

The MOSIX Direct File System Access Method for Supporting Scalable Cluster File Systems

Lior Amar, Amnon Barak * and Amnon Shiloh
Institute of Computer Science,
The Hebrew University of Jerusalem
Jerusalem, 91904 Israel

Abstract

MOSIX is a cluster management system that supports preemptive process migration. This paper presents the MOSIX Direct File System Access (DFSA), a provision that can improve the performance of cluster file systems by allowing a migrated process to directly access files in its current location. This capability, when combined with an appropriate file system, could substantially increase the I/O performance and reduce the network congestion by migrating an I/O intensive process to a file server rather than the traditional way of bringing the file's data to the process. DFSA is suitable for clusters that manage a pool of shared disks among multiple machines. With DFSA, it is possible to migrate parallel processes from a client node to file servers for parallel access to different files. Any consistent file system can be adjusted to work with DFSA. To test its performance, we developed the MOSIX File-System (MFS) which allows consistent parallel operations on different files. The paper describes DFSA and presents the performance of MFS with and without DFSA.

Keywords: cluster computing, cluster file systems, consistent file system, massive parallel I/O

1 Introduction

Recent advances in cluster computing, including the ability to create and assign parallel processes dynamically to many machines, have created a need to develop scalable cluster file systems that not only support parallel access to many files, but also provide consistency between processes that access the same file. Most traditional network file systems such as NFS [19], AFS [1] and Coda [10] are inadequate for scalable parallel processing because they do not provide cache consistency. A new generation of file systems, such as PolyServe's Matrix Server cluster file system [16], Sistina's Global File System (GFS) [12], IBM's GPFS [20], SGI's CXFS [21] and xFS [4] are more appropriate for clusters because these systems distribute storage, cache and control among the cluster's nodes and provide means for parallel file access and cache consistency. However, except xFS, these file systems require shared hardware, which is more expensive and likely to limit their scalability.

This paper presents an extension of MOSIX [17] that combines cluster file systems with dynamic work distribution. The target cluster consists of multiple nodes (which can have any combination of single CPU and/or SMP) of compatible architecture, e.g., all x86 based, that work cooperatively, availing the cluster-wide resources to each process. The cluster file system consists of several sub-trees that are placed (off-line) in different nodes, to allow parallel operations on different files. The unique feature of our scheme, which currently is supported by only few production systems, is the ability to bring

* Copyright (c) 2003 Amnon Barak. All rights reserved

(migrate) a process to the file rather than the traditional way of bringing the file's data to the process. As we shall show, this capability creates a scalable, easy to use cluster computing environment that provides a significant performance improvements over traditional methods. We note that our scheme can improve the performance even when used with existing packages for static work distribution, such as MPI [18] or PVM [11], which by themselves do not support process migration.

MOSIX is a cluster management system that supports a preemptive process migration. MOSIX is particularly efficient for load-balancing CPU-bound processes. However, in order to maintain complete Unix compatibility, the MOSIX scheme became inefficient for running processes with significant amount of I/O and/or file-system operations. To overcome these inefficiencies, MOSIX was enhanced with a provision for Direct File System Access (DFSA), in which most I/O oriented system-calls of a migrated process can be performed directly on the node where the process currently resides, as opposed to being redirected to the node in which the process was created. The main advantage of DFSA is to allow a process with a moderate to high volume of I/O to migrate to the node in which it performs most of its I/O, to take advantage of accessing local data. Other advantages include a reduction of the communication overheads and network congestion, since less I/O operations use the network; increased scalability and better scope for migrating I/O-bound and mixed processes that require a significant amount of both I/O and CPU.

Correct operation of DFSA requires a *single-node consistency* among processes that run on different nodes, which currently only a few distributed file-systems provide [12, 16, 20, 21]. To demonstrate the performance of DFSA with a low-cost, consistent file system that does not require shared hardware, we developed the MOSIX File-System (MFS), a prototype file system that places all files and directories within a MOSIX cluster under a single directory.

One of the results is that if a very large file is partitioned to several smaller files that are placed in different nodes, then it is possible to migrate parallel processes from a client node to the respective nodes to enable parallel local access to the different files. The MOSIX Parallel I/O (MOPI) [2] scheme is an example of such a system.

1.1 Contribution and organization of the paper

This paper demonstrates an operational system that improves performance by migrating I/O-bound processes to file servers versus the traditional methods of bringing the data to the process.

The paper is organized as follows: Section 2 presents a short overview of MOSIX. Section 3 presents DFSA and Section 4 presents MFS. Section 5 presents the performance of MFS with and without DFSA vs. NFS and Section 6 presents some related works. Our conclusions are given in Section 7.

2 MOSIX Background

MOSIX is a management system that can make a Unix cluster run almost like in an SMP. Its main features are a preemptive process migration and supervising algorithms for load-balancing [6], to prevent excessive paging when a node is thrashing [5] and for parallel I/O [2]. MOSIX is transparent to the applications. Users can run parallel (and sequential) applications by initiating processes in one node, then allow the system to assign and reassign processes to the best available nodes throughout the life of the application [13]. The granularity of the work distribution is the Unix process: when a user launches several processes, some of which may migrate away, but if the user runs "ps", it will report the status of all the processes, regardless of where they run.

2.1 The system image model

In MOSIX, each process has a unique home-node (where it was created), which is usually the Login node of the user. The system image model is a cluster, in which every process seems to run at its home-node and all the processes of a user's session share the run-time environment of the home-node. Processes that migrate to a remote (away from the home) node use its local resources, e.g., CPU and memory, whenever possible to run the application processes but continue to interact with the process' environment via the home-node. For example, "kill" always refers to process-ID's on the home-node rather than where the process may be running.

2.2 Process migration

MOSIX supports preemptive (completely transparent) process migration, that can migrate almost any process, any time, to any available node. This tool along with the set of supervising algorithms, constitutes the MOSIX process migration policy, which continuously attempts to assign and reassign processes to nodes, to take advantage of the best (cluster-wide) resources as they become available. The algorithms used by this policy are based on commodity market heuristics that measure and weight the amount of different resources, e.g., CPU, memory, I/O, used by all the processes vs. the cluster-wide available resources, in order to maximize the overall performance [15].

