The Network RamDisk : Using Remote Memory on Heterogeneous NOWs*

Michail D. Flouris and Evangelos P. Markatos †

Computer Architecture and VLSI Systems Division
Institute of Computer Science (ICS)
Foundation for Research & Technology – Hellas (FORTH),
P.O.Box 1385
Heraklion, Crete, GR-711-10 GREECE
{flouris, markatos}@ics.forth.gr

August 4, 1998

Abstract

Efficient data storage, a major concern in the modern computer industry, is mostly provided today by the the traditional magnetic disk. Unfortunately the cost of a disk transfer measured in processor cycles continues to increase with time, making disk accesses increasingly expensive.

In this paper we describe the design, implementation and evaluation of a device that uses main memory of remote workstations as a faster-than-disk storage device. Our Network RamDisk has been implemented both on the Linux and the Digital Unix operating systems, as a block device driver without any modifications to the kernel code. In our study we propose various reliability policies, making the device tolerant to single workstation crashes. We show that the Network RamDisk, is portable, flexible, and can operate under any of the existing Unix filesystems. Using several real applications and benchmarks, we measure the performance of the Network RamDisk over an Ethernet and an ATM network, and find it to be usually four to eight times faster than the magnetic disk. In one benchmark, our system was two orders of magnitude faster than the disk.

We believe that a Network RamDisk can be efficiently used to provide reliable low-latency access to files, that would otherwise be stored on magnetic disks.

1 Introduction

Reliable and efficient data storage is a major concern in the modern computer industry. This type of storage is mainly provided by magnetic and optical devices, the most common being the

*This project was supported in part by a Usenix Student Research Grant. This work was supported in part by PENED project "Exploitation of idle memory in a workstation cluster" (2041 2270/1-2-95) funded by the General Secretariat for Research and Technology of the Ministry of Development. We deeply appreciate this financial support.

†The authors are also with the University of Crete.

traditional magnetic disks. However, since processor performance improves at a higher rate than disk performance, the cost of a disk access (measured in processor cycles) continues to increase with time. As a result, several applications whose effective execution depends on low latency access to their files, are going to suffer an increasingly high performance penalty if they store their data on traditional magnetic disks. Such applications include transaction-based systems, visualization systems, web servers, web proxies, compilers, etc.

Several file systems [20, 22, 25] attempt to speedup file accesses by using a portion of the file server's main memory as a disk cache. Unfortunately, such caches can not be larger than a single workstation's main memory. To overcome this memory size limitation, research file systems use all client and server caches in a NOW as a single cooperative cache [1]. Unfortunately, such file systems are not widespread yet, and their principles have not been incorporated into commercial operating systems.

In our study we describe a new device, the Network RamDisk, which attacks the high latency problem in a very convenient manner. A Network RamDisk is a block device that consists of the all idle main memories in a Network of Workstations (NOW). It behaves like any other disk, allowing the creation of files and file systems on top of it. However, since it is implemented in main memory (RAM), it provides lower latency and higher bandwidth than most traditional magnetic disks.

In this paper, we propose to organize the (otherwise unused) client and server memories of a NOW, into a Network RamDisk. Idle workstations donate their memory which will be used to store a portion of the Network RamDisk. When an application reads a file, the file system (e.g. NFS, UFS) requests a block from the device driver of the Network RamDisk. The driver knows in which workstation's main memory the block resides, asks the workstation for this block, and returns the requested block. Write operations proceed in the same way. Since block accesses involve only memory and interconnection network transfers, they proceed with low latency and high bandwidth. For example, typical disk latency is around 10 ms, while typical network latency is around 1 ms. Modern interconnection networks provide latency as low as a few microseconds [4, 6, 14, 17, 19]. Thus, Network RamDisks may result in significant performance improvements over magnetic disks, especially when application performance depends on latency.

The Network RamDisk, much like network memory file systems [1, 15] exploits network memory to avoid magnetic disk I/O, but unlike these file systems, the Network RamDisk, being a device driver, can be easily incorporated into any environment, without modifying existing operating system sources, or changing file systems. Thus, by using a Network RamDisk, users will be able to exploit all network memory to store their files, without changing their operating system, or their file system. For example, files created on a Network RamDisk can be accessed through any file system, e.g. NFS. NFS will not be aware of the fact that the files do not reside on magnetic disk.

The rest of the paper is organized as follows: Section 2 present the design of a Network RamDisk. Section 3 describes the implementations of two Network RamDisk clients as device drivers of the Digital Unix 4.0 and the Linux operating systems, and several Network RamDisk servers. Section 4 presents our experiments and performance results with the Digital Unix NRD client over the Ethernet interconnection network, and the Linux NRD client over the ATM interconnection network. Section 5 presents related work, while section 6 presents our conclusions.

2 The Design of a Network RamDisk

The Network RamDisk (NRD) is a scalable distributed low-latency storage system. It consists of one machine that functions as a NRD Client, and a number of workstations functioning as NRD Servers.

2.1 Server and Client Workstations

The workstations that participate in the Network RamDisk by letting a certain amount of their memory be used, are known as NRD servers, while the workstation that can access the Network RamDisk and the filesystems on it, is called the NRD client. Any workstation in a NOW may act as a server, making a selectable amount of its memory available for the Network RamDisk, depending on its workload and memory capacity. The client machine must have the NRD device driver linked into its operating system, and must be connected over an interconnection network with the server workstations. The Network RamDisk functions then as a normal disk mounted on the client host, and any computer on the network can access the NRD by mapping it (e.g. by NFS) from the client. It is possible that a machine can function both as a client and a server.

2.2 Device Operation

All server workstations run an NRD server process, which stores a number of disk blocks and handles all block read and write requests. The NRD client accepts read or write as a normal disk, and when it wants to read or write a number of blocks, it searches its block table to find out on which server the corresponding blocks reside, and starts sending requests to that server.

In case of native memory-demanding processes starting on a server workstation, part of the server's virtual memory is swapped to disk. Future requests will be serviced from the disk, thus degrading performance. The problem can be handled by the server which can send a message to the client notifying of its memory-loaded state. The latter will try to find another server having enough free memory and migrate the disk blocks that were stored on the loaded server to a new one. If this is not possible, the delay of requests to the loaded server are inevitable.

2.3 Reliability Issues

In a distributed system, a workstation may crash at any time (e.g. due to software or human error). If the crashed workstation acts as an NRD server, it will lose the disk blocks of the client. Clearly, it is not acceptable for applications running on the client workstation to lose their data files due to remote server crash. Instead, we would like to be able to recover their disk blocks. Otherwise a remote server crash is certain to cause, not only an application crash and important data loss, but to result in a client crash as well, especially in the case where the Network RamDisk is used to store swap space of applications.

If the crashed workstation acts only as a client, after recovering from the crash, it can reconnect to the server hosts and find the Network RamDisk in the same state as its magnetic disks were after the crash. Running `fsck` will certainly correct inconsistency problems.

There are many types of crashes. First of all there may be machine crashes due to loss of power. This situation is not addressed in this paper, since most computer buildings are equipped with UPSs. The most frequent cause of crash is a software crash. To avoid loss of data due to a server crash, some systems write all network memory data to the disk as well ([1, 11]). Instead, we

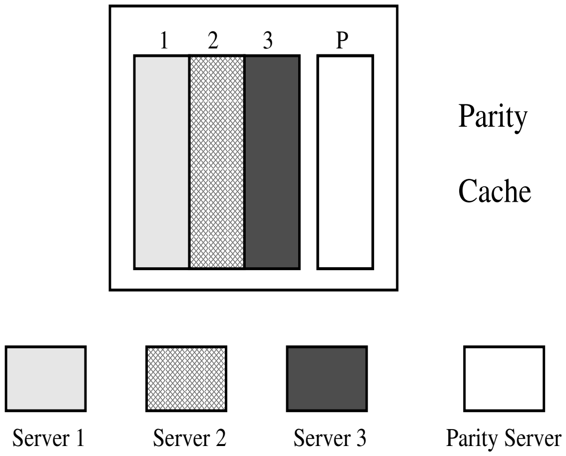

Figure 1: **The Parity Cache.**

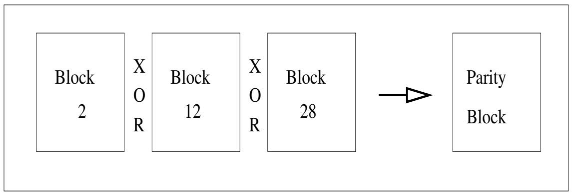

Figure 2: **A Parity Group.**

design a *reliable* Network RamDisk that is able to reconstruct the lost disk blocks when *one server* fails.

