request indicating that a *WACK* is needed, the server responds as quickly as possible with the *WACK* response. Semantically, the *WACK* indicates that the server has gotten a request and is the primary carrier of flow control credits as described below. However, to avoid having two acknowledgements per data block on *WRITE* requests, only the first request of the original cluster is marked as requiring a *WACK*. Figure 7 shows the packet exchange for reads with *WACK* while Figure 8 represents the exchange for writes with *WACK*.

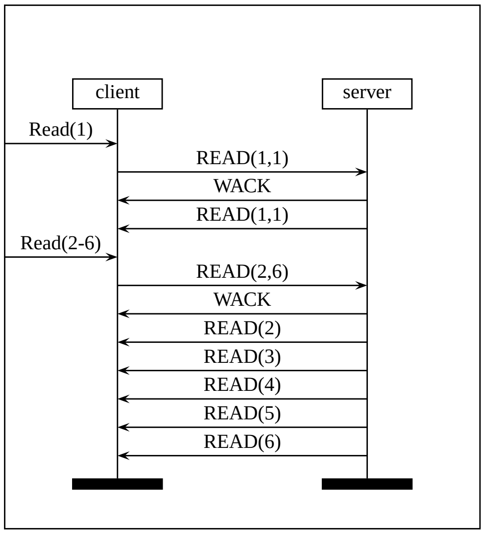


Figure 7: Read Requests with WACK

Finally, the implemented protocol adds two additional request types that are unacknowledged. The first of these is sent by the client when the client driver is terminating and is used by the server to reclaim resources. This request is informational and may not be sent in all cases such as, for example, that of a client end-point crash. The second request type is the converse and is almost the only example in EBD of an unsolicited packet sent by the server. It indicates that the server is shutting down and is used to notify the client. The expected client behavior is immediately to return I/O errors to all subsequent requests for any minors supported by that server.

2.4 Error Recovery

There are three possible sources of error in the EBD environment. First, the client may request an operation that the server's disk cannot do such as ones that encounter a media error on the server or which make requests outside the bounds of the target disk. These errors are handling using the protocol's negative acknowledgment (*NAK*) response.

If a request cannot be serviced by the server due to a logical or media error, the server responds to the client with

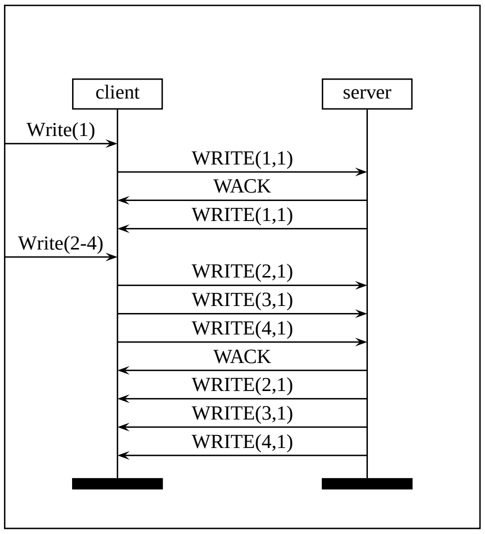


Figure 8: Write Requests with WACK

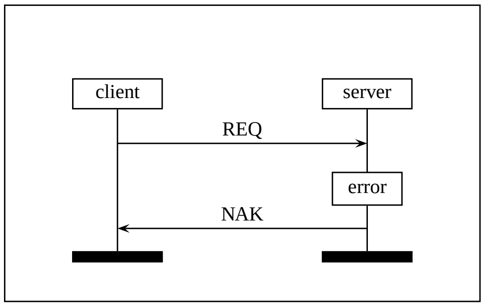


Figure 9: NAK Response

a *NAK* for the request. The EBD driver on the client then forwards the error up to the file system layer where it is handled as if the error occurred on a local storage medium.

The second source of error in the EBD protocol is due to client or server overload or failure. It is detected by the time out of an unacknowledged request. Since overload conditions are often transient, the error recovery strategy is a simple timeout mechanism. The idealized protocol does a single timeout on each request that it sends while the implemented protocol does separate timeouts for the *WACK* and the actual response with the *WACK* timeout interval somewhat shorter than the response timeoout. When the timout occurs, unless a maximum waiting time has passed, the client retransmits the request to the server. Partial receptions will only request blocks which have not yet been received. Reception of duplicate responses, resulting from unnecessary retransmission of requests, is silently ignored. This design insulates the server from the retransmission policy and any need to check for duplicate requests. The retransmission policy is governed entirely by the client side. Setting the timeout values is considered to be a part of the flow control and congestion avoidance mechanisms, but the values chosen are constrained to ensure that there are minimum and maximum values and to guarantee multiple retries. Any request that remains unacknowledged after a maximum waiting period is reported to the file system layer as an I/O error.

Since the EBD protocol is intended for reliable networks with very low error rates, true network errors, as opposed to end-point problems, are exceptionally rare. However, any packets dropped by the network are recovered using the

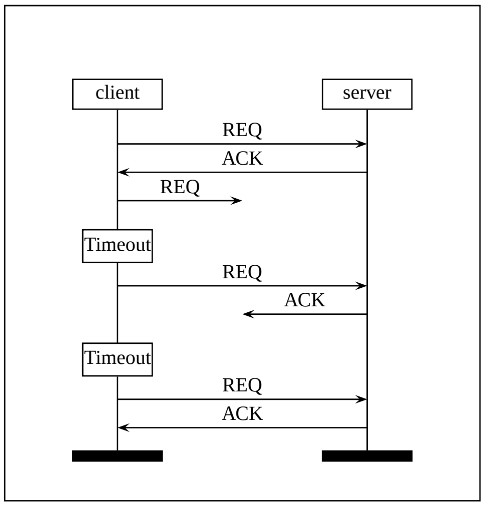


Figure 10: Retry After Timeout

timeout-and-retransmission mechanism. Any corrupted packets are dropped at the link level by using CRC checking.

2.5 Flow and Congestion Control Mechanisms

While switched gigabit ethernet networks (and other high-performance interconnects) do not suffer from congestion on the network or at the switch, the limitations of the end-point systems force the EBD protocol to include three separate, but inter-related mechanisms to deal with flow and congestion hazards. Following the very helpful distinction made by [27], EBD distinguishes between flow and congestion control where flow control manages the flow of requests and responses so as to ensure that both client and server can keep pace and congestion control deals with congestion in the network and the apparent response time of the combination of the network and the server.

In the case of EBD, two resources are critical and can either be overrun or the source of congestion – memory, especially server memory, and the end-points' network transmitter (TX) hardware, generally that on the client end. Having solutions to these problems is especially important in light of the mis-match between disk and network hardware discussed in 2.1. Since the disk is best used in sorted bursts, the block I/O software in the operating system creates bursts to take advantage of that phenomenon. Although it would be nice to have the server catch and use these bursts, they can overwhelm the client's transmitter and the server's memory resources and, hence, they have to be smoothed out by the flow and congestion control mechanisms. By the same token, since there is some advantage to mimicking the naturally bursty disk behavior, complex flow and congestion control mechanisms such as the one in TCP are not aggressive enough in transmitting bursts and do more traffic smoothing than the EBD environment requires. Although some ethernet devices and drivers offer ways of tuning the number of transmit and receive buffers, potentially alleviating these problems, such tuning is beneath the level of the EBD protocol and very device-specific. Other than the use of both standard and jumbo ethernet frames, no experimentation has been done to determine what effect such tuning would have.

2.5.1 Flow Control

The primary job of flow control on both the client and the server is to ensure that the end-point has enough memory to buffer bursts and is able to keep up with the other side. As discussed later in 3, memory shortages are surprisingly common, even on machines with large main memories.

Simple flow control can be performed on the client by limiting the number of outstanding requested blocks. Clients can prevent themselves from being overrun by never requesting more than they are willing to handle. Servers use the *flow-control* field in the header of the EBD frame to give the client a sending credit: the credit represents the maximum number of sectors that the server is willing to let the client request at any instance in time. This number varies according to the load on the server, and every response, including *WACK* responses in the implemented protocol, contains a new credit value. When the number falls below a threshold, generally set to be the product of the number of sectors in a maximum-sized clustered request and a small integer, the client stops sending new requests until the value exceeds the threshold again. Note that this scheme implies that the maximum client clustered request size must be smaller than the maximum server flow control credit. Although neither protocol requires it, the server may enforce its flow control credits by dropping packets from clients that exceed their credits.

The intent of the flow control mechanism is that the server be self-calibrating and adaptive. It should be able to calculate the initial maximum flow control credit for each client based on its capacity and the number of clients that it has and then track the number of in-flight sectors and adjust the credit accordingly.

While in most cases the server will be able to adjust client behavior during the course of normal transactions, in extreme cases of congestion, the server may change the flow control credit by sending a *CONGESTION* packet. In truly extreme cases of congestion, the server may broadcast a *CONGESTION* packet to all client end-points for all client minors.

2.5.2 TX Congestion Control

The second hazard in the EBD protocol is the network interface transmitter or TX. There are limits on the number of frames that the TX can send without presenting an interrupt and having the operating system provide it with a new set of frames to transmit. Without introducing some pauses in transmission, it is quite easy, especially on the client side and even with a relatively slow processor, to flood the TX, causing the network device driver to reject new frames queued for transmission. The problem occurs primarily because writes have to be declustered and transmit their data inline in the request packets.

To deal with this problem, the EBD protocol requires that both client and server reliably queue every frame to the network interface every frame that they send although the protocol does not specify how this is to be done. This does not require that the frame actually be sent, just that the interface accept responsibility for it. In addition, there has been some consideration of mechanisms for controlling the rate at which frames are queued, the current protocols, idealized and implemented, do not attempt to do this. In Section 3 there is some discussion of the use of low-level kernel primitives and waiting mechanisms for dealing with this problem.

2.5.3 Setting Timeout Values

Finally, the EBD protocol on the client side must set timeout values for data responses and, in the case of the implemented protocol, for *WACK*s as well. In the case of the implemented protocol, the *WACK* timeout value is expected to be much less than the data response timeout. In every case, the goal of setting the timeout value(s) is to ensure that the following two conditional probability equations are correct. $P(lossin < timeout|loss) \approx 1$ and $P(timeoutexpires|noloss) \approx 0$ These properties ensure that the timeout is both effective in detecting and retrying lost frames and does not consume resources retransmitting frames that are not lost. Even though the protocol requires that the responses to redundant retransmissions be ignored, they consume both client and server resources.

The EBD protocol uses adaptation within limiting values to set timeout values. The upper limit is determined based on a reasonable understanding of when disk device drivers would report I/O errors to programs using the disk while the lower limit is set to avoid excessive execution time in the retransmission mechanism. Within the these bounds the protocol requires that the client keep a rolling average of the responsiveness of the server to its requests and use that to modify the timeout value or values. In the idealized protocol the client uses the round-trip time of the *HANDSHAKE* request plus a server-provided value for its response latency on data requests to initialize the values that mechanism has not been formalized in the implemented protocol.

There are no timeouts defined for the server side of the protocol.

2.6 Error Handling and Recovery

As with the rest of the EBD protocol, for error handling and recovery, the design mimics the bus and disk architectures used for locally attached storage.

Although the server tracks the clients that are using it and the client and the server exchange tokens, the server index and the client minor, there is no formal connection in the sense that term is used in TCP/IP. All of the state that exists is maintained within the EBD driver and server, and there is no additional information about what systems are communicating and using storage stored anywhere else. This form of containment dramatically simplifies the error and recovery portion of our protocol.

The protocol deals with errors through a very simple, two-level re-transmission scheme driven by the client: this is very much like retrying disk I/O operations. The first level is used to track the receipt of WACKs from the server and uses a relatively short timeout. If no WACK has been received for a request when the timeout occurs, the client retransmits the request to the server. On a longer timeout cycle, the client retransmits those requests that are not yet finished but which have been weakly acknowledged. For these requests, there is reason to believe that the server has them but for whatever reason did not complete them. There are two potential problems with this scheme. First, for timing reasons, it is easy for the client to receive duplicate responses for a request. All responses to requests that are not outstanding are quietly discarded. Second, if the server is congested, re-transmission tends to make it more congested. However, one does not want to stop the re-transmission if flow control is on because it may be the case that responses that would turn flow control off are lost. The solution to this dilemma is to increase proportionally the timeout intervals as the server drops the value of the flow control credit and to decrease it in the same proportion as it increases. Initial timeout values are subject to negotiation to match the characteristics of the network, client and server.

The number of retry attempts is limited at both levels. When all of the retries are exhausted, the client reports an I/O error on the request to the kernel. One important simplification is that the protocol assumes that the network interface code alway succeeds even though in practices problems such as memory buffer shortages can cause it to experience failures. The fact that the network interface code does not successfully send or receive a packet is not significant. If the problem disappears, a retransmission succeeds. If it fails, in the end, there is what appears to be a disk I/O error to the kernel, much the way that there would be with a locally attached device.

Another set of hazards that can occur are those associated with client or server shutdown or failure and restart. We have include a handshake request from the client to inform the server that it is terminating its use of a particular minor device as well as one that is sent by the server to all of the clients indicating that it is shutting down. The more interesting cases are those in which one end or the other disappears and then re-appears in an uninitialized state. In the case of the client, when it restarts, it sends the initial handshake request. The server must recognize this, drop all requests for the previous version of the connection and accept the new handshake as the initial contact from a new client. In the case of the server, it responds with an identifiable NAK to all requests that it receives from clients that transmit requests without first doing the handshake. The client must then send the proper handshake requests and retransmit any requests already sent but not retired before resuming normal operation.

3 Implementation

The EBD protocol described above is designed for client implementation as a low-level driver within an operating system kernel. As a result, the implementation is affected by design of both the current generation of network hardware as described above and standard operating systems. On the other hand, the server side of the protocol can be implemented either as a user-level daemon program or as a kernel module. This report describes in detail the kernel-level implementation that is evaluated later. Although the implementation described here is specific to the Linux 2.4.18 kernel, many of the constraints and design choices are similar in other operating system environments. While the client supports multiple device minors associated with multiple devices on potentially different server systems, the server can export multiple minors of the same disk driver, and each minor may have multiple clients accessing it. The combination of multiple clients with multiple servers was extensively tested with earlier versions of the implementation running on earlier kernels but has not yet been re-validated with the current implementation.

3.1 Client Implementation

The client implementation is written as a kernel module for the Linux 2.4.18 kernel, using the standard Linux block driver kernel-level interfaces.

3.1.1 Block Driver Structure

The client-side driver conforms to all of the required interfaces including the block `ioctl` needed to make it appear as though it is driving a local disk. It also sets up the global data arrays used by the block driver code to hold information about block devices such as the size of the device, its block size, the maximum number of sectors that the kernel may request at once and the read ahead value.

Although there are two types of block drivers in Linux at the 2.4.18, those that use a request queue and those that directly handle I/O one buffer head at a time [32] and bypass the request queue processing, the client uses the request-queue style to take advantage of the request clustering done by the kernel block interfaces. In the case of read operations, this optimizes the operation of the EBD client since it creates clustered read requests. On the other, for writes, it creates a clustered request that the client must decluster into individual block-at-a-time writes since the data on write operations is transmitted in line in the EBD write request frames.

Since the driver is designed to support multiple device minors, potentially accessing different devices on different servers and even over different network interfaces, it uses the multiple request queues, one per device minor, and implements a function that the kernel calls when it queues a request that locates the request queue for the minor. The kernel then puts the block request structure on that queue. The driver also has a separate disabled spin lock for each minor to serialize access to its private data structures between request processing, the network packet handling that it does and the kernel thread used to do retransmissions.

Since the client driver conforms to the current 2.4 Linux block driver interfaces, the kernel treats it as *pluggable* driver, which means that the kernel puts requests on the queues as they arrive but does not directly invoke the driver's request processing code. Instead, it uses the task queue mechanism to run the disk task queue, which, when it runs, invokes the request functions for all of the drivers that have queue requests. Thus, the request function is called under the control of the kernel thread used to run task queues. In cases where there are multiple minors, the kernel may call the request function multiple times on each disk task queue execution, once for each minor device that has at least one request on its queue.

In Linux, the kernel issues requests to block devices using a request structure containing information on the desired operation, a start sector, the request size and a chain of buffers, which are each a block in length and represent a contiguous piece of data. All EBD requests are either for a single buffer or an integral number of buffers.

When the request function gets control, it immediately pulls all of the requests on the request queue off of it, timestamps the request structures in an unused field and puts them on an internal list, `pending`, which holds the kernel request structures that the driver has received but not yet transmitted to the server. This processing is done holding the kernel `io_request_lock`, which is dropped as soon as the request function has moved all of the requests. The code then acquires the device lock for the minor. Then the request function processes all or as many of the elements of the pending list as it has flow control credit to cover, creating EBD request structures and transmitting them as described below. As described in Section 2, *WRITE* operations are declustered at this point, and the code creates a separate EBD request for each block being written. On the other hand, *READ* requests generate a single EBD request. The request structure is then put on another list, `outstanding`, that is used to hold requests that have been transmitted but not yet complete. Each request transmitted has both the client minor and the address of the kernel's request structure in it, making it easy to locate the correct request when the response arrives. Figure 11 illustrates the flow of requests from reception by the driver to the `outstanding` queue.

Unfortunately, for reasons discussed previously, the transmission of EBD requests to the server is not hazard-free, and since the driver is very aggressive, in practice, it easily floods the network transmitter. When this happens, the request function saves pointers to the current kernel request and the next block to request, schedules a tasklet to restart the operation and returns. The reason for this mechanism is in Section 3.1.2. If, however, the request function runs out of flow control credits, it simply returns without scheduling the tasklet.

The driver uses the kernel request to track the status of the request. As the individual blocks are read or written, the network portion of the client invokes request completion functions. The Linux block driver architecture provides for processing completion in two parts: the first marks a particular buffer head complete, representing the successful transmission or receipt of a single block, while the second indicates the completion of an entire request. Rather than using the generic kernel implementations of these functions, the driver uses private functions, primarily for locking reasons. All of the processing of the `outstanding` list, including block completion, occurs with the device spin lock for the minor held. However, the kernel structure requires that request completion occur with the `io_request_lock` held. To avoid deadlocks, the driver never attempts to hold both locks at once.

The current implementation has been validated for both 1024-byte using standard ethernet frames and an MTU of 1500 and 4096-byte blocks using jumbo frames and an MTU of 9000. As discussed in Section 2, EBD does not block, de-block, fragment or reassemble blocks. While our buffers do not make full use of the available MTU sizes for jumbo frames, the complexity and overhead of segmenting and reassembling buffer fragments are excessive. Using the buffer structure of the file system layer and buffer cache make the implementation on the client as well as the server simpler and more efficient. Although 8192-byte blocks make more efficient use of the available jumbo frame space, ext2 does not support file systems blocked at 8192 bytes per block, limiting their usefulness.

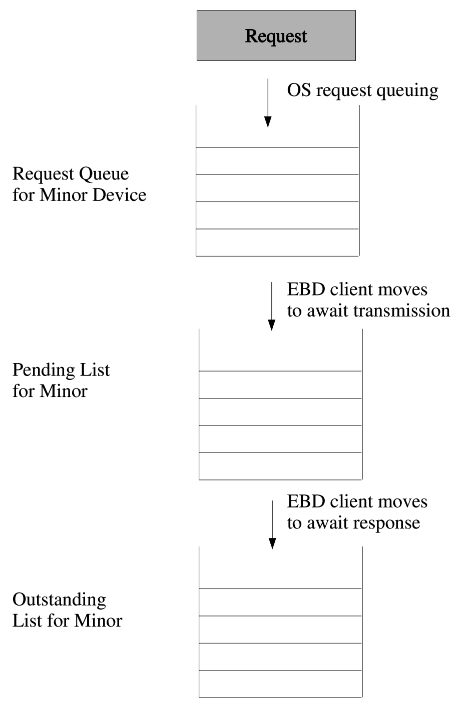

Request
OS request queuing
Request Queue for Minor Device
EBD client moves to await transmission
Pending List for Minor
EBD client moves to await response
Outstanding List for Minor

Figure 11: Implemented EBD Frame Format

3.1.2 Network Operations and Packet Processing

In addition to the block device driver logic, the EBD client has code to send and receive link-layer frames. The transmission code uses the Linux `sk_buff` fragmentation mechanism to do zero-copy transmissions of outgoing data for frames such as those for *WRITE*s that contain it. The transmission mechanism used is to queue the outgoing frame to the network driver, which has two important implications. First, the frame is subject to any packet scheduling that the network code is doing: all EBD testing to date has been using the default FIFO packet scheduler. Second, as explained in Section 2.1, the hardware can transmit only a limited number of packets without presenting an interrupt and being given a new set of frames to send. In Linux, processing the transmitter interrupt requires not only that interrupts are enabled but also that the tasklet scheduler get control and run the `NET_TX_ACTION` tasklet. Each network device has a queue of fixed, finite but settable size with a default of 100 entries, In practice, it is extremely easy for the block driver logic to fill the queue, even when it has been expanded by a factor of 5 or 10, creating the hazard mentioned above. When that occurs, the EBD driver must back off and allow `NET_TX_ACTION` to run. To keep overhead to a minimum, the code schedules a tasklet that invokes the same EBD request generation code used by the request function. The logic of the tasklet scheduler guarantees that `NET_TX_ACTION` runs before the EBD tasklet, ensuring that EBD can make progress. In addition, by limiting the number of sectors that the kernel can request with a single request structure also helps reduce the pressure on the network transmitter. Although the protocol allows up to 255 sectors per request, in practice values of around 8 work much better.

To receive incoming response, the client registers a packet handler with the network stack for the EBD link-layer frame type. It assumes that packets received from the link-layer are valid. For ethernet this means the network driver has already checked the CRC. Using the client minor and the address of the kernel request structure, the packet handler determines the minor device and locates the outstanding request. For standard *READ* and *WRITE* requests, there are two types of responses, *WACK*s and data responses. When it receives a packet, the packet handler looks at the header for the response information and checks to determine if the response is a *WACK*. If so, then it invokes the *WACK*-handling code described in Section 3.1.3. Otherwise, it uses the client minor and the request structure address in the response header to locate the correct kernel request structure. Then it

- locates the buffer head for the block
- copies incoming data if the operation is a *READ*
- updates the flow control credit and driver statistics
- calls the completion function for it
- updates the request structure.

If there are no more blocks for the request, the packet handler removes the request from the `pending` list and invokes the request completion code. Unlike *WRITE*s where the transmission code does a zero-copy transfer of the outbound data, *READ*s require a copy operation, for two reasons. First, the Linux kernel has two different buffer management schemes, one for networks and one for block devices, and moving data areas between them is complex and exposes the code to locking problems. Second, although, in fact, the incoming data responses to *READ*s are solicited, that fact is not apparent to the lowest of the networking code. It must have the buffers required by unsolicited network packet arrival and only after the data is in the buffers can it recognize that this data is in response to EBD *READ*s.

What about applying gather/scatter I/O to disk block operations? But then how does that solve this problem? This is one of the reasons for the still-confidential multiple personalities for the network interface.

Duplicate responses are discarded as is any response which does not have an associated request on the `outstanding` queue or whose request does not have a matching buffer still outstanding. Since response ordering need not match request ordering, there is special code to handle out-of-responses although, in practice, due to the point-to-point nature of the network and the server's ordering of disk requests, they are rare.

Prior to giving up control the packet handler invokes the flow control logic described in Section 3.1.6.

3.1.3 *WACK* Handling

The client handles *WACK*s with specialized code that is invoked by the packet handler as soon as it discovers the packet is a *WACK*. *WACK* processing has two major functions. First, it marks the kernel request to indicate that a *WACK* has been received from the other side, giving some, although not complete, assurance, that the server actually has the request and will process it. The second function of the *WACK*-specific code is to update the flow control credit and, if necessary, change the flow control state of the client. If the new credit is less than that required to transmit two full requests, the function turns on flow control, so that the only new transmission allowed is one that completes the earlier transmission of a request that was interrupted due to a transmitter flood. While simplicity motivates the use of an integral multiple of the request size, the choice of two is arbitrary. If the new credit is at least enough to send two full-sized requests and if flow control is current in force, the *WACK* handler turns it off and calls the transmission wrapper to resume sending retransmission and requests on the `pending` list.

3.1.4 Handshake Processing

The EBD client currently transmits *HANDSHAKE* EBD requests only during initialization although one obvious enhancement is to initialize the device minors dynamically, in which case, the client would have to send *HANDSHAKE* packets whenever a new minor initialized. When the response is received, the packet handler invokes a function to check for consistency, particularly, block size consistency and its compatibility with the network MTU, and then to copy the information from the server to the appropriate data structures.

3.1.5 Timeout and Retranmission

To maintain reliability, a separate kernel thread periodically scan the `outstanding` list. As each device minor has its own retransmission thread, the thread operates safely by obeying client's locking structure. In order to simplify the handling of the transmitter flooding problem, it also uses the common EBD request transmission wrapper. Any retransmission that actually occurs moves the request from the `outstanding` list to a `retransmission` list for the minor.

Periodically, the retransmission thread for a minor checks each kernel request on the `outstanding` list for three conditions.

1. Has the request been outstanding so long that it is unlikely that the server will ever respond?
2. Has a *WACK* been received for the request?
3. Has any buffer been outstanding long enough that the client should try transmitting the EBD request again?

Using the timestamp written into the kernel request structure upon arrival, the retransmission thread is able to determine if the request has been outstanding long enough that "a reasonable disk device driver" would report an I/O error. If so, it uses the completion processing routines to finish the request with all remaining outstanding buffers marked as having an I/O error. Similarly, the second check is done only on the kernel request structure to determine if the *WACK* time out has passed with no *WACK* received. If so, the thread retransmits the request. Since the client updates the kernel request structure and its buffer chain as individual buffers are finished, the third of the checks applies only to the buffers that remain when the retransmission thread runs. The thread compares the current time to the flush time for the first remaining buffer, and if it is now later than that, it retransmitted the request, or rather, what part of it is still outstanding.

The EBD client must set four inter-related values to make this mechanism meet the criteria for timeout values established in Section 2 – the scheduling interval of the thread itself, the *WACK* timeout value, the standard data response timeout value and the time at which to report an I/O error. In practice, these values are quite difficult to set by hand or even to calibrate for adaptation. The current implementation use a rather small value for the *WACK* timeout and a rather larger one for the thread scheduling interval. It sets the buffer flush time as a multiple of the thread scheduling interval with a goal of trying to have a small number of retransmissions and the error time to give up after that number of retransmissions. Even worse, the proper settings for the timeout values interact with the settings of the maximum request size and the maximum flow control credit from the server. It is, however, easy to tell if the timeout settings are wrong. If the timeout values are too short, the client discards sequeunces of responses whose sector numbers increase monontonically by the number of sectors per block. If they are too long and there is some other problem such as server failure, the driver either hangs or waits for a long time before producing a run of I/O error responses.

An implementation more in keeping with the intent of the protocol has to calibrate and adapt the timeout values. The best way to do this is to track the current latency of *WACK*s and data responses and set the timeout values for each based on a weighted rolling average of the previous value and the most recent latency value bounded within a range. Thus, for both *WACK* and standard timeouts,

$$TO_{new} = \frac{TO_{prev} + LATENCY_{latest}}{2}$$

where

$$UPPER \geq TO_{new} \geq LOWER$$

and n, UPPER and LOWER are constants. A more sophisticated decay function is clearly possible, but there is no reason to believe that one is required. The bounds are somewhat arbitrary, but they should be set such that the retransmission thread does somewhere between two and four retransmissions before the error timeout. Once the other thread values are known, then the retransmission thread can adjust its scheduling interval to ensure that it runs often enough to send the right number of retries before the error timeout. One can argue that the final error timeout value should be a constant based on the expectations of application and kernel code and is somewhat independent of the behavior of EBD.

3.1.6 Flow Control

Most of the flow control mechanism has already been described above. The client has no direct influence on the flow control credit issued by the server, but the size of the credits issued by the server do affect the appropriate maximum request size. Generally, speaking the maximum request size needs to be smaller than the maximum (and initial) flow control credit divided by a small integer k where $2 \leq k \leq 16$. Although the current implementation does not do so, the client should, based on its capacity, bid for a particular maximum request size, and then accept any value that the server returns to it. Section 3.2 describes more fully how these values interact.

3.2 Server Implementation

Like the EBD client, the server is a Linux kernel module composed of

- a group of network transmission routines
- a packet handler for incoming frames
- a kernel thread that does the I/O to and from the target device.

3.2.1 Network Transmission Routines

Since the network transmission code in the server is very similar to that in the client, this section covers only the differences betweent the two. Since on the server side the outbound data consists of the read responses, they are transmitted using the zero-copy mechanism. Unlike the client, the server rarely, if ever, floods the transmitter because most of the tranmissions, including all of the ones transmitting frames with a data payload run under the control of a kernel thread rather than under the disk task queue. The transmission routines are wrapped by code that increases the flow control credit by the size of the response in sectors.

3.2.2 Packet Handler

Again like the EBD client, the server's packet handler registers with the link-layer code to receive all incoming packets with the EBD frame type. When it gets control, it is running in the `NET_RX_ACTION` tasklet handler as a part of the system's low-level network receive processing. `NET_RX_ACTION` is the highest priority tasklet handler and runs before `NET_TX_ACTION`.

HANDSHAKE requests are handled in much the same way as on the client side and generate immediate responses. Otherwise, the packet handler must pass the request to the kernel thread for processing, and it uses a one of a set of pre-allocated structures to pass the required information to the thread. The packet handler's processing of data requests is dependent on whether they are *READ*s or *WRITE*s. Since *WRITE*s have already been declustered by the server, the packet handler pulls a one of a set of pre-allocated structures off of a free list, clones the incoming `sk_buff` to capture the incoming data and wakes up the kernel thread to do the I/O. If the EBD request indicates that a *WACK* is required, the packet handler creates an EBD *WACK* response and calls the transmission routines to send it. However, if the request is a *READ*, the packet handler declusters it, pulling off and filling in a structure for every block requested and queuing each structure to the kernel thread. It only wakes up the kernel thread when all of the structures have been queued for processing. The packet handler then does the same *WACK* processing as it does for a *WRITE*.

3.2.3 Kernel Thread Processing

When the kernel thread wakes up, it processes the entire queue, running until there are no more structures on it. For each structure, it does a `getblk` , if it has not already done one, for the block being requested, which returns a buffer structure for it.

READ requests are handled in multiple passes. If the block in memory is up-to-date, the thread calls the transmission routines to send a response containing the data using the zero-copy mechanism. Otherwise, the thread submits the buffer for reading from the disk and puts the communication structure back on the work queue. The thread keeps a counter that is incremented for each block that requires another pass, and when this counter reaches a set limit, the thread schedules the execution of the disk task queue and calls the scheduler to allow the system an opportunity to execute the task queue. The effect is to do a long enabled spin for a fixed number of iterations before giving up control to the disk request processing code. How many passes this processing requires depends on the disk scheduling and load, but generally, one disk task queue execution should be sufficient to bring into memory every block that was not present on the initial pass. Once the read data is finally transmitted to the client, the thread releases the buffer and returns the communication structure to the free list.

For *WRITE*s the payload is copied to the buffer, which is then marked as dirty, and the buffer is released back to the operating system to be flushed to the disk. Except in the case of synchronous writes, as soon as the block is released to the underlying operating system, the thread calls the transmission routines to send the acknowledgement response back to the cleint. Synchronous writes are flushed immediately to the disk, prior to the sending of any acknowledgement. This requires that the thread put the communication structure back on the work queue. Just as for *READ*s, the thread maintains a counter of the number of outstanding synchronous writes, and if a set limit is exceeded, it waits for the buffer to be done. Eventually, when the buffer has been written, the thread gets control back. Generally, waiting on one buffer has the effect of finishing a number of them, so that some number of subsequent waits return immediately.

3.2.4 Server Hazards

As with the client, there are some hazards in server operation. As mentioned above, transmitter flooding, while certainly possible, rarely occurs. However, it is quite easy to exhaust the memory that is allocatable at interrupt time. The packet handler has to clone the `sk_buff` for incoming writes as well as transmit any required *WACK*s. If packets arrive in large, very frequent bursts, as from a very aggressive client with a large flow control credit, and if active buffers consume large amounts of memory, a server system with 512 MB of physical memory and no other load can run out of memory allocatable at interrupt time. The effect manifests itself in one of three ways.

- The server's packet handler cannot clone an `sk_buff`.
- The packet handler is unable to transmit a *WACK*.
- The network driver cannot allocate a fresh set of receive buffers.

This phenomenon is especially pronounced with jumbo frames for a very interesting reason. A receive buffer actually has to have 9022 bytes of space for the incoming data, and even though EBD cannot use more about 4030 bytes of it, each receive buffer still uses at least part of three pages and may force the allocation of the three free pages. When they fill up, the network driver allocates new receive buffers in a group to form a new receive ring for hardware. This behavior is specific to the driver for the particular gigabit ethernet card used in the test systems, and other drivers and interfaces might exhibit different behavior.

The best way to solve these problems is to avoid them by restricting the flow control credit to relatively small values and limiting the maximum number of sectors allowed in a single client request before declustering. Although experience allows one to find a workable set of values, clearly, it would be much better to have the client and server do initial calibrations and then adapt the values to fit the workload and available resources.

On both the client and the server sides of the EBD implementation, the introduction of zero-copy and the use of the gather/scatter features of the network hardware resulted in more pressure on the network and memory resources. Prior to the introduction of zero-copy transmission, both the client transmitter flooding and the server memory problems were never observed and may not have existed at all.

3.3 Discovery, Naming and Calibration

Naming and discovery are beyond the scope of the EBD protocol. In current implementation, they are statically configured at compilation or module load.

Certain parameters such as the capacity of the combination of the network and the server to hold in-flight frames amd the base timeout values are currently set by hand rather than being calibrated automatically by the client and server. Experience clearly shows the need for automatic calibration.

3.4 Control and Statistics

Both the client and the server report specific statistical information through the `/proc` file system. In addition, both have certain control parameters such as the amount of debugging output printed that can be set through `/proc` .

3.5 Kernel-Imposed Limitations

The structure of the Linux 2.4.18 kernel and its device drivers impose some limitations on the implementation, making it less functional and efficient than it might be otherwise.

First, there is one assumption that Linux kernels make about block devices which is false in the case of EBD and which requires significant kernel changes to remove. Linux assumes that the block size of a block device is somewhat arbitrary. There is an internal kernel function `set_blocksize` that sets the block size of a block device. It is used by swap initialization to set the block size of a swap partition to the page size of the machine, by ext2 and ext3 to set the block size of a device being mounted when the superblock is being read and by the generic raw I/O code to set the block size of a block device associated with a raw device to 512 bytes. In the case of EBD, the block size is a property of the remote device and is limited by the MTU of the network: the client cannot arbitrarily set it. There is no way for a block driver to override `set_blocksize`. While there are special versions of the kernel that contain EBD-specific code in swap initialization and in the superblock logic to get around this problem, Linux should have a way for a driver to register a driver-provided function that `set_blocksize` calls for that device.

Second, the long polling loops in the server are the direct result of the absence of a good, asynchronous I/O mechanism inside the kernel and a way of waiting for multiple events.

Third, there is no obvious way to adjust and manage the amount of memory available for allocation at interrupt time, making it difficult to devote additional memory for receive buffer allocation.

Fourth, in situations where the network offers jumbo frames and EBD uses them, the kernel forces it to waste approximately half of each frame because the block I/O subsystem and the ext2 and ext3 file systems do not support block sizes larger than 4096 bytes on the i386 hardware. This is less of a problem on some other architectures that do not have this restriction.

4 Functional Evaluation

The overall goal of EBD is to be ”just like” a local disk, including not only how well it handles ordinary reads and writes through the file system layer but also whether it offers all of the I/O modes and functional support provided by a standard disk driver including that required to support the needs of the operating system itself.

4.1 Operating System Support

One of the difficulties faced in implementing an environment in which all storage is networked is the assumption of a local disk made by operating system components and the relatively large number of dependencies that the full network protocol stack has on the rest of the operating system in the current generation of systems. Among the critical operating system functions that depend on file system and disk access are:

- kernel loading during boot
- root file system access for kernel modules, libraries, binaries and configuration files
- file system mounts
- logging
- dump
- distributed file system serving
- high-performance web and media serving
- swapping.

During boot and kernel initialization, storage access is dominated by reads, typically for a small number of files and generally sequential in nature. Given the simplicity of EBD, it is possible to incorporate it into a second-stage boot loader and use it to read the kernel image or initial ram disks into memory. Second-stage boot loaders are usually relatively small programs that run in a restrictive environment, but the amount of code required by EBD is small, especially in the context of one that already supports remote boot using DHCP and tftp. In addition, the diskless server implementation of Linux, Linux-DSA, is able to use an EBD minor to hold its root file system. In addition, the kernel initialization sequence has to change slightly to recognize that the root file system is to be mounted from EBD and to ensure that EBD is initialized but only after the network device code is operational. Although it should be possible to put the EBD client driver and the network driver in an initial RAM disk, initialize them together and then swap the root over to EBD, this has not been tested.