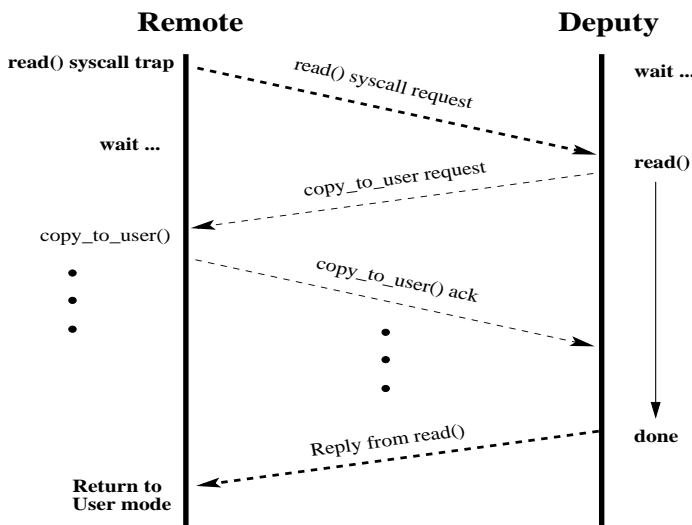


Figure 1: *Remote - Deputy* interaction, *Read* system-call.

After a process is migrated, most of its system-calls are intercepted by a link layer at the *remote* node. If a system-call is site independent, it is performed directly on the *remote* node. Otherwise, the system-call is forwarded to a stub, called the *Deputy*, (see Figure 1), which performs the system-call (synchronously) on behalf of the process in the home-node. The *Deputy* returns the result(s) to the process in the *remote* node which then continues its operation. Figure 1 shows the above sequence of operations for the *read* system-call. Note that such an operation could require up to 4 messages, although in most cases the data is appended to the reply message, thus requiring only 2 messages.

From the illustration in Figure 1 it follows that the process migration policy could be inefficient for processes with intensive I/O and/or file-system operations. Clearly, such processes would be better off not migrating at all. The next section describes one method to overcome this problem.

3 Direct File System Access

The Direct File System Access (DFSA) is a re-routing switch that was designed to reduce the extra overhead of running file (and directory) oriented system-calls of a migrated process. This is accomplished by performing most of those system-calls in the node where the process currently runs. The main advantages of this approach are fast access to files, preventing network congestion (especially at the home-node), reducing the network-communication overheads, and even eliminating it completely when the process runs in the same node as the file(s) it uses.

DFSA checks whether the partition used by a given system-call is declared to be mounted (with the same mount-flags) on all nodes and that its file-system type supports DFSA. If so, it usually runs file and directory oriented system-calls directly on the present node, and only exceptional cases are directed to the home node (for example, when a file-descriptor is shared among several processes).

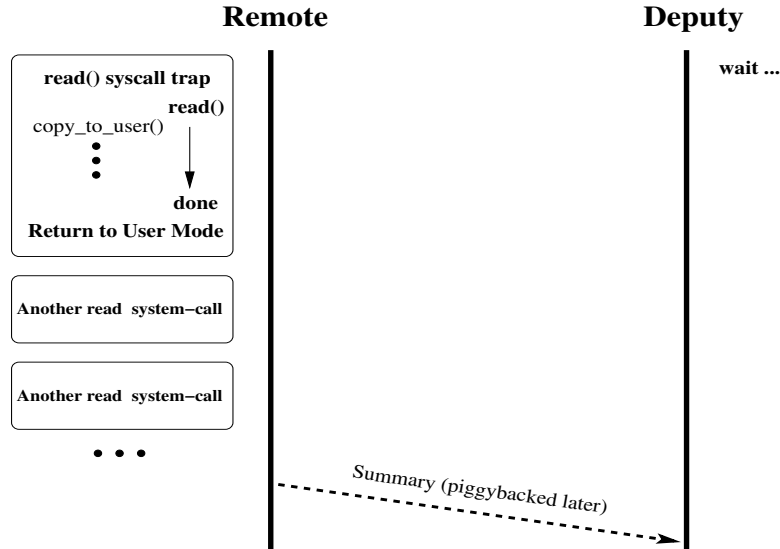


Figure 2: *Read* system-call with DFSA enabled.

Figure 2 shows the sequence of operations for the `read` system-call with DFSA enabled. Unlike the *Remote - Deputy* protocol, shown in Figure 1, the entire system-call is performed in the *Remote* node, and although (for reasons of consistency) the *Deputy* still needs to be eventually updated, DFSA prevents the need to contact the *Deputy* per system-call, and only a short summary of operations is queued to be sent to the deputy, usually after a span of many system-calls, and even then, that summary is piggy-backed to other messages.

3.1 DFSA requirements from the system administrator

DFSA requires that the selected file system(s) are mounted on the same-named mount-points and with the same mount-flags on all nodes.

To operate correctly, the system administrator must make sure that the user/group-ID scheme is either identical throughout the cluster or at least safe enough so that no access-violations can occur when the same file system is accessed by users with ID's assigned by different nodes.

3.2 DFSA requirements from the supporting file-systems

Due to the process migration, it is essential that a migrated process that issues a sequence of operations on a data item will get the same results as if it had not been migrated at all. For example, a process located on node A can write data to an open file and then migrate to node B. After migrating, the process might try to read the block it just wrote. In that case the process must get the same data it would get as if it was not migrated, since the process is not aware that it was relocated to node B.

Due to the above, DFSA can work with any file system that satisfies the following properties:

- A *single-node consistency*: the results of any sequence of (read/write) operations on a data item (a file or a directory) by processes running in a set of nodes could also occur if those processes were all running in one node.

For example, NFS [19] does not provide a *single-node consistency* when used in multiple nodes, since it does not guarantee that when a process on one node writes a data item, a second process, on another node, performing a read after the completion of the write operation will get the result of that write. If however, both processes were running in one node then the reading process would get the result of the latest write operation. In contrast, GFS [12] and MFS (see below) provide a *single-node consistency*, the former by maintaining locks on modified data items and the latter by maintaining a single cache for each data item.

Note that in a server/client model, a *single-node consistency* usually implies either cache invalidation in the client nodes or a single virtual cache, which could be in any node, e.g., in the server. Also note that a sophisticated server may possibly “lend” the cache of particular files and/or blocks to a particular node at any time. File-systems with shared hardware may offer other solutions.

