Design and Implementation of a Novel Distributed Memory File System

Urvashi Karnani[1], Rajesh Kalmady[1], Phool Chand[1], Anup Bhattacharjee[2], and B.S. Jagadeesh[1]

[1] Computer Division, [2] Reactor Control Division,
Bhabha Atomic Research Centre, Mumbai, India
{urvashi,rajesh,phool,anup,jag}@barc.gov.in

Abstract. To improve performance and efficiency of applications, a right balance among CPU throughput, memory performance and I/O subsystem is required. With parallel processors increasing the number crunching capabilities, the limitations of I/O systems have come to fore and have become the major bottleneck in achieving better turnaround times for large I/O bound jobs in particular. In this paper, we discuss the design, implementation and performance of a novel distributed memory file system that utilizes the free memory of cluster nodes over different interconnects to assuage the above-mentioned problem.

Keywords: Clusters, Network Interconnects, Memory File System, RAM Disk, Tmpfs, Free Memory, File System Agent.

1 Introduction

It is a well known trend that the improvements in access times of I/O subsystems have not kept pace with improvements in processor speeds. Processing powers for computing systems have been increasing regularly over the years however; the improvements in hard disk technology have mainly achieved increases in storage density and size [1]. In addition to the increase in CPU speed, the past decade has witnessed a large improvement in computation speedup achieved through parallelizing applications. As a result, the CPU processing times of programs have improved considerably, thereby shifting the performance bottleneck towards the I/O subsystem. The I/O operation is time consuming because it is governed by the speed of rotation of disk (which is a mechanical device) and the speed of the translational movement of read/write head. This surfaces as the major source of delay, resulting in poor turnaround times of jobs. Further, when parallelization brings down the computation time, this delay in I/O subsystem becomes even more apparent.

To address the issues of I/O subsystems in High Performance Computing Clusters we have designed, developed and deployed a Novel Distributed Memory File System that operates in the user space and dynamically grows as per the needs of a given job thereby optimally utilizing the memory resources. The novel idea explored here is to

N. Meghanathan et al. (Eds.): CCSIT 2011, Part III, CCIS 133, pp. 139–148, 2011.

make an 'in-memory' file system exploiting the free memory available in nodes of a cluster that would boost the file I/O performance of an application by logically replacing the disk with memory interconnected over network. This is an 'in-memory' file system with the memory being aggregated from other nodes on the fly, where the data of a file may span over multiple nodes. In this paper, we discuss the design, implementation and results obtained from our work on the Novel Distributed Memory File System.

2 Related Work

To improve the I/O subsystem performance, researchers are focusing on the identification of high performance I/O subsystem architectures and implementations. Utilizing either the local memory or remote memory over some interconnect network for file systems have been explored both from academic and commercial perspectives over the past one and a half decades.

2.1 Memory Based File Systems

A Memory based file system is a file system that resides in memory and does not write data to non-volatile storage like disk. A memory based file system is typically used as storage for temporary files. By keeping as much data as possible in memory it avoids the disk I/O and associated overheads.

2.1.1 RAM Disk

The first attempt in this regard has been the RAM disk. A RAM disk reserves a range of memory and makes it available through a block device interface using a pseudo driver [2]. RAM disks have certain limitations: Memory is reserved at the time of the disk creation, so it is locked from shared system use whether actually in use or not. The block device is of fixed size, so the file system mounted on it is also fixed. RAM disks do not support paging of their memory resulting in very poor system response in cases where the system is running low on free physical memory. Since the system sees a RAM disk as a device rather than a file system, any access to a file stored on it results in a second copy of the data being kept in the file system buffer i.e. it requires unnecessary copying of memory from the block device into the page cache [2].

2.1.2 Tmpfs

Tmpfs is a memory based file system which uses kernel resources relating to the Virtual Memory System and page cache as a file system. Tmpfs is so named because files and directories are not preserved across reboot or unmounts. Tmpfs is designed as a performance enhancement utility which is achieved by caching the writes to files residing on a Tmpfs file system [3]. Tmpfs files are written and accessed directly from the memory maintained by the kernel thus, Tmpfs file data can be swapped to disk, freeing resources for other needs. General Virtual Memory System routines are used to perform many low level Tmpfs file system operations hence reducing the

amount of code needed to maintain the file system. Performance improvements in Tmpfs are most noticeable when a large number of short lived files are written and accessed on a Tmpfs file system. Tmpfs has a limitation that it shares resources (data and stack segment) with the programs executing in a system [3]. Large sized Tmpfs files affect the amount of space left for programs to execute thus, affecting the execution of very large programs. Likewise, programs requiring large amounts of memory use up the space available to Tmpfs.