After kernel initialization, many of the disk accesses are actually normal user-level file accesses by daemon processes and pose no particular problems. However, there is a trend to migrate some of what has been user-level file access back into the operating system kernel. For example, the standard Linux implementation of the NFS server daemon is now a set of kernel threads. Similarly, the very high-performance Tux web server [31] runs and does file access from within the Linux kernel. However, one of the most classical uses for in-kernel file and disk access and one of the most difficult to support using a networked device is swapping. EBD can support swapping both to a minor at the block level and to a file in a file system. To do this required the changes to `set_blocksize` described in 3 as well as minor changes to the swapping code to accept swap partitions whose block size is a divisor of the page size of the machine. Strictly speaking, this change is unnecessary if the network supports jumbo frames and the block size of EBD is set to the page size. While current implementations of the Linux Network Block Driver (NBD) are able to support swapping, they do not do so reliably in low-memory scenerios, apparently due to locking problems and resource usage by of the TCP/IP stack implementation. The simpler implementation of EBD and its limited set of internal dependencies make it easier to provide a robust implementaton of swapping. By bypassing the TCP/IP stack, EBD avoids deadlocks involved with resource allocation and delays associated with stack processing. It acts as a direct transport for read and write operations to the device, which, in this case, happens to a server system with disks attached, and it very little interaction with the relatively complex network protocol stack. In particular, this tremendously reduces the potential for deadlocks, race conditions and resource shortages on the swap-out and swap-in paths. This support has been tested by bringing up the client system with 64 MB of memory and running a program that artificially forces continuous swapping. While system performance is degraded by the swap thrashing, responsiveness is similar to that of a local swap disk and the system remains stable.

4.2 Compatibility with Utility Programs

There are a few utility programs that deal directly with disk devices including `fdisk`, `mkswap`, `mkfs`, `fsck` and a variety of utilities for tuning and examining ext2 file systems. Since different minors under EBD access different remote disk devices or partitions, `fdisk` does not have any function on EBD devices and is not supported. On the other hand, all of the other programs work over EBD since EBD supports all of the `ioctl` calls required of a standard Linux disk device driver.

4.3 Non-standard Access Modes

Current versions of Linux support two access modes on block devices that are different from ordinary read and write – raw I/O and `O_DIRECT` processing.

Since the overriding goal of the EBD work is to create an ethernet block driver that is ”just like” a standard disk block driver and since one of the features of standard disk block drivers is raw I/O, it is reasonable to attempt to provide the same function with EBD.

In Linux, the raw I/O interface to block devices is implemented using the raw character driver. This driver provides an `ioctl` that binds one of its character minor devices to a particular block major and minor pair. Programs can then do raw I/O using the minor of the raw device, and the raw device driver converts the character I/O into sector-at-a-time I/O. However, the internal kernel mechanisms that are used require that the blocksize of the minor device be set to the standard sector size of 512 bytes. The kernel implementation makes use of the `set_blocksize` function described earlier. In this case, setting the block size of the EBD device to 512 bytes as the implementation of raw I/O is compatible with the protocol. However, the kernel needs to invoke the EBD device driver, so that it, in turn, can send a control frame to the server for the minor. The server then does the corresponding `set_blocksize` on the block device for the EBD minor. Just as in the earlier case, where the kernel sets the block size of the device when initializing swapping or reading the superblock, there should be a mechanism to have `set_blocksize` call a driver-provided function for network block devices.

`O_DIRECT`, which is implemented at the file system level but uses some of the same internal kernel mechanisms as raw I/O, functions correctly for both block and file access over EBD. However, the implementation has not been verified for `O_DIRECT` support.

4.4 SMP Safety and Performance

The code in the current implementation on both the client and server sides does very careful locking of the data structures used internally, and there is good reason to believe that it is SMP-safe. Since the client code maintains separate data structures for each minor and minimizes the hold time on the `io_request_lock`, the major impediment to client SMP scaling is likely to be access to the network interface transmitter. Although the protocol allows multiple network interfaces, the current implementation does not fully support it. On the server side, the code is again limited to a single network interface, and it uses a single kernel thread to process the requests from the client. Since that thread can spin waiting for the arrival of buffers from the disk, the current server implementation may not interact well with other applications running on the server system. Also, if the system has multiple I/O busses and disk adapters, an implementation with one kernel thread per processor up to the number of independent I/O paths should offer more parallelism and, hence, more throughput. To date, no SMP testing has been done.

4.5 Execution on Different Network Hardware

To date, most of the testing of the implementation has used one particular type of gigabit ethernet network interface card although one test configuration uses a 100 Mbit ethernet and a different network interface chip. The robustness of the protocol and its implementation, as well as the level of data integrity that both maintain, needs to be tested on a variety of network interface hardware and in environments that force a high retransmission rate.

As specified, the EBD protocol should flow over bridges, VLAN bridges [11] , and within virtual private networks, and, in theory, there should be little difference in performance although some of the congestion and flow control parameters may require tuning or re-calibration. However, this function is entirely untested.

5 Performance Evaluation

Much of the motivation for EBD lies in the hope that its simplified protocol and implementation structure leads to

- better read and write throughput
- reduced access latency
- lower network utilization
- less client CPU utilization

- better scalability due to lower server resouce consumption

than existing alternatives for remote storage access. This section describes a set of experiments run using EBD to test these hypotheses and to compare it with several other networked storage and distributed file system implementations. The experiments collected not only the results reported by the tests but also system and network utilizations as well. The benchmark programs

- bonnie++
- iozone
- PostMark
- tiobench

cover a spectrum of different file and disk uses and implementation styles. All of them are openly available and are built from the standard sources, changed only to ensure clean compilation for test environment.

5.1 Experimental Platform

The hardware used for both client and server in the evaluation is a set of 866 Mhz Pentium III systems running Red Hat Linux 7.3 except for a custom Linux 2.4.18 kernel built from the base Linux source distribution maintained on the kernel.org web site. All experiments reported here use the same hardware and software, configured the same way. In particular, the system executes as uniprocessor even though they are physically two-way SMP machines, and memory is restricted at boot time to 256 MB on the client (and only machine in the local disk tests) and to 512 MB on the server. An earlier set of performance tests used 64 MB of memory on both client and server, but this is smaller than most data-class systems today and prevented the execution of certain tests designed to determine the effect of the server's memory caching on test performance. In all cases, the file system on the target device is the standard Linux ext2 file system created with a block size of 4096 bytes per block. The experiments must use jumbo frames on the gigabit ethernet to allow EBD to support a remote disk with a file system block size of 4096 bytes. The 100 Mbit ethernet is not used in these experiments and is not configured on the test systems. Although the Extreme switch is a shared resource, the network runs as a point-to-point switched Ethernet at wire speed with very limited load on the other links. There is no indication that the shared switch is a source of significant experimental error. For EBD, all of the experiments use the implemented EBD protocol and the prototype client and server described earlier, implemented as loadable kernel modules. EBD runs with a single minor on the client side and the server supporting only that client. The NDB results use the standard nbd.o driver built from the 2.4.18 kernel sources, configured as a loadable module, and the nbd-2.0 distribution of the NDB server and client configuration code from Source Forge. For NFS the code is from the 2.4.18 kernel sources and is configured with default values. The local disk accessed in the local disk runs is the same disk and partition used as the remote disk in the other cases. Finally, the primary mechanism for collecting information about the utilization of the client and server systems is the Linux `sar` utility, running on both the client and the server and sampling at 20-second intervals.

5.2 Bonnie++

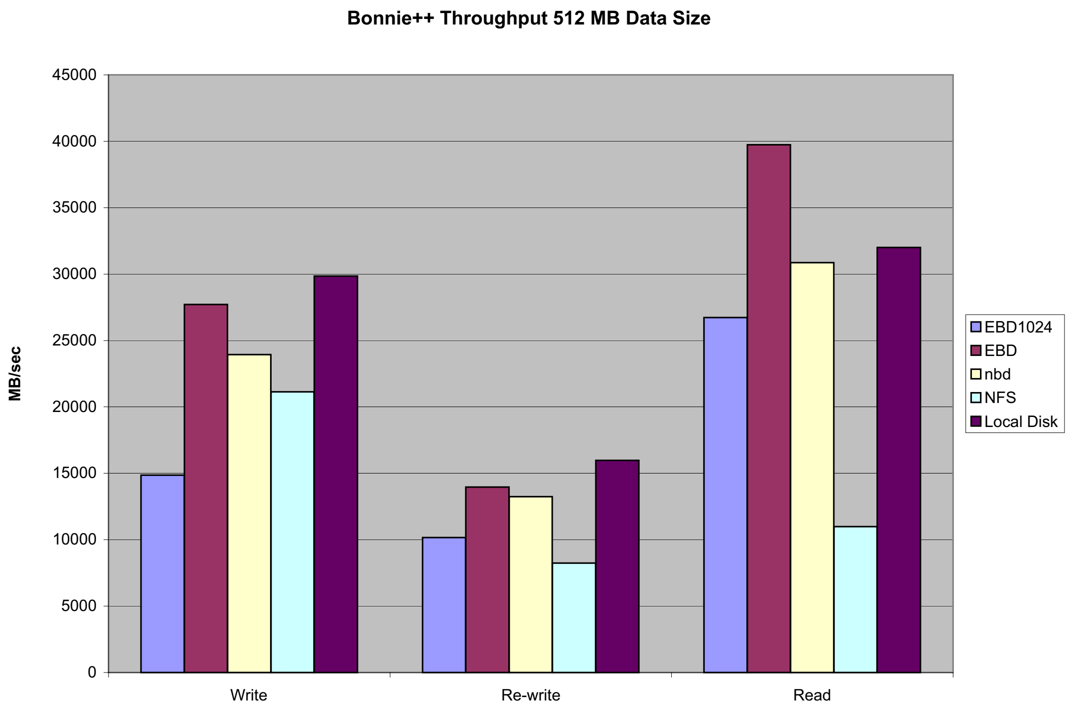


Figure 12: I/O throughput on bonnie++ at 512 MB data size

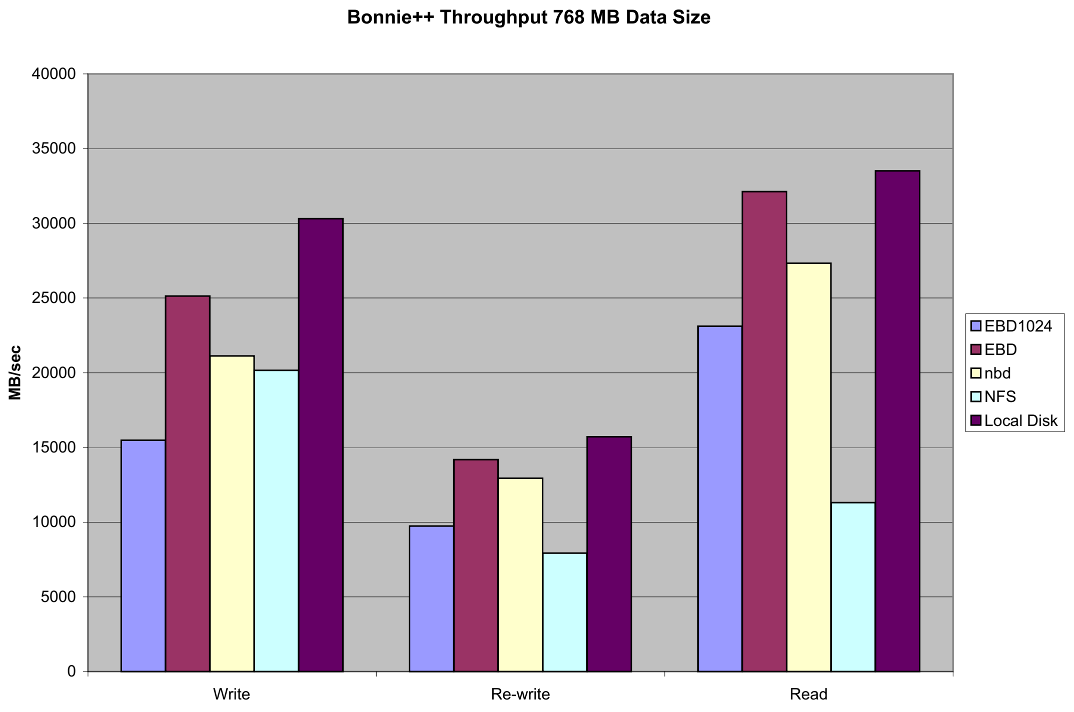


Figure 13: I/O throughput on bonnie++ at 768 MB data size

The first benchmark is the 1.02c version of the bonnie++ [7] benchmark. In all cases, the experiments skip the character I/O tests since they primarily test the implementation of the C library's character I/O routines. The benchmark writes

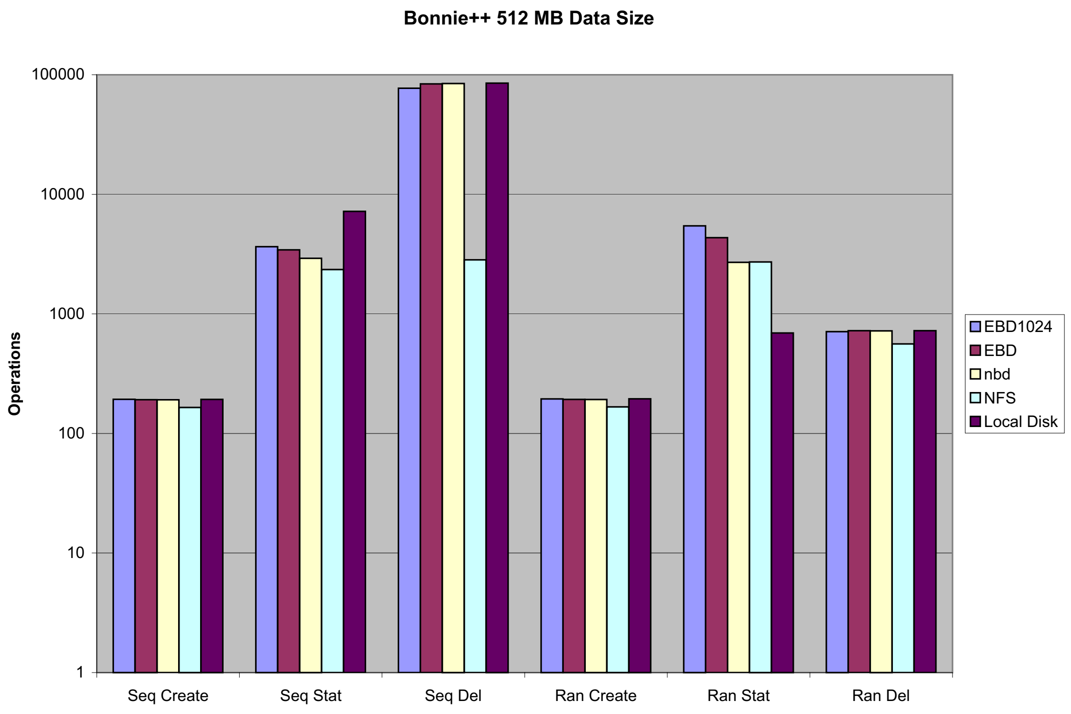


Figure 14: File operations performance on bonnie++ at 512 MB data size

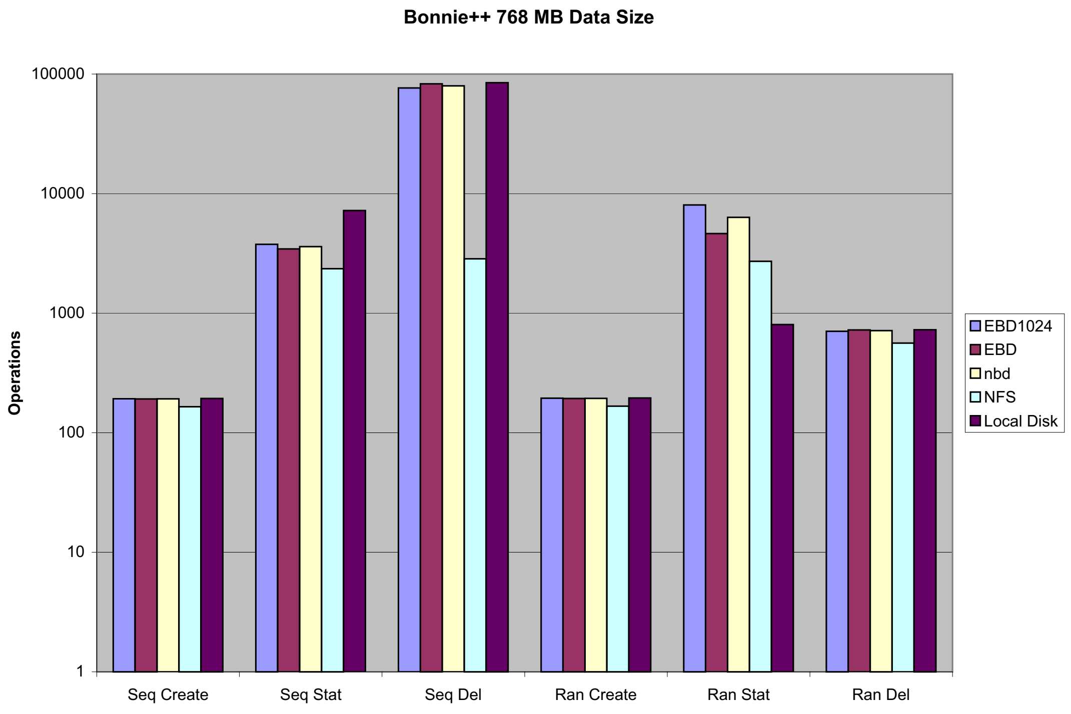


Figure 15: File operations performance on bonnie++ at 768 MB data size

a file of the specified size, then rewrites it sequentially and reads it back sequentially. Once that is done, it does a large number of `lseek()` calls, each followed by a `read()` of a buffer, dirty and `write()`. Finally, bonnie++ does a set of file creation and deletion tests which create files, stat them and then delete them. There are two sets of tests, one with a file size of 512 MB and one with a file size of 768 MB in order to distinguish between the cases where the file nearly fits in the server's memory and where it is too large to cache effectively. The file creation count is 51,200 with the file sizes randomly distributed over a range from 0 to 256K bytes and scattered over 10 directories. The tests use an iteration count of only three (3) since NBD hangs with larger counts. All other parameters are defaulted. For bonnie++, there is one additional comparison point, labeled as EBD-1024 in the results graphs. This is a run of the bonnie++ using EBD but with the network configured for standard 1500-octet frames and the test file system blocked at 1024 bytes per block.

Figure 12 shows the results for the write, re-write and read back phases of bonnie++ at 512 MB data size while Figure 14 presents those for the file system operations at that size. Similarly Figure 13 and Figure 15 show the results at the 768 MB data size.

5.3 Iozone

Iozone is a comprehensive file system benchmark that performs a large number of different file operations using different block sizes and different internal buffer sizes. The results reported here are from running the fully automatic form of iozone using the `iozone -a` command. In fully automatic mode, iozone tests a sequence of file and transfer size combinations with the file sizes ranging from 64KB to 512MB and the transfer sizes ranging from 4KB to 16MB. With file sizes larger than 32MB, transfer sizes size than 64KB are not tested. The operations performed are

- sequential write of a new file
- rewrite, that is, sequential write to an existing file
- sequential read of an existing file
- sequential read of an previously read file
- random read
- random write
- sequential read of a file backwards, that is, reading from the end to the beginning
- record rewrite, that is, writing and rewriting a particular spot in a file
- strided read
- write to a new file using the `fwrite` library function
- write to an existing file using the `fwrite` library function
- read from a file using the `fread` library function
- read from a recently read file using `fread`.

Since the memory size of the client machine is configured to be 256 MB, the 256 MB and 512 MB data sizes provide the best tests of the technologies under comparison. The results for read, reread, write, rewrite, random read and random write are summarized in the following graphs. All transfer rates are normalized to that of the local disk.

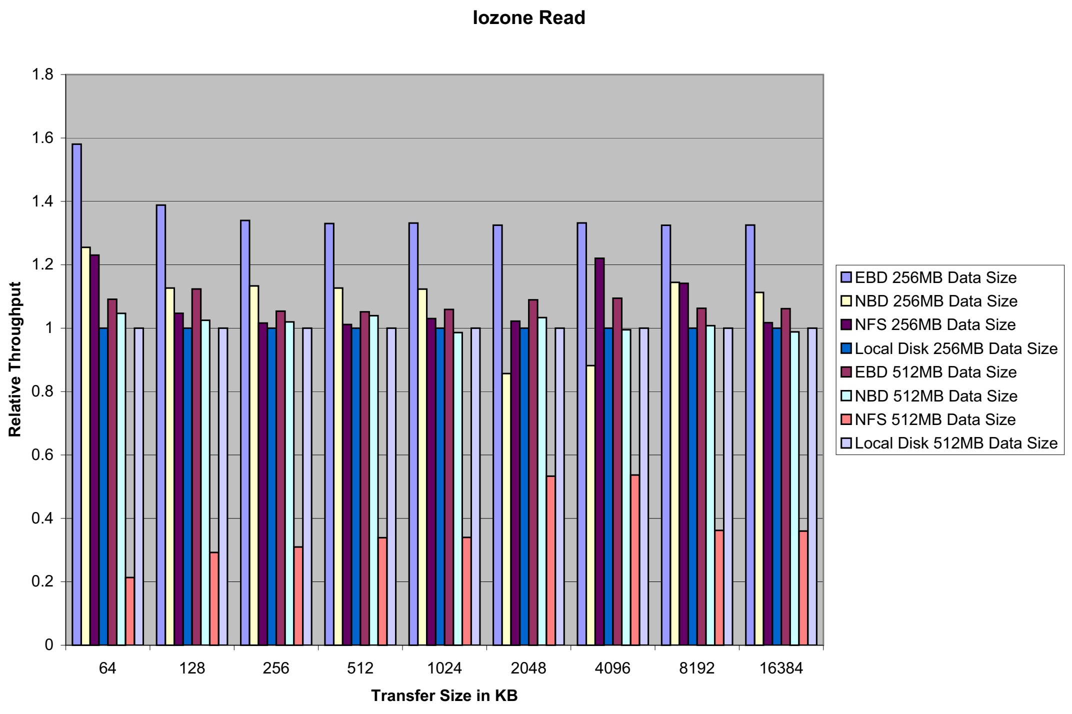


Figure 16: Iozone Read Performance

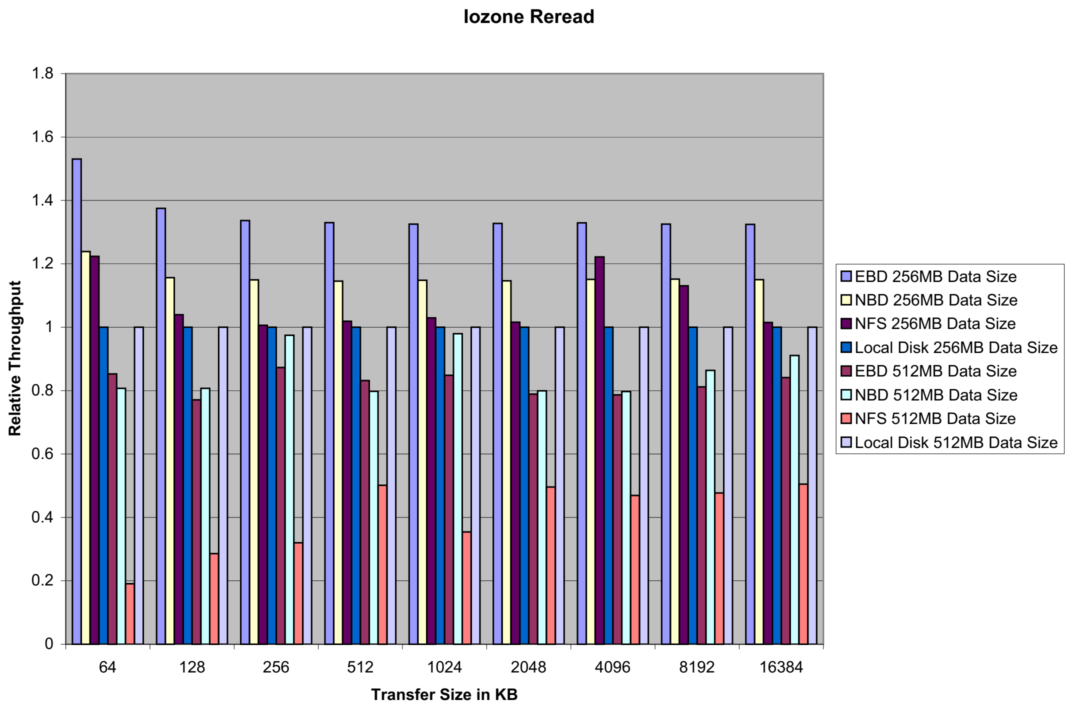


Figure 17: Iozone Reread Performance

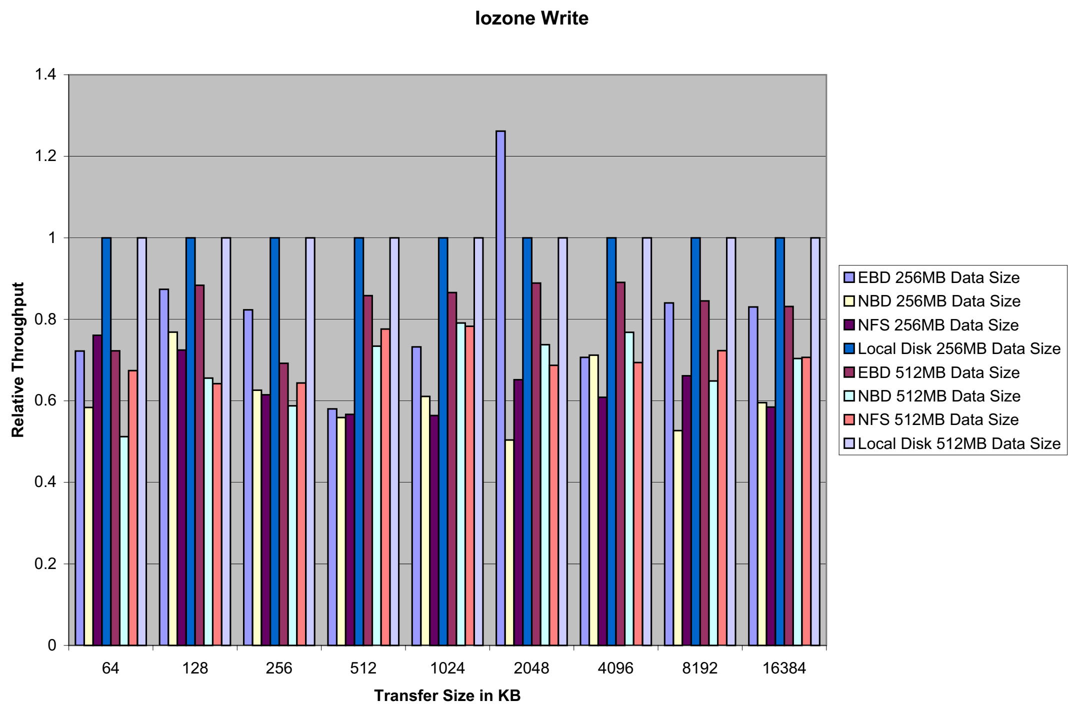


Figure 18: Iozone Write Performance

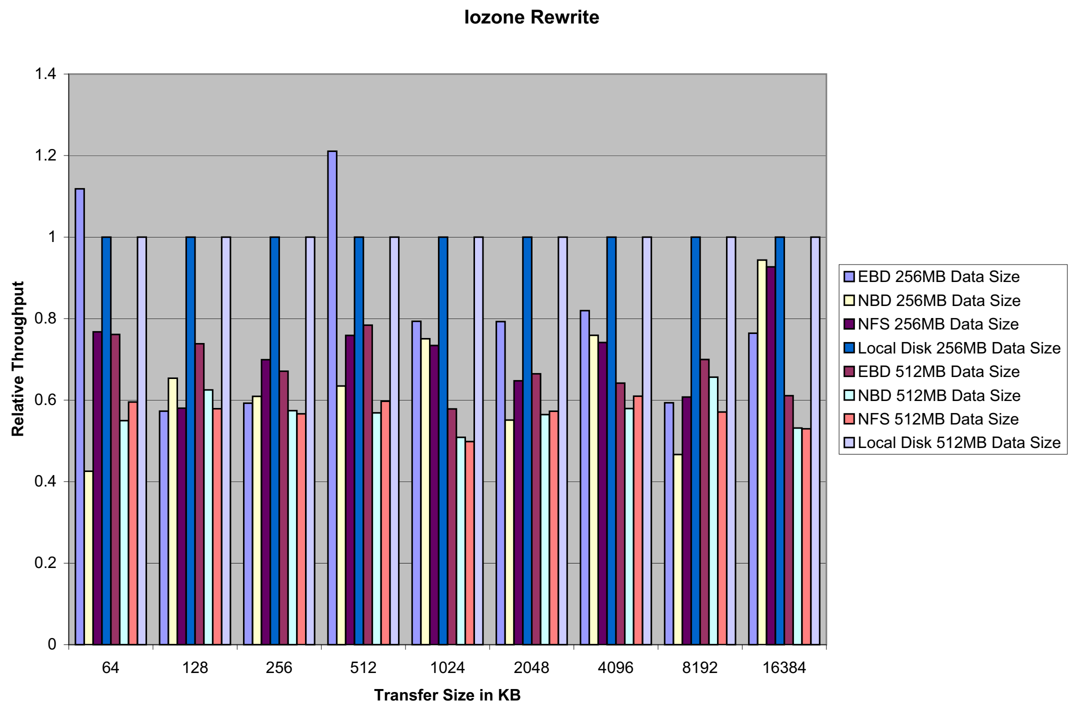


Figure 19: Iozone Rewrite Performance

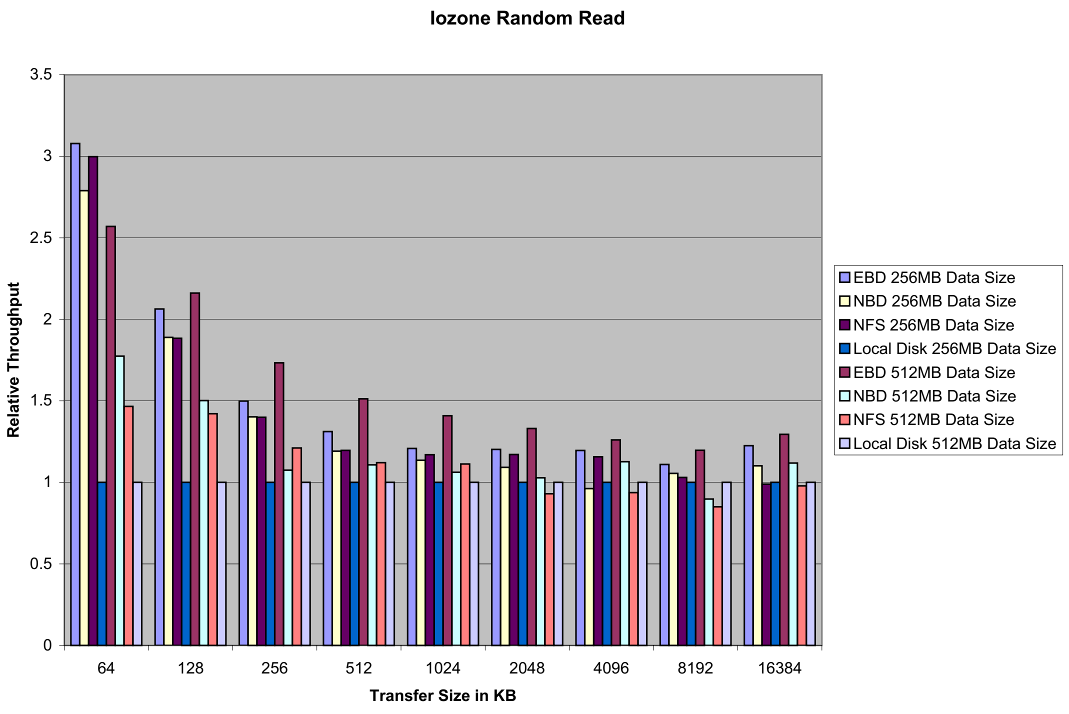


Figure 20: Iozone Random Read Performance

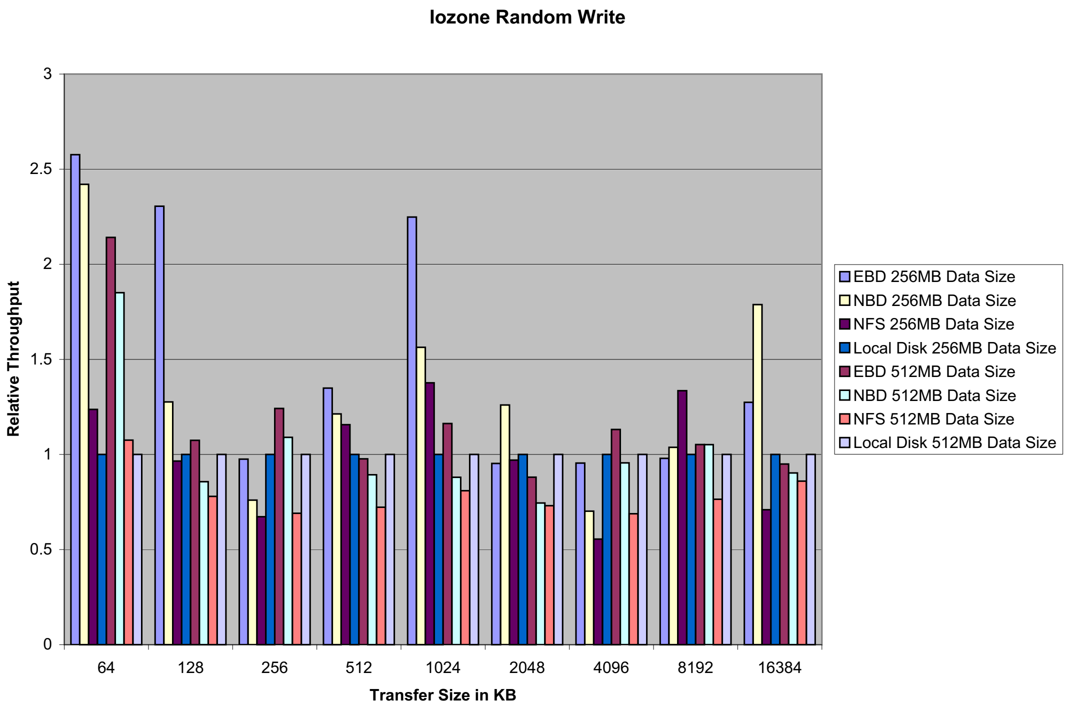


Figure 21: Iozone Random Write Performance

5.4 PostMark

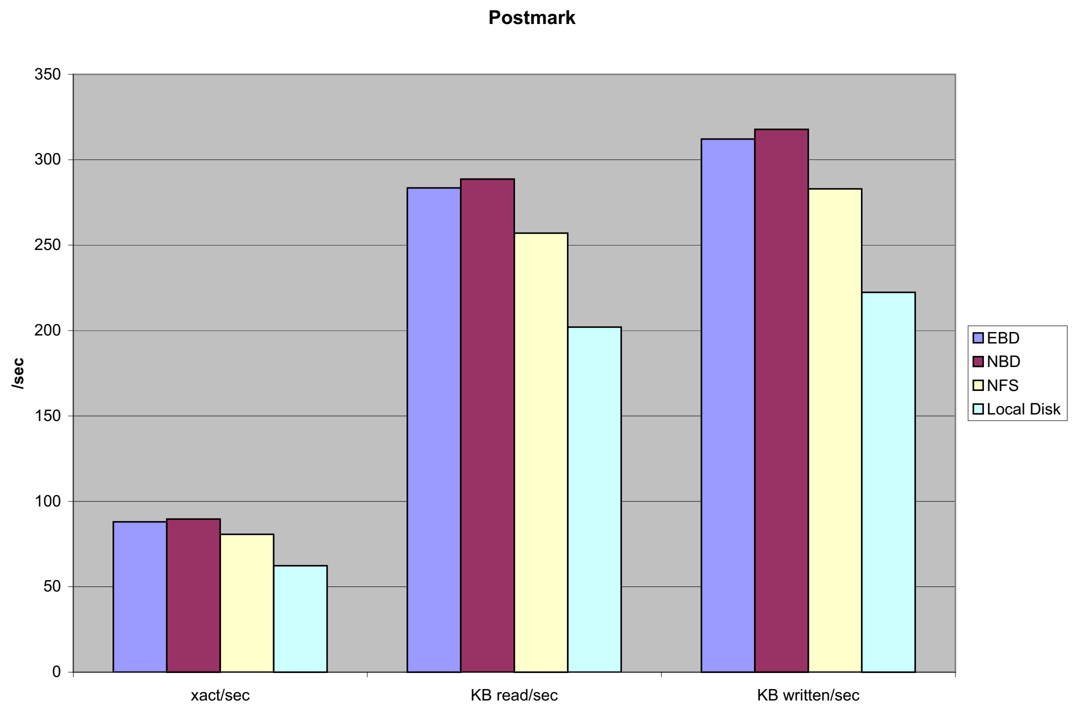


Figure 22: PostMark Performance

PostMark [14] is a benchmark developed under contract to Network Appliance, and its intent is to measure the performance of a system used to support a busy email service or the execution of a large number of CGI scripts processing forms. It creates the specified number of files randomly distributed the range of file sizes: it is typically used with very large numbers of relatively small files. It then runs transactions on these files, randomly creating and deleting them or reading and appending to them, and then it deletes the files. For calibration purposes, the tests include the three workloads reported in [14]. In addition, the Postmark tests add a fourth, very large run using 50000 files and 1000000 transactions. Figure 22 presents the results of only this last test. It is a highly stressful test that runs for many hours, and it is intended to model the file activity of a busy email server or a web server running CGI scripts that are creating and updating files, running over a period of many hours. Somewhat surprisingly, NBD performs slightly better than EBD on this test. Understanding the reason for this requires further analysis although it may be due to the very large number of metadata update operations done by PostMark that, in turn, lead to a large number of writes. As pointed out previously, NBD marks write operations done without necessarily doing anything more than copying the data to be written to a local socket buffer. Even so, EBD is quite competitive.