- Time-stamps on files and between files in the same partition must be consistent and non-decreasing, regardless from which node the modifications are made.
- The file system must ensure that files/directories are not cleared when unlinked, as long as any process in the cluster still holds them open.

4 The MOSIX File System (MFS)

DFSA requires a single-node file and directory consistency between processes that run on different nodes because even the same process (or a group of cooperating related processes) can appear to operate from different nodes.

To use DFSA without any shared hardware, we implemented a prototype file system, called the MOSIX File-System (MFS), that provides a unified view of all files on all mounted file systems (of any type) on all the nodes of a MOSIX cluster as if they were all within a single partition. For example, if one mounts MFS on the `/mfs` mount-point, then the file `/mfs/1456/usr/tmp/myfile` refers to the file `/usr/tmp/myfile` on node #1456. This makes MFS both generic (since `/usr/tmp` may be of any file system type) and scalable (since MOSIX itself is scalable and all MOSIX nodes are included). Figure 3 shows the MFS tree structure.

MFS uses a client/server model, see Figure 4. When a process issues an MFS-related system-call, the local kernel acts as client and redirects the request to the appropriate MFS server, thus each node can be used as both a client and a server. The MFS server accesses its local file system (which can itself be of almost any type).

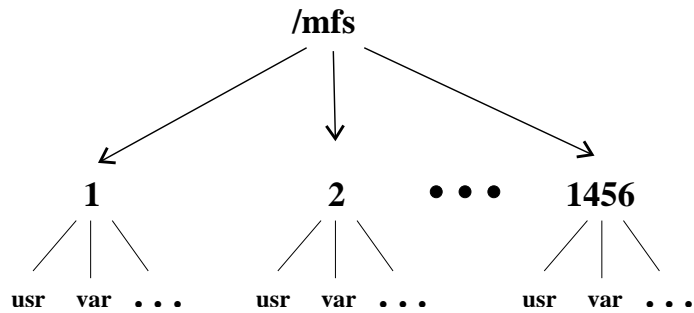


Figure 3: The MFS tree structure.

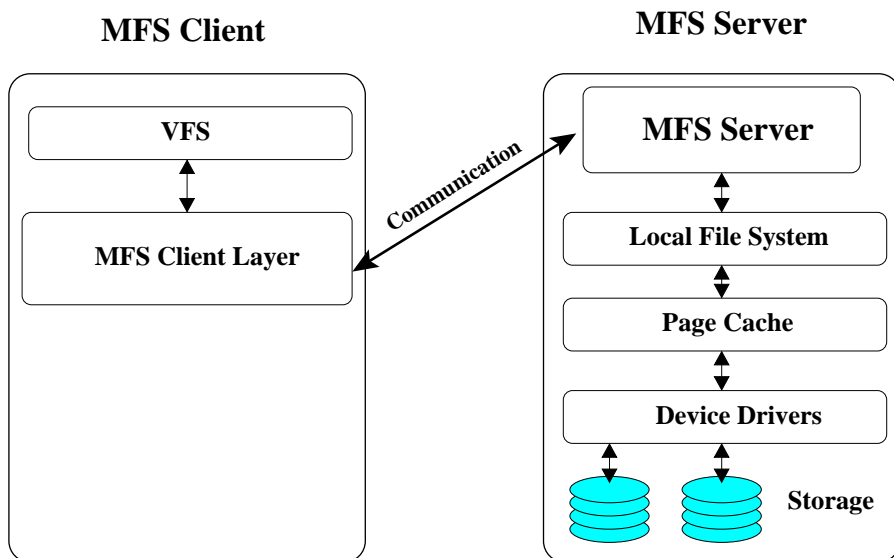


Figure 4: The MFS client server interaction.

Unlike other file systems, MFS provides a *single-node consistency* by maintaining only one cache - on the server(s). To implement this, the standard disk and directory caches of Linux are used only in the server and are by-passed on the clients. The main advantage of this approach is providing a simple, yet scalable scheme for consistency. Another advantage of the MFS approach is raising the client-server interaction to the system-call level which is especially good for large I/O operations. Obviously, having no cache on the client is a major drawback for I/O operations with smaller block sizes. We note that MFS does not support high availability, e.g., a failure of a node prevents any access to files that were located in that node.

4.1 Administration of the cluster

MFS can be used for administering the cluster, since all the mounted file systems of different nodes are accessible from any node. Various tasks can be performed without having to deal with complicated NFS administration tasks, such as maintaining export tables. This feature is particular useful for managing large clusters, where often the system administrator needs to change specific files on some (or all of the) nodes. The system-administrator may also use MFS to update whole file-system partitions by directly

accessing block-devices.

4.2 Bringing the process to the file

Each process collects statistics on its MFS usage, including the locations, rates and the amount of data accessed on each node. For scalability considerations, each process maintains statistics only about a limited (fixed) number of nodes where it does most of its operations. The list of nodes is continuously being updated according to the process' most recent I/O activities. The MFS statistics are incorporated with other collected information such as CPU usage, memory usage and non-DFSA I/O operations and incorporated into the MOSIX process migration policy.

If the supervising algorithm detects that a process is performing an amount of I/O beyond a certain threshold, the process is marked as a candidate for migration. The aim of these statistics is to encourage a process that performs a moderate to high volume of MFS I/O to migrate to the node in which it does most of its I/O. The process migration algorithms monitor and weighs the amount of I/O operations vs. the size of each process, as well as CPU load-balancing considerations, in order to make an intelligent decision whether to migrate the process or not.

Unlike all other networked file systems that bring the data from the file server to the client node over the network, the MOSIX algorithms attempt to migrate the process to the node in which the file resides, eliminating the communication overhead between the process and the file server.

5 Performance

This section presents the performance of MFS with and without DFSA and compares the results with the performance of local I/O operations and remote I/O via NFS. We begin with several benchmarks that test common I/O operations such as *read* and *write*, and then move to more complex benchmarks.

All tests were performed in a cluster with 64 identical workstations, each with a Pentium III 1133MHz, 512 MB RAM, a 20 GB 7200 RPM IDE disk and a 100 Mb/Sec Ethernet NIC, that were connected by a switch and ran under MOSIX-1.7.0 [17] for Linux 2.4.18. To reduce the impact of memory caching and other disk related fluctuations, each test was repeated 5 times, using 1 GB files, and the average throughput was measured.