To provide this level of reliability, some form of redundancy must be used. The main issues that must be taken into account regarding the form of redundancy used are :

1. The runtime overhead introduced must be minimal since it is a cost paid even when no server crashes.

2. The memory overhead introduced must be as low as possible because the memory reserved for reliability could be used locally at each workstation in order to store its memory pages.

3. The runtime overhead of recovery from failure should be low.

We consider and discuss two different reliability policies: mirroring and a novel policy called adaptive parity caching.

2.3.1 Mirroring

The simplest form of redundancy is *mirroring*. In mirroring, there exist two copies of each disk block (block replication). When the client writes a disk block, the block is sent to two different servers, and the block table is updated. Thus, even when one of the servers crashes, the filesystem is still stable and consistent, because all disk blocks of the crashed server exist on the mirror server. Obviously the crash recovery overhead, in case of mirroring, is minimal. However, the runtime overhead is rather high, since each block write requires two block transfers. To make matters worse, mirroring wastes half of the remote memory used.

2.3.2 Parity Caching

To avoid the additional block transfers induced by the basic parity-based redundancy schemes used in RAIDS [7], we have considered the parity logging scheme, discussed in [18], as a suitable reliability policy for the NRD. The key idea of this technique is that a given block need not be bound to a particular server or parity group. Instead, every time a block is written out, a new server and a new parity group may be used to host the block.

Using the parity logging policy with S servers, the runtime overhead of the reliability policy can be reduced to a factor of $(1 + 1/S)$ of the runtime overhead without reliability. However, the parity logging policy *cannot be efficiently used* on our device without a major latency cost. This happens because the parity stripping unit of the parity logging policy must be fixed, and in disk devices the smallest data unit is one disk block (512 or 1024 bytes on most systems). Given that the filesystem sends a variable amount of blocks on each request (usually 1 to 8 blocks), the parity logging policy would make one small network transfer for each block, and as a result the latency of each I/O request would be a multiple of the latency of the network. For example, if we have an I/O request of 16 disk blocks the latency would be 16 times the latency of the network, which is similar, or even higher than the latency of magnetic disk.

Addressing this issue, we have developed a new reliability policy for our device to reduce both the memory overhead and the latency bottleneck for each I/O request. Our policy, which is called *parity caching*, uses a small amount of the NRD client's memory as a parity cache. The parity cache collects a fixed amount of disk blocks, enabling the parity caching policy to use more than one blocks as a parity stripping unit during normal operation, while it still supports recovery for each block unit in case of server failure. Obviously parity caching has much lower latency than the parity logging policy.

Before looking deeper at how parity caching functions, we need to define the concept of the parity group. A parity group as a number of data blocks and the corresponding parity block, which is formed by XORing all the data blocks. If one of the data blocks is lost, we can reconstruct the lost block by XORing all the remaining data blocks and the parity block. A parity group is depicted on figure 2 .

Suppose there exist one (reliable) client and S (unreliable) server machines, and each server can store B disk blocks. We need one server to keep the parity blocks (the parity server), while the other $S - 1$ servers (called storage servers) store normal data blocks. Accordingly, our cache has $S - 1$ buffers for the storage servers (storage buffers) and one for the parity server (the parity buffer). Thus the cache is a total of (X*S) blocks (1 block = 512 bytes). Each buffer in the cache corresponds to exactly one server, meaning that the blocks placed in a particular buffer will be sent and stored on its corresponding server. Such a block cache and the matching of of buffers and servers are illustrated in figure 1 for S=4 (3 storage servers and 1 parity server).

When a write request of X blocks arrives from the filesystem the client places the X blocks into one storage buffer, updating a map that shows where each block is stored. When all $S - 1$ storage buffers in the cache are full, the client computes the X parity blocks into the parity buffer, sends away all the buffers, each to its corresponding server, and then empties the cache.

2.3.3 Adaptive Parity Caching

Another problem we address is the issue of redundancy cleaning. Due to the fact that a block does not have a fixed position, but can be written anywhere on any server, each time a block is overwritten the old block must be invalidated. The invalid block cannot be cleared until all the data blocks in its parity group are invalid, because it is needed in case one of the blocks in its

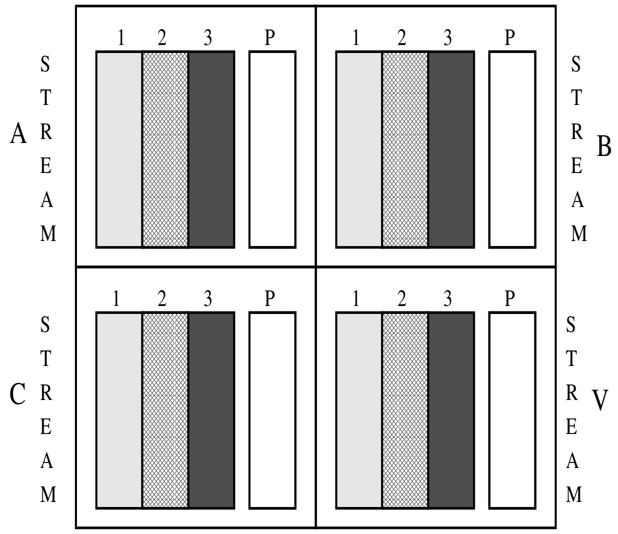

Figure 3: **The Adaptive Parity Cache.**

parity group is lost. Similar behavior is encountered also in the Log-structured FileSystem (LFS) [22, 25, 20], and raises questions about the use of redundancy cleaning.

Based upon our knowledge of disk devices and filesystems we believe that blocks whose block numbers are contiguous, belong either to the same file or to file(s) that are part of a data stream, and are likely to be written, read or rewritten together. If the rewritten blocks were placed in the same parity group, that would make the redundancy cleaning of our device automatic. Using this observation we extended our parity caching policy to an *adaptive parity caching policy* .

In the adaptive caching policy we use P parity caches as the one mentioned above. We call each cache, a stream cache, because it tries to capture a single data stream. Figure 3 shows an adaptive parity cache with 3 stream caches (A, B, C) and 1 victim stream cache (V).

So when the client has to choose a stream cache for placing the X blocks, it performs the following placement:

1. If all stream caches are empty, place the blocks in the first buffer of the first stream cache.
2. If there is one non-empty stream cache :
   (a) Search the non-empty stream caches if the X blocks are contiguous to a buffer already in a stream cache, in which case place the blocks into the first free buffer of the same cache.
   (b) If there is an empty stream cache, place the blocks into its first buffer.
   (c) If there isn't any match of the above place the blocks into the first free buffer of the victim (V) cache.

In case we are waiting on non-contiguous blocks, there is a stream cache timeout. If the victim cache is full *twice* and some caches have not filled yet, the client fills the remaining buffers in a stream cache with 0's, and processes the cache like it was full.