5.5 Tiobench

Tiobench is a threaded I/O benchmark that is under development initially on Linux. It uses a variable number of threads, dividing the work evenly among them. It does sequential I/O plus a specified number of random read and write operations with the work divided evenly among the threads. It permits the user to specify the block size, that is, the transfer size of the individual file operations. The test results that are reported here are based on a set of four independent runs of the tiobench wrapper script using each of the storage access technologies. All four runs iterated three (3) times for each of a total file size of 256MB, 512MB, 1024MB and 2048MB for one (1), two (2), four (4) and eight (8) threads. Tne number of random I/O operations always defaults to 4000. The four runs differ in the transfer size with transfer sizes of 1024, 2048, 4096 and 8192 bytes, respectively, for each of them. Thus, each individual run represents $3 * 4 * 4 = 48$ separate sequences of operations, for a total of 192 test sequences for each technology. Each test aggregates the three data points collected from the base iteration factor and reports the throughput or rate in megabytes per second and the average latency in milliseconds for sequential read, sequential write, random read and random write for each combination of file size, thread count and transfer size. Also for each data point, the benchmark reports the percentage of requests that had latencies greater than two (2) and ten (10) seconds.

To keep the number of graphs within reason, only the throughput, average latency and maximum latency results for the following cases are reported here.

- 256MB data size and 8192-byte transfer size
- 2048MB data size and 8192-byte transfer size

The same graphs for 1024, 2048 and 4096-byte transfers are in Appendix B.

As indicated in Figure 26 NFS outperforms EBD on the sequential write test. The reason for this is apparent from Figure 28 which shows the EBD has a higher maximum request latency than NFS. The late arrival of some responses reduces the measured throughput of EBD when compared to NFS.

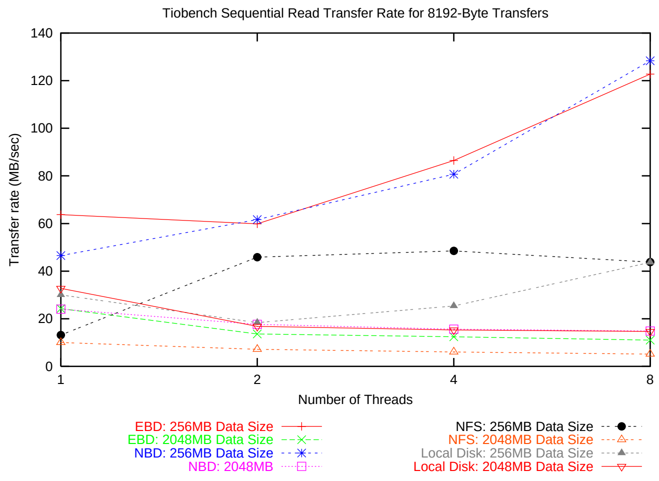


Figure 23: Tiobench Sequential Read Throughput

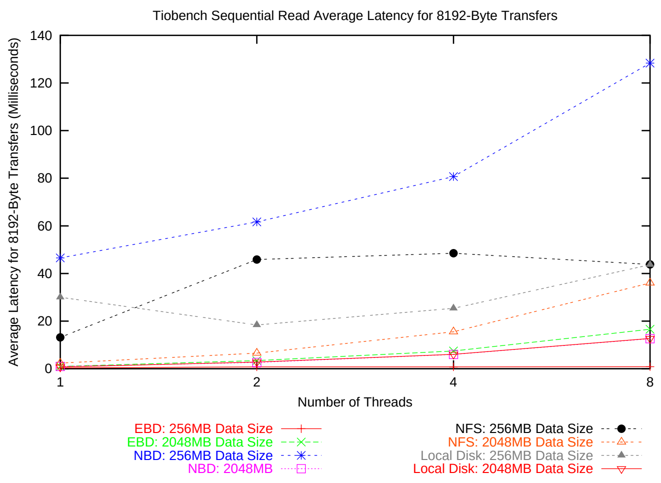


Figure 24: Tiobench Sequential Read Average Latency

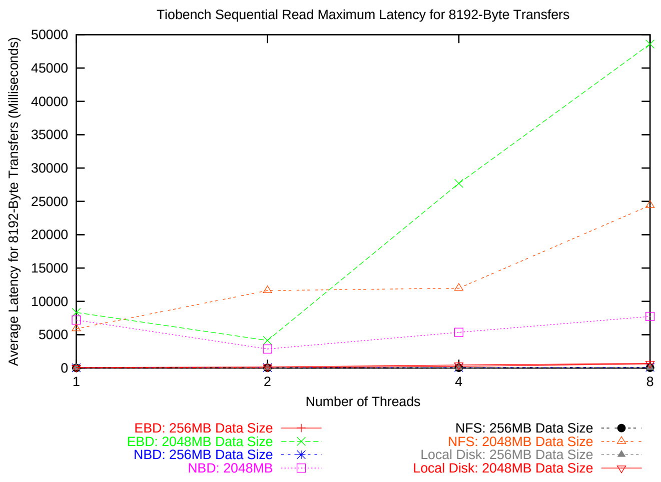


Figure 25: Tiobench Sequential Read Maximum Latency

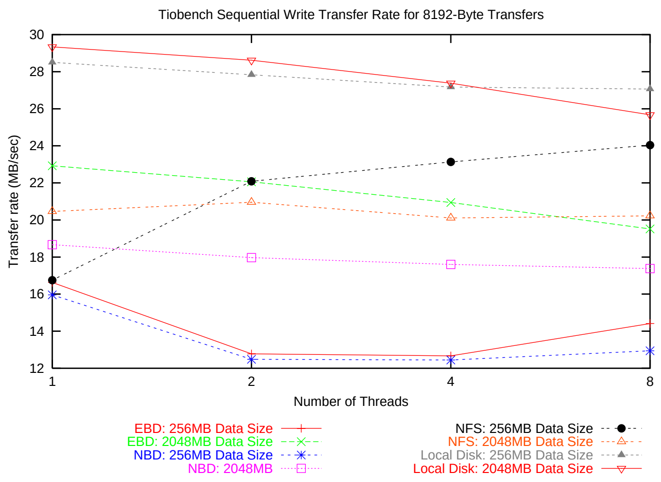


Figure 26: Tiobench Sequential Write Throughput

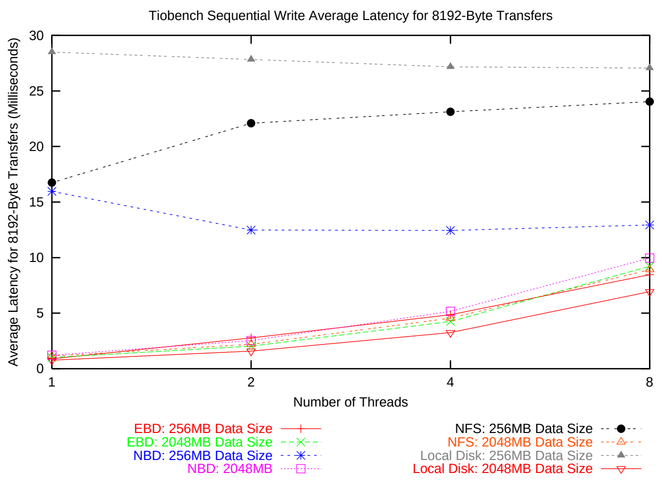


Figure 27: Tiobench Sequential Write Average Latency

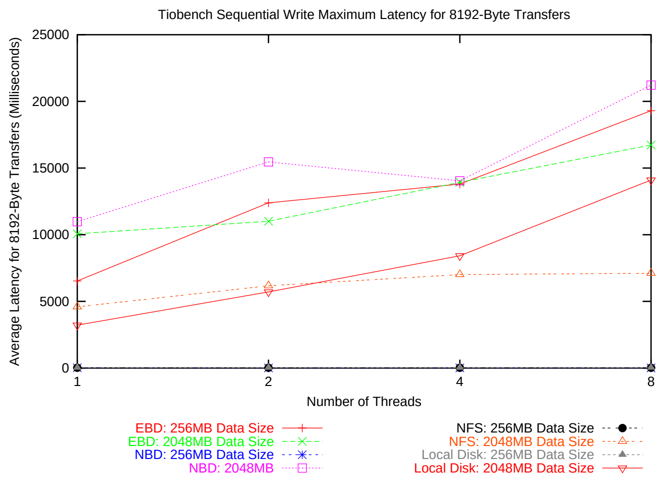


Figure 28: Tiobench Sequential Write Maximum Latency

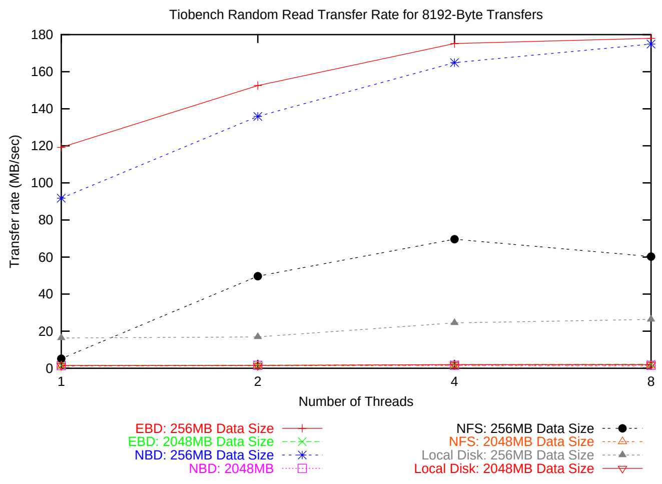


Figure 29: Tiobench Random Read Throughput

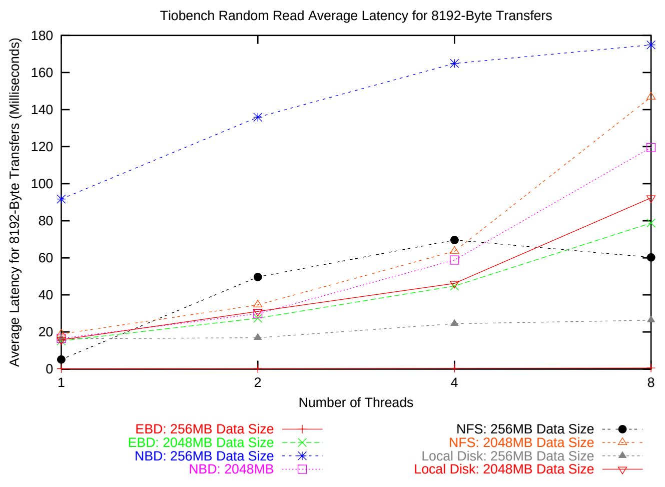


Figure 30: Tiobench Random Read Average Latency

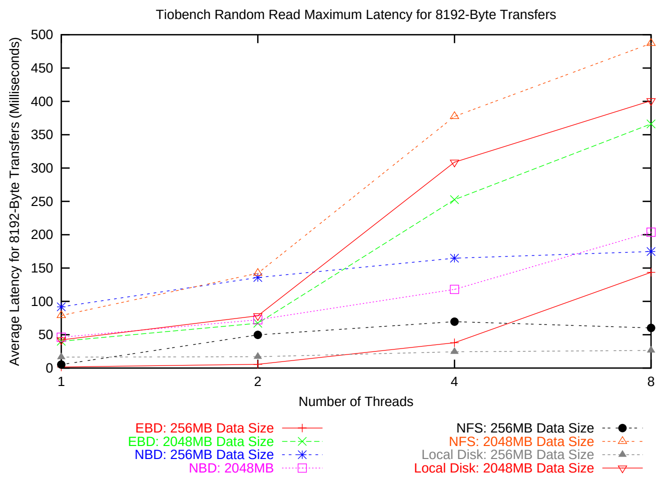


Figure 31: Tiobench Random Read Maximum Latency

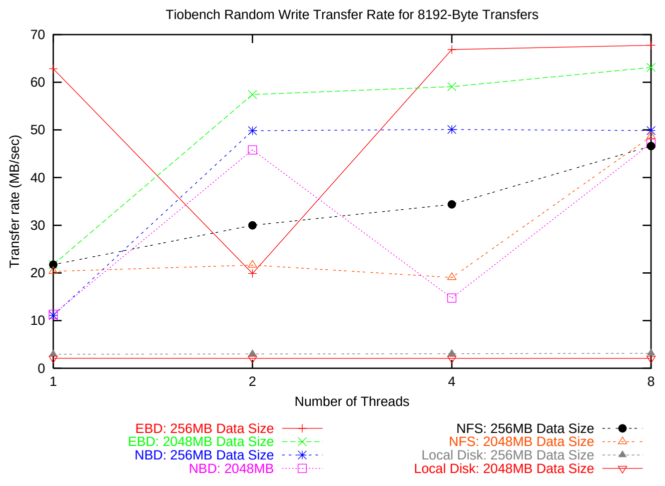


Figure 32: Tiobench Random Write Throughput

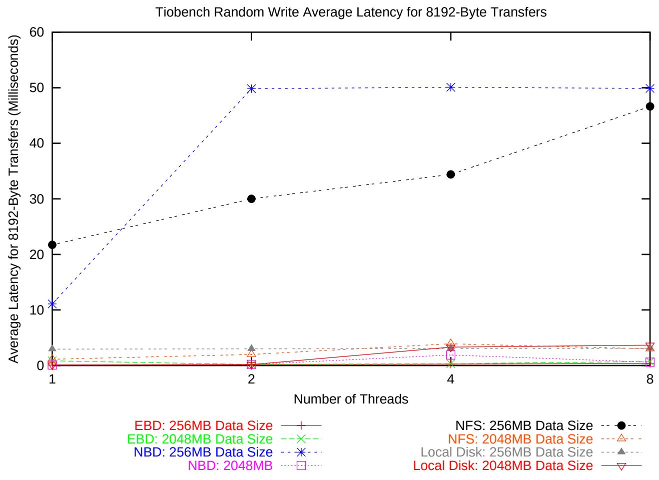


Figure 33: Tiobench Random Write Average Latency

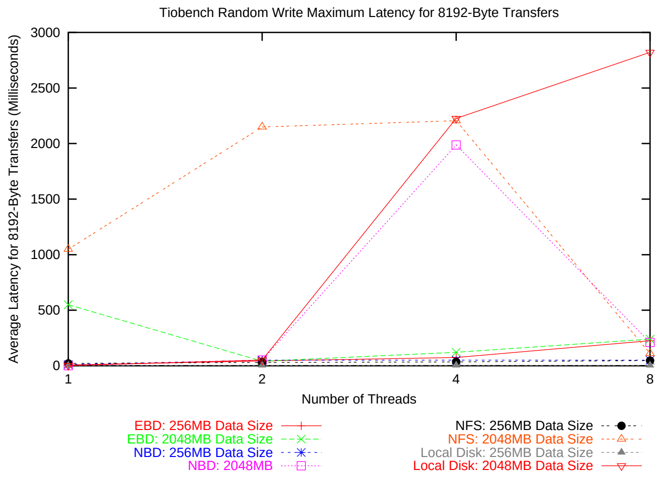


Figure 34: Tiobench Random Write Maximum Latency

5.6 Summary

Overall the data suggest that the performance of EBD is very competitive and exceeds that of the other options in a number of interesting cases. However, there are some surprises such as the fact that NBD outperforms EBD slightly on Postmark. Clearly, these results warrant further analysis. In addition, all of the tests to date have been using a single client, a single server and one particular network interface card. To understand better how EBD scales with the number of clients and on different network hardware requires further testing. Although the test runs collected comprehensive `sar` data, there has no analysis of it. However, Appendix C consists of a set of plots of this `sar` data.

6 Related Work

As mentioned in Section 1, the notions of access to storage over the network, the use of a block-oriented protocol to provide a remote disk device as opposed to a distributed file system and the use of link-level protocols dates back to at least the 1980s if not before. During the 1980s both simple PC file serving and disk-less workstations were popular, and there were academic studies of the performance of at least the disk-less workstation environments. One of the earliest studies of disk-less workstation performance was a set of measurements done on disk-less workstations, using three different operating environments, in the middle 1980s by Lazowska, Zahorjan, Cheriton and Zwaenepoel [17]. They concluded that disk-less workstations using the NFS distributed file system could be made to work acceptably with the right protocol design. A study of a later disk-less workstation environment was done by Nelson, Welch and Ousterhout using the Sprite system and appeared in 1988 [25]. More recently, there have been some newer studies of distributed file system performance and disk-less workstations such as that done by Baker et al using Sprite that appeared in the early 1990s [1].

Despite the heavy criticism and intense scrutiny that it has received, the dominant distributed file system protocol for the Linux/UNIX environment is NFS [4], and much of the work on storage networking either uses NFS or is in reaction to it.

Recently there has been considerable interest in block-level storage access directly over TCP/IP. This work is generally known as IP Storage, which has led to an Internet Standards draft on iSCSI or SCSI over IP [34]. The papers by Katz [15] and Van Meter et al. [23] suggested such approaches, and the work by Van Meter et al. can be considered to be a direct precursor of the iSCSI effort. Like our protocol, iSCSI offers device-level access to storage over the network. However, its protocol structure is radically different since it forwards SCSI device commands over the standard TCP/IP stack. This has many advantages in terms of extending legacy implementations to a networked storage environment, and it also extends the SCSI device protocol in such a way as to make it routable and transportable over a much larger distance than is possible with the physical implementation of the SCSI bus. On the other hand, it limits the underlying storage system to one that can accept and process, in one way or another, the SCSI command set. Moreover, it uses the full TCP/IP protocol stack, making its performance dependent on the quality of the TCP/IP implementation. Due to its complexity and dependencies, the ability to swap over iSCSI may not be available in many system environments. The iSCSI protocol's intended networking environment is the full Internet, rather than the single physical network environment required by layer-2 protocols such as EBD, and it is primarily intended to provide a way to do disk reads and writes over an IP network than to behave ”just like” a disk.

In addition to the use of iSCSI over standard TCP/IP networks to offer block-level remote storage access, there are storage-area networks built using the fibre channel standard. Fibre channel developed as a specialized network for storage and uses its own frame protocol for the transmission of data between processors and storage subsystems. Typically, fiber channel has been associated with storage-area networks for high-end systems although many Network

Appliance Filers use it to connect the Filer's disks to the processing unit. Recently, there has been work to extend fiber channel by sending fiber channel frames over IP networks [30].

The recent paper by Voruganti and Sarkar [38] provides a very good comparison of fiber channel, iSCSI and storage access over the new InfiniBand channel. Network Appliance has defined a distributed file system protocol, DAFS [13], based on NFS for environments such as InfiniBand that support the Virtual Interface (VI) Architecture or similar user-level network access features. As of this writing, the first generation of InfiniBand hardware is just coming on the market, and there are some commercial demonstrations of DAFS running over it but very little in the way of published information or performance data. However, for earlier hardware offering the VI interface, there is a DAFS reference implementation that Magoutis, et.al., describe in [22] .

There have been several papers on higher level constructs which can be implemented on top of block oriented network storage to provide reliability, scalability, and coherent access. The Petal [19] paper describes the organization of several distributed "virtual disk" servers along with mechanisms to reliably replicate data among them and transparently recover from server failure, as well as dynamically allocate new disk resources from the available pool. Frangpani [20] is a related project which implemented a coherent distributed file system on top of Petal. More recently Quinnlinn and Dorward presented a novel new context-referenced storage scheme called Venti [29] which is block based.

More directly related to our work is the existing network block driver in Linux. This driver offers approximately the same function as our prototype but uses a standard TCP connection to transport the data between the client and the server. Furthermore, nbd's implementation utilizes a more synchronous transaction model. There are actually two different implementations of this driver and its supporting software in common use. The first [21], which is included with the standard Linux 2.4.12 sources, is implemented on the client side completely as a modular block device driver and can be paired with either a user-level or an in-kernel server. The second version, known as the enhanced network block driver [3], moves the client-side network code to user-level and supports multiple connections between client and server using multiple threads of execution. Its server implementation is a user-mode program. Like the first network block driver, it uses standard TCP connections between the client and the server. In contrast to both of these network block drivers, our protocol and driver operate at the network-device level rather than using a TCP connection, a difference which increases overall performance and enables us to swap.

7 Future Work

Beyond the more extensive validation and some minor extensions mentioned earlier, there are a number of other areas for possible future work.

7.1 Kernel Changes

One of the mostly extensively changed areas in the Linux 2.5 kernel development series is the block I/O subsystem. An initial investigation of these changes suggests that the EBD implementation would benefit from them, but this is an area that requries further study. But one potential benefit is that the changes may make it easier to implement a unified disk and network buffering scheme, perhaps along the lines of done for BSD Unix in the IO-Lite project [26] . A unified buffering scheme would make it easier to implement zero-copy receives – *READ* s on the client and *WRITE* s on the server.

In addition, several other kernel modifications would help EBD including:

- the introduction of a way of allowing block device drivers to override the standard `set_blocksize` function

with their own private implementations
- the addition of convenient asynchronous I/O primitives
- the extension of the wait functions to allow kernel code to wait for multiple events
- the provision of better controls over the amount of memory that can be allocated at interrupt time.

7.2 Effects of the Network Environment

Ethernet was chosen as a medium for the initial prototype because of its wide acceptance and ready availability. SAN mediums such as VIA and Infiniband provide built-in flow-control and reliability, elements which Ethernet lacks. Since SANs typically implement these mechanisms in hardware (or are accelerated by hardware) they don't complicate the protocol implementation. Additionally, a more abstract media interface would allow us to run the protocol on top of higher layer protocols such as TCP/IP to evaluate performance trade-offs more effectively and allow its use in environments other than SANs.

Another network-related effect is that of packet scheduling and traffic shaping. Although Linux already has a number of different packet scheduling and traffic shaping schemes, the only ever tried with EBD is the standard FIFO packet scheduler. One potential use for a packet scheduler is to ensure that storage and standard network traffic share the network transmitter according to a pre-defined policy.

7.3 Self-Calibration, Adaptation Mechanisms and Performance Improvements

Performance evaluation and profiling have revealed several areas for improvement. In particular, the performance data collected are the result of a large amount of experimentation and hand-tuning of parameters. However, experience shows that hand-tuning is quite difficult, and there is no guarantee that the parameter values selected are, in fact, optimal. A much better approach is for EBD to self-calibrate based on the environment and adapt its behavior to the available resources and current load. Although the current code includes some features for adaptation, most of the required design and implementation has yet to be done.

The performance data strongly suggest that a few EBD requests receive very slow responses. For example, the tiobench sequential write test shows that EBD and NFS have comparable average request latencies, but the maximum request latency reported is sometimes as much as four (4) times that seen for NFS. The cause of this phenomenon is not known, and it clearly warrants further study since eliminating it should have a significantly positive effect on overall EBD performance.

7.4 Block Device Appliance

One can develop an EBD storage appliance along the lines of the NAS and SAN appliances already available in the marketplace. A more tightly coupled interface between the network, EBD, and the disk subsystem, coupled with a poll-driven event mechanism, should provide dramatic performance and scalability improvements on the server. The MIT Click [16] project and the AFPA [12] project have both shown how hybrid event models on network servers often yield significant performance improvements. A targeted hybrid kernel such as the Plan 9 [37] file server kernel would provide the best approach for a storage appliance based on the EBD protocol.

7.5 File System Changes

While this paper describes an effective block delivery mechanism, it provides only half the solution. File systems designed to the unique characteristics of the EBD protocol would provide superior performance and functionality. Custom file systems aimed at providing versioned application and operating system loads, copy-on-write for disk-less root file systems, and file systems tuned for serving web and web-cache data are all potential areas for future exploration.

7.6 Double Burst Elimination

Linux block drivers and the I/O request queue are implemented within the kernel to improve performance by merging contiguous requests before issuing the request to the underlying driver. The kernel implements this optimization by introducing an artificial delay between the time the block request is received by the kernel and the time it is actually issued to the device [2] : this is, of course, the intended effect of the pluggable device driver design and the disk task queue. While this artificial delay dramatically improves disk performance by clustering requests, our prototype introduces it twice – once on the client between the file system and the client-side EBD and again on the server between the EBD server and the target block device. Eliminating the client-side delay by transmitting the request as soon as it is received might improve overall performance and reduce the pressure on the transmitter caused by the bursts of request transmission. It is not clear whether the reduction in the ability to cluster *READ* s offsets this potential gain. Some clustering effect occurs on the server side since the blocks requested by the client appear as in runs of contiguous blocks. Whether changing the server to do some clustering of its own would improve the efficiency of the server's disk and the overall efficiency of the driver is an open question.

8 Conclusions

Leonardo Davinci was quoted as saying "Simplicity is the ultimate sophistication." [9] In this paper we have attempted to apply that virtue to the targeted niche of system area networks supporting disk-less workstations and servers. We believe the characteristics of emerging network technology support the simplification of system protocol stacks without loss of reliability or security while providing an increase in performance and scalability on both the client and the server.

The use of link-layer protocols shortens the instruction path-length on both ends of the storage network, enabling critical operations such as network swap and increases the overall efficiency of the system without the need for expensive protocol off-load hardware or specialized storage networking peripherals such as fibre channel. The tight binding to the underlying network media will facilitate a more straight-forward implementation of zero-copy network operations and performance tuning. The light weight nature of the protocol also improves efficient utilization of the network links and switching hardware.

Block-oriented operations are superior for certain classes of operations - primarily read transactions and scenarios where read-only data allows locally cached and maintained file system meta-data. Root file systems and per-node private directories are ideal targets. With some administrative overhead, block-oriented devices can also support application binaries, libraries and read-only data. On the whole, network block devices are not suitable for shared read/write data where coherency is important.

In short, the use of link-layer protocols and block-oriented network storage can provide a useful complement to coherent distributed file system protocols.

A Iozone Performance Plots

The first set of graphs in this section display the throughput performance of iozone in kilobytes per second for EBD and the comparison points on the write, rewrite, read, reread, random read and random write operations. Each graph plots the transfer rate versus the transfer size for a number of different total data sizes, so that each line represents a different data size and access technology combination. To display all of the data requires a total of thirty (30) graphs.