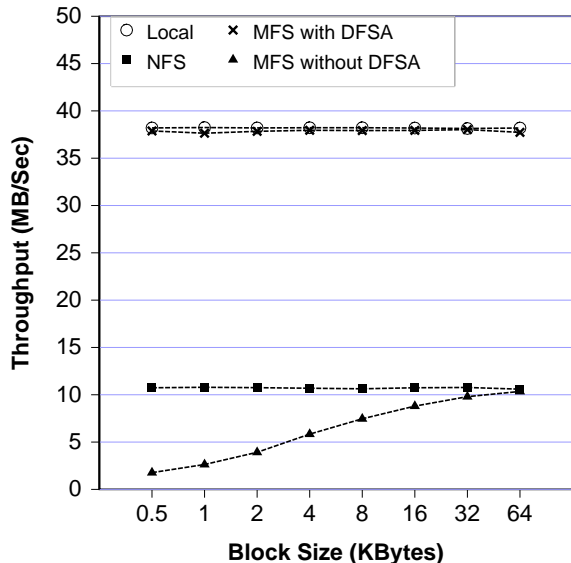
The following 4 file access methods were used:

- **Local:** the process accessed local data.
- **MFS with DFSA:** the process accessed data in a remote node, with DFSA and migration enabled.
- **MFS without DFSA:** the process accessed data in a remote node, with DFSA and migration disabled.
- **NFS:** the process accessed data located in a remote node via NFS.

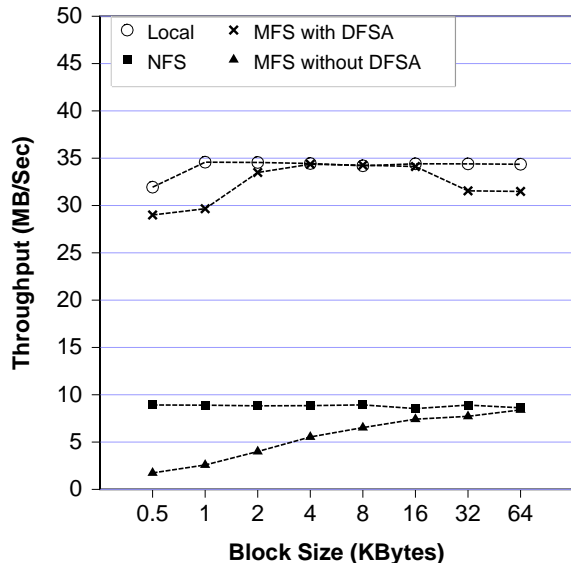
In a preliminary test, we used the `ttcp-1.12` benchmark and measured the speed of the TCP/IP between pairs of nodes at an average rate of 11.17 MB/Sec for 8KB blocks, with less than 0.5% variations for 4KB, 16KB and 32KB blocks.

5.1 Sequential file operations

Figure 5(a) presents the throughput of a sequential *read* for the four access methods with block sizes (i.e., number of bytes read per the *read* system-call) ranging from 512 bytes to 64KB. From the measured



(a)



(b)

Figure 5: Sequential access rates. (a) *read*, (b) *write*.

results it follows that the throughput of **MFS with DFSA** was consistently less than 1% slower than the **Local** access, and on the average, more than 3.5 times faster than **NFS**. Observe that since **MFS without DFSA** did not use a local cache, its throughput was sensitive to the block sizes. As a result, it was much slower than **NFS** for small block sizes and had nearly an identical performance to **NFS** for large block sizes, e.g. 64K. Furthermore, it was only 8% slower than the throughput of TCP/IP.

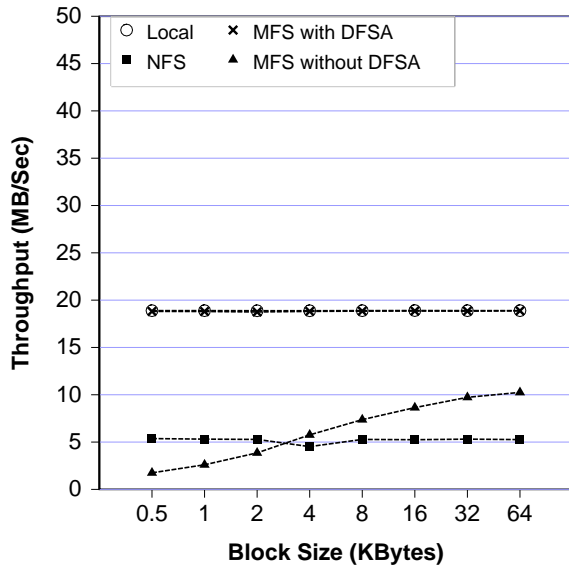
Figure 5(b) presents the corresponding results for a sequential *write*. In this case, the throughput of **MFS with DFSA** was between 0.5% - 13% slower than the **Local** access, with an average of 6.7% for all the block sizes; and on average, over 3.6 times faster than **NFS**. Note that as in the sequential *read*, the throughput of **MFS without DFSA** for large block sizes was only slightly slower than **NFS**.

5.2 Stride file operations

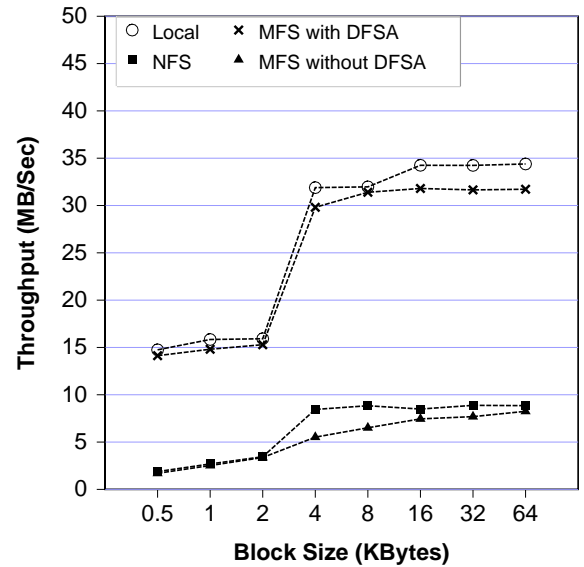
This benchmark presents the performance of stride file accesses, in which every time the process accessed a block of data (for read/write), it performed a seek forward of one block before accessing the data again.