An example of the above policy is illustrated in figures 4 and 5 . In this example we assume that there are $S = 4$ servers, and each buffer is $X = 4$ blocks in size. The filesystem makes the following write requests (in block numbers) : (10,11,12,13), (30,31,32,33), (50,51,52,53), (70,71,72,73), (90,91,92,93), (34,35,36,37), (14,15,16,17), (54,25,26,27), and (74,75,76,77). Then

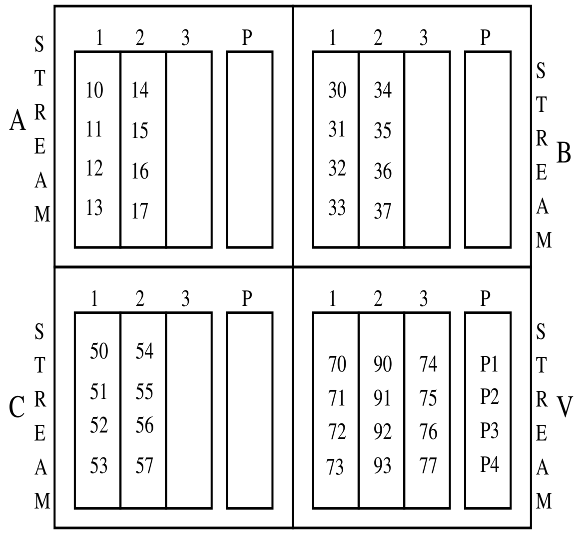

Figure 4: **Parity Cache example (Phase 1) :** Stream cache A accumulates blocks 10-17, stream cache B blocks 30-37, stream cache C blocks 50-57, and stream cache V (the victim stream cache) blocks 70-73,90-93 and 74-77.

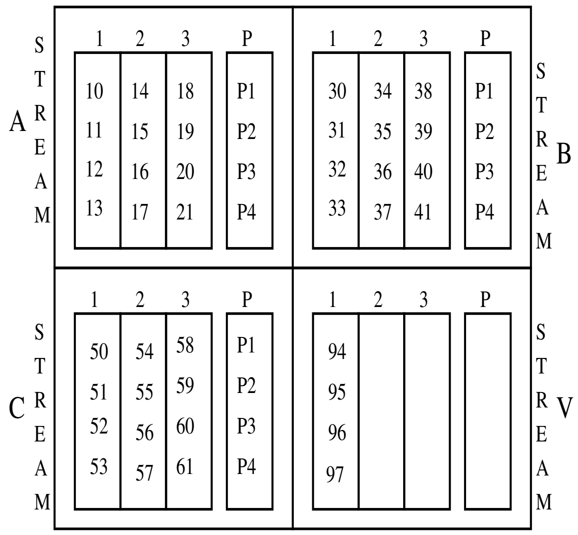

Figure 5: **Parity Cache example (Phase 2) :** Stream cache A accumulates blocks 10-21, stream cache B blocks 30-41, stream cache C blocks 50-61, and stream cache V (the victim stream cache) blocks 94-97. Each of the first three caches have captured a single block stream.

our cache would have the blocks placed as shown in figure 4 . The next write requests are : (18,19,20,21), (94,95,96,97), (38,39,40,41), (58,59,60,61), and our cache would be in the state of figure 5 .

It is clear that this placement policy creates many parity groups of contiguous blocks. Even when the filesystem writes in many different block streams, each stream is placed in the same stream cache. Moreover, this policy degrades gracefully when more than 4 different streams are written together. In the above example we used 5 streams and the cache managed to capture 3 of them, and mixed the last two in the best possible manner.

2.3.4 Simulation of Reliability Policies

To confirm the efficiency of the adaptive parity caching, we simulated the parity reliability policies on our trace-driven simulator [1], using the HP disk I/O traces [2] .

For the simulation we used 12 days of disk I/O traces of the most frequently accessed disk in the traces, and simulated 8 storage servers and one parity server, connected by an ATM network. We have simulated four policies :

- An Unreliable policy, which uses a block cache to reduce the device latency. This policy has the optimum performance.
- The Parity Caching policy as described in section 2.3.2.
- The Adaptive Parity Caching policy, defined in section 2.3.3, using two stream caches (P = 2).
- The Mirroring policy described in section 2.3.1. Mirroring requires 100% extra space.

The disk block size was 1024 bytes, and the parity caches stored 8 disk blocks for each server. The cleaning method we used, cleaned the most redundant parity groups first, by reading the full blocks and writing new non-redundant parity groups. The redundant parity groups were cleared. In the parity policies redundancy cleaning was performed *only* when the disk was full and empty space was needed.

The results of our simulations are presented in figures 6 and 7, and in the table 1. Figure 6 shows the execution time curve for each policy, including cleaning time for the parity caching policies. The basic conclusion derived from this figure is that the execution time for adaptive parity caching is only 11% worse of the optimum unreliable execution time, even when only 20% extra space is given.

Figure 7, which draws the curves of the cleaning time for the parity caching policies, shows clearly that the adaptive caching policy needs much less cleaning time than simple parity caching. For example, cleaning time for the adaptive parity caching is 30% lower than the cleaning time for simple parity caching, when we give 20% extra space. This also means that the adaptive parity caching policy generates less redundant parity blocks.

Table 1 shows execution times for a variable number of stream caches for 20% extra space. The adaptive parity caching policy with one stream cache is equivalent to the (non-adaptive) parity caching policy. It is very interesting to notice that although there is a significant improvement

[1] Our simulator consists of more than 5000 lines of C code. More details and sources of the simulator can be found at http://www.ics.forth.gr/ flouris/projects/pca/

[2] More details about the HP disk I/O traces can be found at http://www.hpl.hp.com/personal/John_Wilkes/traces . These traces were used in [23]

Number of stream caches	Execution time
1	8788.21 sec.
2	8520.39 sec.
3	8525.60 sec.

Table 1: Cleaning time using a variable number of stream caches for 20% extra storage space.

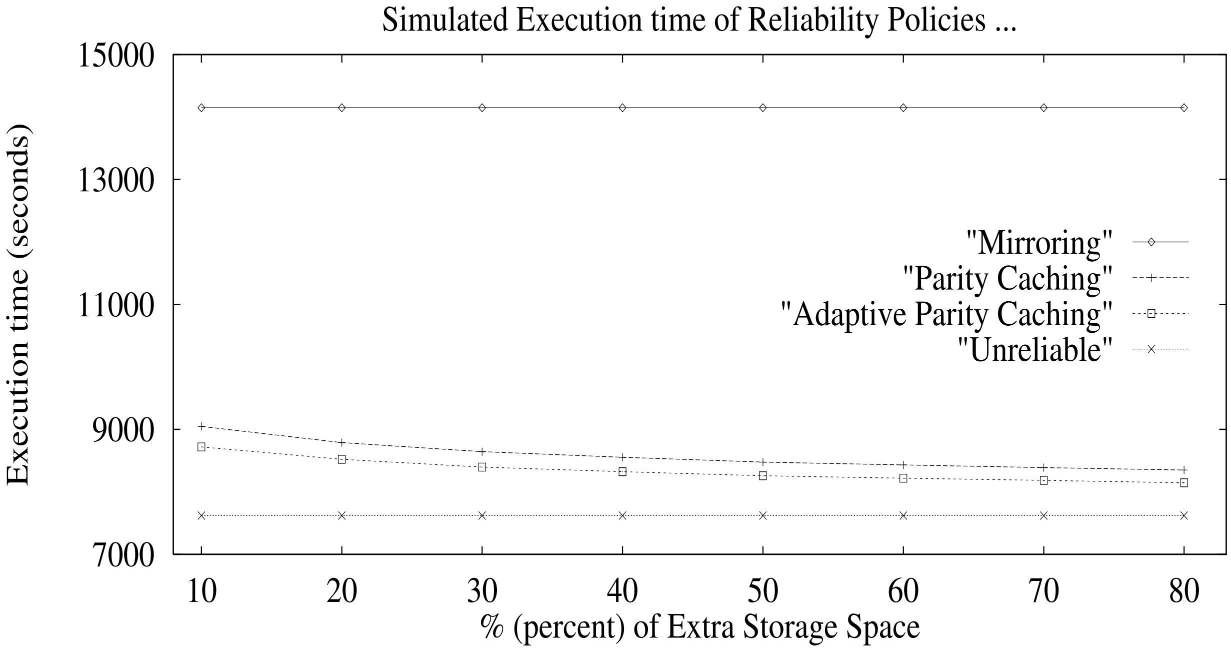

Figure 6: **Simulated execution (network) time for three reliability policies.** For the trace-driven simulation, we used 12 days of disk traces, and simulated 8 storage servers and one parity server for each parity policy. The time on the Y-axis is the network transfer time in seconds (on an ATM network) needed for executing all the traces.

from one to two stream caches, using more than two stream caches, practically does not affect performance.