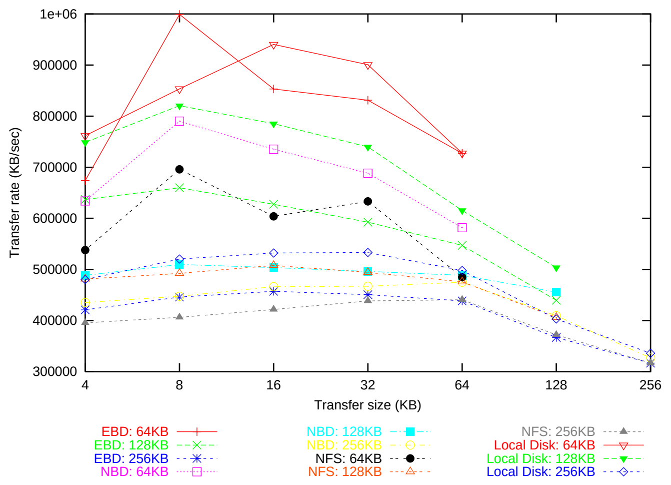


Figure 35: Iozone Read Performance, 64KB-256KB Data Size

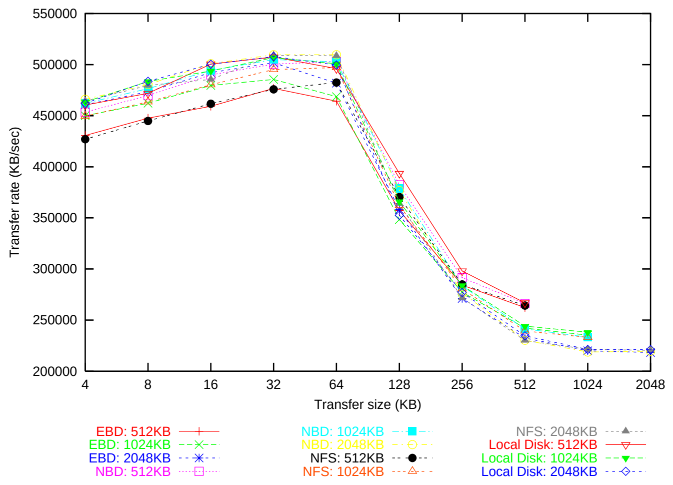


Figure 36: Iozone Read Performance, 512KB-2048KB Data Size

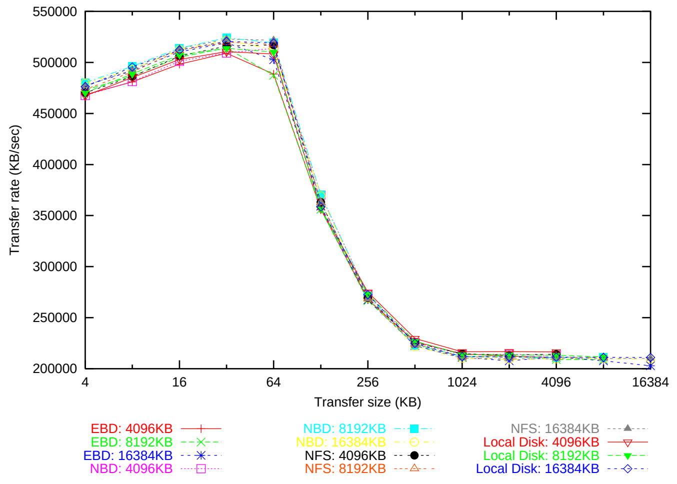


Figure 37: Iozone Read Performance, 4096KB-16384KB Data Size

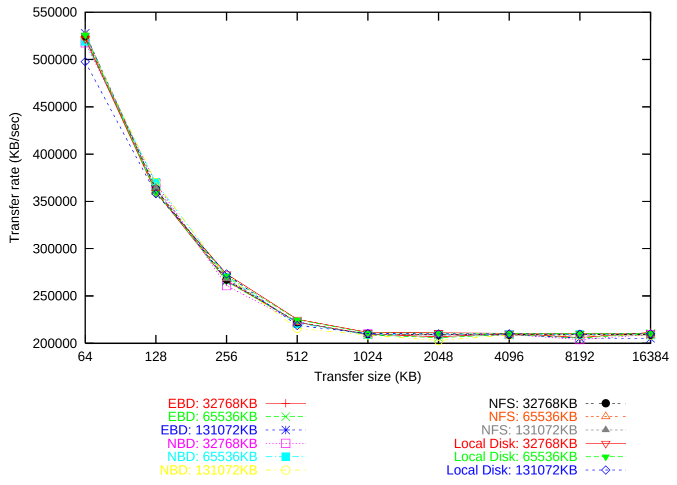


Figure 38: Iozone Read Performance, 32768KB-131072KB Data Size

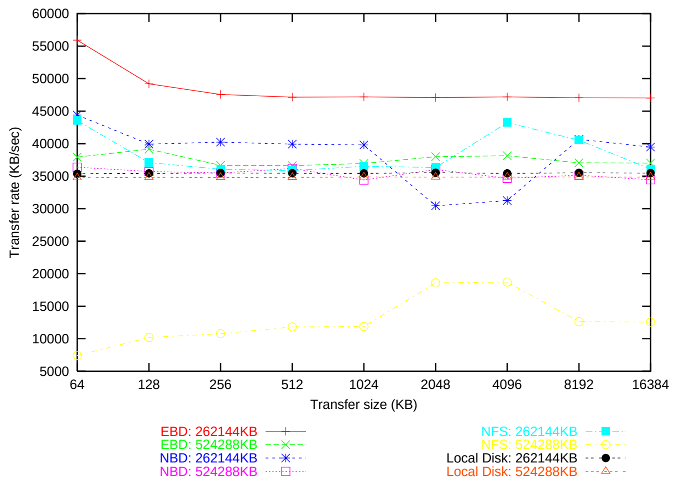


Figure 39: Iozone Read Performance, 262144KB-524288KB Data Size

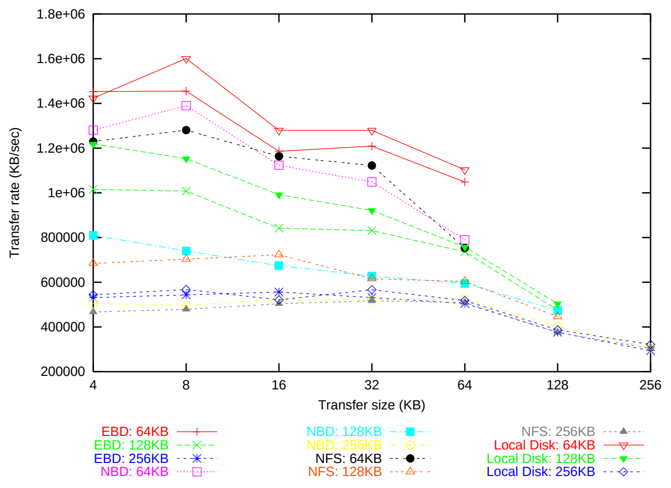


Figure 40: Iozone Reread Performance, 64KB-256KB Data Size

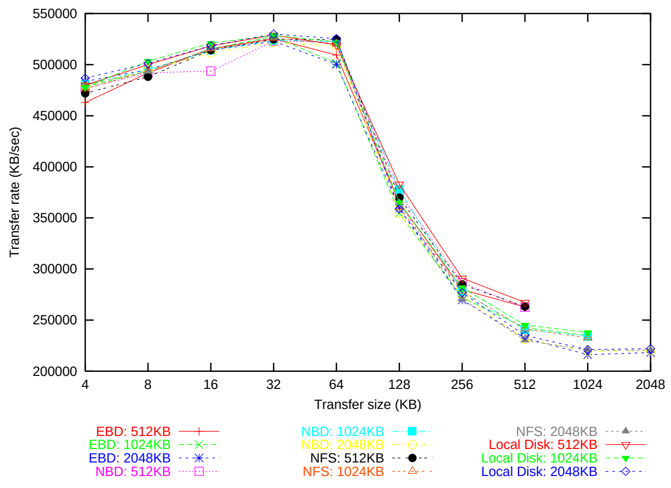


Figure 41: Iozone Reread Performance, 512KB-2048KB Data Size

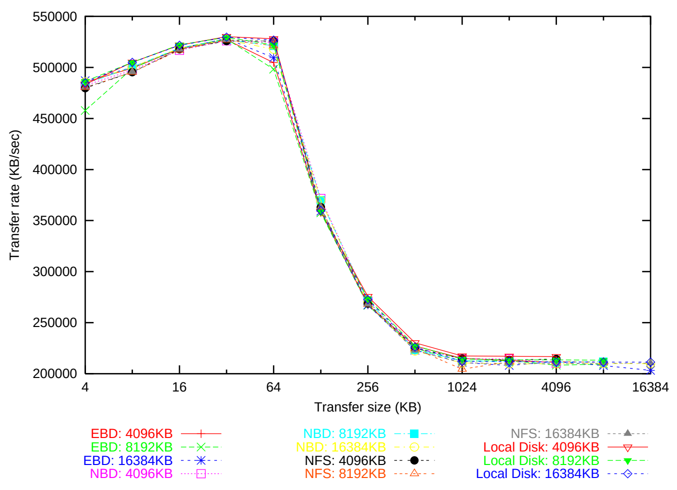


Figure 42: Iozone Reread Performance, 4096KB-16384KB Data Size

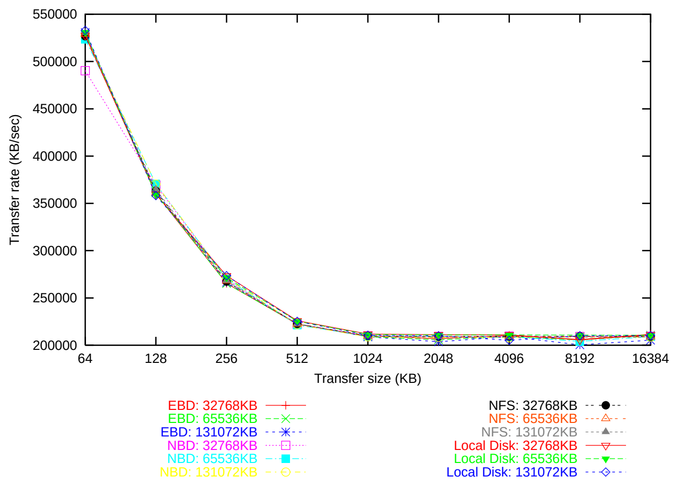


Figure 43: Iozone Reread Performance, 32768KB-131072KB Data Size

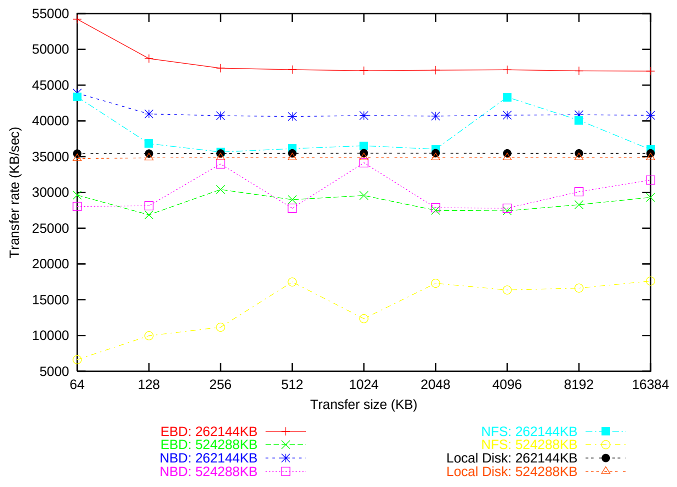


Figure 44: Iozone Reread Performance, 262144KB-524288KB Data Size

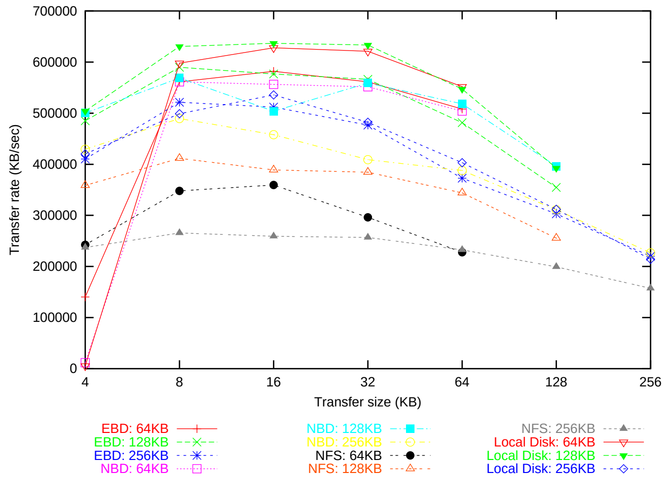


Figure 45: Iozone Write Performance, 64KB-256KB Data Size

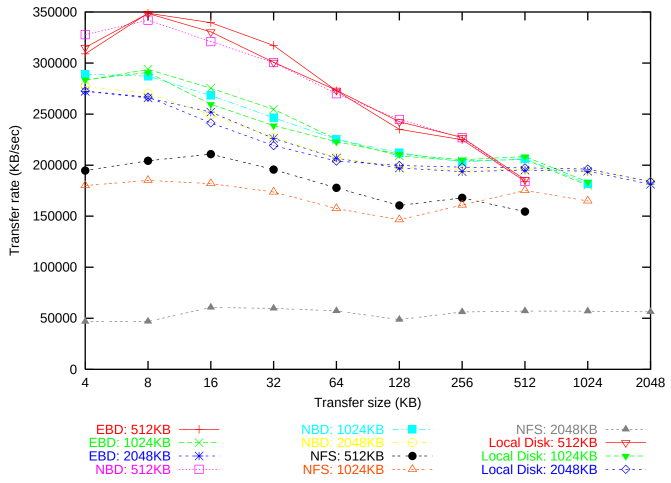


Figure 46: Iozone Write Performance, 512KB-2048KB Data Size

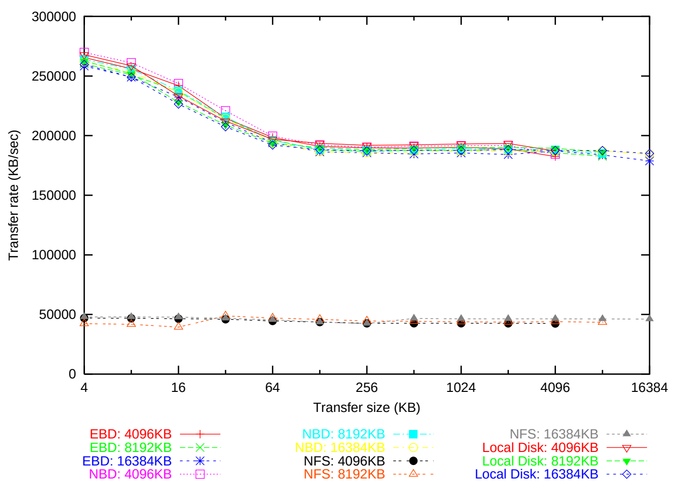


Figure 47: Iozone Write Performance, 4096KB-16384KB Data Size

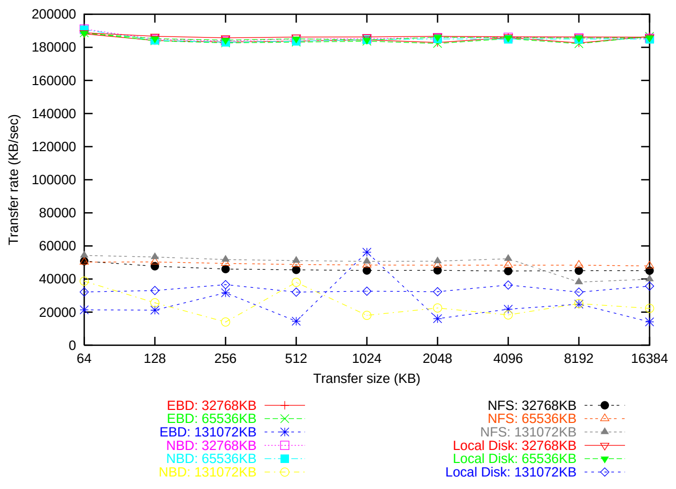


Figure 48: Iozone Write Performance, 32768KB-131072KB Data Size

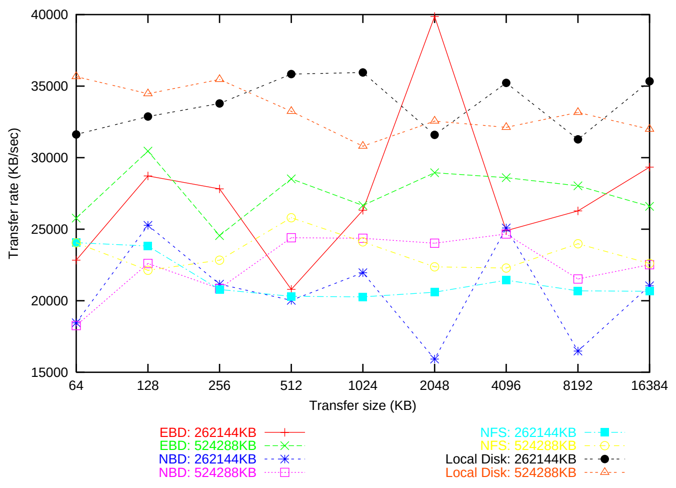


Figure 49: Iozone Write Performance, 262144KB-524288KB Data Size

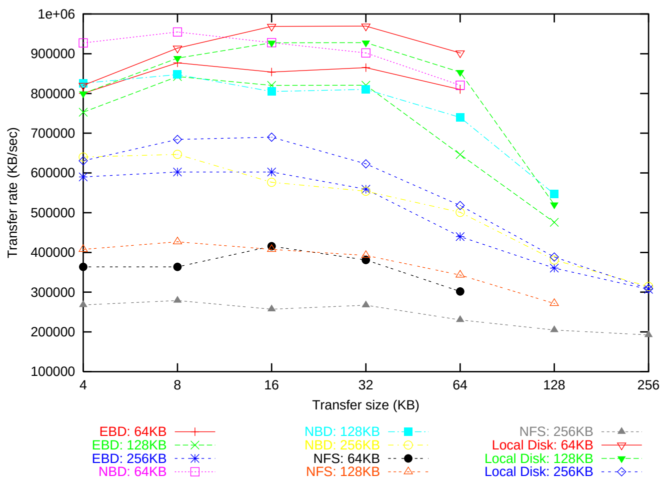


Figure 50: Iozone Rewrite Performance, 64KB-256KB Data Size

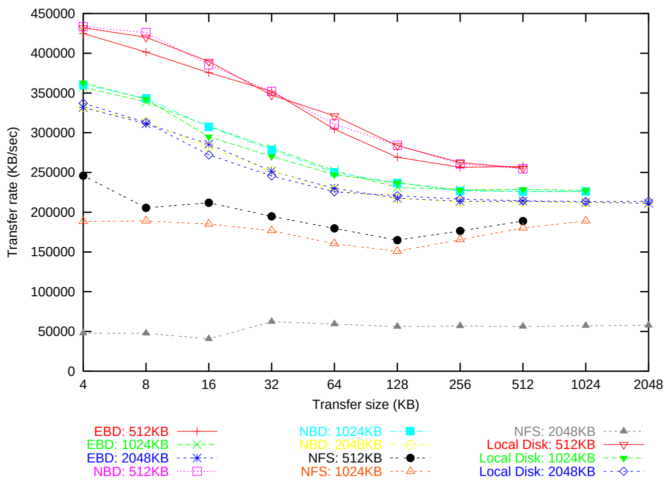


Figure 51: Iozone Rewrite Performance, 512KB-2048KB Data Size

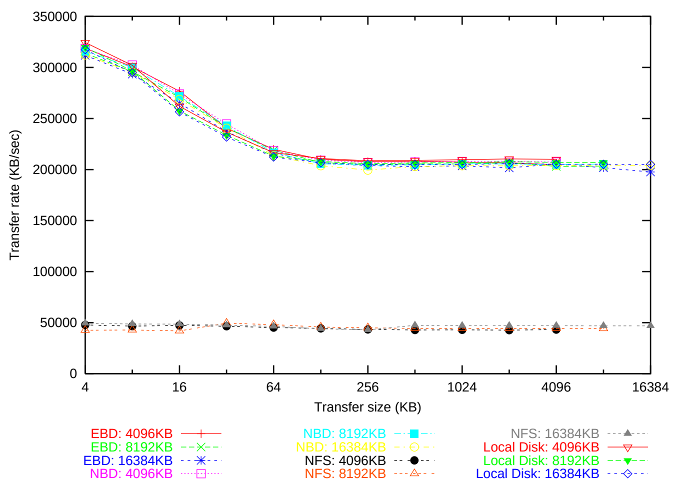


Figure 52: Iozone Rewrite Performance, 4096KB-16384KB Data Size

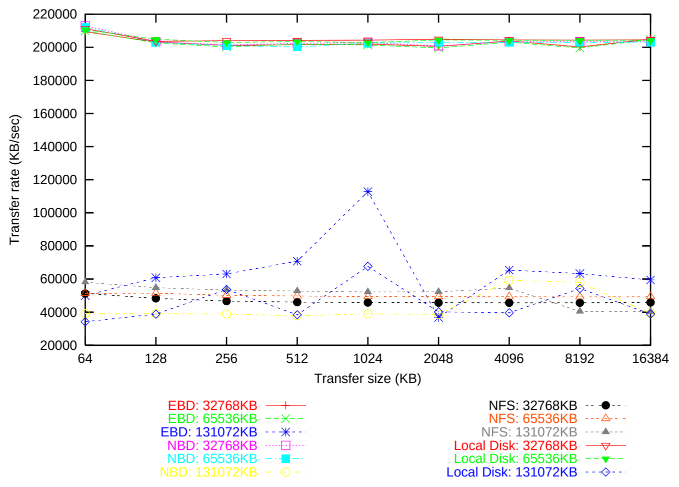


Figure 53: Iozone Rewrite Performance, 32768KB-131072KB Data Size

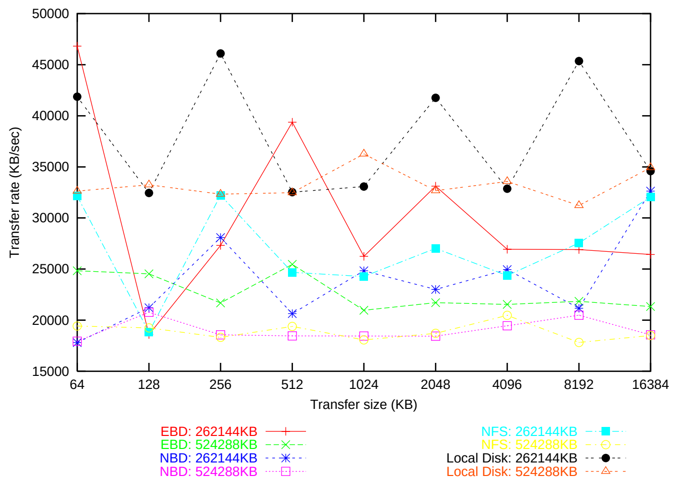


Figure 54: Iozone Rewrite Performance, 262144KB-524288KB Data Size

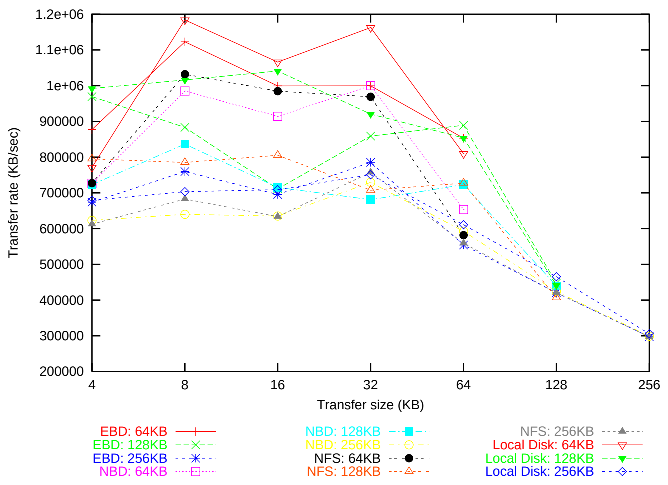


Figure 55: Iozone Random Read Performance, 64KB-256KB Data Size

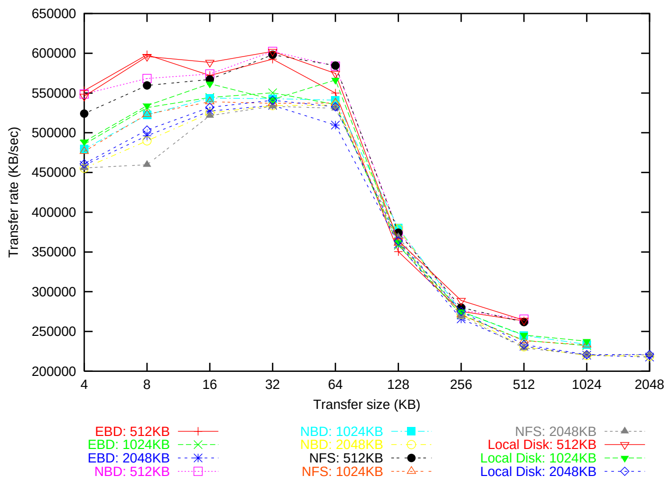


Figure 56: Iozone Random Read Performance, 512KB-2048KB Data Size

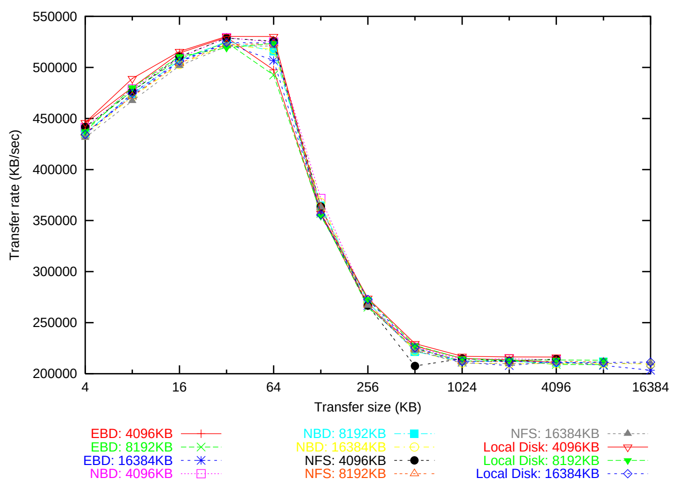


Figure 57: Iozone Random Read Performance, 4096KB-16384KB Data Size

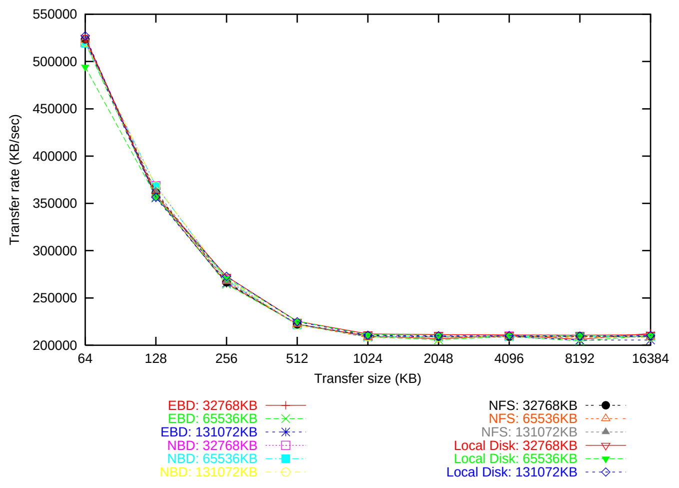


Figure 58: Iozone Random Read Performance, 32768KB-131072KB Data Size

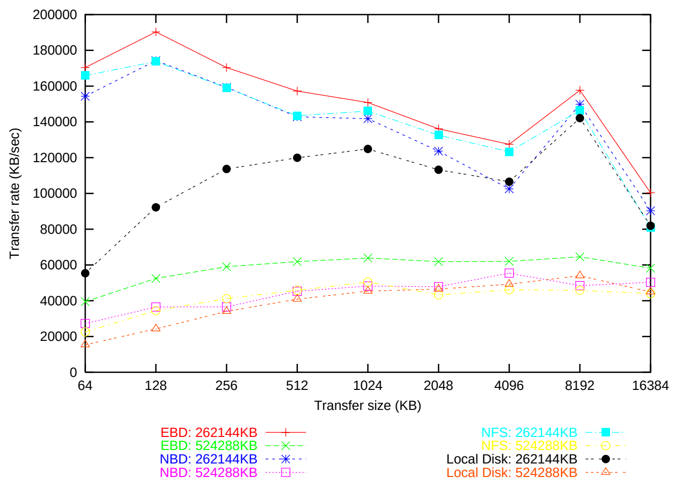


Figure 59: Iozone Random Read Performance, 262144KB-524288KB Data Size

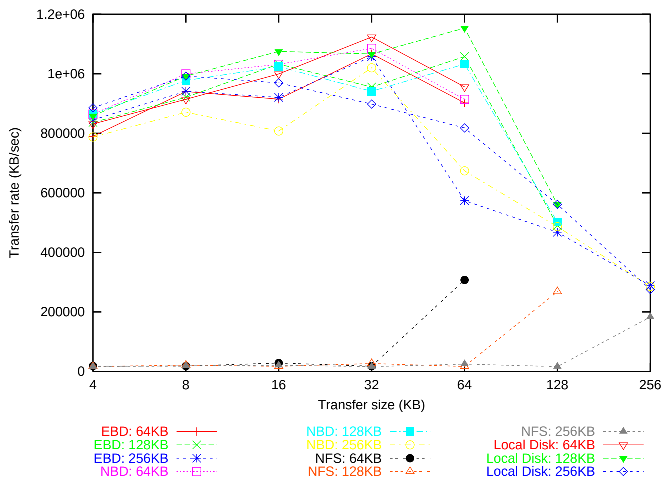


Figure 60: Iozone Random Write Performance, 64KB-256KB Data Size

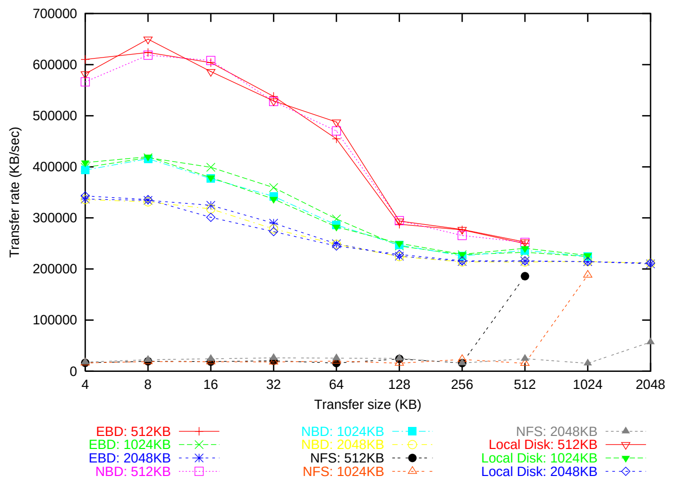


Figure 61: Iozone Random Write Performance, 512KB-2048KB Data Size

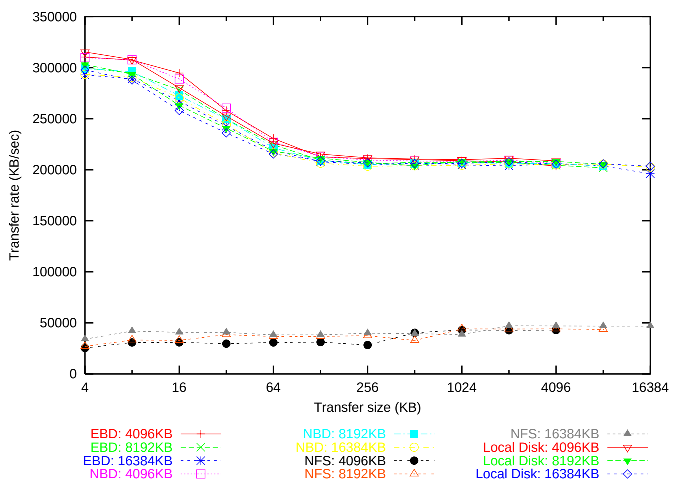


Figure 62: Iozone Random Write Performance, 4096KB-16384KB Data Size

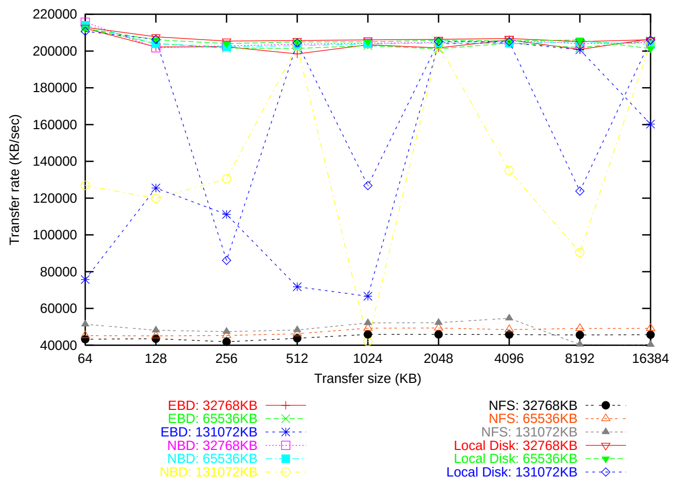


Figure 63: Iozone Random Write Performance, 32768KB-131072KB Data Size

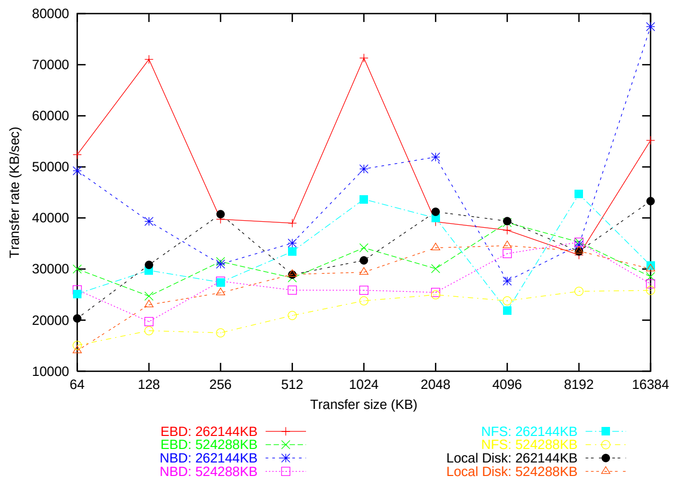


Figure 64: Iozone Random Write Performance, 262144KB-524288KB Data Size

B Additional Tiobench Performance Plots

This section contains the additional performance plots for tiobench.