The throughput of *stride read* are shown in Figure 6(a). As expected, the **Local** access method obtained the best results, with **MFS with DFSA** less than 1% behind. Note that in both of these cases the access rates are about half the rates obtained by the sequential *read* benchmark. The **NFS** throughput presented a similar decline in the performance for all block sizes (about 50% of the rates obtained in the sequential read). In contrast, **MFS without DFSA**, which does not maintain local cache, did not suffer any slowdown (vs. the sequential *read*). It outperformed **NFS** for block sizes greater than 2KB, because stride operations defeated the NFS read-ahead mechanisms.

Figure 6(b) presents the throughput of the *stride write*. As in the *stride read*, the **Local** access method obtained the best results, with **MFS with DFSA** behind. Unlike *stride read*, in these cases, for small block sizes, the throughput was considerably slower than the corresponding rates of the sequential



(a)



(b)

Figure 6: Stride access rates. (a) *read*, (b) *write*.

write, while it improved considerably for block sizes greater than 2KB. Similar anomalies were obtained for **NFS**, while the **MFS without DFSA** performance remained almost identical to the sequential case.

5.3 The Bonnie benchmark

Bonnie [8] is a benchmark for measuring the performance of Unix file system operations. It was designed to expose bottlenecks in server applications. The benchmark performs the following sequence of tests on a file of a given, fixed size:

- A character mode sequential *write* using the `putc` `stdio` macro.
- A block mode sequential *write* using the `write()` system-call.
- The file is rewritten (first a block is read then it is modified and then it is written back).
- A character mode sequential *read*.
- A block mode sequential *read* using the `read` system-call.
- Random *seek*, in which 3 processes perform a total of 4000 random seeks on the file. In each case the block is read using the `read()` system-call and in 10% of the cases the block is modified and written back.

The performance of the Bonnie benchmark for each of the above tests, using a file of size 1 GB and block sizes of 16KB are shown in Table 1. In the table, each line presents the throughput (MB/Sec) of each of the above Bonnie tests, except the last line, which shows the number of *seeks* per second.

Table 1: Bonnie tests performance - 16KB blocks.

<i>Test (mode)</i>	<i>Access Method</i>			
	Local	MFS with DFSA	MFS w/out DFSA	NFS
write (character)	10.04	9.92	4.01	8.56
write (block)	34.34	33.63	7.40	8.49
rewrite	16.60	16.47	3.84	4.03
read (character)	12.77	12.73	4.09	9.82
read (block)	38.27	38.28	8.74	10.48
Random Seek	195.87	196.02	195.14	138.05

From the table it can be seen that the performance of **MFS with DFSA** was only a few percents different from that of the **Local** access method, with a few cases where it was slightly better (due to parallel operation of the *Remote* and the *Deputy* vs. the sequential **Local** process). In contrast, the **NFS** performance were substantially lower than both of the above, with as much as 75% slower for the block write and rewrite cases and about 72% for the block read test. As expected, since **MFS without DFSA** uses no local cache, its throughput were much slower than **NFS**, with almost 53% slower in the character mode write test and 58% slower in read test. However, in the cases of the block-sized system-calls the differences were much smaller, e.g., **MFS without DFSA** was only 13% slower for the write block test, 4.7% slower for the rewrite test and 16.6% slower for the read block test. As expected, **MFS without DFSA** works better with larger block sizes. In the Seek test, **MFS without DFSA** achieved good results and was 41% faster then **NFS**. We note that this is due to the fact that **MFS without DFSA** does not use pre-fetching, which is useless in the case of random access and results in lower performance.

5.4 The Postmark Benchmark

The PostMark [14] benchmark simulates heavy file system loads, e.g., as in a large Internet electronic mail server. It begins by creating a large pool of random size text files, then it performs a sequence of transactions, recursively, until a predefined workload is obtained. We run the benchmark between a pair of nodes. Each test included 50000 transactions using 1000 files, with file sizes from 0.5 BK to 1.02 MB and block sizes ranging from 1K byte to 64K bytes. We note that the Unix buffered file I/O was not used.

The results are presented in Table 2. In the table, the second column shows the **Local** times, when both the process and the files were in the same (server) node; the third column shows the **MFS with DFSA** times, i.e., the benchmark started in a client node then migrated to the server node. The fourth and the fifth columns show the **MFS without DFSA** and the **NFS** times respectively, from a client (in its home) node to a server node.

From the results in Table 2 it follows that on average (for all block sizes) **MFS with DFSA** is only 16.6% slower than **Local**, and more than 10.3 times (900%) faster than **NFS**. We note that this last result motivated the development of MOPI [2]. As expected, since **MFS without DFSA** uses no local cache, its throughput were on average twice slower than **NFS**, with smaller slow down ratios for larger block sizes.

Table 2: Postmark file systems access times (Sec).

<i>Block size</i>	<i>Access method</i>			
	Local	MFS with DFSA	MFS w/out DFSA	NFS
1K	14.4	18.0	656.6	162.0
4K	13.2	15.4	322.6	162.0
8K	13.0	15.0	261.8	161.6
16K	13.0	15.0	232.0	162.0
32K	13.8	15.4	220.2	162.8
64K	14.2	16.4	217.2	163.0

5.5 Common Unix commands

The next set of tests present the performance of some common Unix file-system related commands. The specific tests are similar to the AFS [1] benchmark. The tests were executed on a data-set of size 145 MB, consisting of the entire Linux kernel source-tree (126 MB tar uncompressed and 29 MB tar compressed).

The following commands were issued:

- **gunzip**: decompress the data.
- **du -s**: find the total size of the data-set.
- **ls -lR**: obtain status of every file (size, owner, etc.) in the data-set.
- **tar -x**: untar the data-set.
- **cp**: copy the data-set.
- **find . -type f — xargs wc**; counts the lines of all the files in the data-set.
- **find . -type f — xargs grep kangaroo**: scan every file in the data-set for a non-existent word.