3 Implementation

The Network RamDisk consists of a client issuing byte/block read and write requests and remote server processes satisfying these requests. Fully operational prototypes of the proposed system have been built, with clients on Linux 2.0.33 and the Digital Unix 4.0 Operating Systems, and servers running on Digital Unix 4.0 and Solaris 2.5 operating systems. The reliability schemes mentioned previously, were fully implemented, tested and measured on the Linux NRD client.

3.1 The Network RamDisk Client

The NRD client is a disk device driver that handles all read and write requests. In order to service these requests, it may forward them to user level NRD servers running on remote machines. The above design minimizes the modifications needed to port the system to another operating system,

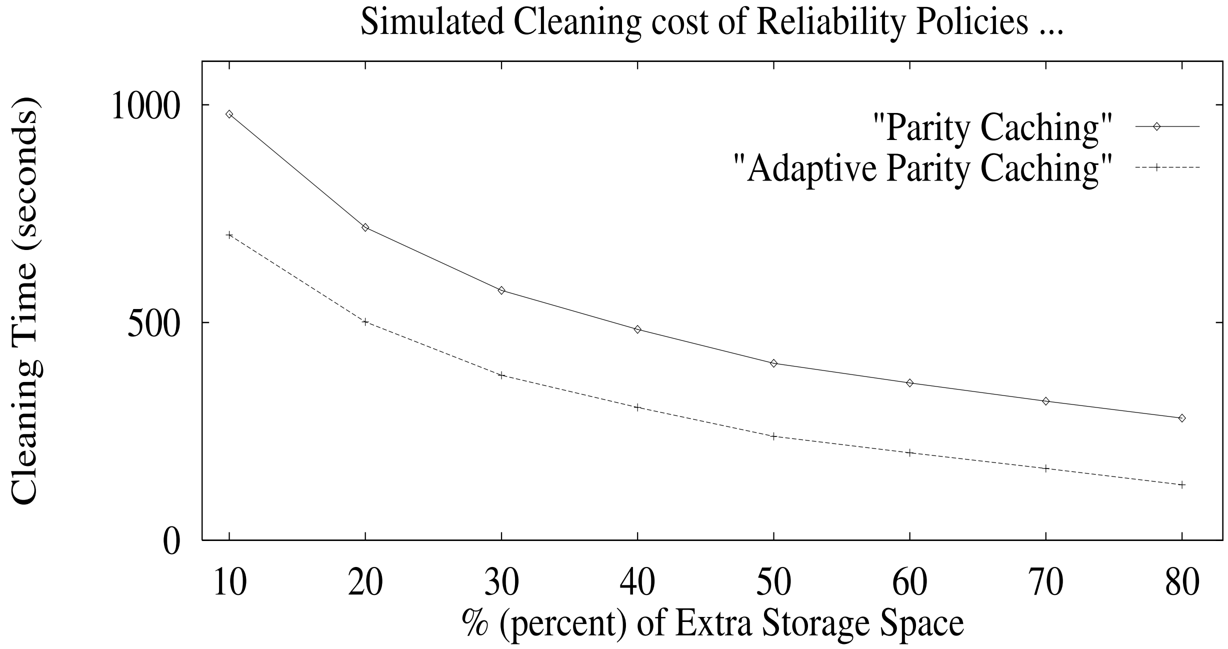

Figure 7: **Simulated Cleaning cost of Reliability Policies.** For the trace-driven simulation, we used 12 days of disk traces, and simulated 8 storage servers and one parity server for each parity policy. The cleaning time on the Y-axis is the network transfer time in seconds (on an ATM network) needed for cleaning the device (each time it was needed to write new blocks).

and avoids modifications to the operating system kernel. The Digital Unix Client has been linked with the Digital Unix 4.0 kernel of a DEC-Alpha 3000 model 300 with 32 MB main memory, while the Linux Client was linked with the Linux 2.0.33 kernel of a Pentium II 233 MHz PC, with 64 MB of RAM.

When the NRD client starts, is passes through a configuration phase where it finds out which servers it will use, how many blocks each server will store, which reliability or caching policy will be used, and then initializes its data structures. The NRD client keeps two maps in its memory, a block table with the location of each block, and a bitmap with the full or empty blocks on each server. The size of these maps is quite small. For example for every 1024 byte block and a maximum of 256 servers, we need 5 bytes for the location of each block (1 byte for the server and 4 bytes for the block number on that server), so for a NRD of 256 MB the block table is only 1.25 MB.

After the configuration phase, the NRD client connects to the NRD servers, and functions as a normal disk accepting block I/O requests from any filesystem that is created on it. Depending on the reliability policy used the client issues block read/write requests to the servers using TCP/IP sockets over the network interface(s) the NRD client machine has installed. Security is ensured by allowing access to our device only to the superuser and by using privileged ports for the communication among the NRD client and the servers.

The operating system is not aware that we use remote main memory instead of magnetic disk as a disk I/O device. It just performs ordinary block I/O activities through the Virtual File System (VFS) using what it considers a disk, mainly due to the device driver's response to all normal ioctl() commands of an ordinary magnetic disk. The disk's geometry (capacity, cylinders, tracks, sectors, etc.) are customizable from the driver's ioctl interface. We also added a disk type entry to the /etc/disktab (or /etc/fstab) file of our experimental NRD client machines, so that the disklabel (or the fdisk) command could easily identify the NRD as a disk. The amount of blocks transferred with each request to the NRD server, is also a matter of disk's geometry and can be adapted according to the current interconnection network, so that the performance is optimum.

The filesystem we used on our Network RamDisk is the classic UFS (Unix FileSystem) on Digital Unix (exists on most Unix-based operating systems), and the Ext2 filesystem on Linux. We are aware that these do not take full advantage of our disk being a Network RamDisk, however we wanted to measure performance resulting purely from the device and not from the filesystem on it. Moreover using a common filesystem, allowed us to use the ordinary system administrative tools for disks (disklabel, newfs, fdisk, mkfs, mount/umount, fsck, df, etc.) to install, create, and test the filesystem on our device. It is quite certain that the performance would be much higher if a more efficient FS was used (e.g. AdvFS). This is an area we could experiment in the future.

The current implementation of the Digital Unix Network RamDisk client contains only an unreliable version, and runs on top of a low-bandwidth 10 Mbps Ethernet.

The current Linux Network RamDisk client implementation contains three different reliability/caching policies:

1. A simple unreliable version similar to the Digital Unix implementation.

2. An unreliable version, which uses caching of blocks and large batched network transfers to reduce network latency. This version has the optimum performance.

3. An adaptive parity caching version mentioned in section 2.3.3 . This version performs also quite well as we will see later in our experiments.

All these versions of the Linux Network RamDisk client have been tested on top of a 155 Mbps ATM network. Lower latency and higher bandwidth networks like Gigabit ATM, SCI (Scalable Coherent Interface) [17] or Myrinet [5] should offer even more promising performance, especially when faster communication protocols are used [26, 2].

3.2 The Network RamDisk Servers

The Network RamDisk server is a user level program listening to a socket and accepting connections from the NRD clients. Each client is served by a new instance of the server which accesses the same portion of the local workstation's main memory to store the client's disk blocks. When the client makes a read request, the server transfers the requested block(s) over the socket. When the client makes a write request, the server reads the incoming block(s) from the socket, and stores them in its main memory. The server is also responsible for providing periodically (or on demand) information to the client concerning its memory load. An NRD parity server process is by no means different than a storage NRD server one. It just performs block read and write requests responding to client requests without knowing whether it stores memory blocks or parity blocks.

The implementation of the NRD server as a user process offers the advantage of using machines with different Unix-based operating systems as servers, because the porting of such user-level programs is very simple. We have used machines running Digital Unix 4.0 and Solaris 2.6 as NRD servers, and have also ported the server side on the SunOS and Linux operating systems.