B.1 1024-Byte Transfer Size

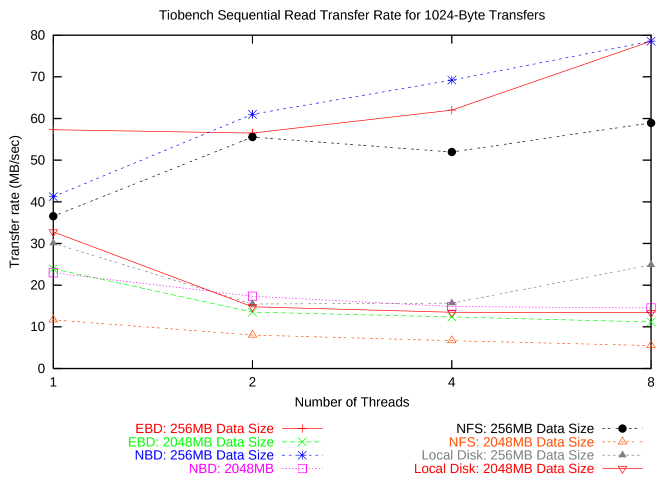


Figure 65: Tiobench Sequential Read Throughput

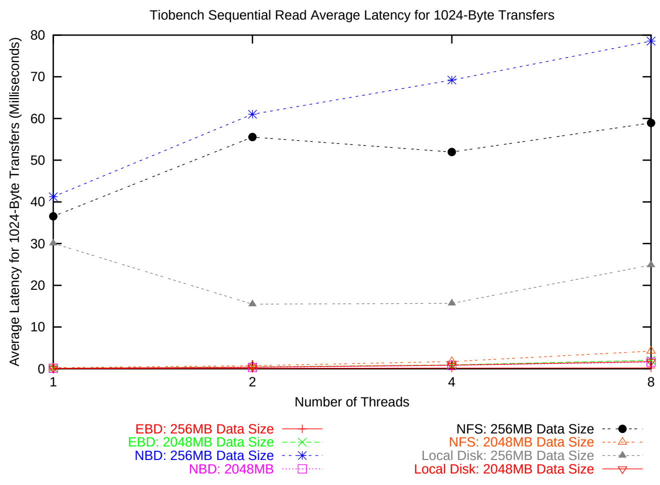


Figure 66: Tiobench Sequential Read Average Latency

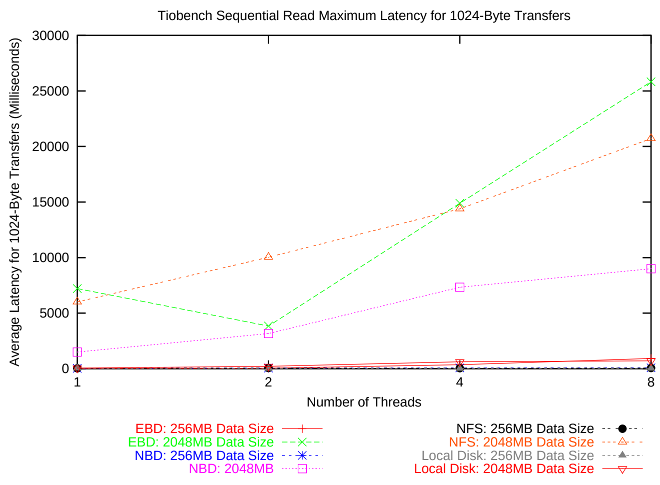


Figure 67: Tiobench Sequential Read Maximum Latency

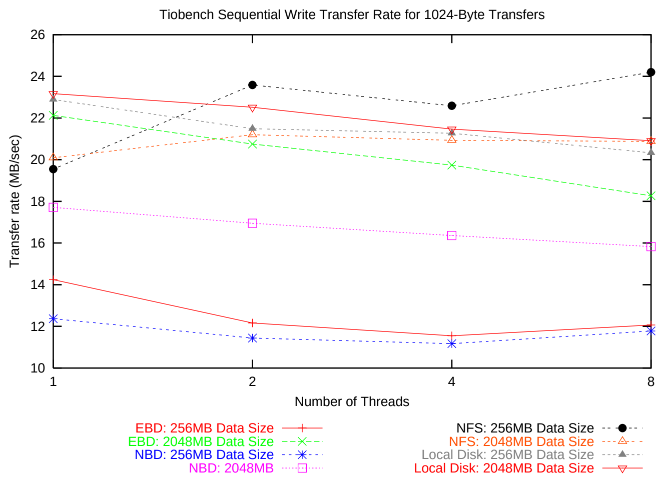


Figure 68: Tiobench Sequential Write Throughput

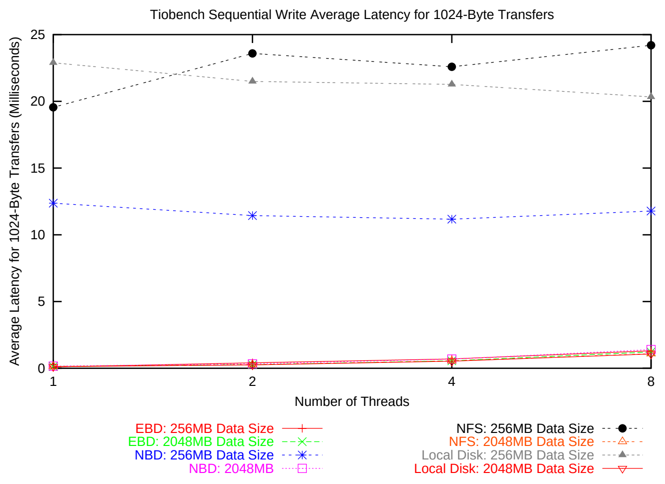


Figure 69: Tiobench Sequential Write Average Latency

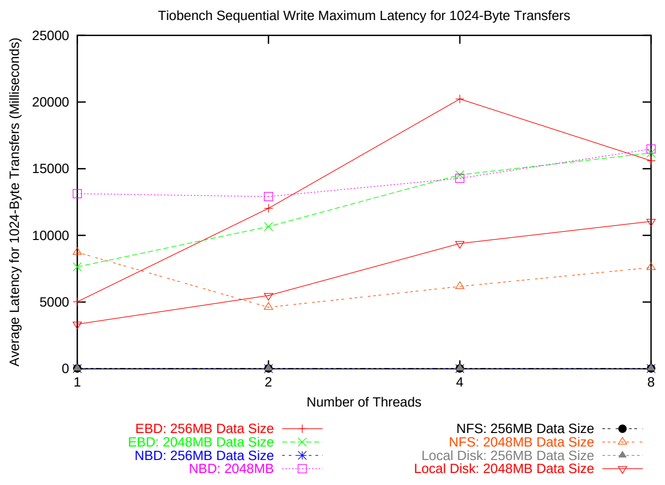


Figure 70: Tiobench Sequential Write Maximum Latency

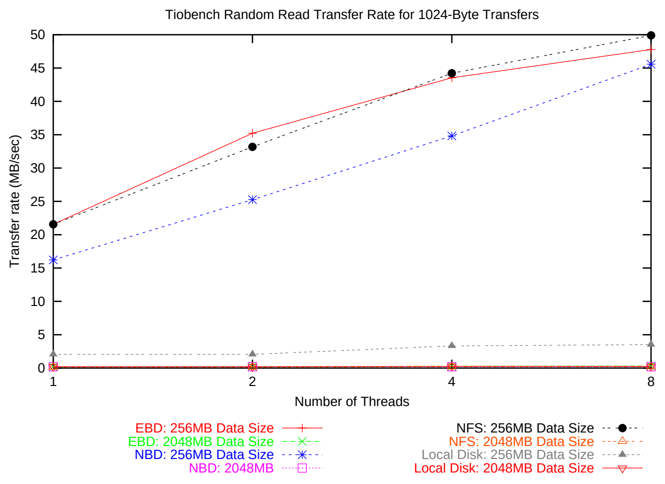


Figure 71: Tiobench Random Read Throughput

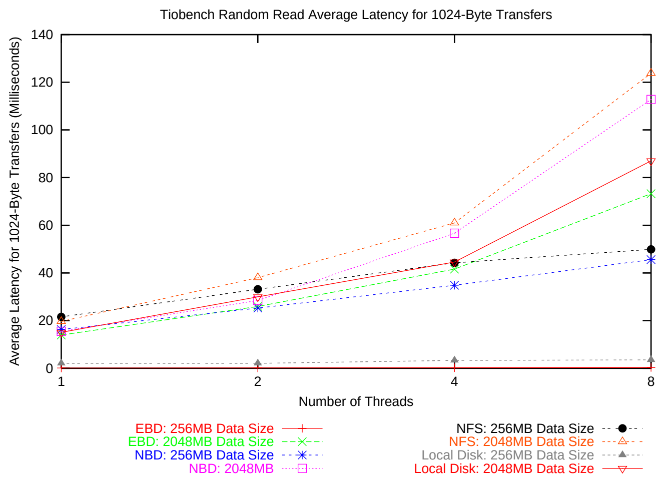


Figure 72: Tiobench Random Read Average Latency

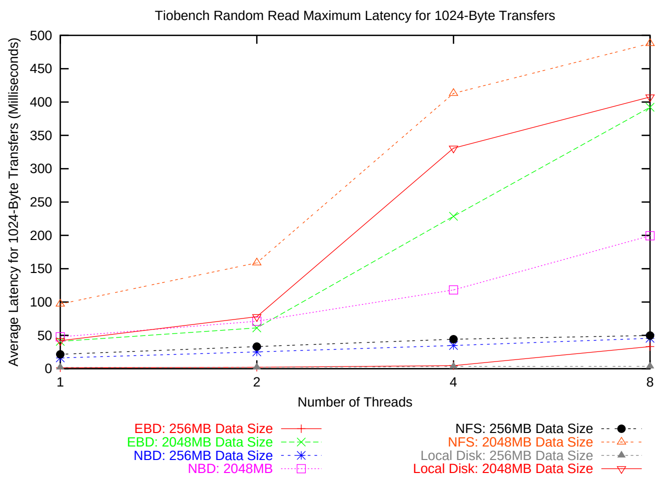


Figure 73: Tiobench Random Read Maximum Latency

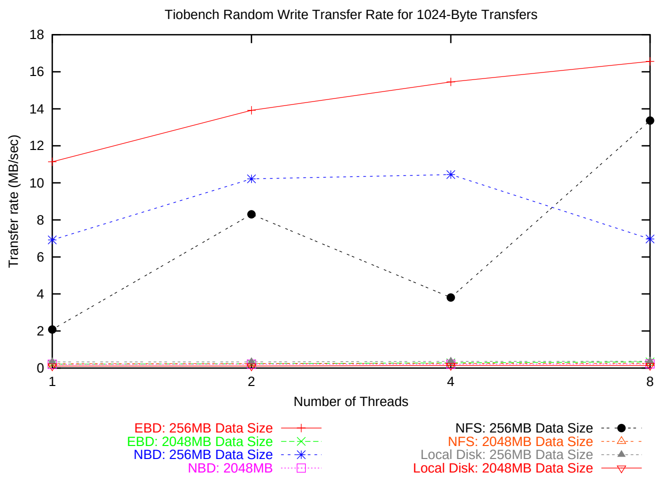


Figure 74: Tiobench Random Write Throughput

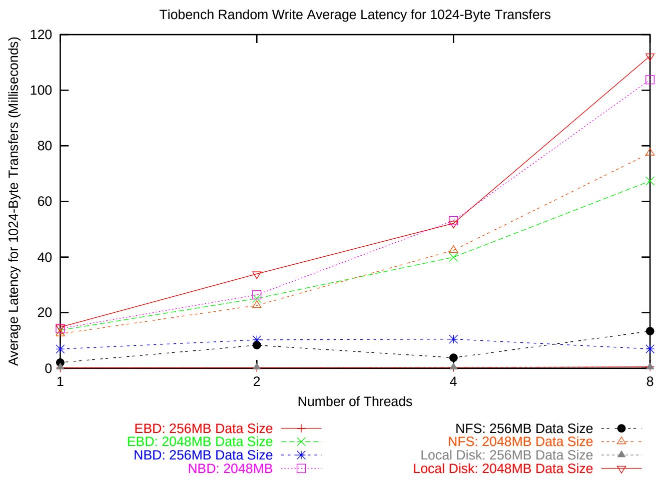


Figure 75: Tiobench Random Write Average Latency

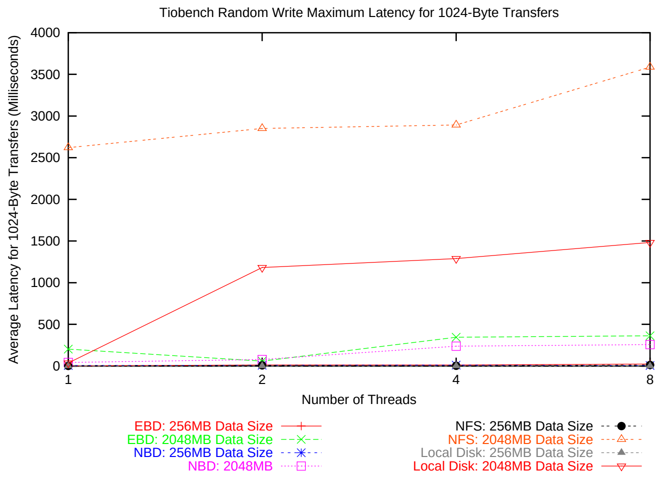


Figure 76: Tiobench Random Write Maximum Latency

B.2 2048-Byte Transfer Size

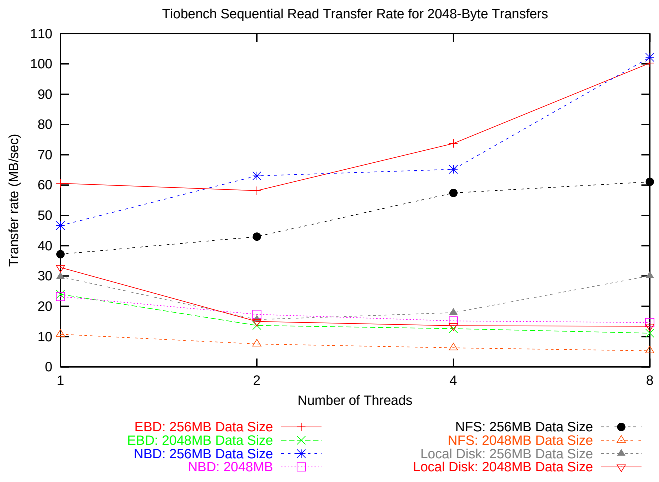


Figure 77: Tiobench Sequential Read Throughput

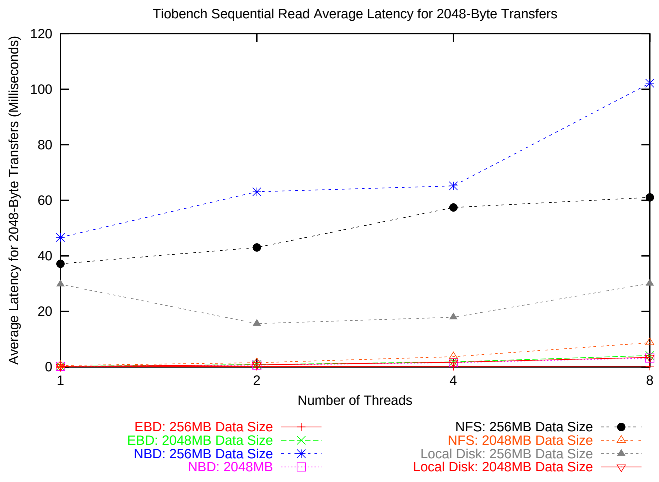


Figure 78: Tiobench Sequential Read Average Latency

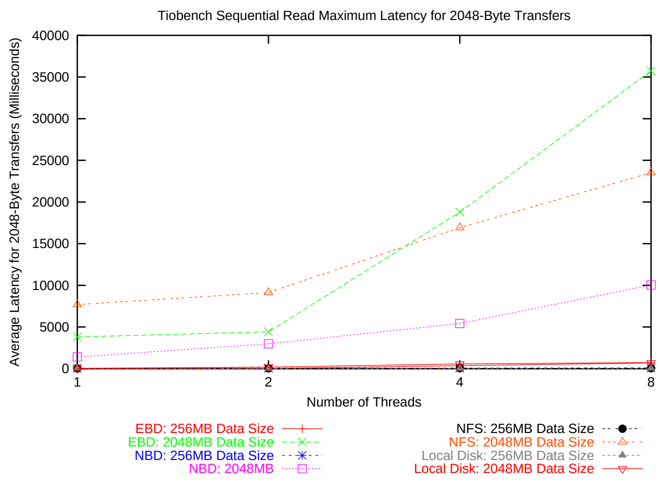


Figure 79: Tiobench Sequential Read Maximum Latency

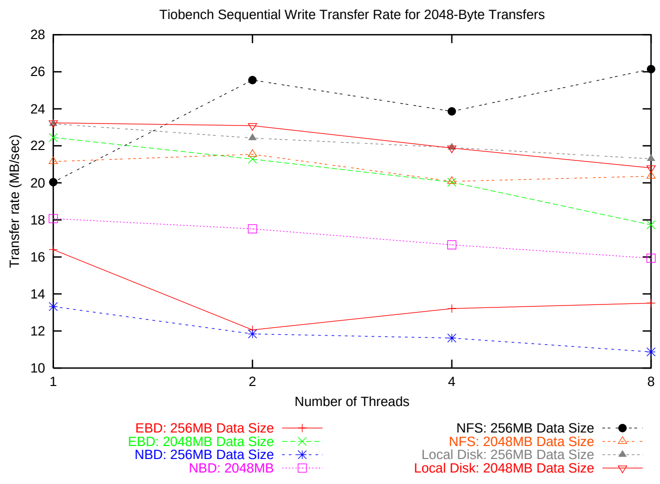


Figure 80: Tiobench Sequential Write Throughput

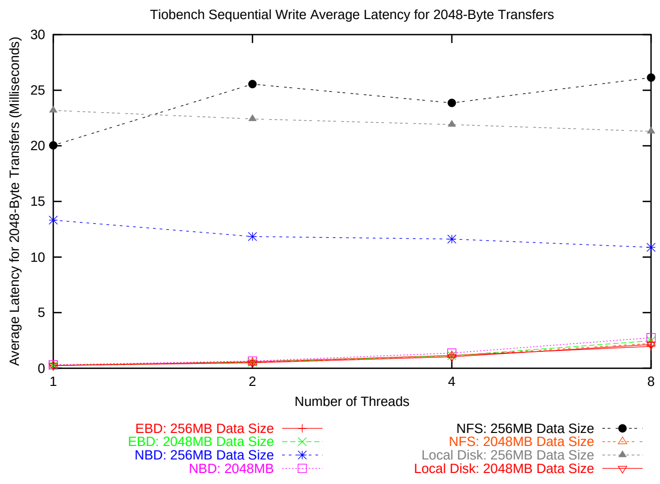


Figure 81: Tiobench Sequential Write Average Latency

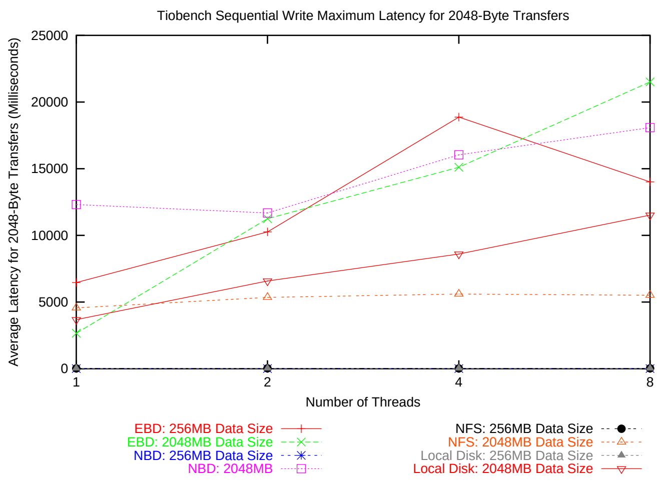


Figure 82: Tiobench Sequential Write Maximum Latency

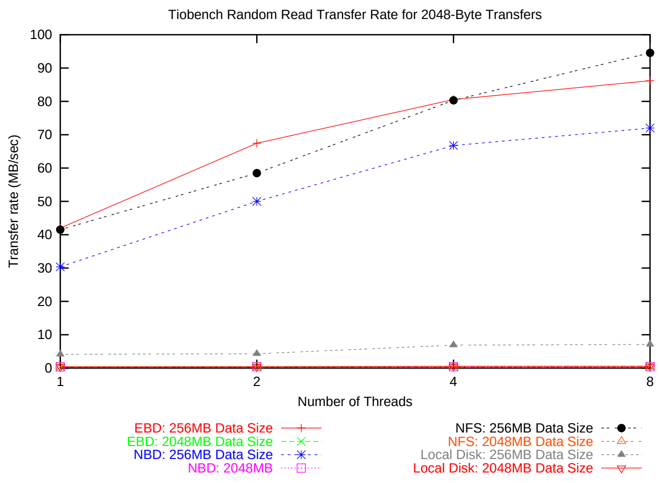


Figure 83: Tiobench Random Read Throughput

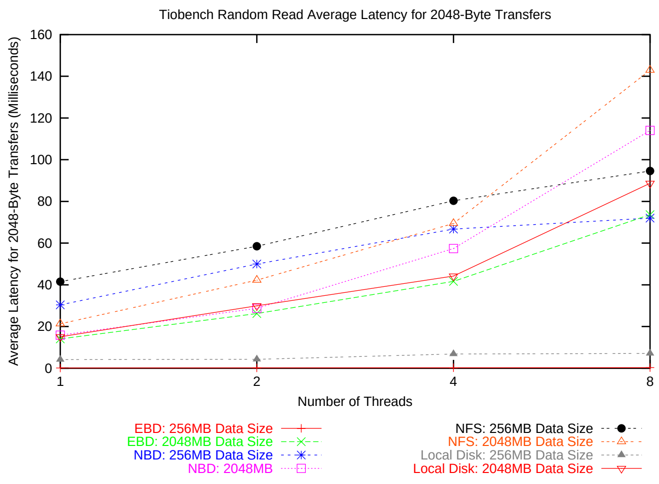


Figure 84: Tiobench Random Read Average Latency

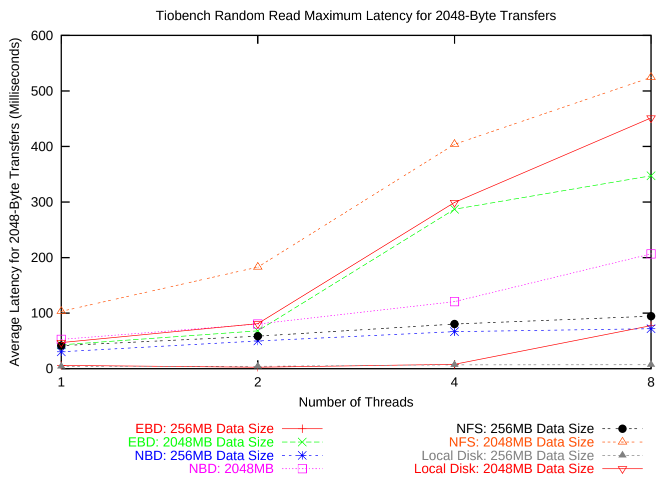


Figure 85: Tiobench Random Read Maximum Latency

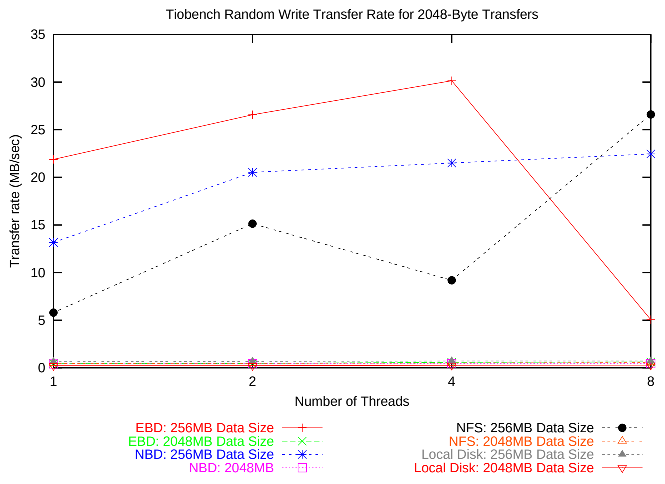


Figure 86: Tiobench Random Write Throughput

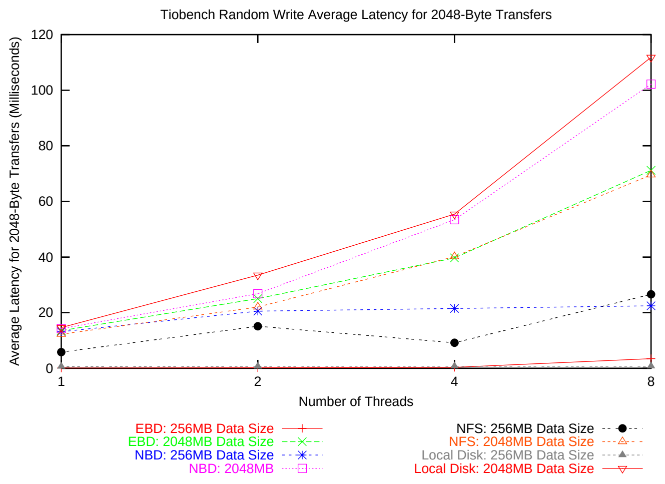


Figure 87: Tiobench Random Write Average Latency

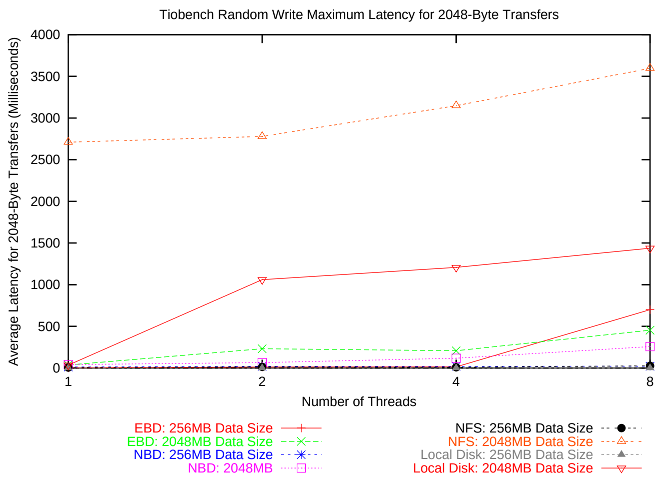


Figure 88: Tiobench Random Write Maximum Latency

B.3 4096-Byte Transfer Size

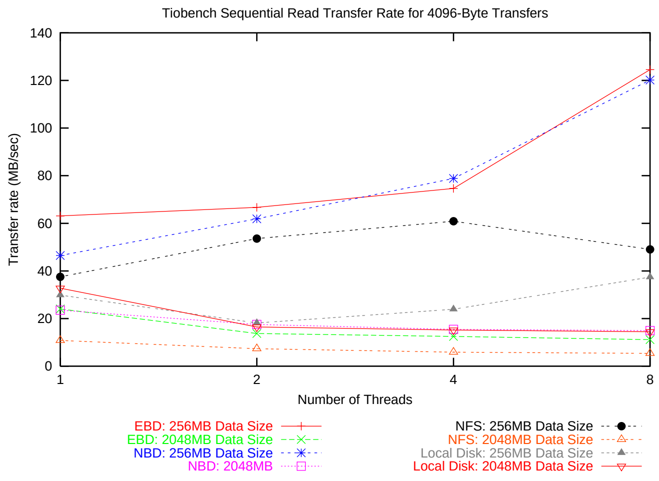


Figure 89: Tiobench Sequential Read Throughput

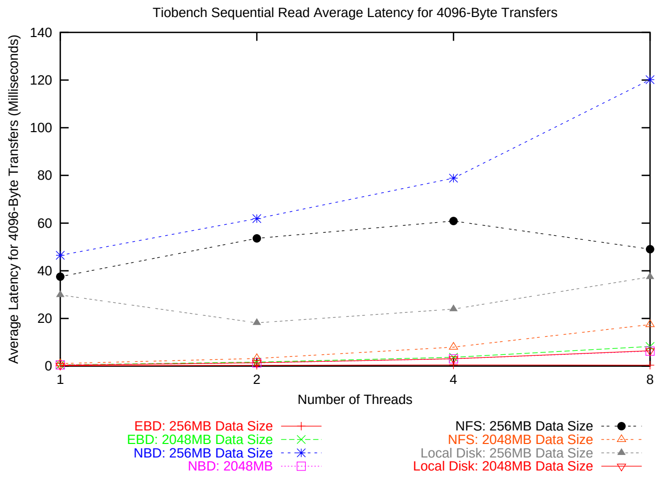


Figure 90: Tiobench Sequential Read Average Latency

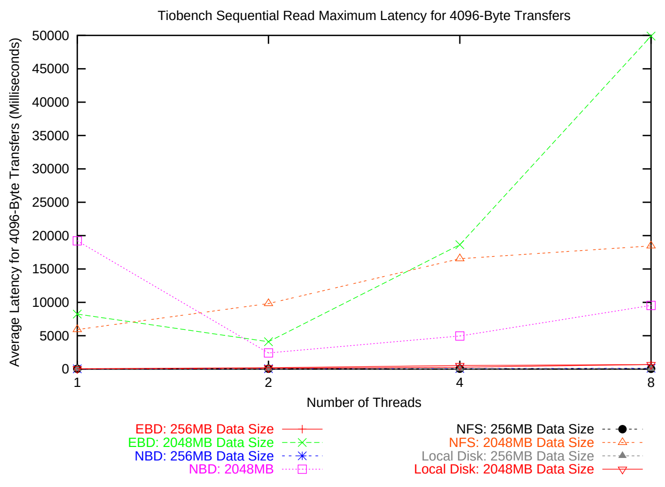


Figure 91: Tiobench Sequential Read Maximum Latency

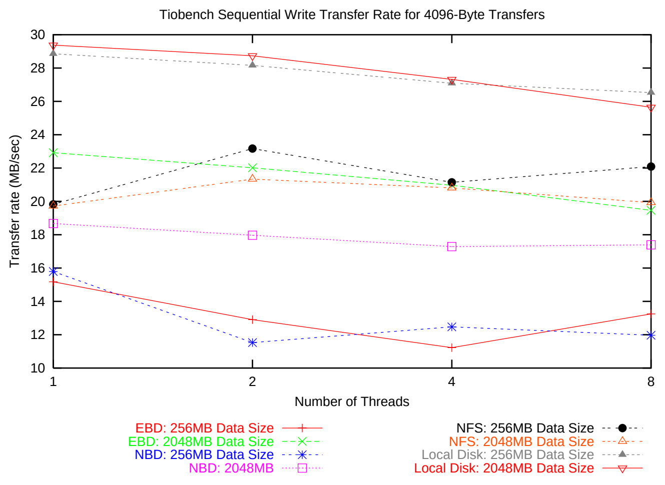


Figure 92: Tiobench Sequential Write Throughput

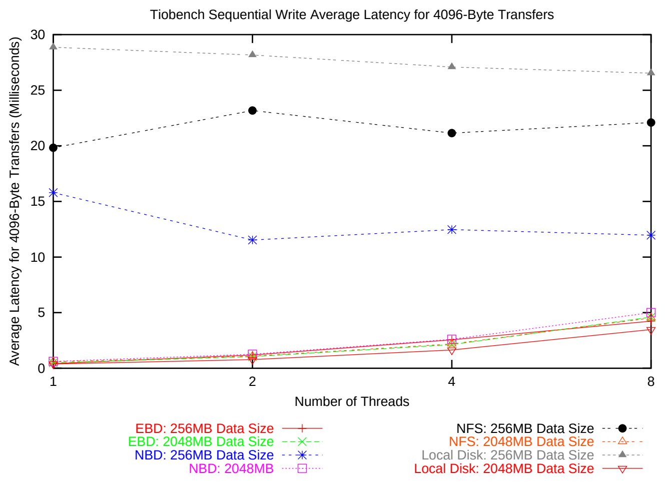


Figure 93: Tiobench Sequential Write Average Latency

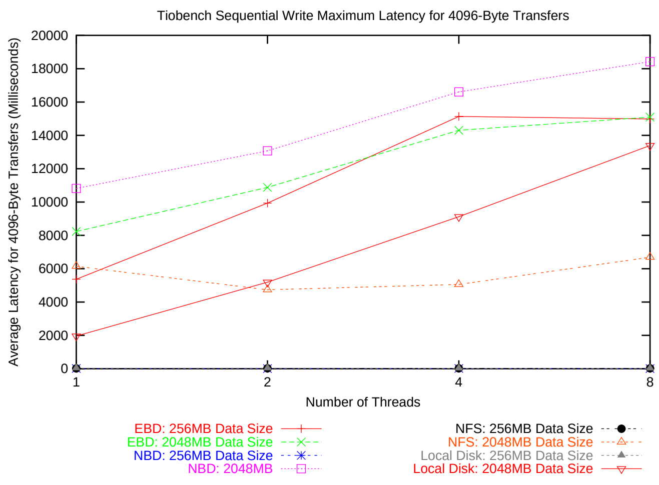


Figure 94: Tiobench Sequential Write Maximum Latency

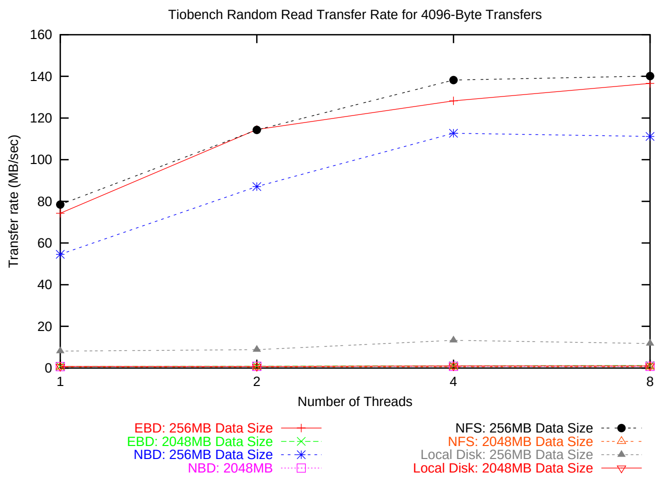


Figure 95: Tiobench Random Read Throughput

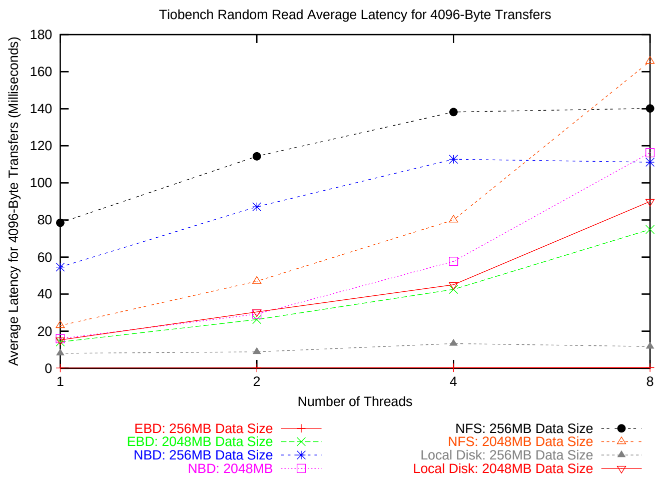


Figure 96: Tiobench Random Read Average Latency

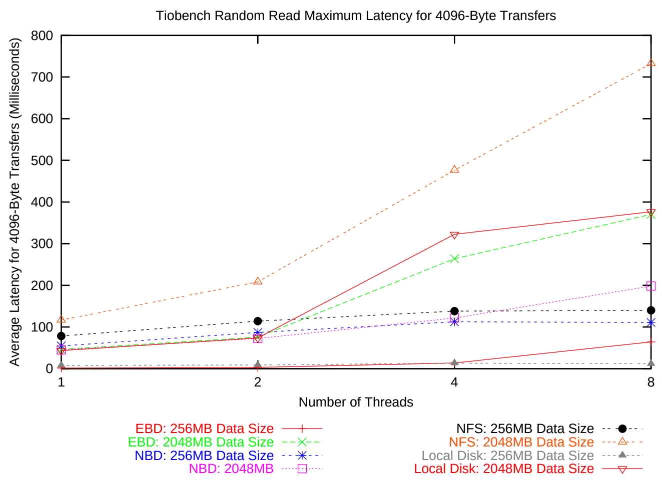


Figure 97: Tiobench Random Read Maximum Latency

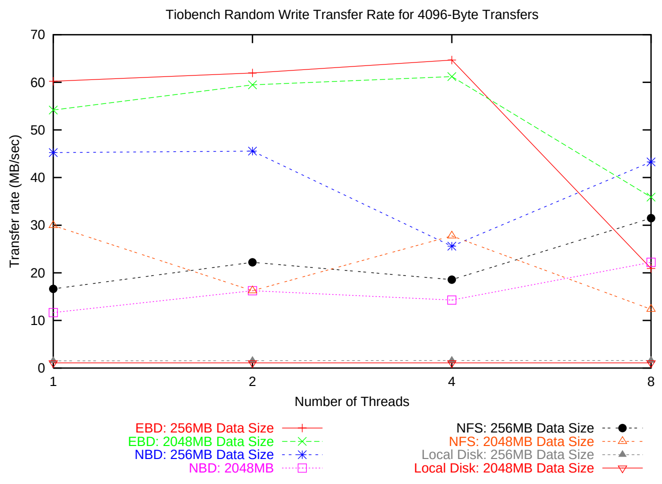


Figure 98: Tiobench Random Write Throughput

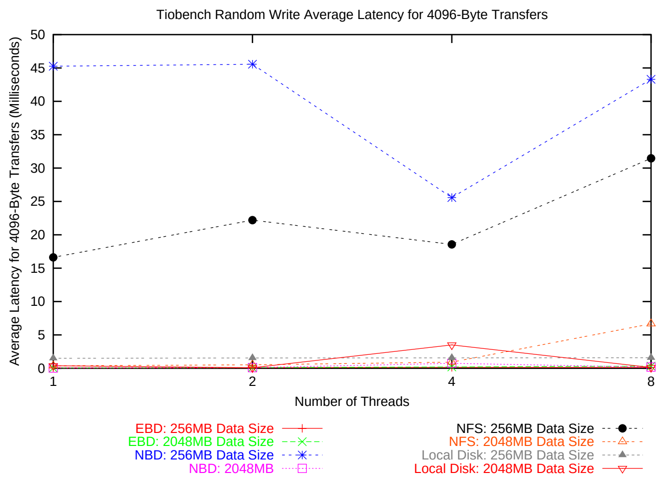


Figure 99: Tiobench Random Write Average Latency

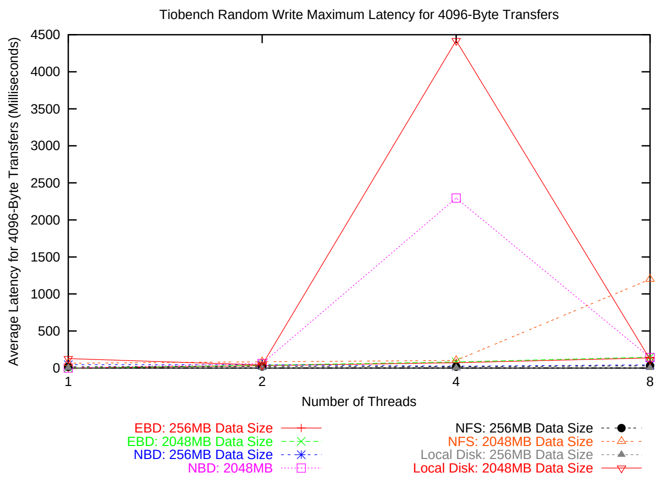
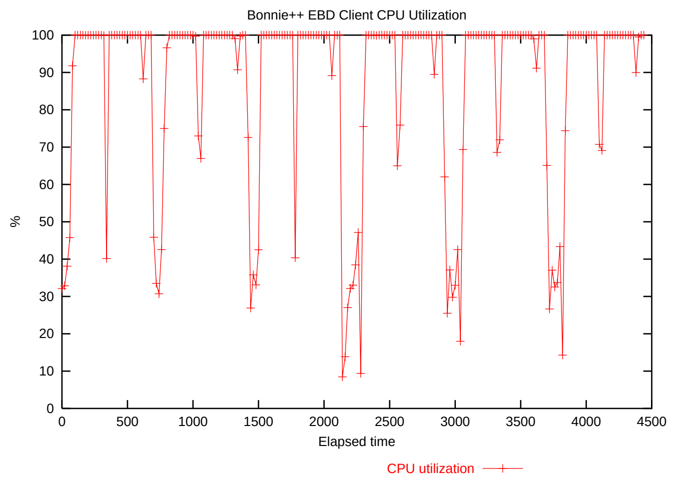


Figure 100: Tiobench Random Write Maximum Latency

C System Analysis Report Plots

The figures in this section plot the data collected during the performance evaluation runs using `sar`: there are client and server graphs for all cases except local disk, which has only a single plot. Only CPU and network data are plotted for the client while for the server, the plots include block I/O access rates. In the local disk case, only the CPU utilization and block I/O rates are shown.

C.1 Bonnie++



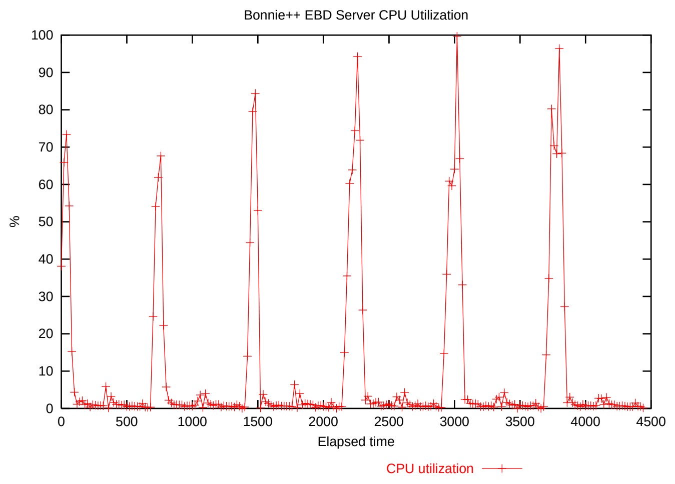
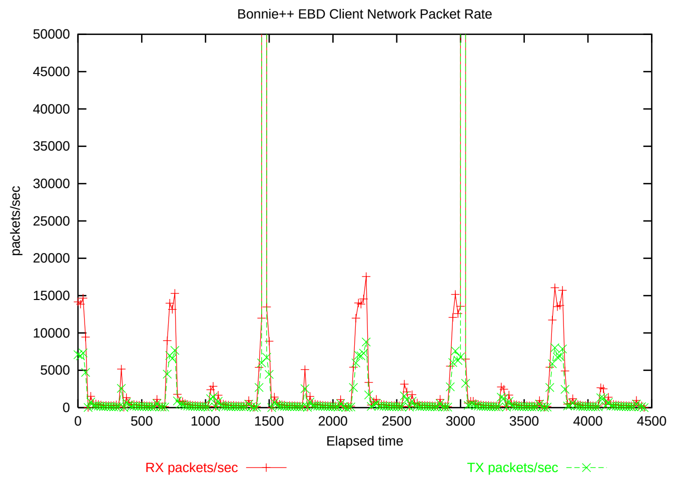


Figure 101: EBD CPU SAR Data



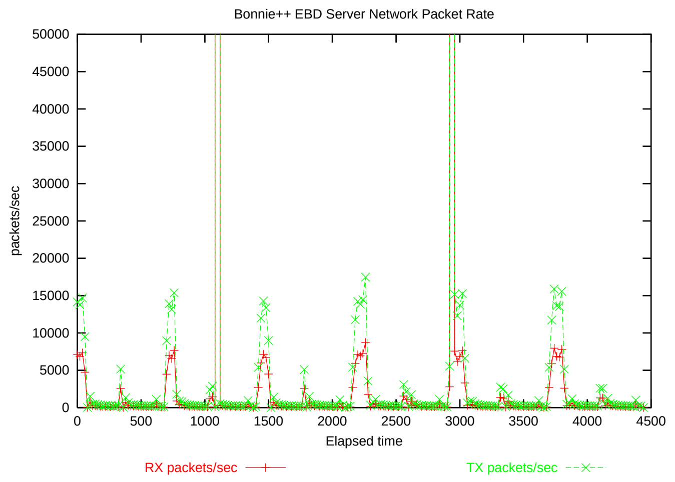


Figure 102: EBD Network Packet SAR Data

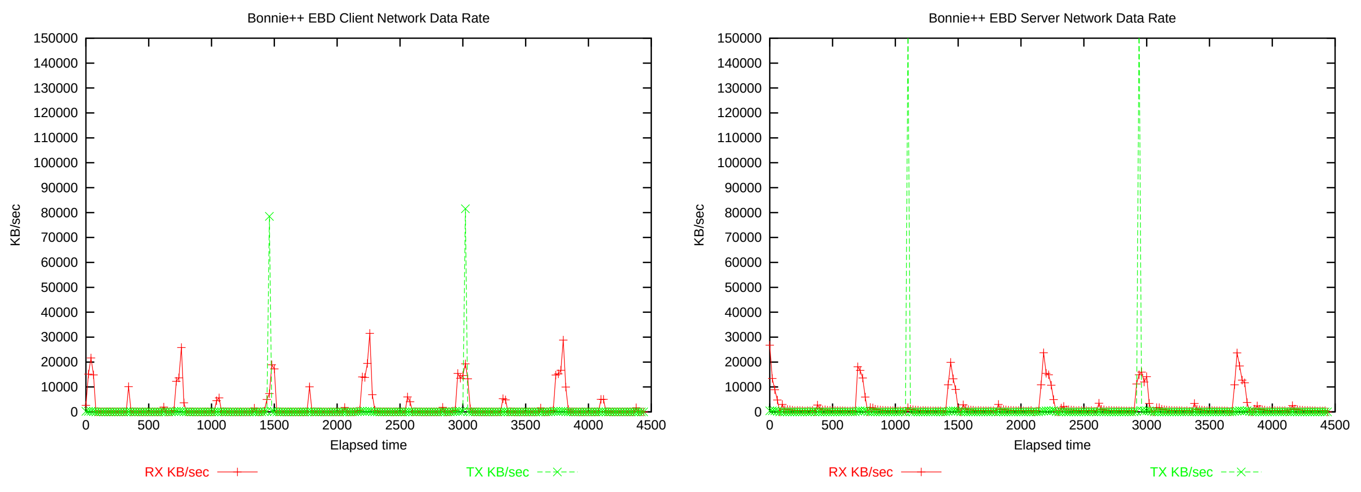


Figure 103: EBD Network Bytes Transferred SAR Data

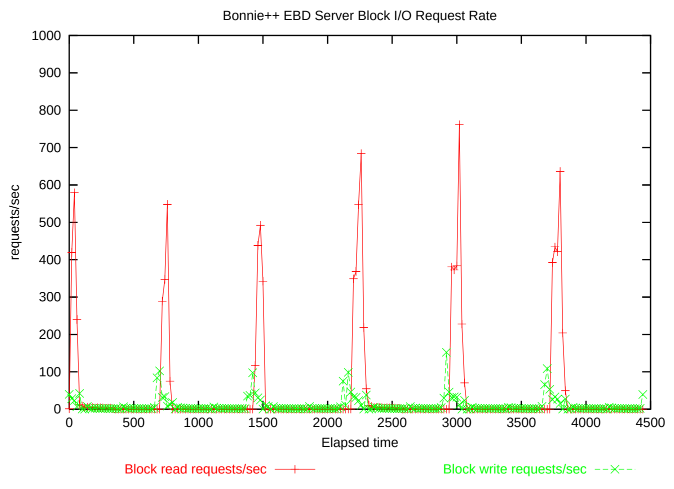


Figure 104: EBD Disk Request Rate SAR Data

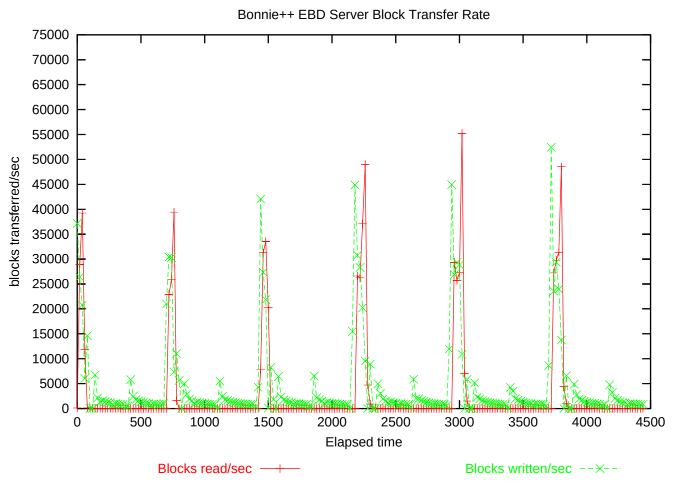
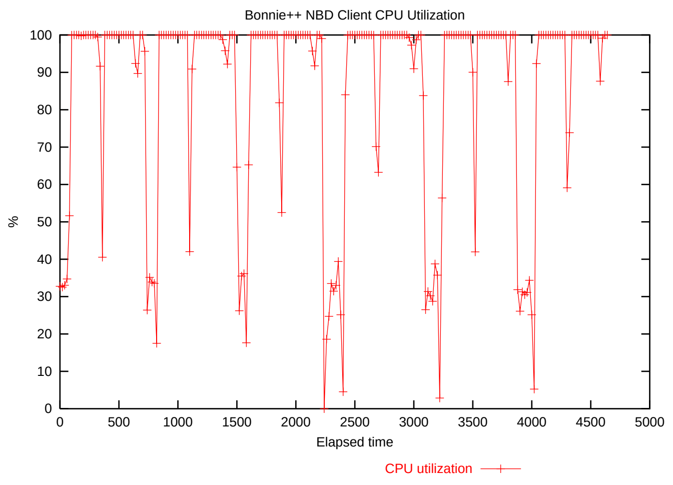


Figure 105: EBD Disk Blocks Transferred SAR Data



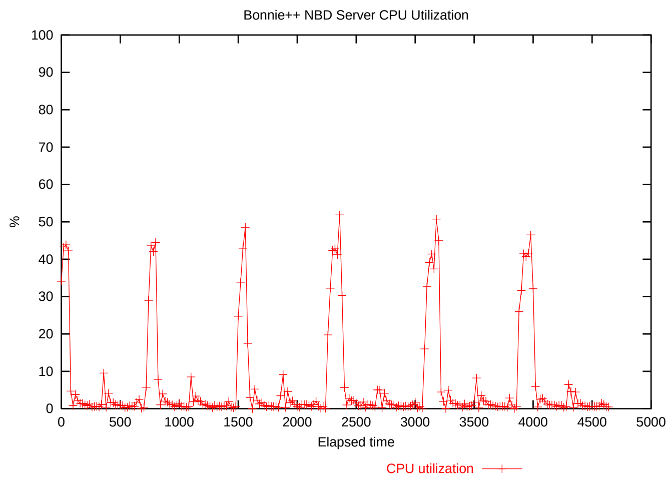
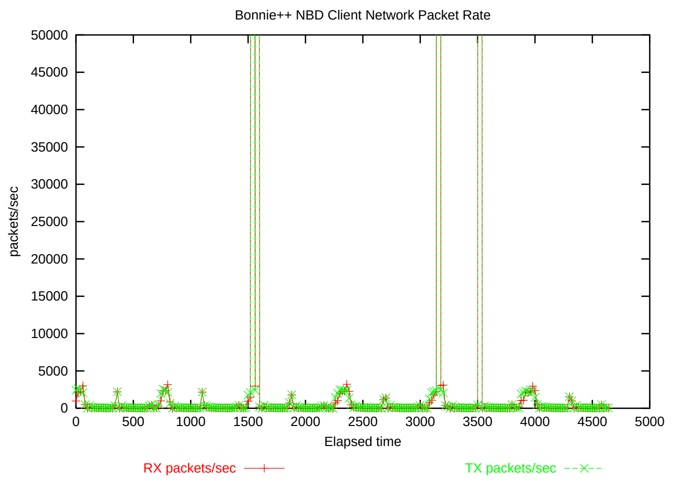