Figure 7 presents the completion times of each of the above tests. The results show that the performance of **MFS with DFSA** was within 3% – 7% of the **Local** access method, with a few cases where it was even better (due to parallel operation of the *Remote* and the *Deputy* vs. the sequential **Local** process). We note that the variation in the slow-down factor is due to the duration of the operation: it was larger for short-lived processes, because the time it took to migrate the process to the server took a larger proportion of the run time. The performance of **NFS** was substantially lower than both the **Local** and the **MFS with DFSA** access methods, and somewhat better than that of **MFS without DFSA**.

It is interesting to analyze the influence of the block size on performance differences between **NFS** and **MFS without DFSA**. The **cp** command uses a 4KB block size. Due to this small block size **NFS** was 16.5% faster than **MFS without DFSA**. The block size used by **tar** are 10KB for reading and up to 10KB for writing (larger than 10KB files). The larger block size resulted in only a 5% difference between **NFS** and **MFS without DFSA**. Both **gunzip** and **grep** use 32KB block sizes. The **grep** command only reads the files, thus the difference between **NFS** and **MFS without DFSA** was only 4.5%. For the **gunzip** command, which involves more writes than reads (with a ratio of 4.34 to 1),

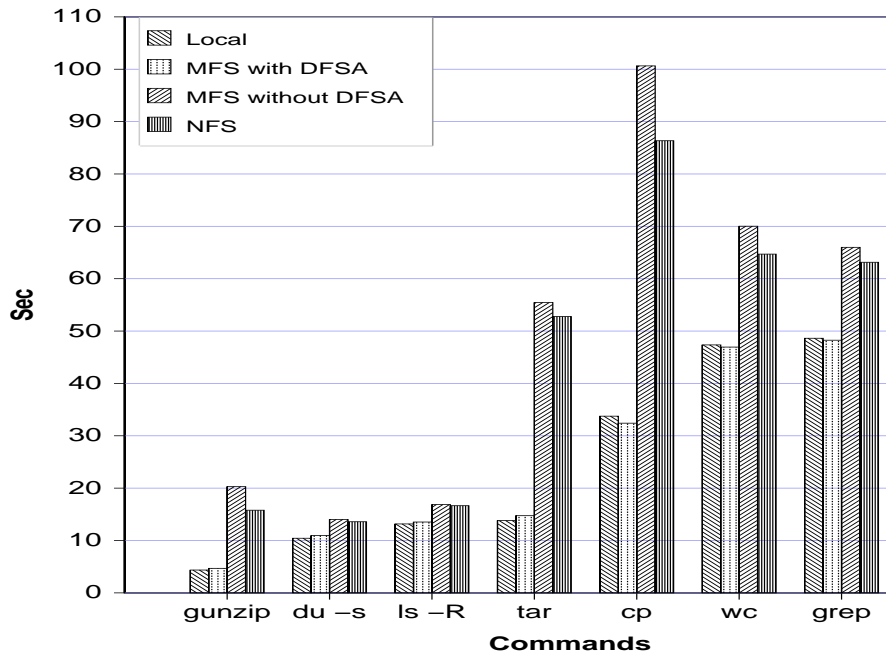


Figure 7: Common file system command times.

the performance difference was 28.5%. We note that this relatively large difference is due to the write caching used by **NFS**, in which not every write request is actually sent to the server. The **ls** and **du** commands, which access only the meta-data, had relatively low differences of 1.26% and 3.24% respectively. This is expected because only small amounts of data are passed between the server and the client.

5.6 Processes with uneven I/O

In some cases such as compression/decompression of a file located in one node to an output file in another node, the process's location can vary the amount of networked-I/O. In such cases, it would be better to run the process in the node where it does most of its I/O.

To demonstrate how MOSIX handled such cases we ran test programs that were intentionally started on the node where less I/O was performed. They were performed on the entire Linux kernel source tree using the **MFS with DFSA**, **MFS without DFSA** and the **NFS** access methods. We used the following programs:

- **gzip**: compress files. The amount of data read was 4.34 times larger than the amount written.
- **gunzip**: uncompress files, same ratio as in **gzip**.
- **tar c**: create an archive of files, using many read operations (especially when the files are small) and less write operations. The read/write ratio was 1.7.
- **tar x**: expand archives, using more write than read operations. Same ratio as in **tar**.

The results are presented in Figure 8. From the figure it can be seen that except in the case of **gzip**, the performance of **MFS with DFSA** was better than **NFS** and both were better than **MFS**

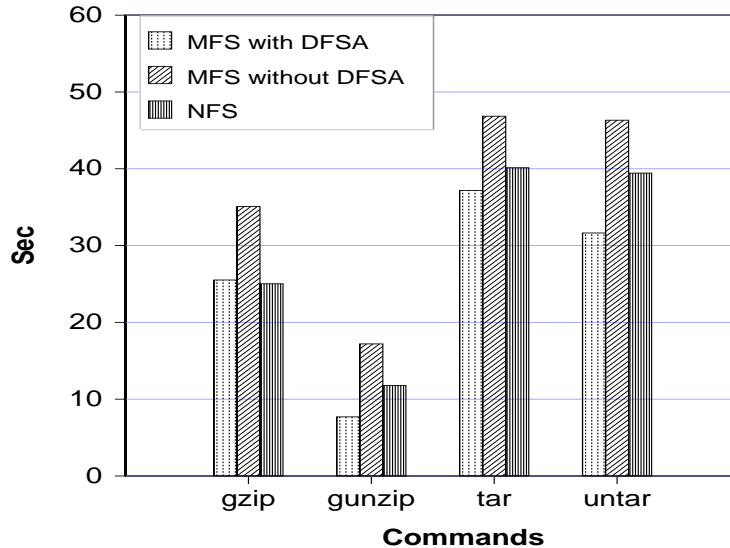


Figure 8: Processes with uneven I/O times.

without DFSA. It is interesting to note that **gunzip** performed better than **gzip** (relative to **NFS**): this is because **gzip** performs more computations and therefore writes its output slower, allowing **NFS** to take advantage of its local caching. **gzip** under **MFS with DFSA** performs 2% slower than **NFS**, while **gunzip** performs 34.7% better. The corresponding improvement for **tar** was 7.4% and for **untar** was 19.8%. We note that **MFS with DFSA** reduced the network usage of **gzip** and **gunzip** by a factor of 4.34; it also reduced the number of operations of **tar** and **untar** by a factor of 1.7.