4 Experiments

To test the operability, evaluate the performance of our Network RamDisk prototype described above, and compare it to traditional disk I/O, we conducted a series of performance measurements

Device	Time
Magnetic Disk	129.8 sec.
Network RamDisk	103.6 sec.
Execution time improvement	25%

Table 2: Find and grep performance

Device	Time
Magnetic Disk	120 sec.
Network RamDisk	67 sec.
Execution time improvement	79%

Table 3: Tar file-unpacking performance

using a number of representative benchmarks and applications, that depend in various ways and capacities on the disk I/O performance. Our applications include searching the files on the disk for a particular string using `find` and `grep`, and using `tar` to unpack the archive file of Gnu Compiler version 2.7.2. Our benchmarks test various important aspects of filesystem performance, such as write throughput and filesystem latency (create/delete many small files).

We have tested both the Digital Unix client which was built first and implements only an unreliable NRD, and the Linux client which implements also the adaptive parity caching reliability policy described in section 2.3.3 .

4.1 The Digital Unix NRD Client

4.1.1 Experimental Environment

All applications and benchmarks for the Digital Unix NRD client were executed from user level on the DEC-Alpha 3000 model 300 with 32MB of main memory running Digital Unix 4.0, and were compiled with the standard system C compiler (cc). The workstations that contributed their main memory for storing the Network RamDisk blocks were DEC-Alpha 3000 model 300 also with 32 MB of main memory, connected via a standard 10Mbits/sec Ethernet. In all experiments the amount of idle memory was larger than the amount of memory needed for storing the Network RamDisk blocks, and was equally distributed among all workstations. During the test we tried to keep the Ethernet as idle as possible, in order to gain the maximum bandwidth from the 10 Mbits/sec available. The local disk that was used for testing and comparison is a DEC RZ55, providing 10Mbits/sec bandwidth, and average seek time of 16 msec. Although both the disk and the interconnection network are slow, they have comparable bandwidth, which is what we would expect from a more modern disk and a more modern network.

4.1.2 Performance of “find” and “grep”

We have measured the file read performance of the Network RamDisk using the `find` Unix command. First we copied a directory tree (specifically the sources of Gnu Compiler version 2.7.2), which `du` reported to be `28 MBytes` in size and having `1075` files, into the two devices, the Network RamDisk and the magnetic disk. We cleared the filesystem cache from the recently written blocks, by reading the whole directory tree of /usr which is `280 MBytes` large. Using a

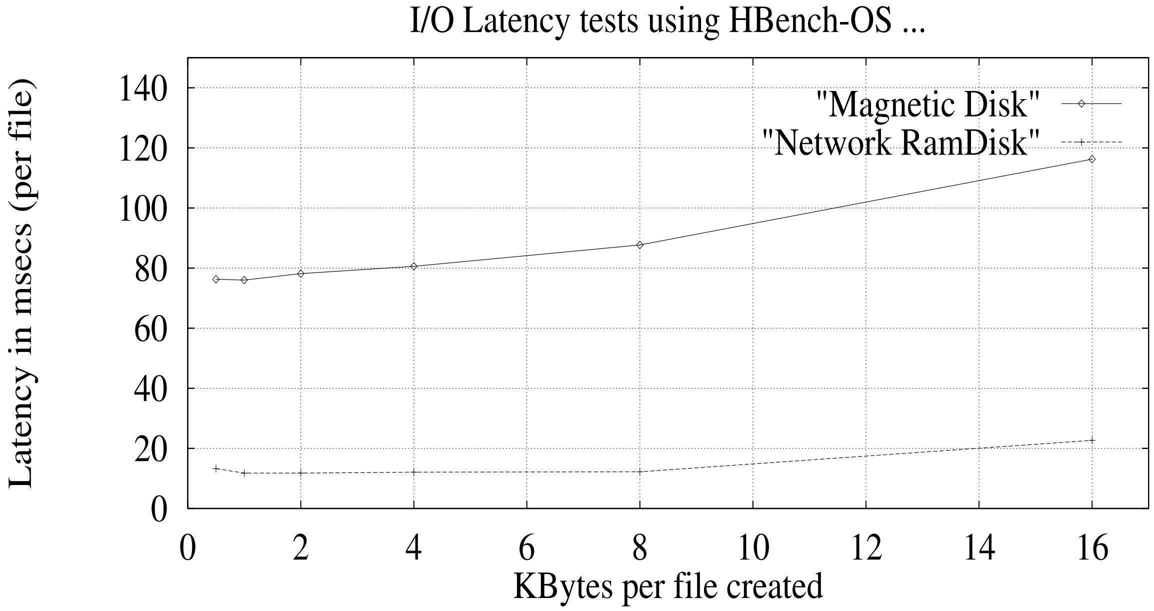

Figure 8: **Comparison of the I/O Latency performance of Magnetic Disk vs. Network RamDisk, using HBench-OS:** *This test measures the latency of performing certain file system metadata operations, in particular file creates.The latency shown is the number of milliseconds (ms) per file creation for a total of 2000 on each of the devices. The benchmark were executed from user level on the DEC-Alpha 3000 model 300 with 32MB of main memory running Digital Unix 4.0, and compiled with the standard system C compiler (cc). The Network RamDisk used as servers two DEC-Alpha 3000 model 300 workstations, each with 32 MBytes of main memory.*

simple shell script we searched all the files in the directory tree for a non-existing string, using the applications `find` and `grep`. The results shown in Table 2, point out that it takes 129.8 seconds for `find` and `grep` for the search while on the Network RamDisk it takes only 103.6 seconds. We see that our application is `25%` faster when run on top of the NRD.

4.1.3 Using "tar" to unpack files

Trying to measure the write performance of real applications we used the Unix `tar` command used to unpack many files from a single archive. We must note that `tar` does not compress the files, it simply packs a whole directory tree into a single file. As a test file we used the sources of `Gnu Compiler version 2.7.2`, a popular package used in most Unix machines. The directory tree contained in the tarfile was `28 MBytes` in size, and had `1075` files. We timed the command to unpack the archive file into the Network RamDisk and the magnetic disk. Table 3 presents the results. Thus the application is `79%` faster on the NRD, than on the magnetic disk. This performance advantage can be exploited by many applications which do many sequential file writes, such as visualizing tools, proxies and databases.

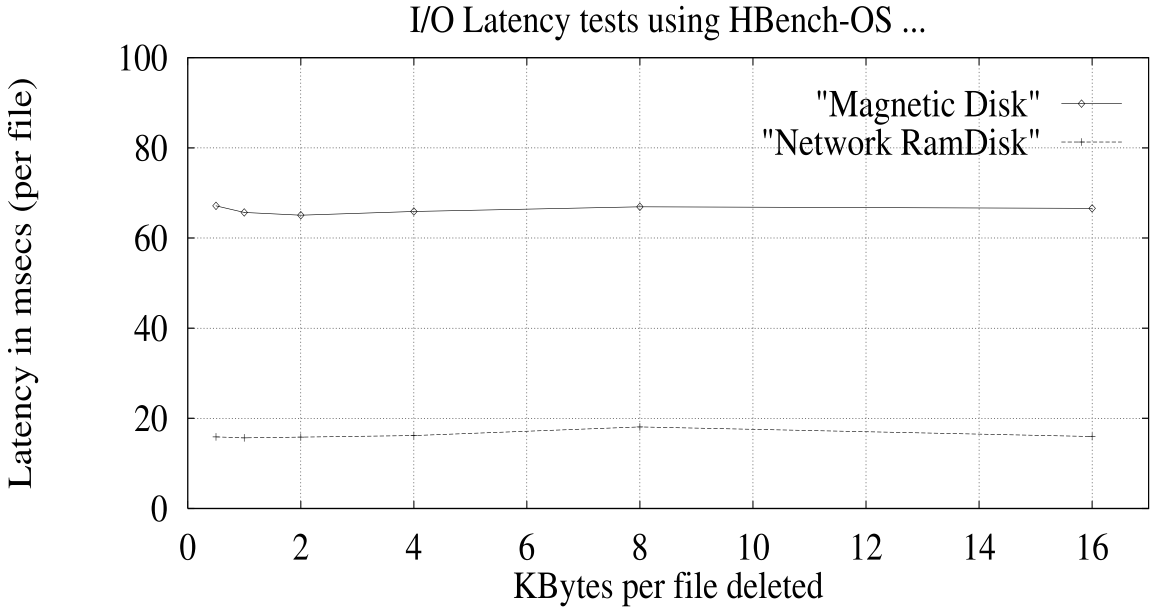

Figure 9: **Comparison of the I/O Latency performance of Magnetic Disk vs. Network RamDisk, using HBench-OS:** *This test measures the latency of performing certain file system metadata operations, in particular random file deletes.The latency shown is the number of milliseconds (ms) per file deletion for a total of 2000 on each of the devices. The benchmark were executed from user level on the DEC-Alpha 3000 model 300 with 32MB of main memory running Digital Unix 4.0, and compiled with the standard system C compiler (cc). The Network RamDisk used as servers two DEC-Alpha 3000 model 300 workstations, each with 32 MBytes of main memory.*