Figure 106: NBD CPU SAR Data



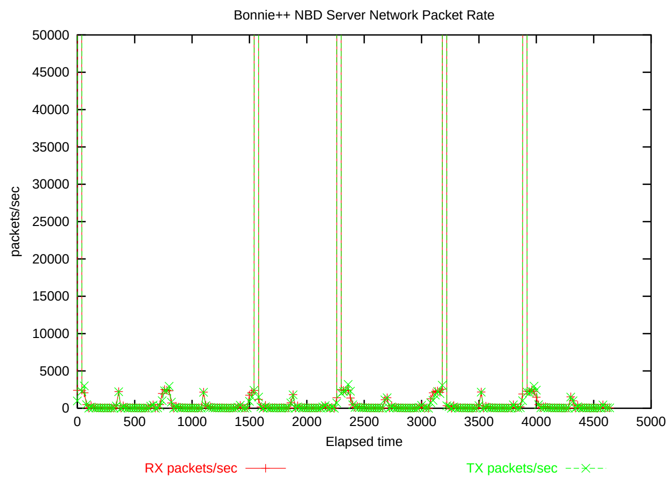
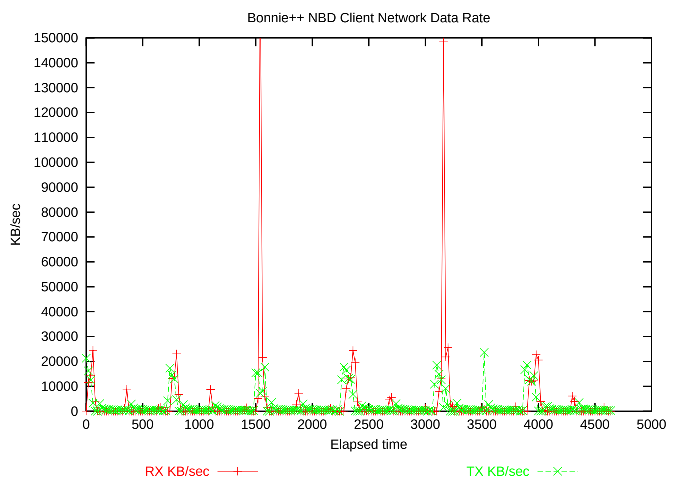


Figure 107: NBD Network Packet SAR Data



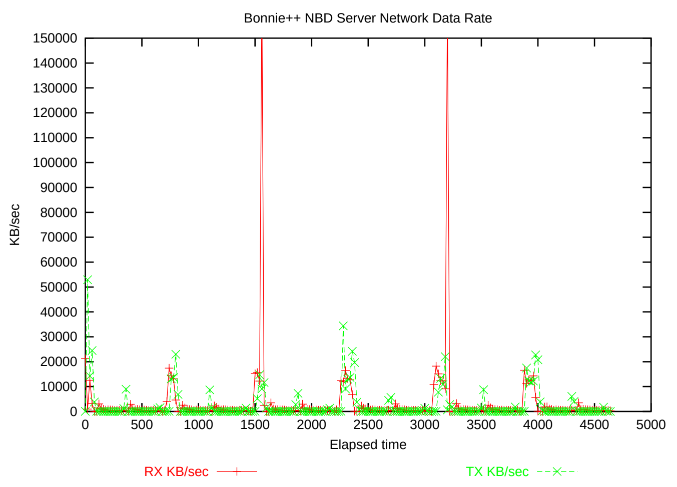


Figure 108: NBD Network Byte Count SAR Data

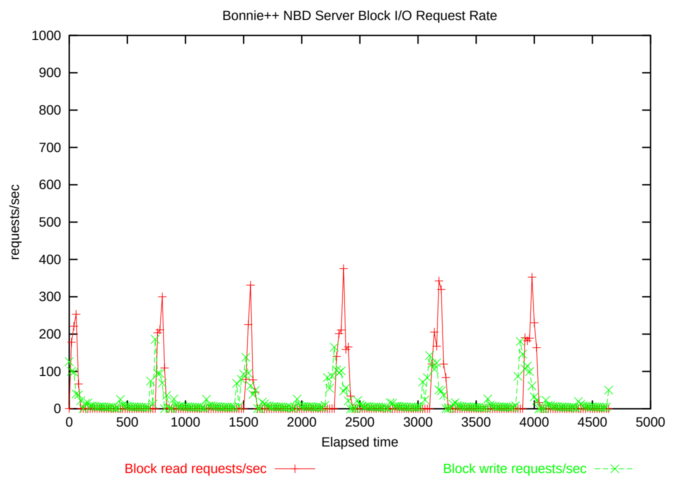


Figure 109: NBD Disk Request Rate SAR Data

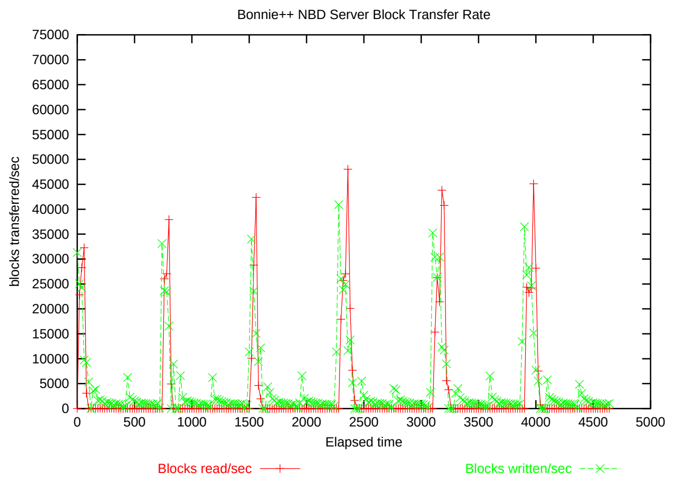
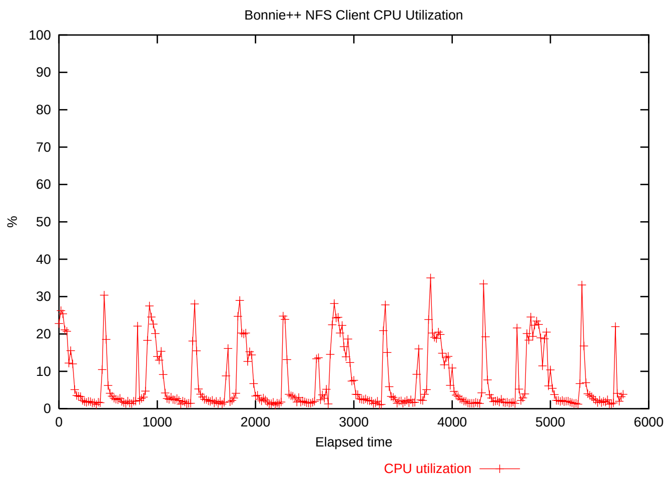


Figure 110: NBD Disk Blocks Transferred SAR Data



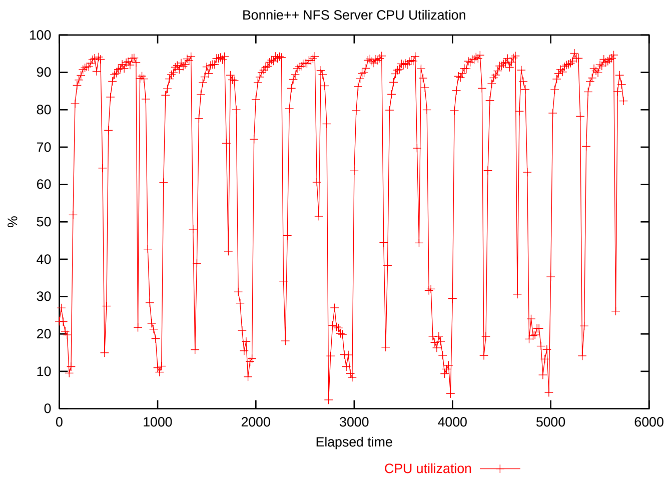
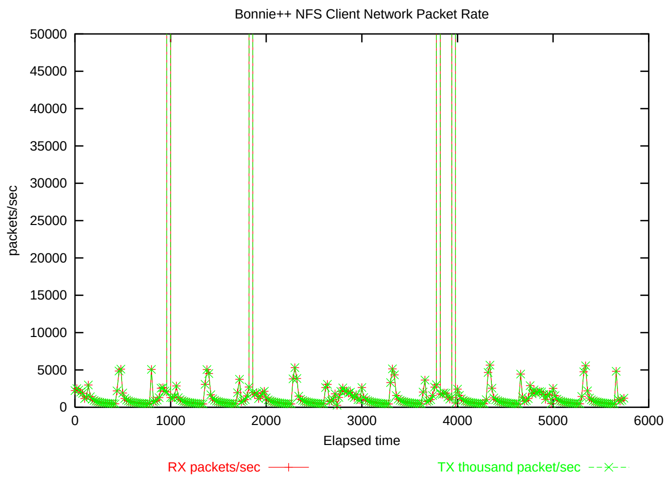


Figure 111: NFS CPU SAR Data



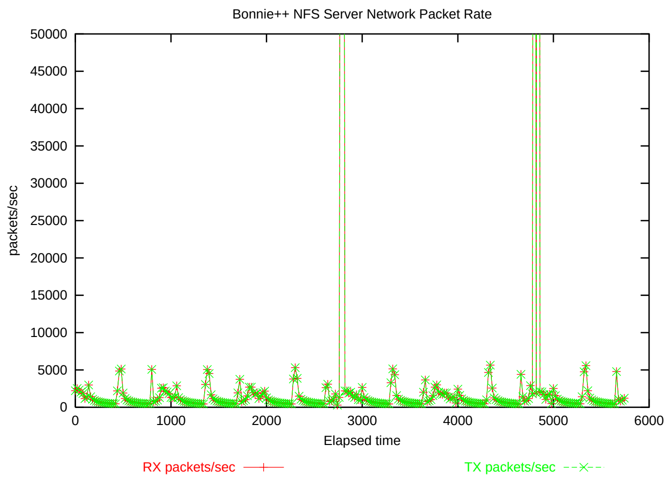
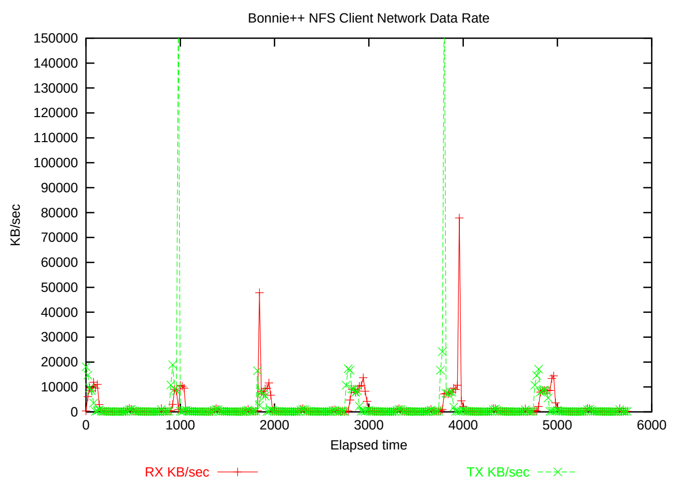


Figure 112: NFS Network Packet SAR Data



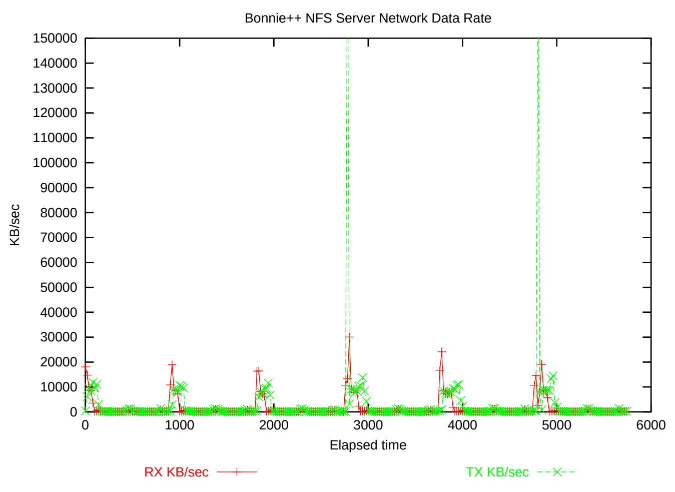


Figure 113: NFS Network Byte Count SAR Data

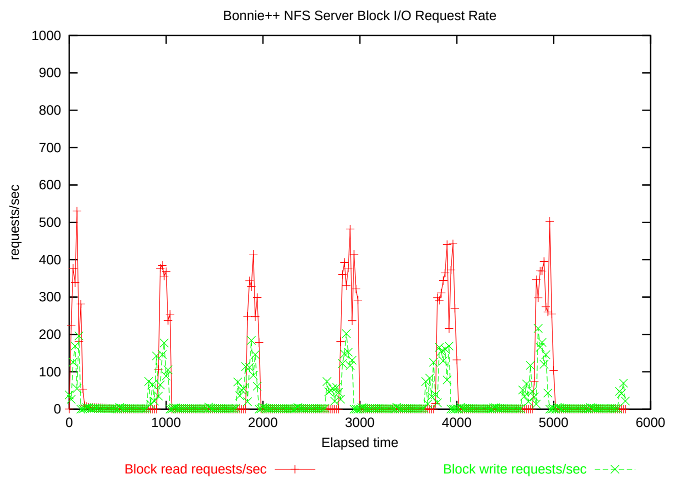


Figure 114: NFS Disk Request Rate SAR Data

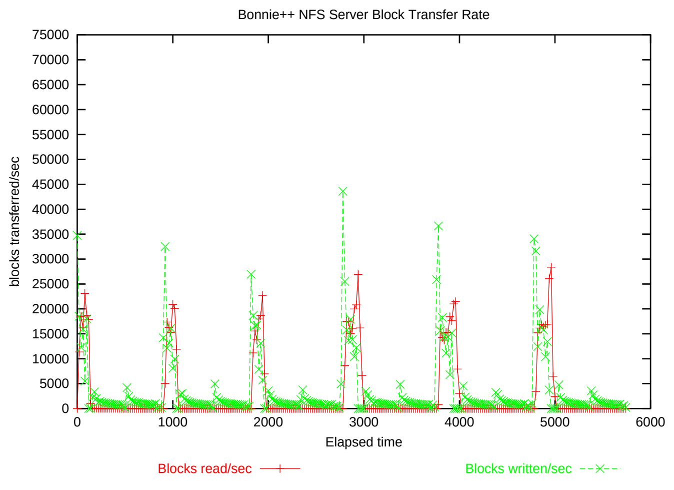


Figure 115: NFS Disk Blocks Transferred SAR Data

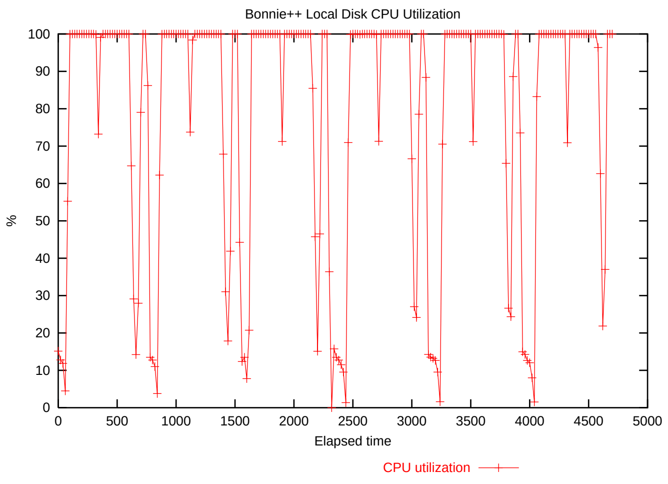


Figure 116: Local Disk CPU SAR Data

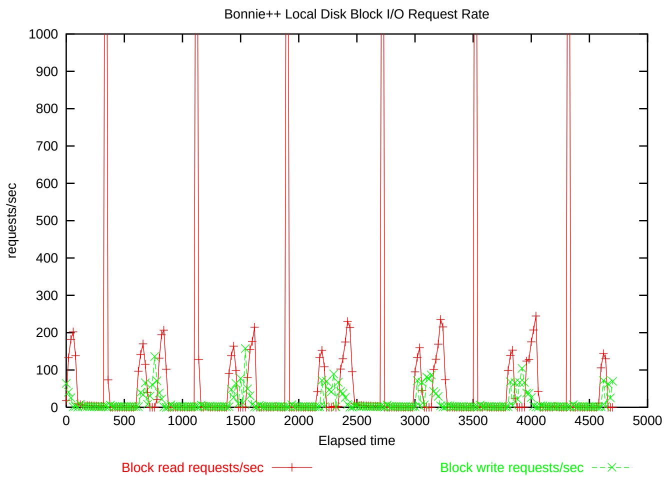


Figure 117: Local Disk Request Rate SAR Data

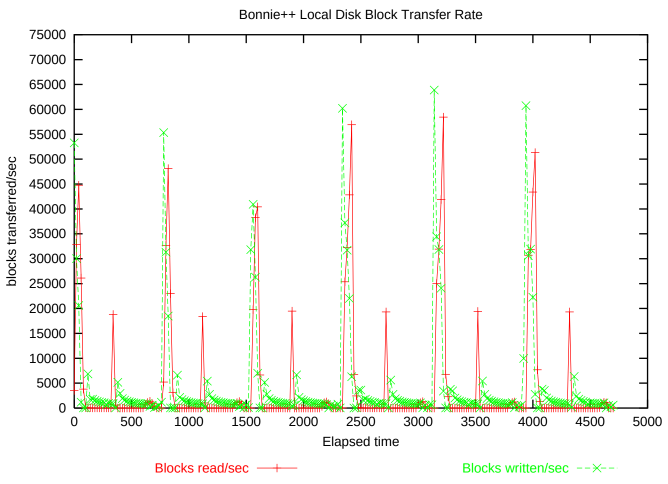
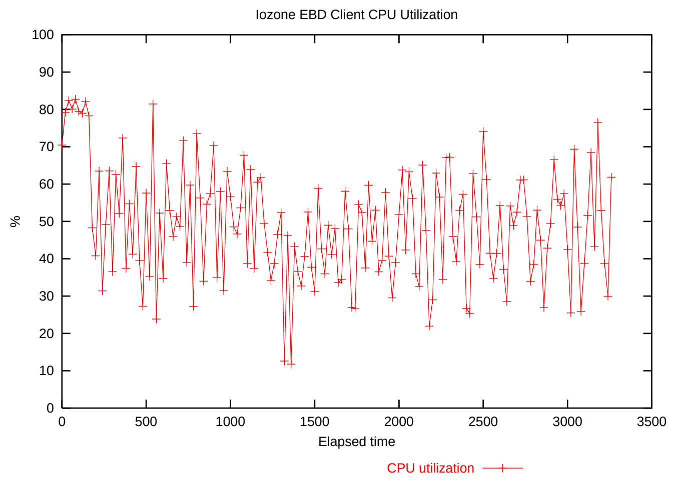


Figure 118: Local Disk Blocks Transferred SAR Data

C.2 Iozone



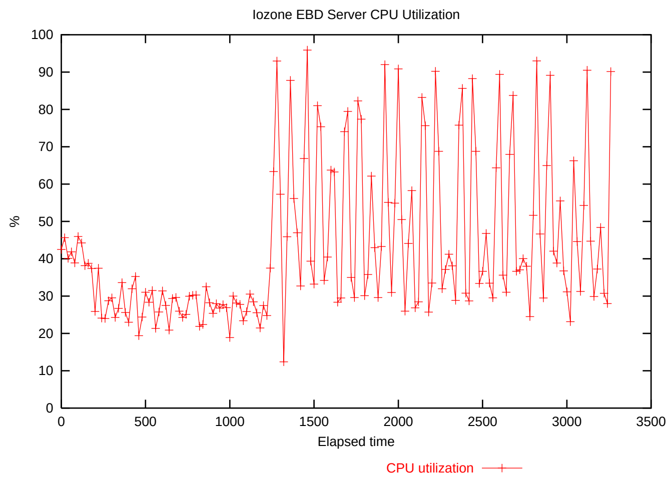
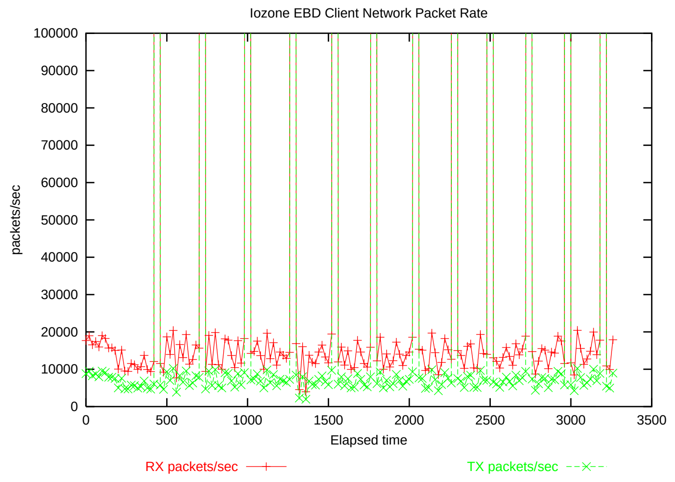


Figure 119: EBD CPU SAR Data



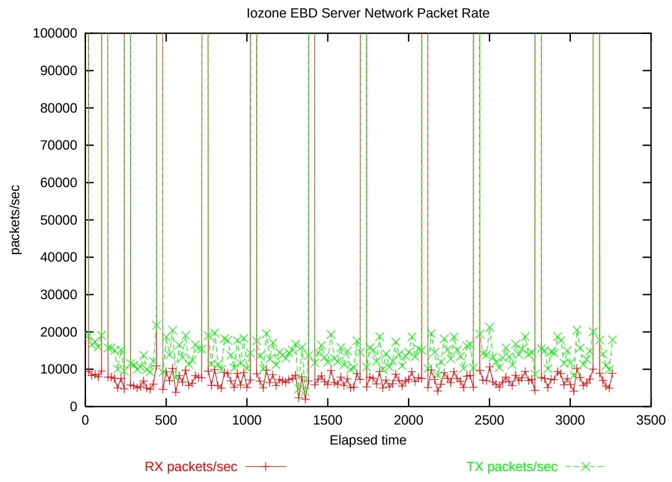
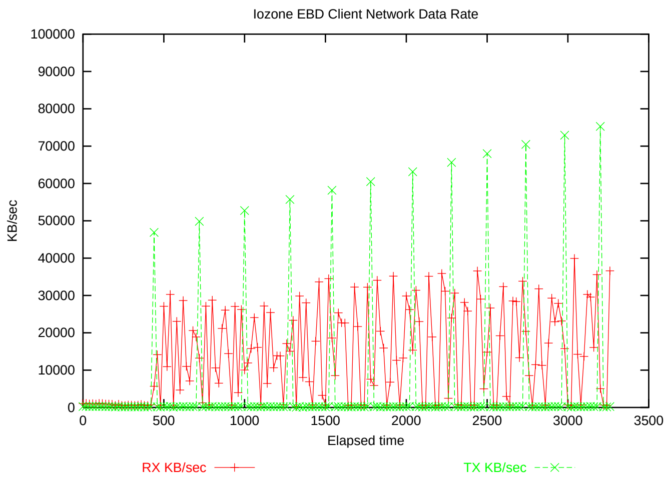


Figure 120: EBD Network Packet SAR Data



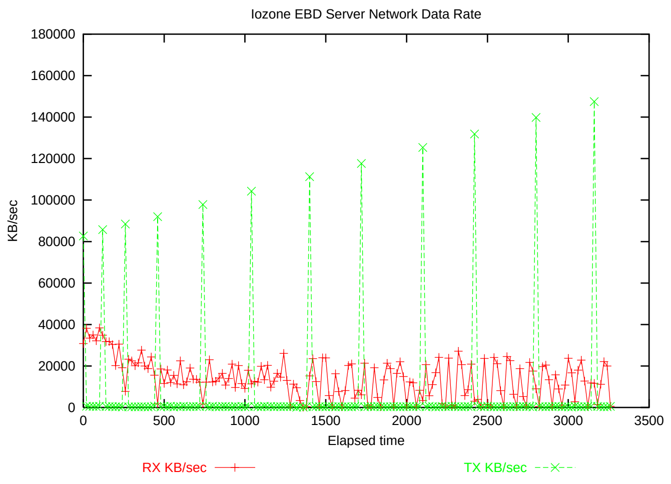


Figure 121: EBD Network Bytes Transferred SAR Data

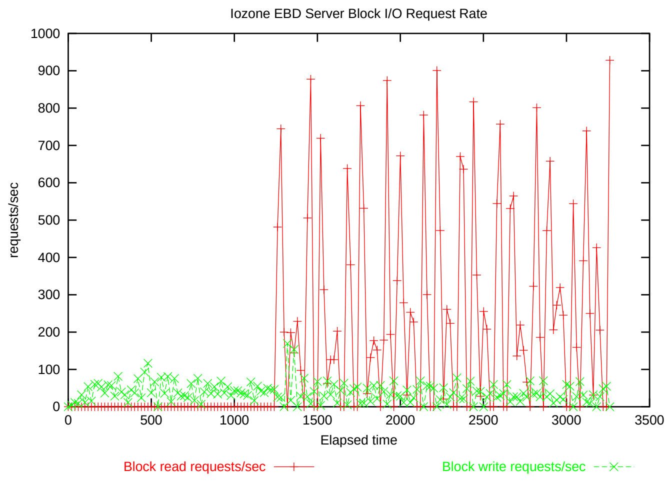


Figure 122: EBD Disk Request Rate SAR Data

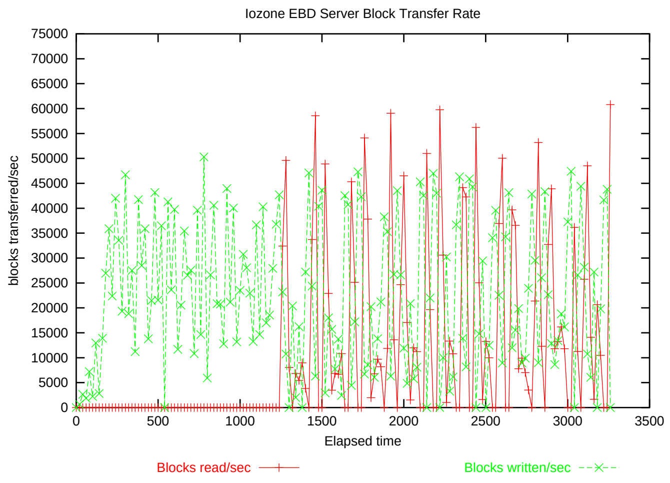
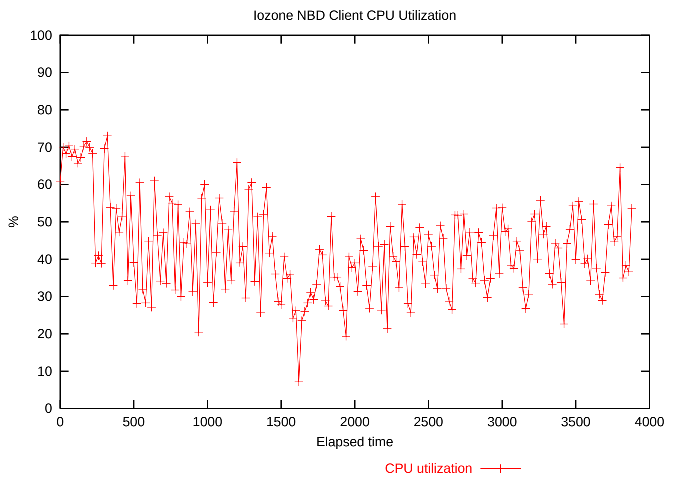


Figure 123: EBD Disk Blocks Transferred SAR Data



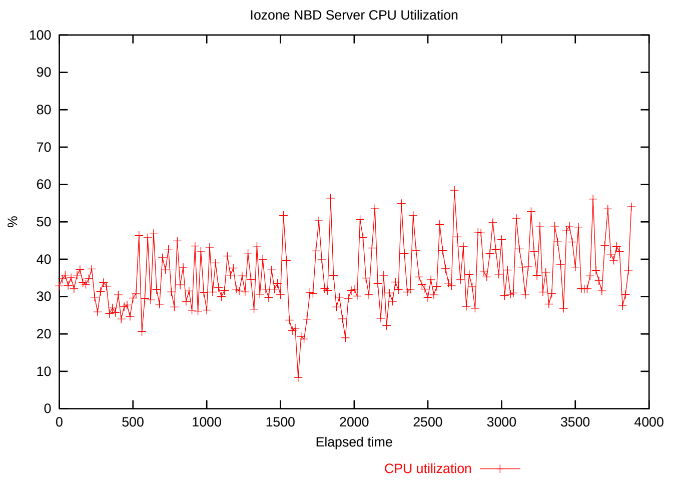
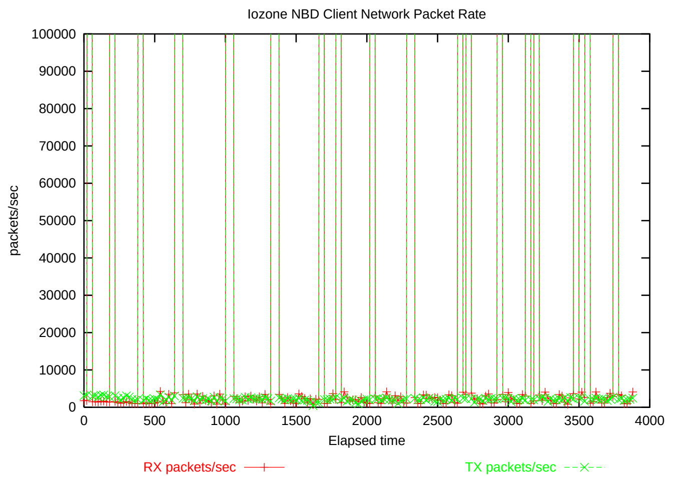


Figure 124: NBD CPU SAR Data



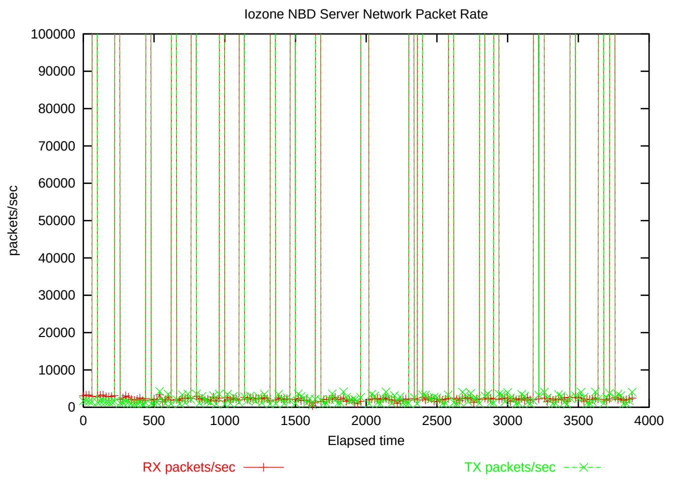
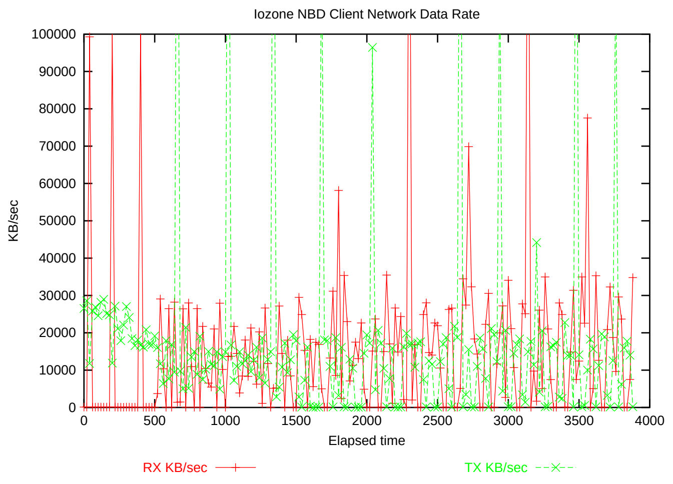


Figure 125: NBD Network Packet SAR Data



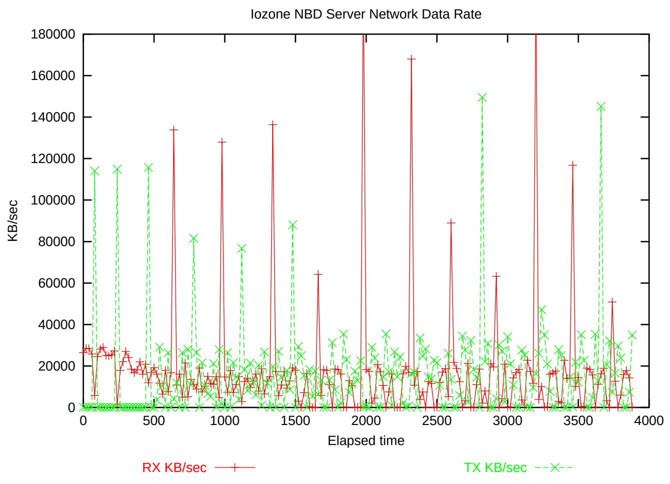


Figure 126: NBD Network Byte Count SAR Data

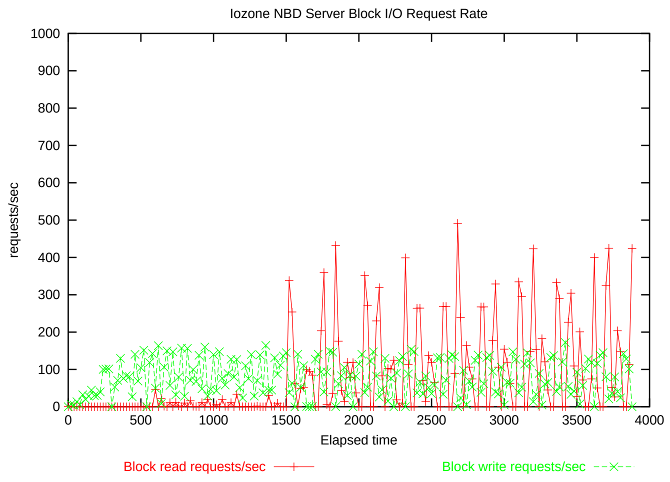


Figure 127: NBD Disk Request Rate SAR Data

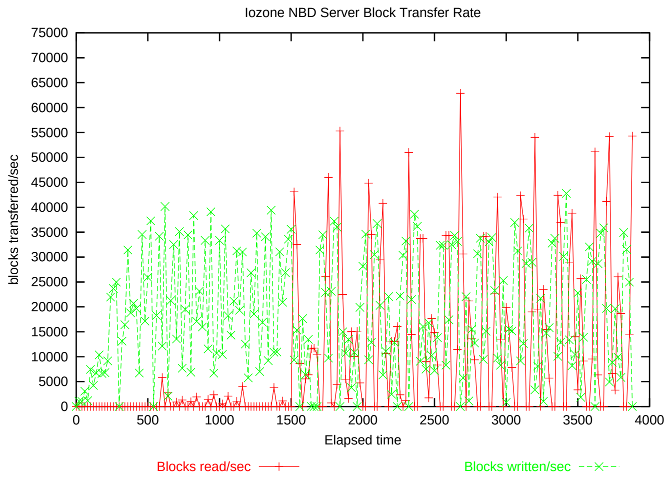
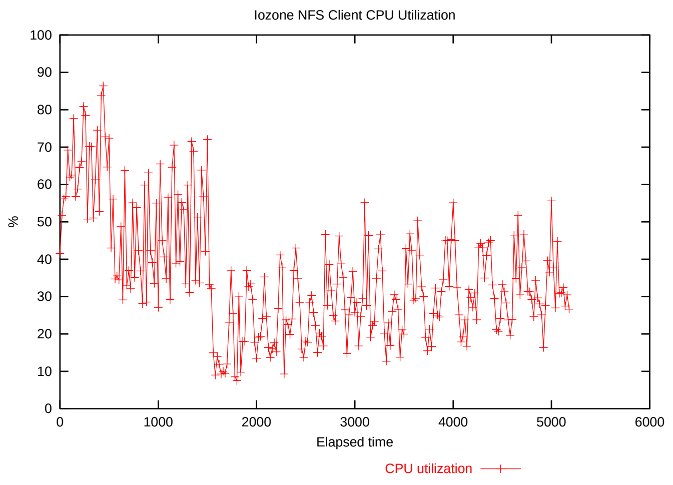