5.7 Multiple servers single client

In many cases a program running in the user's workstation may need to access data that resides in multiple servers, e.g., when running **diff**, **cp**, **tar** and **gzip**. Such cases involve 3 nodes, with a non negligible network overhead. One way to reduce this overhead is to migrate the program to one of the servers.

For this test we used a 3-node cluster, with one client node and 2 servers. The programs ran in the client node and accessed files in the 2 servers using the **MFS with DFSA**, **MFS without DFSA** and **NFS**. As before we used the Linux kernel source tree as our data. The following programs were tested:

- **gzip/gunzip**: compress/decompress the tar ball of the data-set
- **tar/untar**: create/extract a tar ball of the data-set (the source files were in one server and the results in the other). (input files and output files were located on different servers).
- **cp**: copy the data-set from one server to the other.
- **diff**: compare 2 copies of the data-set which were located in the 2 servers.

The results of these tests are shown in Figure 9. From the figure it can be seen that process migration improved the performance in all the cases. The average improvement was 41.37% and the maximum improvement was 138.51%, achieved by the **gunzip** program. We note that in all the cases the network

usage by **MFS with DFSA** was 50% lower than the other methods. This reduction improved the performance, reduced the network load and also freed the client node to perform other tasks.

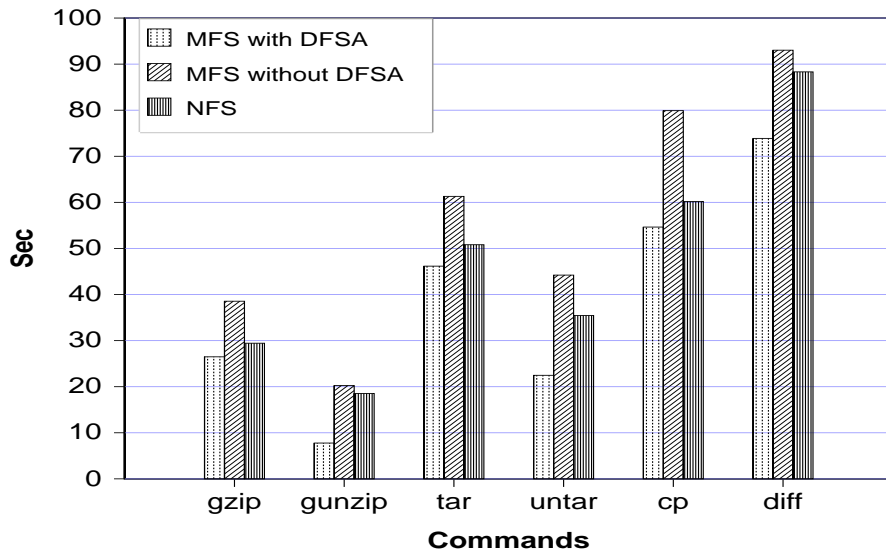


Figure 9: Client - servers run times.

5.8 Multiple clients single server

This set of tests show the performance of parallel processes that started simultaneously, each running in one client node and performing I/O on a shared file server. For reference, all the processes were also ran (**Locally**) in the server. The number of processes was increased from 1 - 12 and the total sum of the run time of each process was recorded.

Table 3: Parallel *gunzip* run times (Sec).

Number of clients	Access Method			
	Local	MFS with DFSA	MFS w/out DFSA	NFS
1	4.0	4.4	19.8	19.0
2	24.8	35.1	83.4	47.5
4	104.6	173.0	356.5	214.6
8	496.3	590.5	1162.8	892.6
12	1178.6	1124.1	2156.8	1876.4

The results of the parallel **gunzip** and parallel **grep** tests are shown in Table 3 and Table 4 respectively. From these tables it can be seen that **MFS with DFSA** performed much better than both **NFS** and **MFS without DFSA**, and one case even better than **Local**. This shows that the optimal balance of processes among the server and the clients is delicate. While the optimal distribution is NP-hard and is different for each application, the tests show that the MOSIX scheme performed better than the other methods.

Table 4: Parallel *grep* run times (Sec).

<i>Number of clients</i>	<i>Access Method</i>			
	Local	MFS with DFSA	MFS w/out DFSA	NFS
1	47.3	47.1	60.9	67.0
2	93.4	91.1	149.9	130.5
4	180.2	182.9	392.7	311.6
8	348.2	398.2	1118.5	836.7
12	530.4	618.9	2143.6	1700.4

In the next test we run a set of processes with a fixed CPU to I/O ratio. In this test each process performed a certain amount of CPU operations for each MB of data read (from a common file). The I/O rate (for a single process) was fixed to 7 MB/Sec, much below the maximal disk rate of 38 MB/Sec, previously shown in Figure 5(a). The main point of this test was to highlight the benefit of a dynamic vs. static process allocation methods.

Table 5: Parallel *read* run times (Sec).

<i>Number of clients</i>	<i>Access Method</i>			
	Local	MFS with DFSA	MFS w/out DFSA	NFS
1	100.1	100.8	146.8	150.0
2	356.3	316.3	382.2	328.5
4	1355.7	993.1	1123.9	1021.7
8	5361.0	3460.8	4444.0	3890.4
12	12054.8	6916.8	9862.2	8683.2

The results of this test are shown in Table 5. As before, it can be seen that **MFS with DFSA** performed much better than **NFS** and **MFS without DFSA**, and in most cases better than **Local**. As expected, the differences in the performance between the access methods is increased with the number of processes. Although the optimal allocation of processes to nodes is unknown, the improved performance of **MFS with DFSA** proves that the fine-tuned, adaptive process reallocation algorithms of MOSIX can outperform the static allocations used. We note that **MFS without DFSA** was slower than the other methods, but still makes it a reasonable alternative for a cluster file system.

6 Related work

There are several systems that reduce the network usage by increasing the local I/O.

Panda [9] is a parallel I/O library for multidimensional arrays that supports the I/O strategy of “part-time I/O”, where each node can be both a client and/or a server as needed, similar to NFS. As opposed to MOSIX, where network usage is minimized by migrating processes closer to their data, Panda optimizes performance by initially selecting the I/O nodes (among the cluster nodes) which will make the cluster use the minimal network resources for a given description of the anticipated I/O requests from clients to servers.