4.1.4 I/O Latency using HBench-OS

To measure the I/O Latency of the Network RamDisk and compare it with the disk, we ran a microbenchmark program called lat_fs which is part of the HBench-OS Benchmark Suite [3] . Lat_fs measures the latency of performing certain file system metadata operations, in particular file creates and deletes. We ran the benchmark both for 2000 file creations and deletions, with variable file size. The resulting graphs are depicted in Figures 8 and 9. We see that the latency advantage of the Network RamDisk makes the performance improvement for this benchmark very high. For example for 2000 file creations of 16 KBytes each, the creation time for each file on the magnetic disk is equal to `116.22` milliseconds (or `8.60` files per second), whereas on the Network RamDisk the creation time is `22.68` milliseconds (or `44.09` files per second). This is a `412%` real time improvement for a 16 KByte file creation.

4.1.5 Performance Test of Sequential File I/O using IOZONE

To compare the performance of sequential file I/O of our device over the Ethernet with the traditional magnetic disk, we measured it using a microbenchmark called IOZONE, V2.01 by Bill Norcott. This benchmark writes a selectable-sized file in small blocks and measures the time to perform these operations. Such workload of writing a large file sequentially in small blocks, is applied

[3] More information on the HBench-OS Benchmark Suite can be found at http://www.eecs.harvard.edu/vino/perf/hbench/

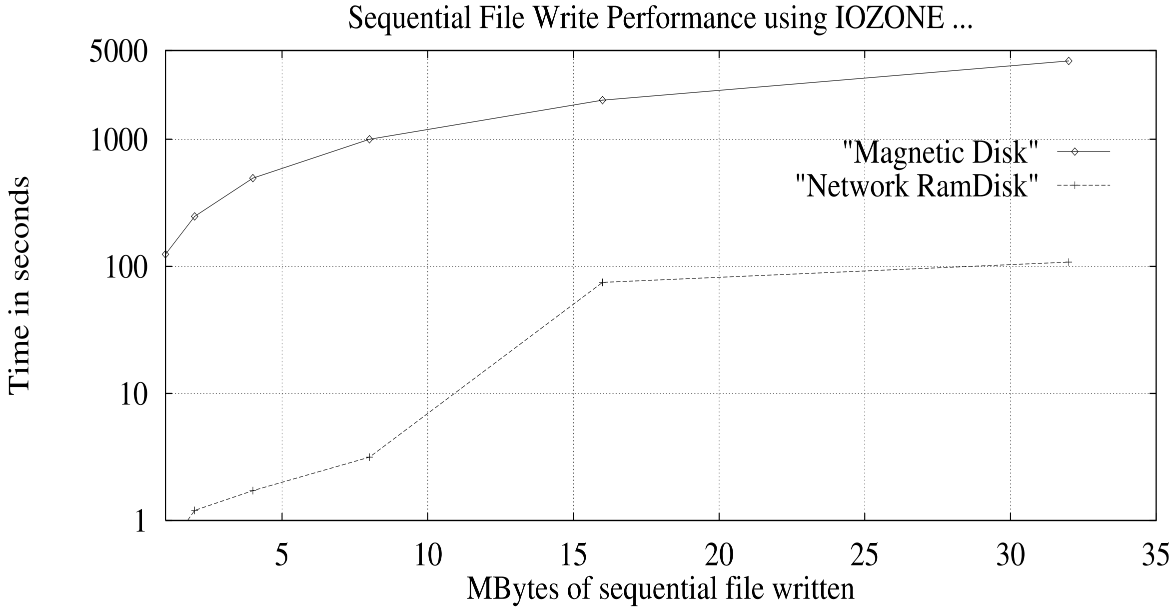

Figure 10: **Comparison of Sequential File I/O on either the disk, or the Network RamDisk:** *In this test we measured the performance of Sequential File I/O, using a microbenchmark called IOZONE, V2.01 by Bill Norcott. We see that the use of remote memory results in significantly higher sequential file I/O performance compared to the magnetic disk. All benchmarks were run on a DEC-Alpha 3000 model 300 workstation, under Digital Unix 4.0. The Network RamDisk used as servers two DEC-Alpha 3000 model 300 workstations, each with 32 MBytes of main memory.*

by applications like multimedia systems (e.g. writing frames of video or audio), compression programs, and databases.

The results and the comparison graph can be seen in Figure 10. It is obvious from the graph that the Network RamDisk even over the Ethernet, and using TCP/IP has **nearly three times higher sequential file I/O performance** .

4.1.6 I/O Performance using IOstone Benchmark

To compare the I/O performance of our device over the Ethernet with the traditional magnetic disk of the same bandwidth we ran the IOstone Benchmark.

IOstone is meant for benchmarking file I/O and buffer cache efficiencies. It does this by creating NSETS (4) of SET_SIZE (99) files. Then iostone performs I/O on each file in each set. The type of I/O is randomly picked (read or write).

More detailed description of the benchmark process is :

- Allocates space for the filenames of NSETS (4) of SET_SIZE (99) files each, and creates the filenames.
- Reads in some files to flush the buffer cache.
- Order filenames in a random permutation for reading/writing.
- Start the timer (using ftime()).
- Perform reads/writes (read:write = 2:1) to the files ITER (4) times.

Device	IOstones/sec
Disk	31854
Network RamDisk	269651
Exec. time improvement	746.5%