Figure 128: NBD Disk Blocks Transferred SAR Data



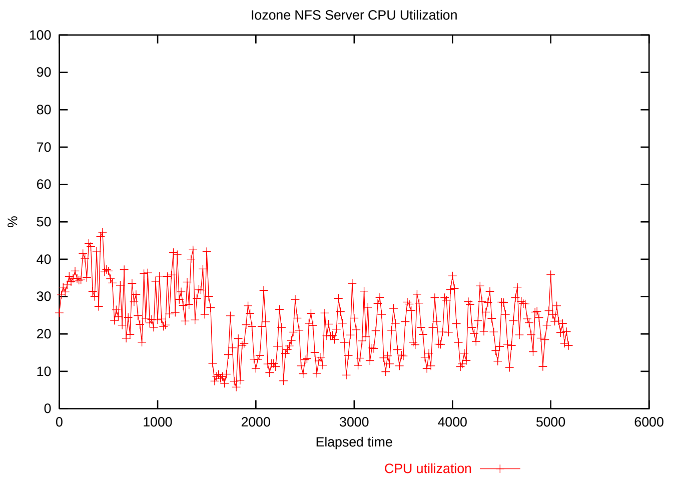
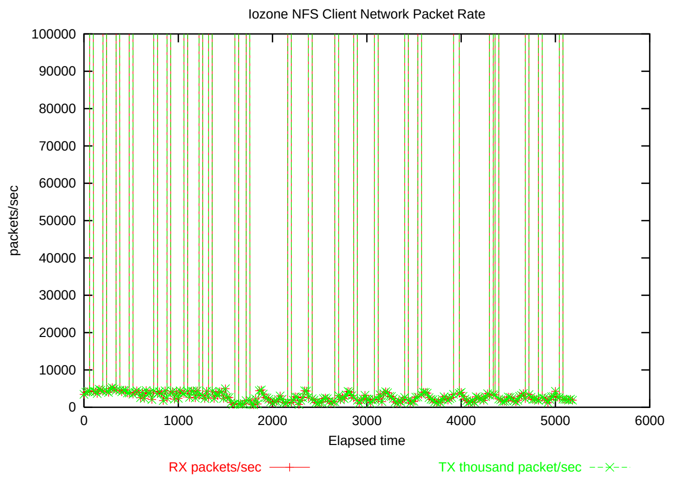


Figure 129: NFS CPU SAR Data



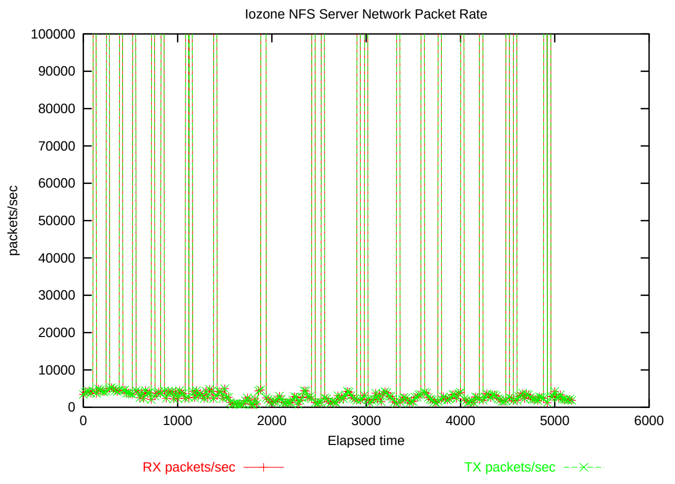


Figure 130: NFS Network Packet SAR Data

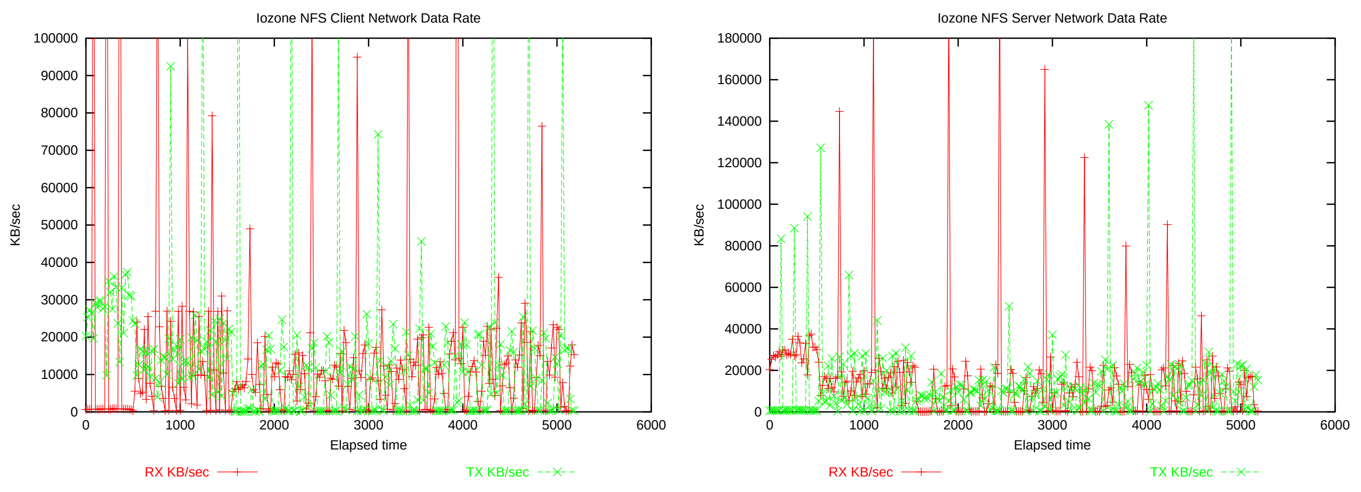


Figure 131: NFS Network Byte Count SAR Data

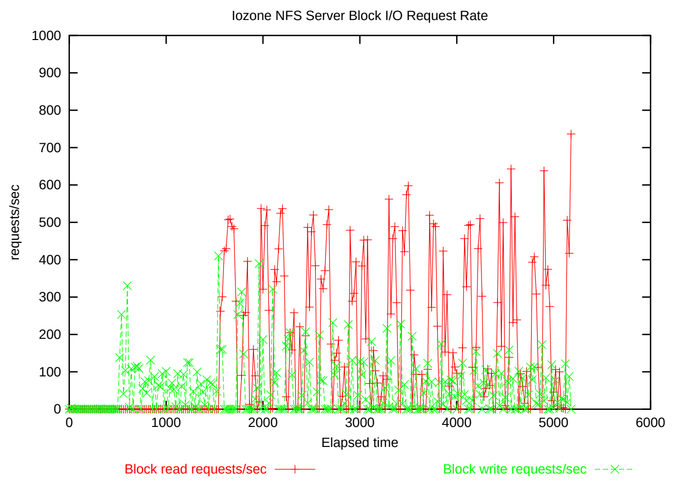


Figure 132: NFS Disk Request Rate SAR Data

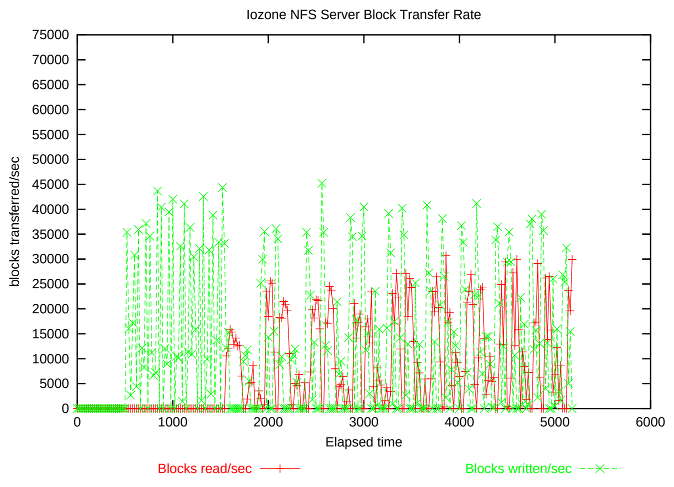


Figure 133: NFS Disk Blocks Transferred SAR Data

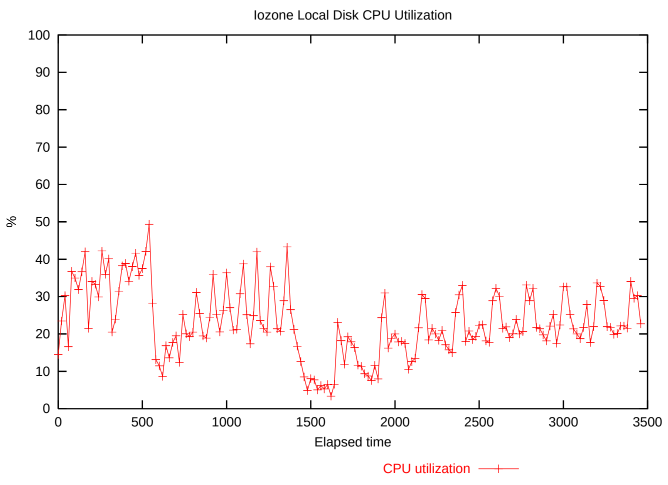


Figure 134: Local Disk CPU SAR Data

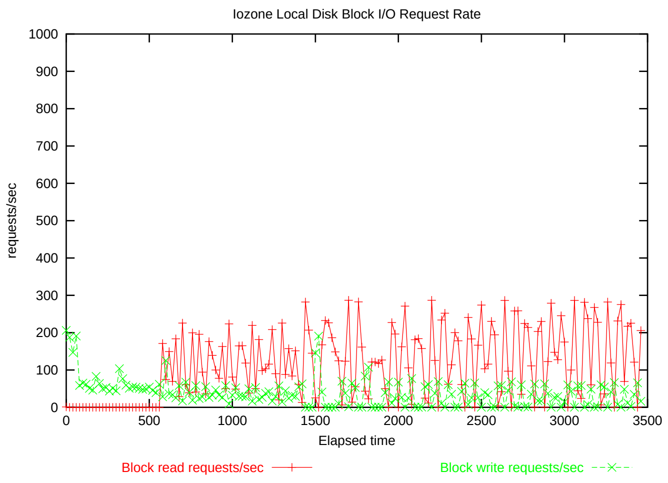


Figure 135: Local Disk Request Rate SAR Data

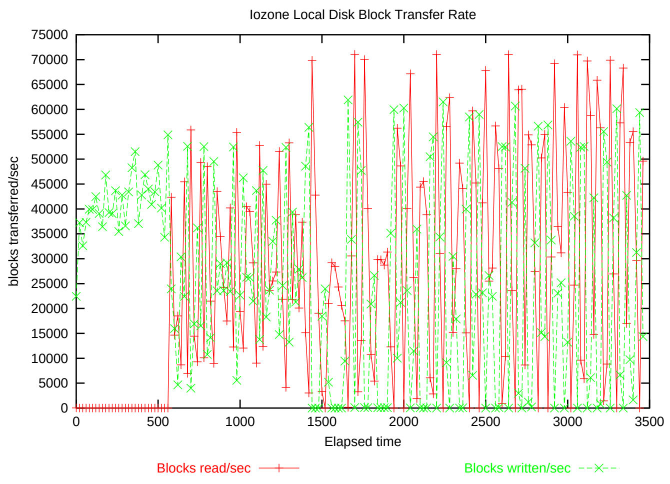
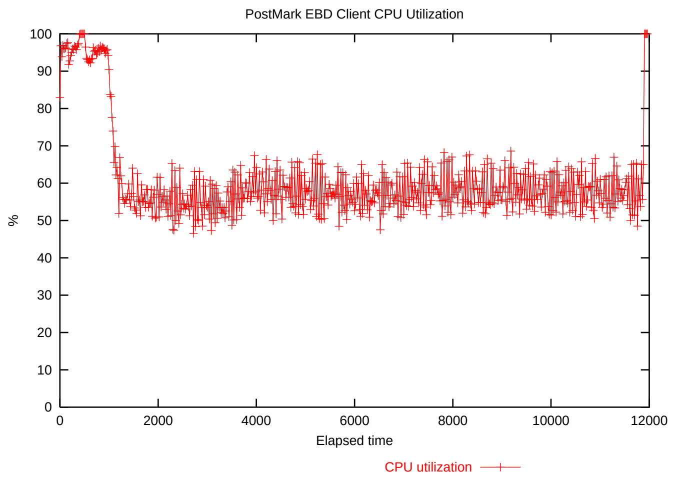


Figure 136: Local Disk Blocks Transferred SAR Data

C.3 Postmark System Analysis Report Plots



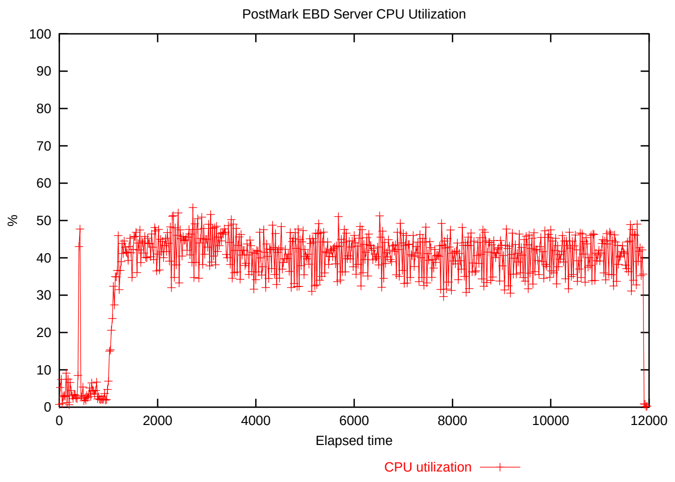
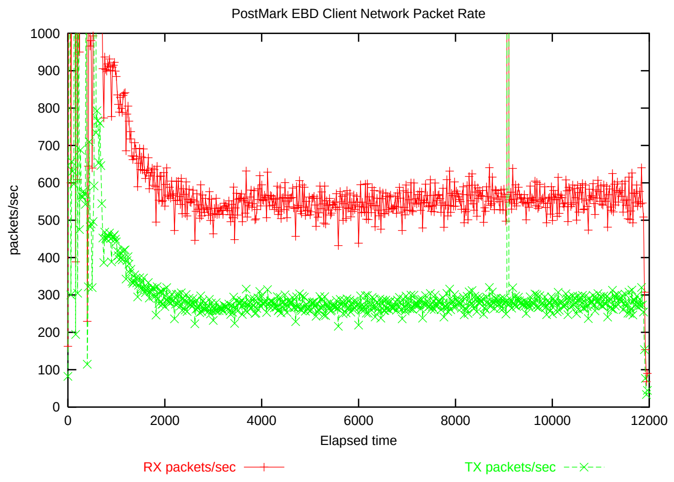


Figure 137: EBD CPU SAR Data



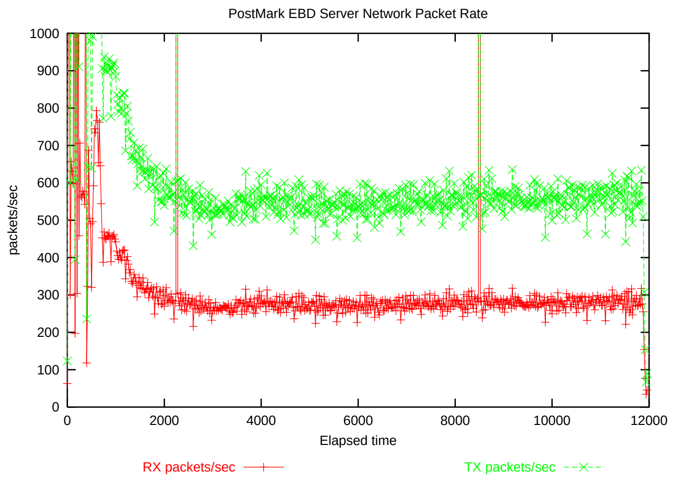


Figure 138: EBD Network Packet SAR Data

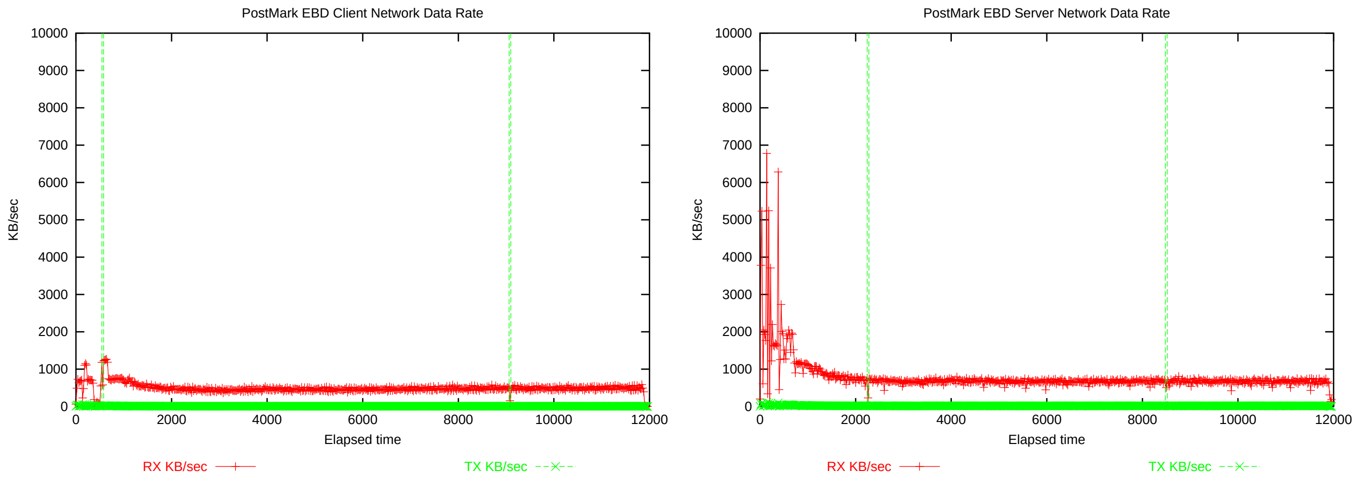


Figure 139: EBD Network Bytes Transferred SAR Data

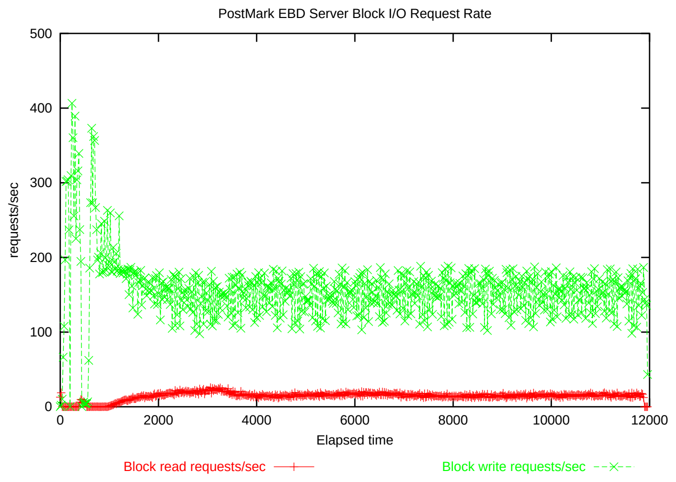


Figure 140: EBD Disk Request Rate SAR Data

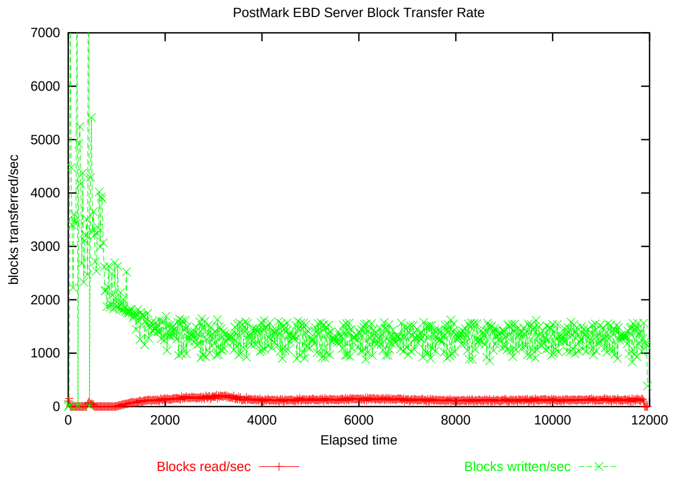


Figure 141: EBD Disk Blocks Transferred SAR Data

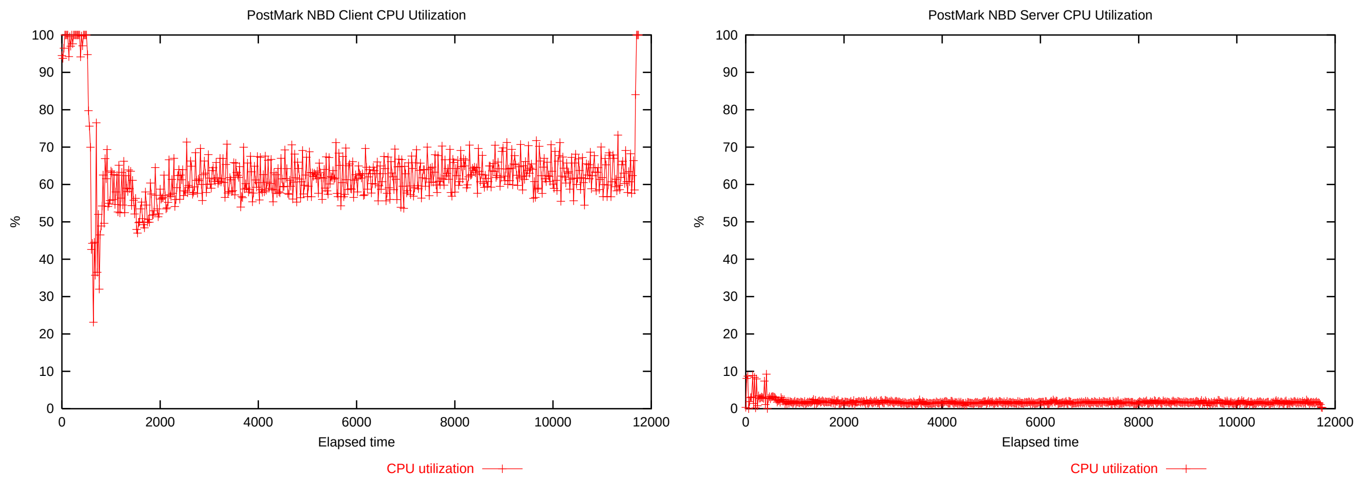


Figure 142: NBD CPU SAR Data

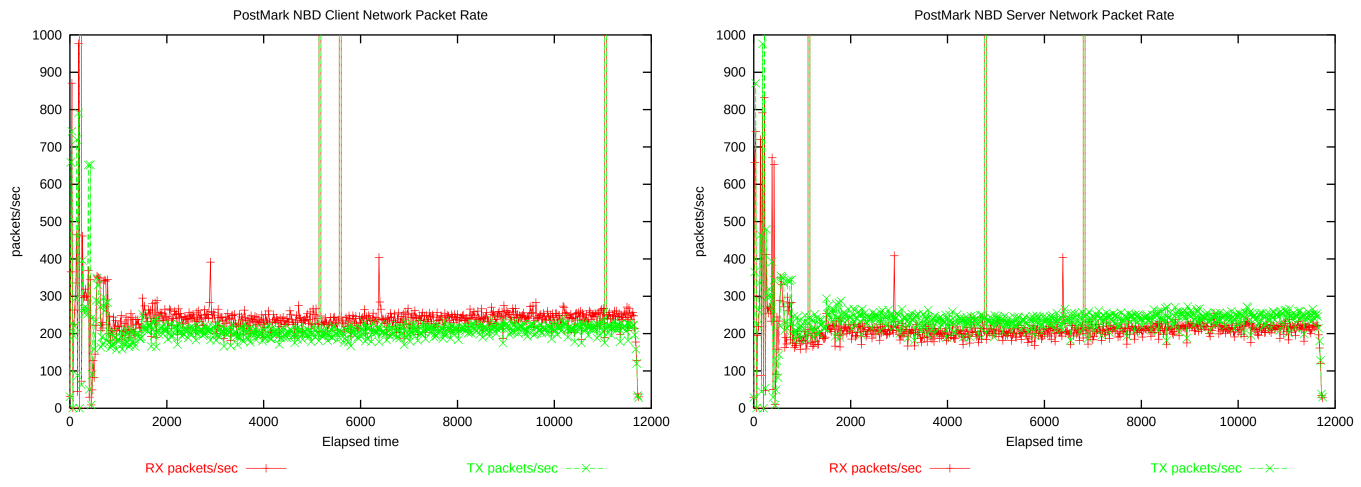


Figure 143: NBD Network Packet SAR Data

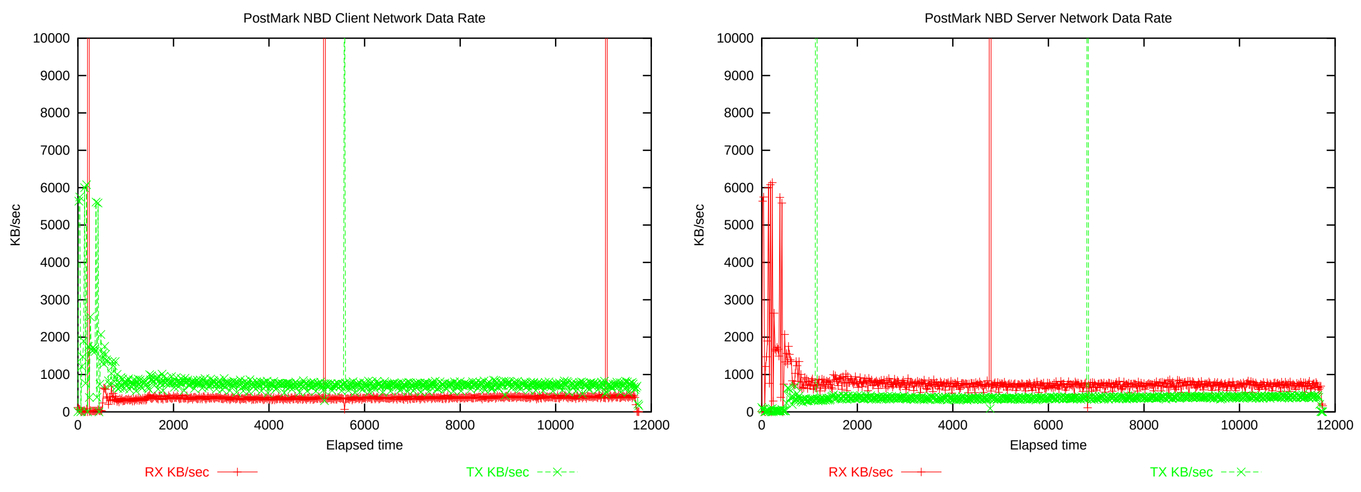


Figure 144: NBD Network Byte Count SAR Data

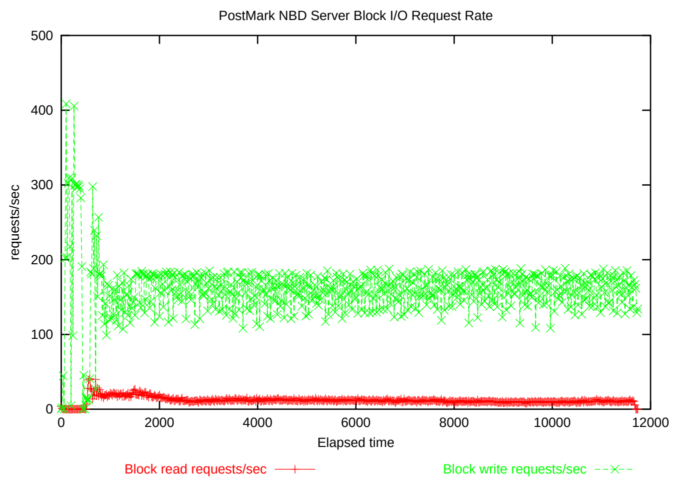


Figure 145: NBD Disk Request Rate SAR Data

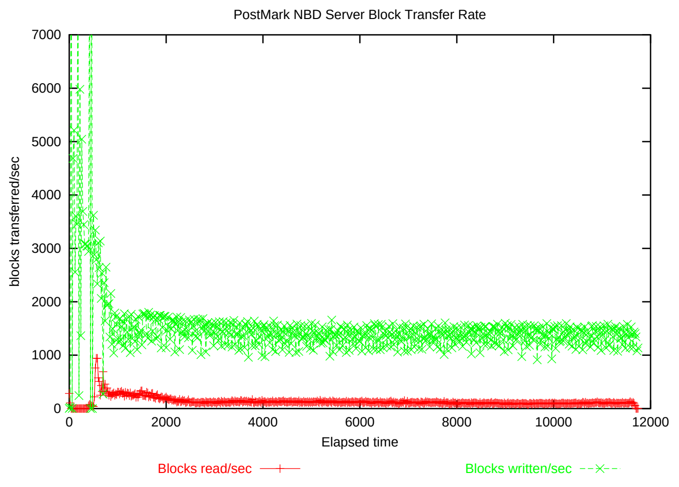
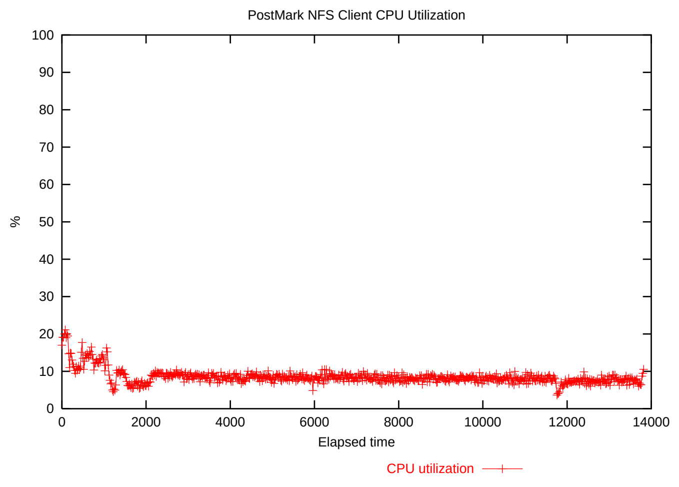


Figure 146: NBD Disk Blocks Transferred SAR Data



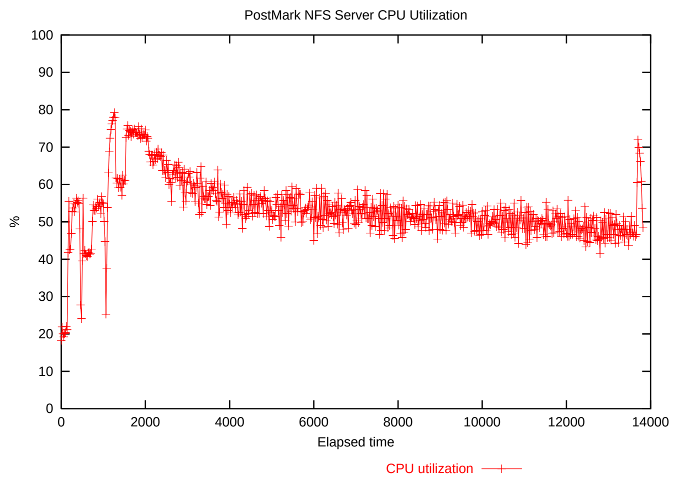
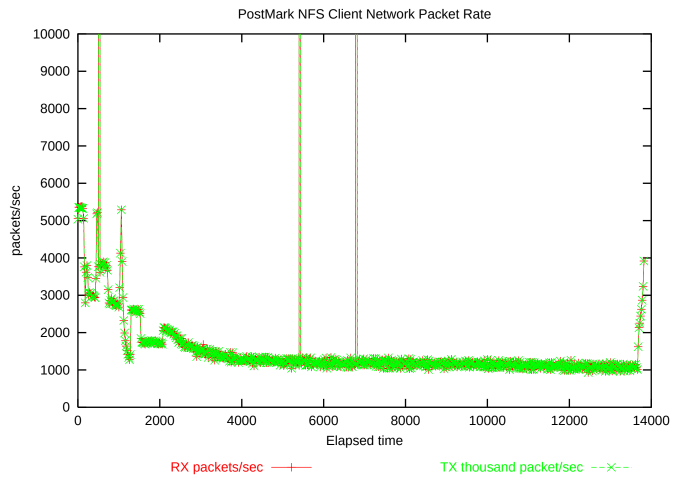


Figure 147: NFS CPU SAR Data



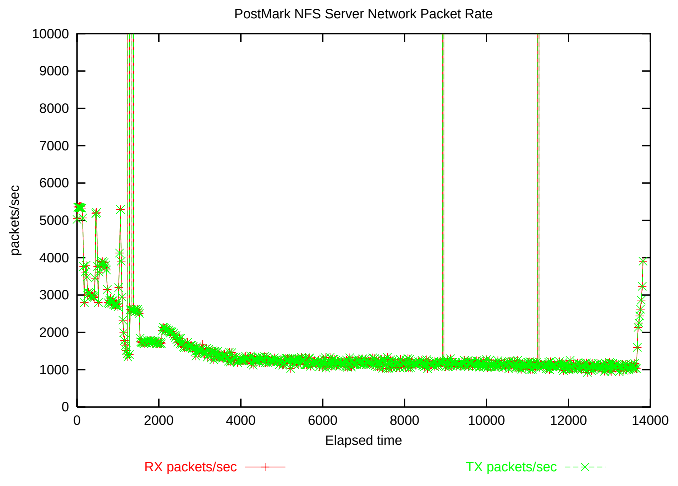


Figure 148: NFS Network Packet SAR Data

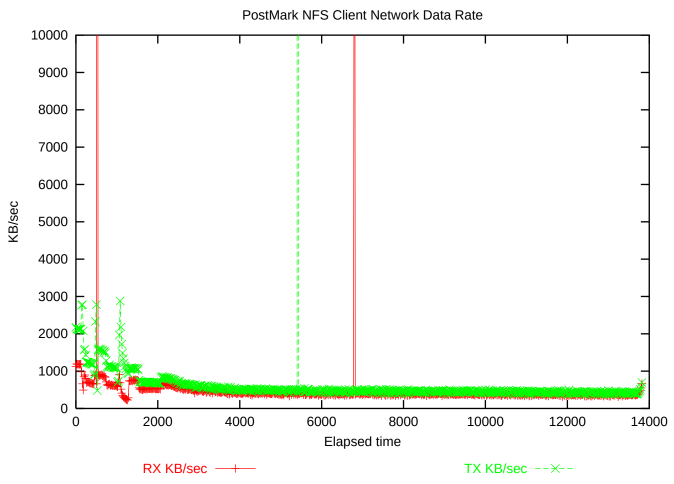


Figure 149: NFS Network Byte Count SAR Data

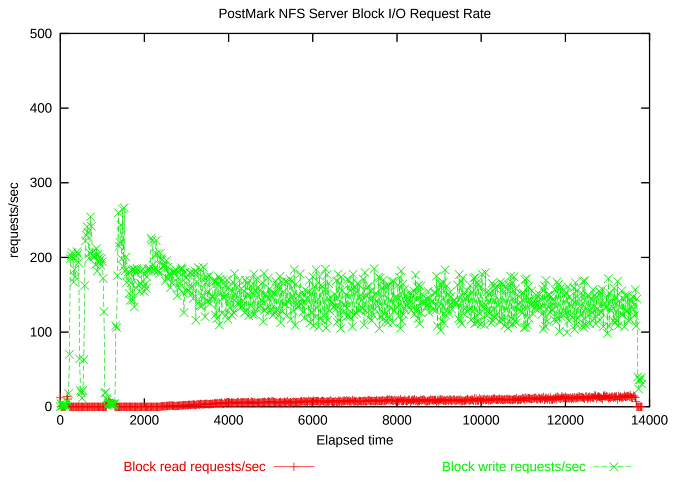


Figure 150: NFS Disk Request Rate SAR Data

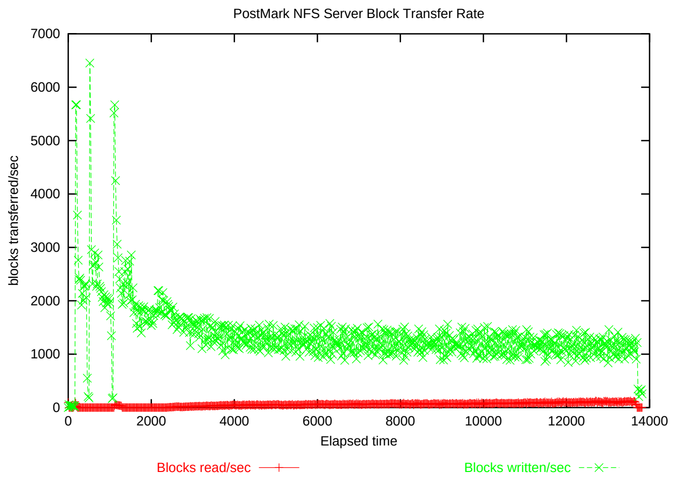


Figure 151: NFS Disk Blocks Transferred SAR Data

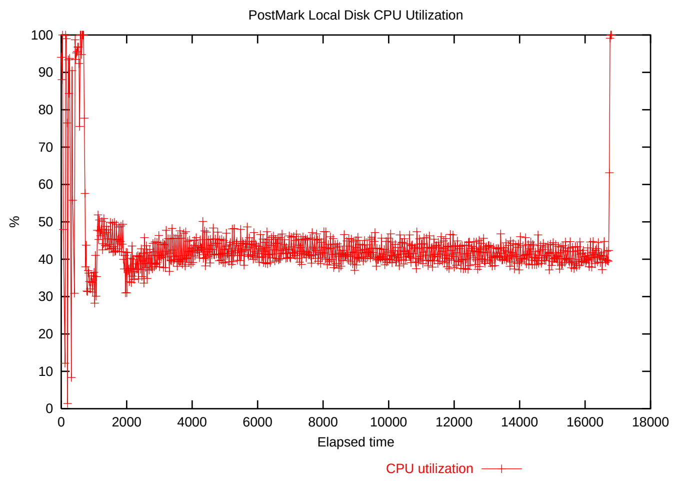


Figure 152: Local Disk CPU SAR Data

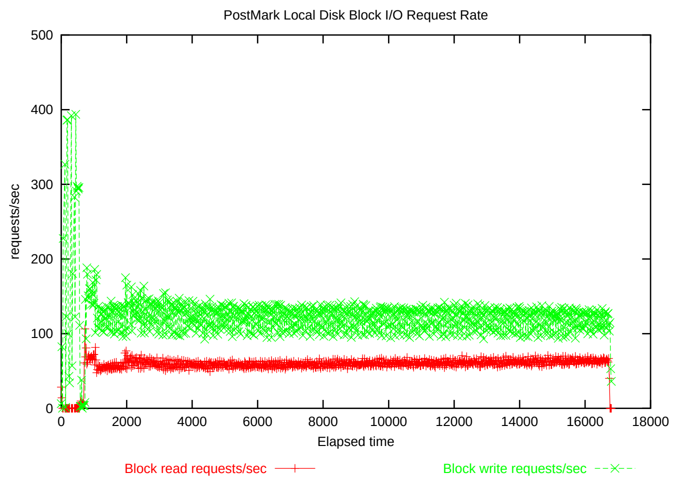


Figure 153: Local Disk Request Rate SAR Data

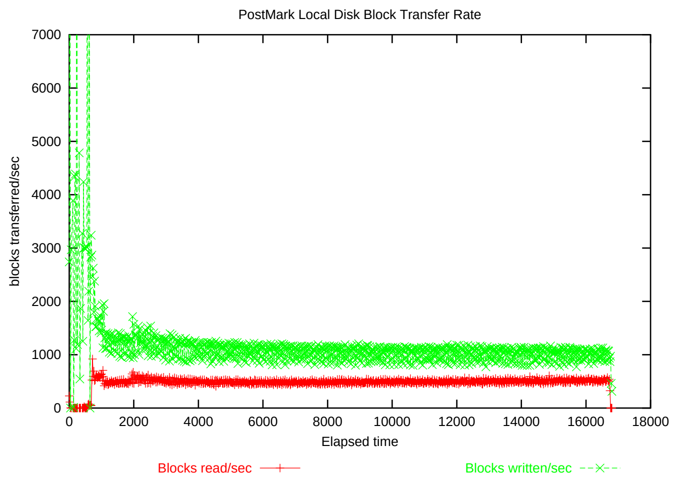
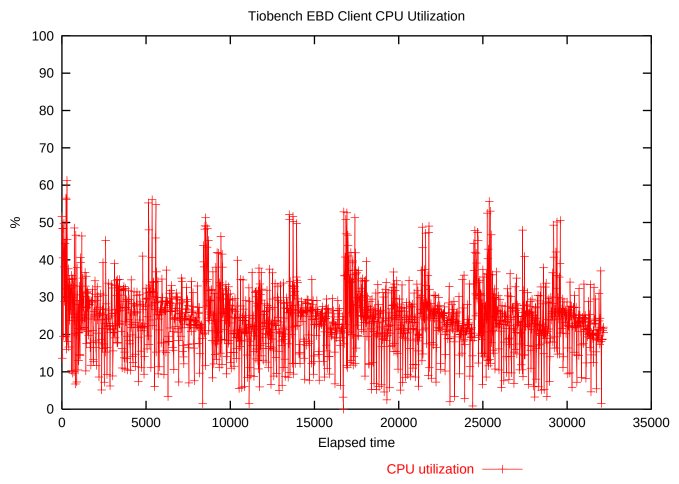