DataCutter [7] is a framework designed for developing data intensive applications in distributed environments. The programming model in DataCutter, called “filter-stream programming”, represents components of data-intensive applications as a set of filters. Each filter can potentially be executed on a different host across a wide-area network. Data exchange between any two filters is described via streams, which are uni-directional pipes that deliver data in fixed size buffers. The idea of changing the placement of program components to better use the cluster resources is similar to MOSIX. However, In MOSIX there is no need to change the applications, and MOSIX incorporates automatic load balancing, whereas in DataCutter assignments are manual.

Abacus [3] is a programming model and a run time system that monitors and dynamically changes function placement for applications that manipulate large data sets. The programming model encourages programmers to compose data-intensive applications from explicitly-migratable functions, while the run time system monitors those functions and dynamically migrates them in order to improve the application performance. This project is quite similar to MOSIX, both perform dynamic load balancing of processes/functions based on run-time statistics-collection. The difference is that MOSIX works with generic programs whereas Abacus requires a programming model.

7 Conclusions and future work

This paper presented a method for scalable cluster I/O that combines cluster file systems with process migration. Our scheme allows a migrated process to access file systems directly from its present node, e.g., an I/O-bound process could be migrated to the node where it performs most of its I/O. One outcome is that parallel processes of a job could be migrated from a client node to file servers, to enable efficient parallel access to different files. We showed that our scheme is “easy to use” and it provides improved performance and scalability.

Correct operation of our scheme requires a *single-node consistency* between processes that run in different nodes. This property, which is currently supported by only few file systems, e.g., GFS[12], PolyServe’s Matrix Server cluster file system [16] or GPFS [20], combined with DFSA, creates an opportunity to build high I/O performance systems that can scale up without saturating the network or any one node.

A possible following project to improve the performance, would be to add some cache-lending facility to MFS, e.g., as in xFS [4]. Other possible followup projects could be to adjust one of the above file systems to DFSA and MOSIX. Similar extensions could be developed with other file systems that meet the DFSA requirements, especially with shared storage devices such as SAN.

An interesting research problem is where to locate a process when it performs I/O operations on files that are located in different nodes while also communicating with other processes.

Beyond that, it would be interesting to build specific packages for scalable parallel I/O. For example, the MOSIX Parallel I/O (MOPI) [2] package is a library and set of daemons that enable a user to work with very large data sets in a MOSIX cluster. With MOPI, it is possible to partition a large file to smaller segments and to place the segments in different nodes. A process can then access the segments of a file (by using the MOPI interface) without being aware of the segment locations. Part of this project includes adaptation of ROM-IO, which is an implementation of the MPI-IO standard, to support MOPI. This allows programs using MPI [18] (and MPI-IO) to use MOPI in a transparent way.

Acknowledgments

We are grateful to D. Braniss and A. Eizenberg for their help. This research was supported in part by grants from the Ministries of Defense and Science and from Dr. and Mrs. Silverston, Cambridge, UK.

References

- [1] AFS, <http://www.transarc.com> (2003).
- [2] L. Amar, A. Barak and A. Shiloh, The MOSIX parallel I/O system for scalable I/O performance, in: *Proc. of the 14th International Conference Parallel and Distributed Computing and Systems (PDCS 2002)*, Cambridge, MA (November 2002) pp. 495–500.
- [3] K. Amiri, D. Petrou, G.R. Ganger and G.A. Gibson, Dynamic function placement for data-intensive cluster computing, in: *Proc. of the USENIX Annual Technical Conference*, San Diego, CA (June 2000) pp. 307–322.
- [4] T.E. Anderson, M.D. Dahlin, J.M. Neefe, D.A. Patterson, D.S Roselli and R.Y. Wang, Serverless network file systems, *ACM Tran. on Comp. Systems*, 14(1) (1996) 41–79.
- [5] A. Barak and A. Braverman, Memory ushering in a scalable computing cluster, *Journal of Microprocessors and Microsystems*, 22(3-4) (1998) 175–182.
- [6] A. Barak and O. La'adan, The MOSIX multicomputer operating system for high performance cluster computing, *Future Generation Computer systems*, 12 (1997/98) 361–372.
- [7] M. Beynon , C. Chang, U. Catalyurek, T. Kurc, A. Sussman, H. Andrade, R. Ferreira and J. Saltz, Processing large-scale multidimensional data in parallel and distributed environments, *Parallel Computing*, 28(5) (2002) 827–859.
- [8] Bonnie, <http://www.textuality.com/bonnie> (2003).
- [9] Y. Cho, M. Winslett, M. Subramaniam, Y. Chen, S. Kuo, and K.E. Seamons, Exploiting local data in parallel array I/O on a practical network of workstations, in: *Proc. 5-th Workshop on I/O in Parallel and Distributed Systems*, San Jose, CA (1997) pp. 1–13.
- [10] Coda, <http://www.coda.cs.cmu.edu> (2003).
- [11] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM: Parallel virtual machine* (MIT Press, 1994).
- [12] Global File System (GFS), <http://www.sistina.com> (2003).
- [13] M. Harchol-Balter and A. Downey, Exploiting process lifetime distributions for dynamic load balancing, *ACM Tran. on Comp. Systems*, 15(3) (1997) 253–285.
- [14] J. Katcher, PostMark: A new file system benchmark, <http://www.netapp.com> (2003).
- [15] A. Keren and A. Barak, Opportunity Cost Algorithms for Reduction of I/O and Interprocess Communication Overhead in a Computing Cluster, *IEEE Tran. on Parallel and Distributed Systems*, 14(1) (2003) 39–50.
- [16] Polyserve Matrix Server, <http://www.Polyserve.com> (2003).
- [17] MOSIX, <http://www.MOSIX.org> (2003).
- [18] P. Pacheco, *Parallel programming with MPI* (Morgan Kaufmann Pub. Inc., 1996).
- [19] NFS version 3 protocol specification, <http://www.faqs.org/rfcs/rfc1813.html> (2003).
- [20] F. Schmuck and R. Haskin, GPFS: a shared-disk file system for large computing clusters, in: *Proc. of the USENIX conference on file and storage technologies (FAST '02)*, Monterey, CA (January 2002) pp. 232–244.
- [21] CXFS, <http://www.sgi.com/products/storage/cxfs.html> (2003).