Table 4: IOstone Benchmark performance

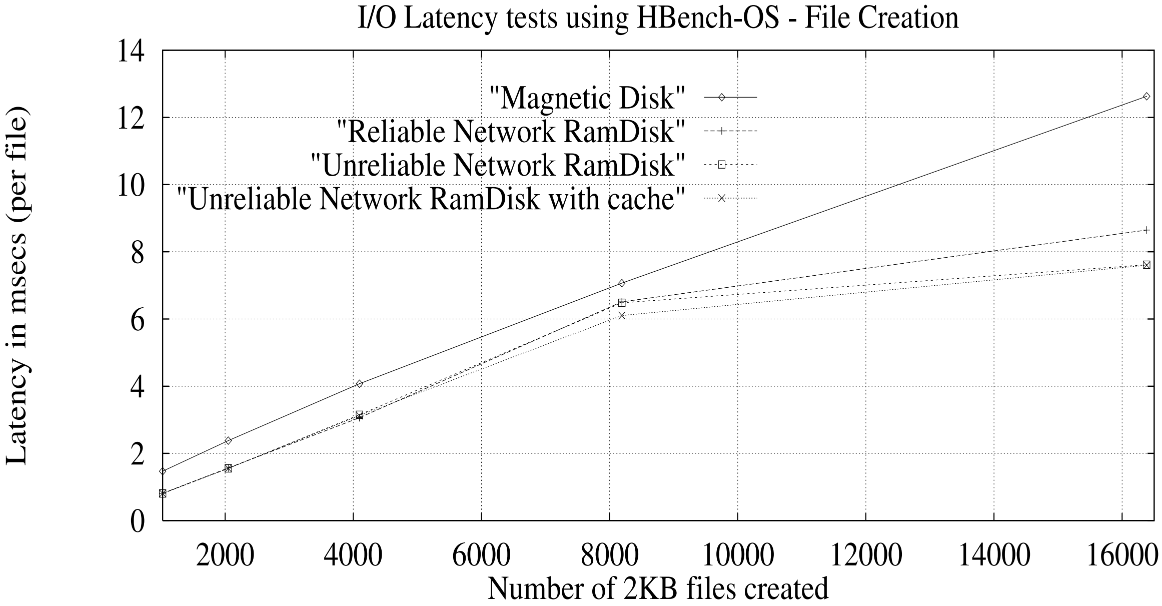

Figure 11: **Comparison of the I/O Latency performance of the Linux NRD Client on the Magnetic Disk vs. Network RamDisk over the Ethernet and the ATM interconnection network, using HBench-OS:** *This test measures the latency of performing certain file system metadata operations, in particular file creates.The latency shown is the number of milliseconds (ms) per file creation for a variable number of 2KB file creations on each of the devices. The benchmark were executed from user level on a Pentium II 233Mhz PC with 64MB of main memory running Linux 2.0.33, and compiled with the GNU C compiler (gcc 2.7.2.1). The Network RamDisk used as servers two Sun Ultra 1 workstations, each with 128 MBytes of main memory and Sun ATM NIC, running Solaris 2.6 .*

- Stop the timer and prints the iostones of the system (a constant divided by the number of msec that were measured).

The comparison of IOstones/second can be seen in Table 4. It is obvious from the table that the Network RamDisk even over the Ethernet, and using TCP/IP has **over seven times higher performance**, than the magnetic disk.

4.2 The Linux NRD Client

4.2.1 Experimental Environment

The Linux NRD Client is a fully operational prototype which implements three reliability/caching policies, as mentioned above : a simple unreliable version, an unreliable version with block caching, and a reliable version with parity caching. We have run experiments on all three versions of the

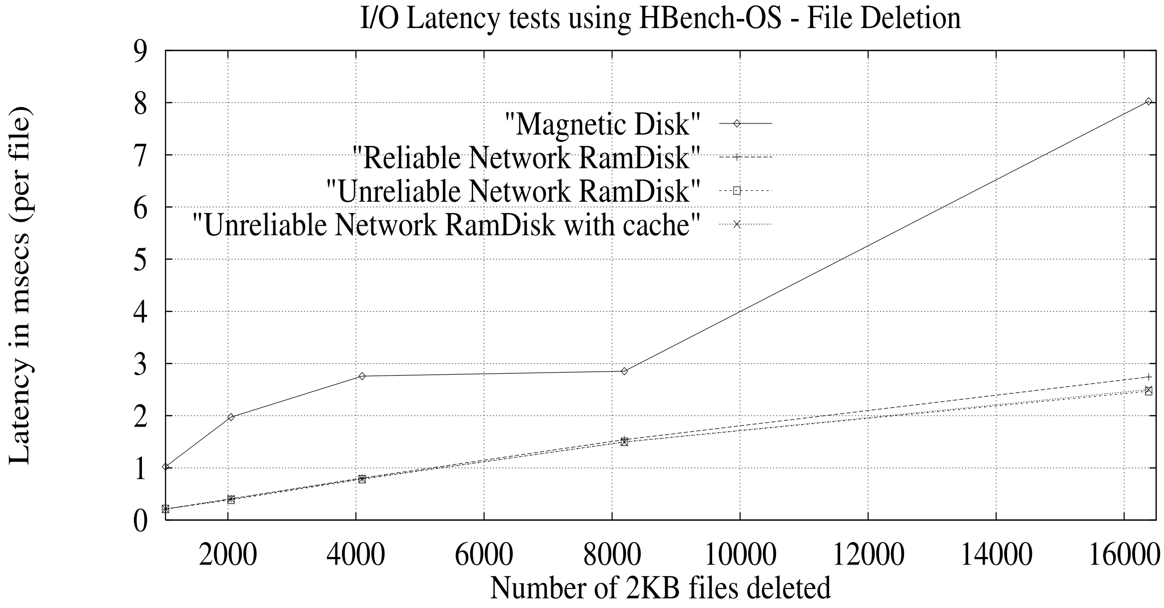

Figure 12: **Comparison of the I/O Latency performance of the Linux NRD Client on the Magnetic Disk vs. Network RamDisk over the Ethernet and the ATM interconnection network, using HBench-OS:** *This test measures the latency of performing certain file system metadata operations, in particular random file deletions. The latency shown is the number of milliseconds (ms) per file deletion for a variable number of 2KB file deletions on each of the devices. The benchmark were executed from user level on a Pentium II 233Mhz PC with 64MB of main memory running Linux 2.0.33, and compiled with the GNU C compiler (gcc 2.7.2.1). The Network RamDisk used as servers two Sun Ultra 1 workstations, each with 128 MBytes of main memory and Sun ATM NIC, running Solaris 2.6 .*

NRD client.

The experiments on the Linux client were conducted on a 233MHz Pentium II PC with 64MB of RAM, a 2GB Western Digital Caviar EIDE Hard Disk, and a FORE PCA200E 155MBps ATM NIC, running Linux 2.0.33 . For the ATM experiments we have set up a LANE (LAN Emulation) ATM subnet, using the experimental Linux-ATM distribution 0.31 [4] , along with the also experimental FORE PCA200E 0.2 Linux device driver [5] .

It is important to mention that the Linux-ATM distribution 0.31 is a development distribution and had quite unstable performance, closing the open network connections often, and sometimes dramatically degrading the performance of the ATM network. Under a more stable ATM network we strongly believe that the NRD client would have much better performance.

4.2.2 I/O Latency using HBench-OS

To measure the I/O Latency of all versions of the Network RamDisk over the ATM (LANE) interconnection network and compare it with the magnetic disk performance, we ran the lat_fs microbenchmark program creating and deleting a variable number of 2KB files, measuring the latency for each file reported by the benchmark.

The results of lat_fs for the file creations on all versions of the Linux NRD Client on the ATM,

[4]More information on the Linux-ATM distribution can be found at http://lrcwww.epfl.ch/linux-atm.

[5]Information about the FORE PCA200E device driver can be found at http://os.inf.tu-dresden.de/project/atm.

and the magnetic disk are illustrated on figure 11 . It is obvious that every version of the Network RamDisk has much lower latency on the ATM, than the magnetic disk. For example for 16384 file creations of 2 KBytes each, the creation time for each file on the magnetic disk is equal to 12.629 milliseconds (or 79 files per second), whereas with the unreliable version of Linux NRD on the ATM with block caching, the creation time is 7.601 milliseconds (or 131 files per second). The results of lat_fs for the random file deletions are depicted in figure 12 , where we can see that the performance of all versions of the NRD client is again superior compared to the the magnetic disk.

An important conclusion derived from figures 11 and 12 is that the reliable version of the Linux NRD client with the adaptive parity caching policy, performs very close to the unreliable version. We conclude that the NRD can provide reliability, retaining its high performance.

5 Previous Work

The idea of using main memory as a fast block storage device is not new. Supercomputers and databases frequently use *solid state disks*, as performance accelerators. These "disks" are peripheral devices filled with DRAM chips that behave like regular block storage devices. Since solid state disks have high throughput and zero seek time, they are perfectly suitable for I/O intensive applications. In our work, we organize main memories to behave much like a solid state disk. The main advantage of our approach, compared to solid state disks, is its low cost, since we exploit the (otherwise) unused main memory that already exists in a workstation cluster. A network RamDisk may be used in several cases to store temporary data, including a web cache, intermediate compilation files, "/tmp" files, etc. Reliable versions of the Network RamDisk can be used to provide low-latency high-bandwidth storage to applications that need it, like databases, storage managers, etc.