Figure 154: Local Disk Blocks Transferred SAR Data

C.4 Tiobench System Analysis Report Plots



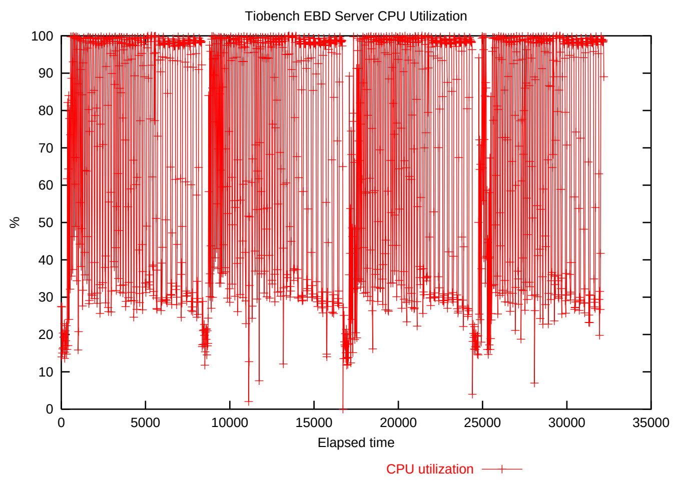
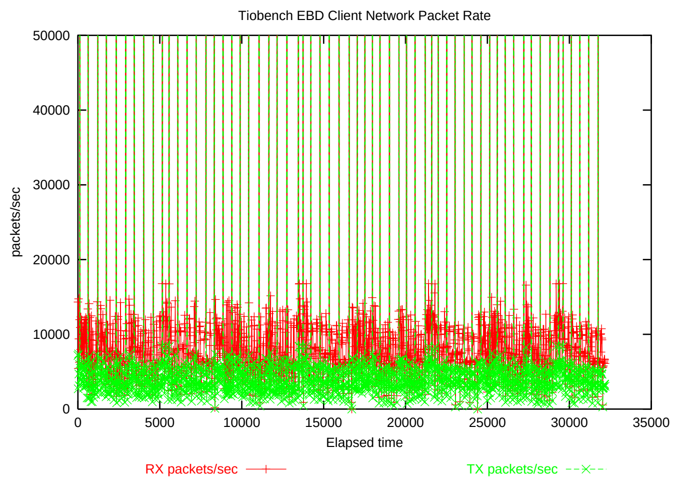


Figure 155: EBD CPU SAR Data



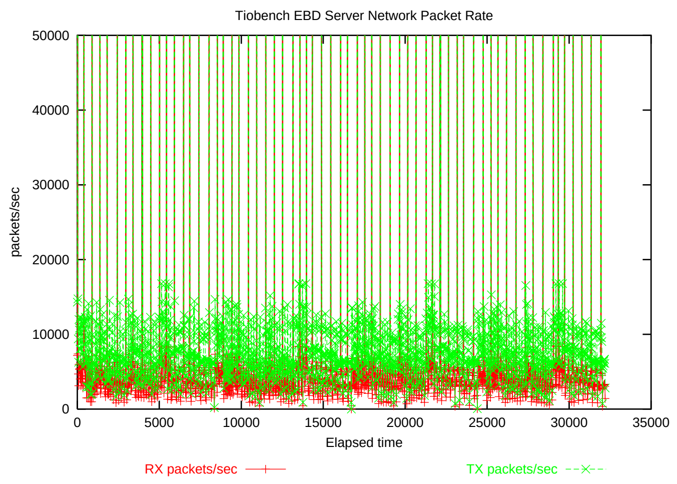
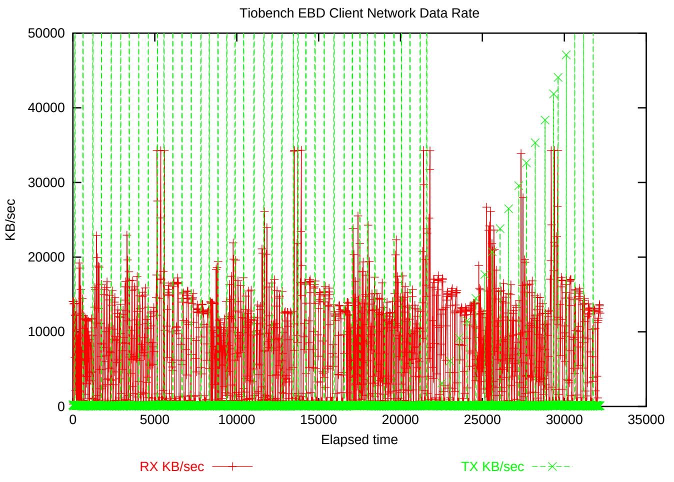


Figure 156: EBD Network Packet SAR Data



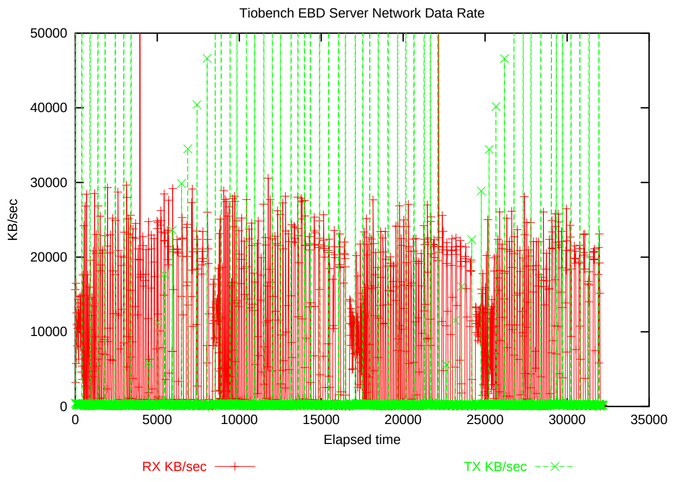


Figure 157: EBD Network Bytes Transferred SAR Data

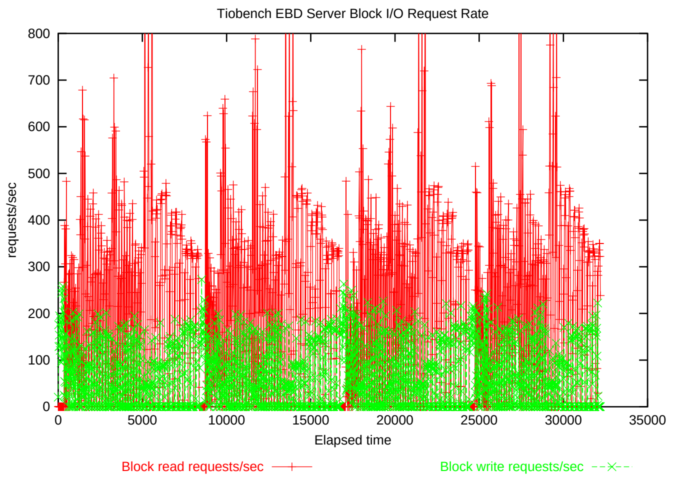


Figure 158: EBD Disk Request Rate SAR Data

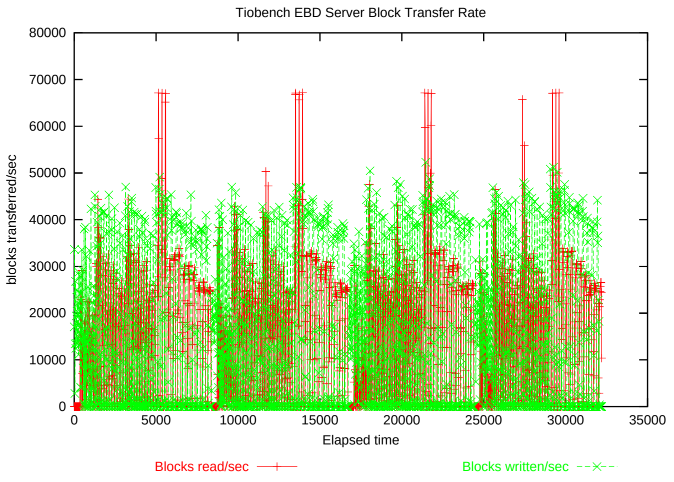
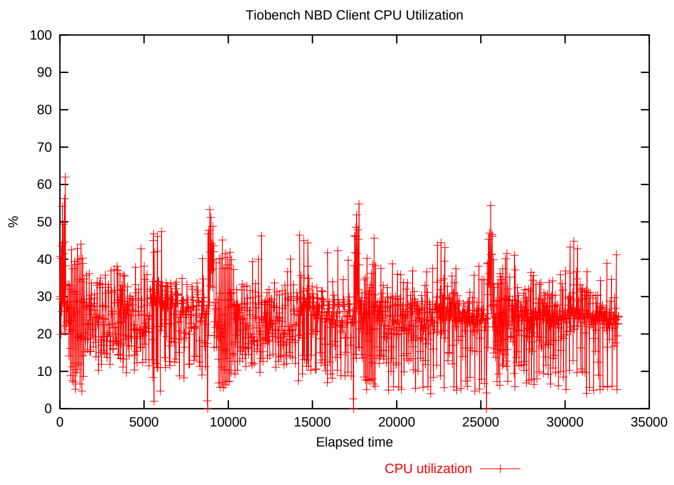


Figure 159: EBD Disk Blocks Transferred SAR Data



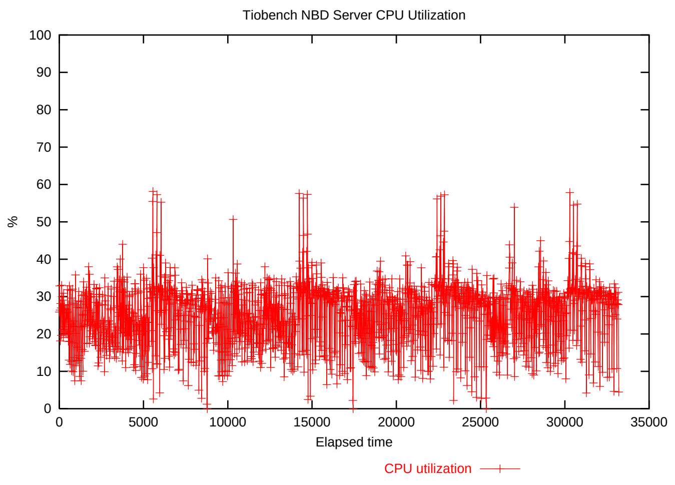
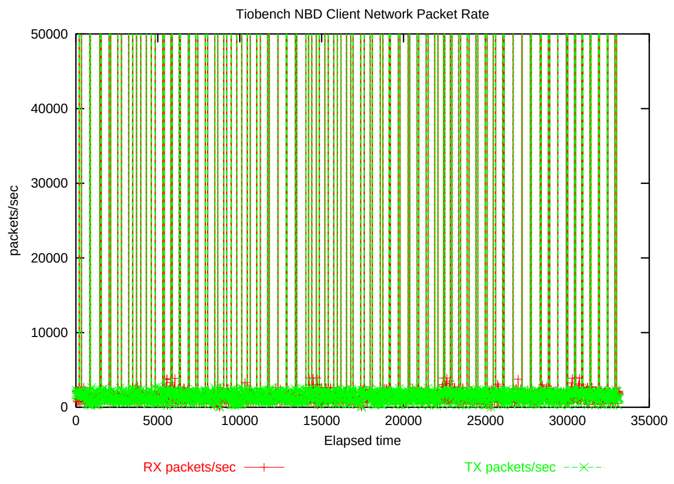


Figure 160: NBD CPU SAR Data



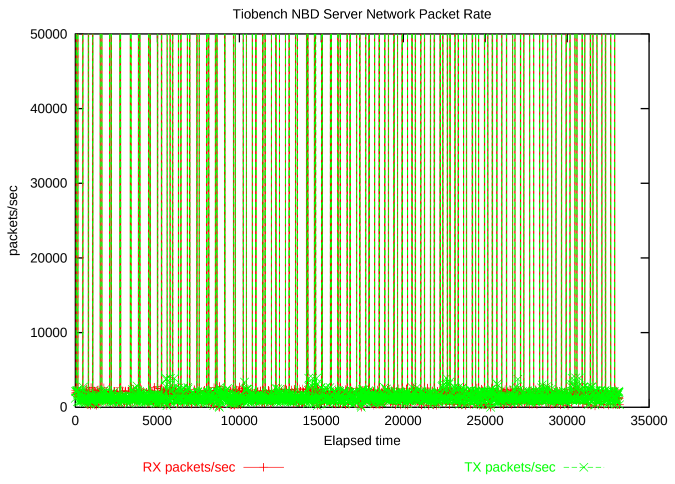
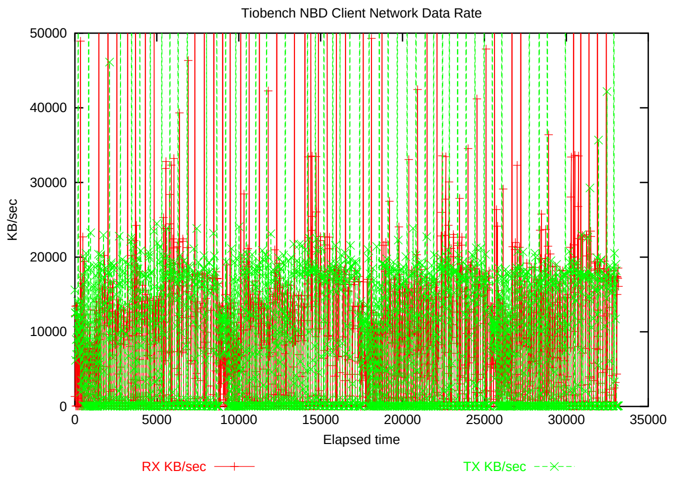


Figure 161: NBD Network Packet SAR Data



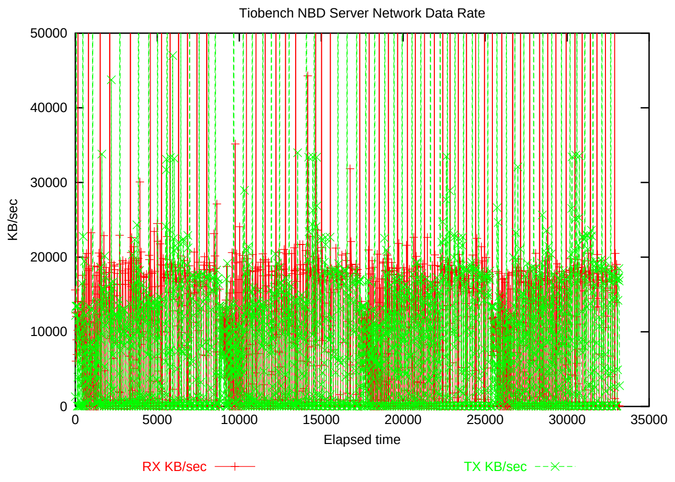


Figure 162: NBD Network Byte Count SAR Data

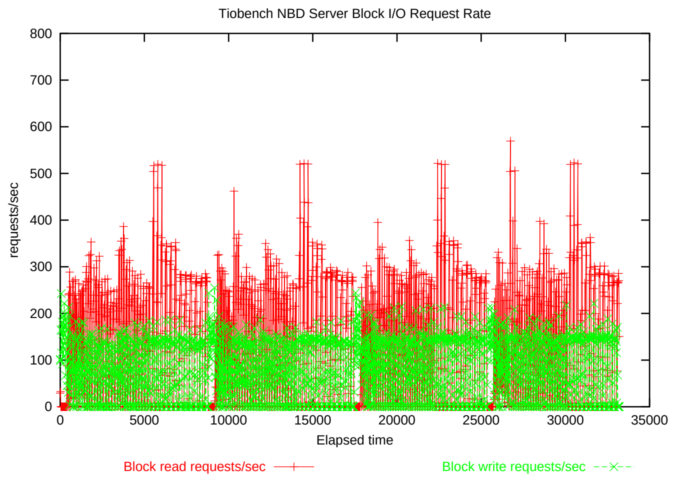


Figure 163: NBD Disk Request Rate SAR Data

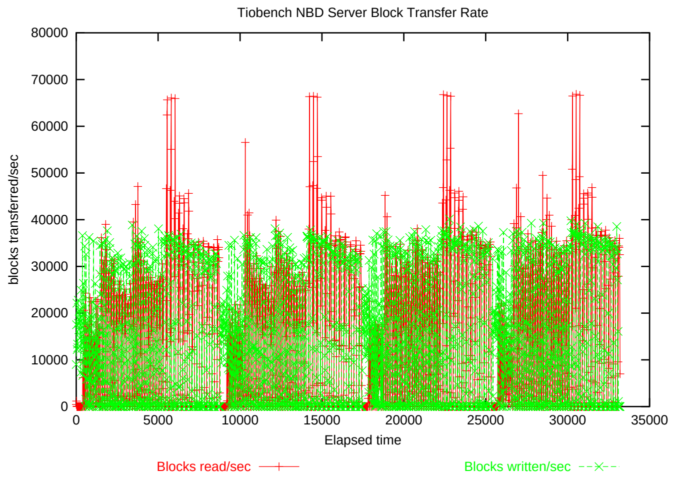
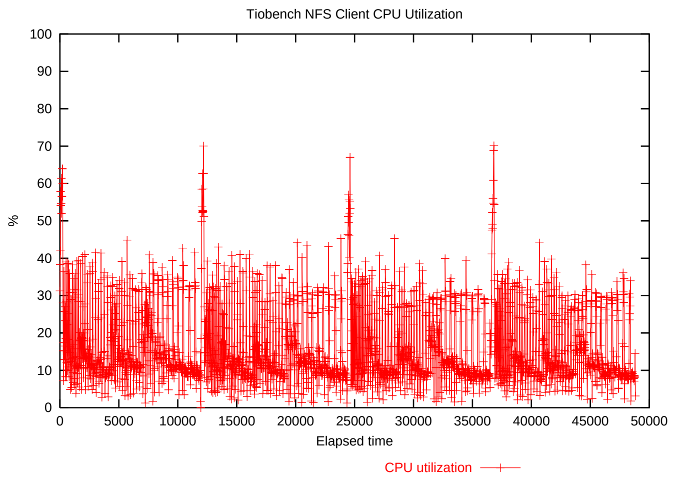


Figure 164: NBD Disk Blocks Transferred SAR Data



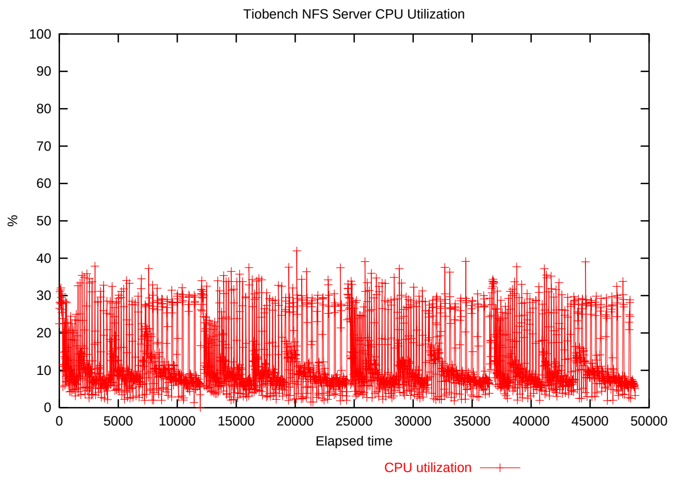
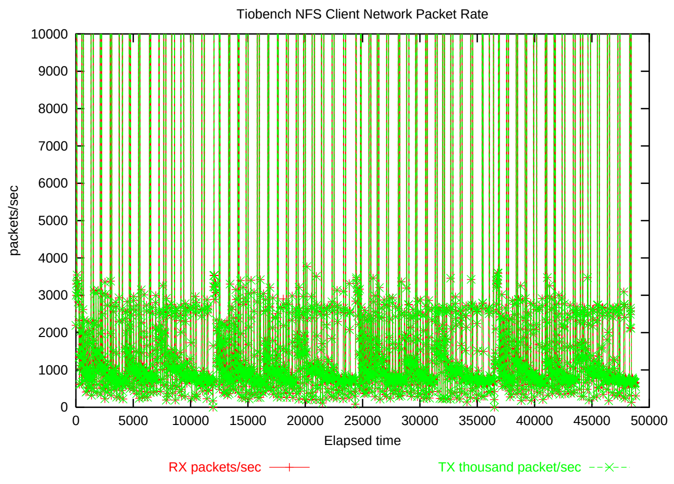


Figure 165: NFS CPU SAR Data



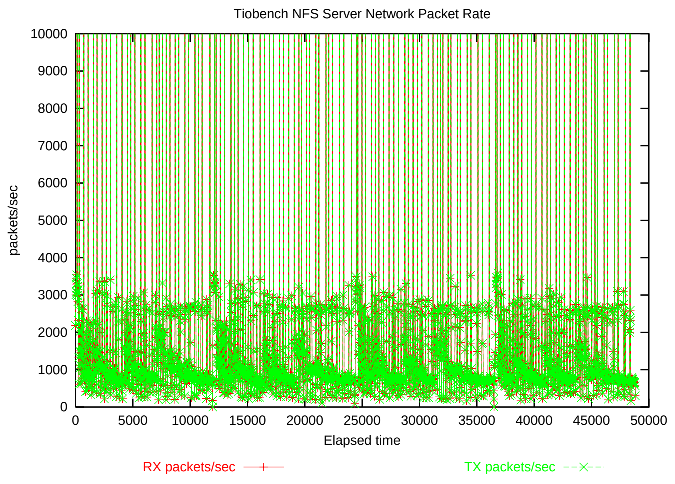
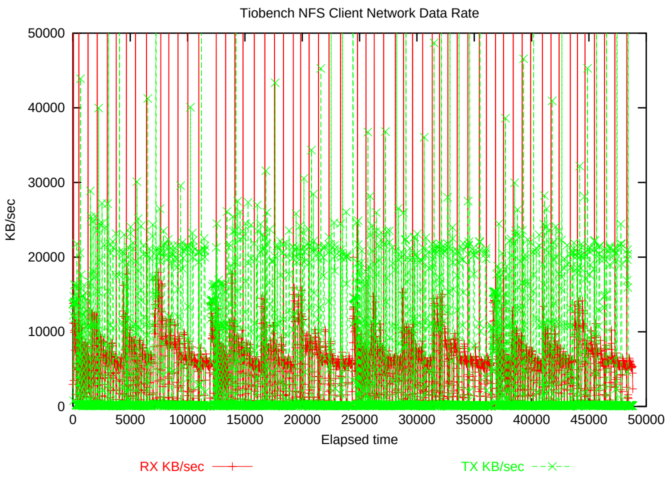


Figure 166: NFS Network Packet SAR Data



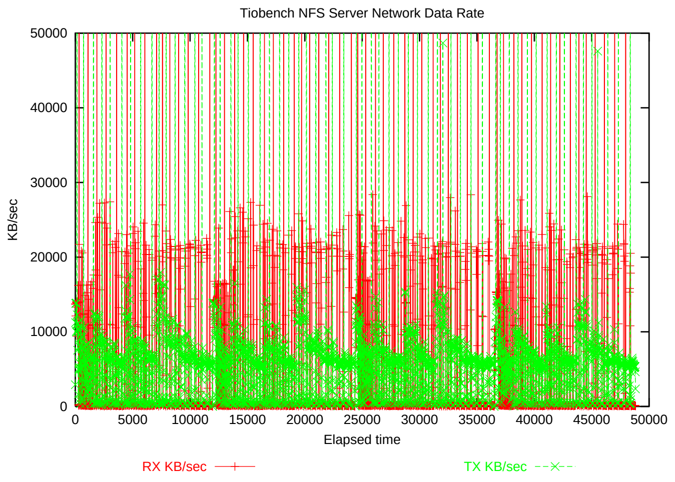


Figure 167: NFS Network Byte Count SAR Data

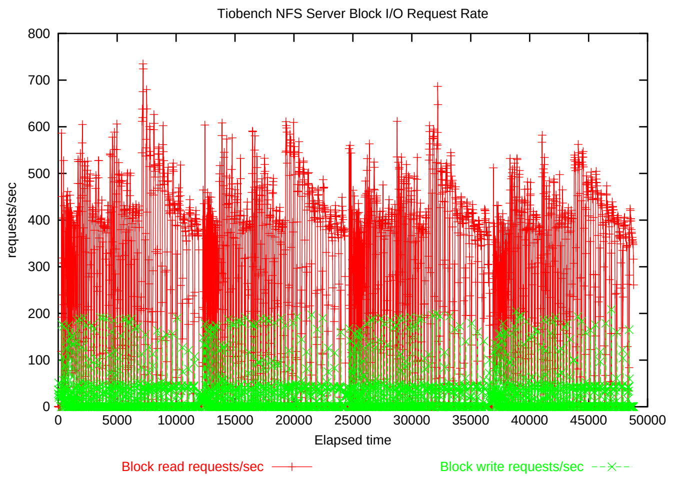


Figure 168: NFS Disk Request Rate SAR Data

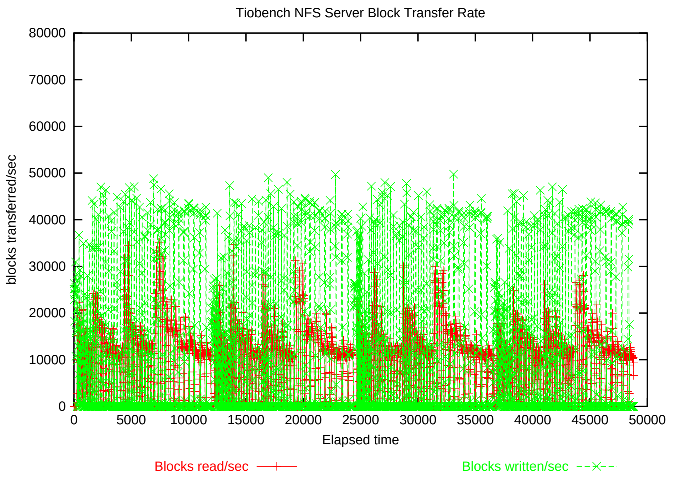


Figure 169: NFS Disk Blocks Transferred SAR Data

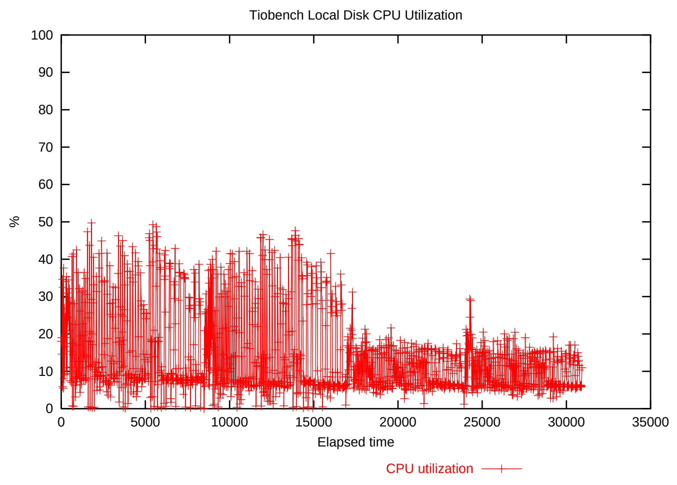


Figure 170: Local Disk CPU SAR Data

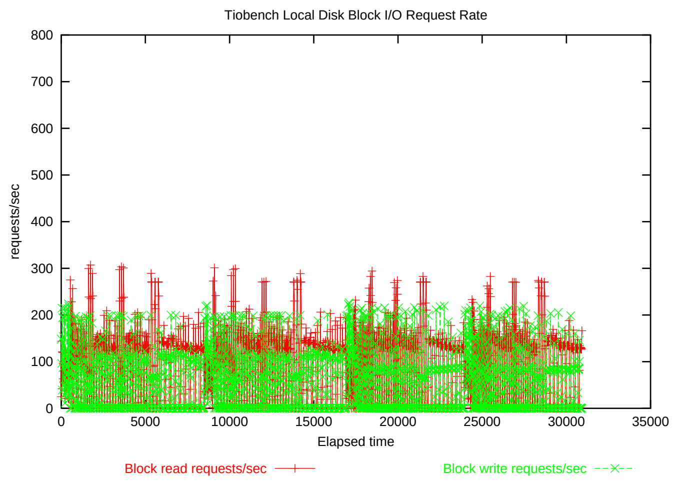


Figure 171: Local Disk Request Rate SAR Data

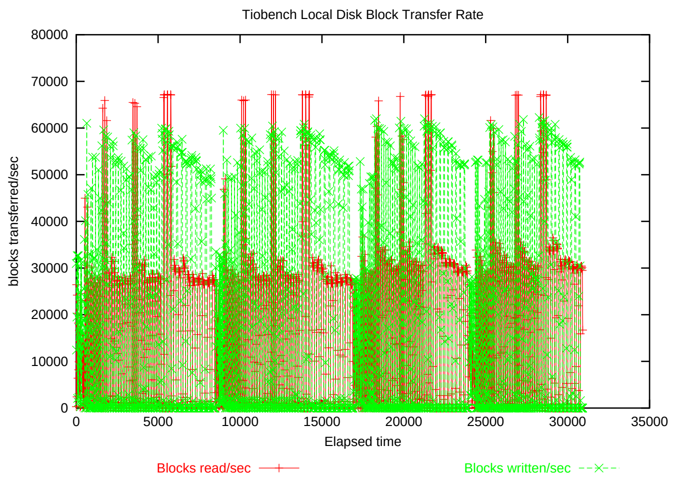


Figure 172: Local Disk Blocks Transferred SAR Data

References

[1] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212. Association for Computing Machinery, 1991.

[2] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly and Associates, Inc., 2001.

[3] P. Breuer, A. Marin Lopez, and A. Garcia Ares. The network block device. *Linux Journal*, (Number 73), May 2000.

[4] B. Callaghan. *NFS Illustrated*. Addison-Wesley, 2000.

[5] D. Cheriton. Vmtp: A transport protocol for the next generation of communication systems. In *Proceedings of the 1986 ACM SIGCOMM Conference on Communication Architectures and Protocols*, pages 406–415. ACM, 1986.

[6] D. Cheriton. The v distributed system. *Communications of the ACM*, 31(3), March 1989.

[7] R. Coker. Bonnie++ home page. http://www.coker.com.au/bonnie+, 2002.

[8] D. Comer. *Internetworking with TCP/IP, Volume 1: Principles, Protocols and Architectures, Third Edition*. Prentice-Hall, 1995.

[9] Leonardo Davinci, 1452-1519.

[10] Riccardo Gusella. A measurement study of diskless workstation traffic on an ethernet. *IEEE Transactions on Communications*, 38(9).

[11] IEEE. IEEE standards for local and metropolitan area networks: Virtual bridge local area networks, ieee standard 802.1q-1998.

[12] Philippe Joubert, Robert King, Richard Neves, Mark Russinovich, and John Tracey. High-performance memory-based web servers: Kernel and user-space performance. In *Proceedings of the 2001 USENIX Annual Technical Conference*. USENIX Association, 2001.

[13] J. Katcher and S. Kleiman. An introduction to the direct access file system. http://www.dafscollaborative.org.

[14] Jeffrey Katcher. Postmark: A new file system benchmark. Network Appliance Technical Report 3022.

[15] R. Katz. High-performance network and channel-based storage. *Proceedings of the IEEE*, 80(8), August 1992.

[16] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click moduler router. *ACM Transactions on Computer Systems*, 18(3), August 2000.

[17] E. Lazowska, J. Zahorjan, and D. Cheriton. File access performance of diskless workstations. 4(3):238–268, 1986.

[18] Paul Leach and Dan Perry. Cifs: A common internet file system. *Microsoft Interactive Developer*, November 1996.

[19] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[20] Edward K. Lee, Chandramohan A. Thekkath, and Timothy Mann. Frangpani: A scalable distributed file system. In *Symposium on Operating Systems Principals*, 1997.

[21] Pavel Machek. Network block device (tcp/ip version). http://nbd.sourceforge.net, 2002.

[22] 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 the 2002 USENIX Annual Technical Conference*, 2002.

[23] R. Van Meter, G. Finn, and S. Hotz. VISA: Netstation's virtual internet scsi adapter. In *Proceedings of the 8th Internation Conference on Architectural Support for Programming Languages and Operating Systems*, pages 71–80. ACM, 1998.

[24] C. Monia, R. Mullendore, F. Travostino, W. Jeong, and M. Edwards. iFCP – a protocol for internet fibre channel networking. http://www.ieft.org/internet-drafts/draft-ietf-ips-ifcp-13.txt.

[25] M. Nelson, B. Welch, and J. Ousterhout. Caching in the sprite network file system. 6(1):134–154, 1988.

[26] 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.

[27] L. Peterson and B. Davie. *Computer Networks, Second Edition*. Morgan Kaufman, 2000.

[28] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, P. Winterbottom, and P. From. Plan 9 From Bell Labs. *USENIX Computing Systems*, 1995.

[29] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *First USENIX Symposium on File Systems and Storage Technologies*. USENIX Association, 2002.

[30] M. Rajagopal, E. Rodriguez, and R. Weber. Fibre channel over tcp/ip (fcip). http://www.ietf.org/internet-drafts/draft-ietf-ips-fcovertcpip-12.txt.

[31] Red Hat. Red Hat content accelerator 2.2: Reference manual. http://www.redhat.com/docs/manuals/tux/rh-content-accelerator-2.2.pdf, 2002.

[32] A. Rubini and J. Corbet. *Linux Device Drivers, Second Edition*. O'Reilly and Associates, 2001.

[33] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network file system. In *Proceedings of the 1985 USENIX Summer Conference*. USENIX Association, 1985.

[34] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. iscsi. http://www.ieft.org/internet-drafts/draft-ietf-ips-iSCSI-17.txt.

[35] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts, 6th Edition*. John Wiley and Sons, Inc., 2001.

[36] Sun Microsystems. Nd(4p) - network disk driver. In System Interface Manual for the Sun Workstation, 1984.

[37] K. Thompson. The Plan 9 File Server, 1995.

[38] K. Voruganti and P. Sarkar. Analysis of three gigabit network protocols for storage area networks. In *Proceedings of the 2001 International Performance, Computing and Communications Conference*, 2001.