Several operating systems provide software RamDisks. A software RamDisk is a part of a computer's main memory, that is being used as a block storage device [21]. Such RamDisks are being used to store temporary files in order to improve system performance. Our work extends the notion of a software RamDisk into a Network of Workstations. Instead of using the main memory of a single computer as the RamDisk does, we use the collective main memory of several computers in a NOW as a network RamDisk. Furthermore, we allow users to configure the network RamDisk using various reliability policies, so that in the event of a computer crash, the contents of the network RamDisk will not be lost. On the contrary, the contents of traditional software RamDisks are lost in the event of a crash.

Several research groups have proposed various methods of exploiting network main memory to avoid disk accesses. In remote memory paging systems, for example, applications use the remote memory as swap area [2, 3, 8, 12, 16, 24]. As a generalization to remote memory paging, several researchers propose to use the remote main memory as a file system cache [1, 9, 10, 11, 15]. For example, Feeley *et al.* have implemented a global memory management system (GMS) in a workstation cluster, using the idle memory in the cluster to store clean pages of memory loaded workstations [11]. Anderson *et al.* have implemented xFS, a serverless network file system [1, 10]. Both GMS and xFS have been incorporated inside the kernel of existing operating systems and their performance has been demonstrated. Our approach differs from the above in that the Network RamDisk can be easily incorporated into an existing commercial operating system without any kernel changes. Both GMS and xFS have been closely integrated within their underlying operating system kernel (although a device-driver implementation for xFS has been reported [13]). Thus, in order for users to be able to exploit the performance benefits of remote memory, they have to use the GMS and/or xFS systems and their associated operating system kernels. On the contrary, with our approach, ordinary file systems (e.g. NFS, UFS) are able to exploit the benefits of remote memory. Thus, an existing file system (like UFS) can be easily turned into a network memory file system without any modifications to it.

Moreover, we believe that the Network RamDisk due to its simple implementation has the potential to perform better than Network Memory Filesystems (e.g. like xFS). For example, using NFS over an xFS distributed filesystem, a block read operation, will require 2 network accesses if the data block exists in the client's filesystem cache, or 5 network accesses if it resides on another client's cache (the client would have to query the block's manager first and then the storage server holding the block). The data block write operations would need in xFS, from 2 to 4 network accesses (depending on whether the client contacted has write ownership of the data block). Accessing data blocks using NFS over our device would need 4 network accesses. Even in this case, because of its simplicity and portability, the memory and runtime overhead of our driver would be much lower (we do not use distributed maps, etc.), resulting in faster real time responses to I/O requests. Network Memory Filesystems (like xFS) may offer scalability, but our approach offers simplicity and efficiency.

6 Conclusions

In this paper we propose a new device : the Network RamDisk. This is a block storage device that consists of (idle) main memory of workstations within a (heterogeneous) workstation cluster. We describe our prototype implementations of a Network RamDisk implemented on top of the Digital Unix 4.0, and the Linux operating systems as a device driver. No modifications were made to the kernel of the Digital Unix 4.0, or the Linux operating system.

The contributions of this paper are:

- We describe how to build a remote memory disk storage system. The major advantage of our design is the portability it offers. Our device driver can be installed on any Digital Unix or Linux system, and can exploit the main memory of *any workstation supporting TCP/IP*.

- We propose a new adaptive parity reliability strategy for distributing disk blocks to many servers. Studying the performance of our policy, we show that it effectively reduces the time for redundancy cleaning, thus providing reliability at a very low performance cost.

- We show that storing disk blocks to remote memory results in substantial performance improvements over the local magnetic disk, mostly due to latency reduction.

- We provide a new efficient way of *making remote memory available to every application* through the common filesystem structures.

Based on our implementation and our performance results we conclude:

- *Storing data to the Network RamDisk results in significant performance improvement over storing them on the traditional magnetic disk.* Applications that use our Network RamDisk even when running on top of *traditional* Ethernet technology show performance improvements of up to 412% (Figure 8).

- *Organizing remote memory as a Network RamDisk under any ordinary filesystem, is an inexpensive way to let applications use the memory of any workstations in a heterogeneous* *cluster*. Everyday applications can now use remote memory for their data files without modifying their code, under any Unix filesystem.

- *Remote memory data storage provides good performance and reliability* with almost no extra hardware support.
- *The benefits of storing data blocks to remote memory will only increase with time*. Current architecture trends suggest that the gap between processor and disk speed continues to widen. Disks are not expected to provide the latency needed by several applications unless a breakthrough in disk technology occurs. On the other hand, interconnection network bandwidth keeps increasing (and network latency decreasing) at a much higher rate than (single) disk bandwidth, thereby increasing the performance benefits of the remote memory disk storage.

Based on our performance measurements we believe that using remote memory as a Network RamDisk is a cost-effective and performance-effective way to execute I/O intensive applications on a network of heterogeneous workstations.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[2] T.E. Anderson, D.E. Culler, and D.A. Patterson. A Case for NOW (Networks Of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[3] G. Bernard and S. Hamma. Remote Memory Paging in Networks of Workstations. In *Proceedings of the SUUG International Conference on Open Systems: Solutions for Open Word*, April 1994.

[4] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. 21-th International Symposium on Comp. Arch.*, pages 142–153, Chicago, IL, April 1994.

[5] N.J. Boden, D. Cohen, and W.-K. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29, February 1995.

[6] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Second Symposium on Operating System Design and Implementation*, pages 245–259, October 1996.

[7] P.M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[8] D. Comer and J. Griffioen. A new design for Distributed Systems: the Remote Memory Model. In *Proceedings of the USENIX Summer Conference*, pages 127–135, 1990.

[9] T. Cortes, S. Girona, and J. Labarta. PACA: A Cooperative File System Cache for Parallel Machines. In *2nd International Euro-Par Conference (Euro-Par'96)*, pages 477–486, 1996. Lecture Notes in Computer Science 1123.

[10] M.D. Dahlin, R.Y. Wang, T.E. Anderson, and D.A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *First Symposium on Operating System Design and Implementation*, pages 267–280, 1994.

[11] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proc. 15-th Symposium on Operating Systems Principles*, pages 201–212, December 1995.

[12] E. W. Felten and J. Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical Report 91-03-09, Computer Science Department, University of Washington, November 1991.

[13] D.P. Ghormley, D. Petrou, and T.E. Anderson. SLIC: Secure Loadable Interposition Code. URL: http://now.CS.Berkeley.EDU/Slic/.

[14] R. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12–18, February 1996.

[15] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. *Proc. 14-th Symposium on Operating Systems Principles*, pages 29–43, December 1993.

[16] L. Iftode, K. Li, and K. Petersen. Memory Servers for Multicomputers. In *Proceedings of COMPCON 93*, pages 538–547, 1993.

[17] D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi. Scalable Coherent Interface. *IEEE Computer*, 23(6):74–77, June 1990.

[18] E.P. Markatos and G. Dramitinos. Implementation of a Reliable Remote Memory Pager. In *Proceedings of the 1996 Usenix Technical Conference*, pages 177–190, January 1996.

[19] E.P. Markatos and M. Katevenis. Telegraphos: High-performance networking for parallel processing on workstation clusters. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, pages 144–153, February 1996.

[20] J.N. Matthews, D. Roselli, A.M. Costello, R.Y. Wang, and T.E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proc. 16-th Symposium on Operating Systems Principles*, October 1997.

[21] A.K. McKusick, K.J. Karels, and K. Bostic. A Pageable Memory Based Filesystem. In *Proceedings of the Summer 1990 Usenix Technical Conference*, pages 137–145, June 1990.

[22] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[23] Chris Ruemmler and John Wilkes. UNIX disk access patterns. In *Proceedings of the Winter'93 USENIX Conference*, pages 405–420, January 1993.

[24] B.N. Schilit and D. Duchamp. Adaptive Remote Paging for Mobile Computers. Technical Report CUCS-004-91, University of Columbia, 1991.

[25] M. Seltzer, M. K. McKusick, K. Bostic, and C. Staelin. An implementation of a log-structured file system for unix. In *Proceedings of the 1995 Winter Usenix Technical Conference*, San Diego, CA, January 1993.

[26] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. 19-th International Symposium on Comp. Arch.*, pages 256–266, Gold Coast, Australia, May 1992.