RAM disk and Tmpfs use local memory (which is limited on a single machine) for storing the data. Hence, maximum file size on RAM disk or Tmpfs is calculated based on the physical memory available on that machine and if the application requires a much larger file as compared to the one supported by any of the memory based file systems it is written on to the disk.

2.2 Related Work on Use of Remote Memory

With high bandwidth and low-latency network becoming affordable, interesting efforts have been made to utilize remote memory. The use of remote memory has been explored in several contexts such as an extension of main memory, cooperative-caching and network paging. The Global Memory Service (GMS) provides an example of use of remote memory as an extension of main memory [4]. Dahlin et al. describe a co-operative caching algorithm called NChance Forwarding. Cooperative caching seeks to improve network file system performance by coordinating the contents of client caches and allowing requests not satisfied by a client's local in-memory file cache to be satisfied by the cache of another client [5]. To boost file system performance Network Ram disk can be used which is a virtual block device that uses idle main memories in a cluster for storage through the disk interface [6]. Network Swap (NSwap) is an example of a remote paging system [7].

We have developed a Novel Distributed Memory File System to provide better performance to I/O intensive applications. In the following sections we discuss the design, implementation and the performance obtained for the same.

3 Distributed Memory File System Design

The traditional file system manages two types of data namely file content that the application can access via the read, write functions and metadata i.e. the information related to file attribute and file system structure. In our design we have followed the paradigms of UNIX/Linux based file systems [8, 9]. The file system abstractions information such as superblock, inode and data blocks is maintained in the memory. The file data is written and accessed directly from memory of the nodes in the cluster. Figure 1 depicts the schematic design of the Distributed Memory File System. It consists of a Metadata Server Node, Distributed Memory File System (DMemfs) Nodes, File System Agent and a thin library at the client node whose functions are incorporated in the client application source code.

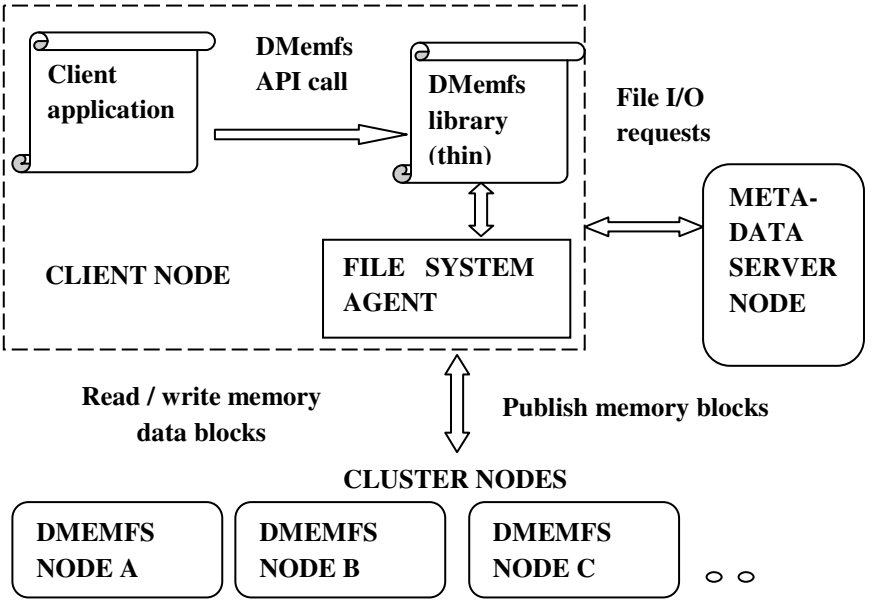


Fig. 1. Distributed Memory File System

3.1 Metadata Server Node

Metadata Server Node is the heart of the file system, which is responsible for maintaining the state of the file system that needs to be updated regularly to preserve the consistency of file system. Some of the data structures maintained at the metadata server are as follows:

- **Superblock structure** holds data about the file system such as total number of blocks, number of free blocks, total number of inodes and number of free inodes.
- **Inode data structure** is used to uniquely identify file. It holds information about a file such as owner, permissions and timestamps. The inode structure also contains address fields that give information about the data blocks where the file data resides. Similar to the UNIX paradigm [8], we have designed addressing scheme using various levels of indirections to incorporate large file size. In the current design of inode with file system block size of 8K we can support file size in Terabytes. Each indirect addressing block can hold 1024 entries. With 12 direct block addressing fields we can have file sizes up to 96K (12 * 8K) bytes. With the two single indirect addressing we can support 16Mbytes (2* 1024 * 8K) file size. Four double indirect addressing blocks account for next 32GB (4 * 1024 * 1024 *8K) of file size and the two triple indirect addressing blocks can account for another 16 Terabytes (2 * 1024 *1024 * 1024 * 8K) of file size.
- **Directory structure** maintains a mapping between the name of the file and inode numbers.

- **Metaopen file table** at the metadata server node maintains the state of open files across all the nodes. Each entry in this table consists of the access permission, count and pointer to an inode in the inode table whenever a file is opened an entry is made in this table.
- **Nodememoryinfo table** maintains a mapping of number of memory blocks published by each DMemfs node for the distributed memory file system.
- **Block map table structure** maintains mapping between the logical number of memory blocks and its actual physical location that consists of node identifier and local block number in the same node.

3.2 File System Agent

File system agent is responsible for communication between the metadata server node and other nodes in the cluster for reading and writing of blocks of data dedicated to the distributed memory file system. File system agent processes the commands and requests demanded by the client application for manipulating file data as well as metadata by initiating corresponding functions.

3.3 Distributed Memory File System (DMemfs) Nodes

The DMemfs nodes are the nodes of the cluster that can publish free memory blocks available for the use of distributed memory file system. Every node has a *local open-file table* that holds the information about the files opened by a program running on the node. It has an offset field that is required for the file I/O operations on the files opened by the program. It contains a *meta_open table* index that relates the file to an entry made at the metadata server. Whenever a read/write request is sent to the metadata server the *meta_open table* index is also sent along with the request which is required by the metadata server module to check for accessibility permissions. It also maintains *blockaddress table* structure that maps local memory blocks published for the file system with their actual physical addresses.

3.4 Client Application

The client application incorporates the library function calls that correspondingly initiate specified file handling operations such as open, close, read, write, mkdir, rmdir. The *DMemfs library* has been implemented as a thin library i.e. it contains only the wrapping functions, where as the core implementation of various functions is in the file system agent. It was required to keep this library thin so that any changes made in the file system agent do not necessitate a rebuild of the application program.

4 Implementation and Working

Currently the file system has been developed in user space with the client side API available as a library. Any client application can incorporate the API calls in the source code and use the file system. A communication protocol consisting of request and response messages has been developed for interaction of the metadata server node, DMemfs nodes and the file system agent that uses TCP/IP as the underlying

communication protocol. Each message consists of a string defining the operation to be performed followed by a delimiter and then appended with the data necessary for the operation *(RQ_operation; arguments)*.

The library communicates with the file system agent that in turn forwards the file I/O request to the metadata server. The metadata server module handles one session for every client where in multiple threads can be created for various I/O requests. At the DMemfs node module one thread is dedicated for communication with the metadata server and one thread serves the data transfer request for reading or writing data to the memory blocks.

4.1 Initialization

Initially, the DMemfs nodes do a handshake with the metadata server node at a predefined well known port and publish memory blocks. The DMemfs node sends a *"NODE_UP"* request followed by the number of memory blocks to be published. According to the available memory blocks the metadata server makes a file system and initializes all the data structures. A root directory is also created during the initialization that acts as the parent directory for other directories created by the users. The various tables for storing the metadata of the file system, residing at the metadata server are created and initialized.

4.2 Directory Operations

Directory maintains a mapping between the name of a file and its inode number that helps in uniquely identifying the file in the file system. In the current implementation the directory structure and related information is maintained at the metadata server node. Memory blocks are reserved at the metadata server node that holds the data corresponding to the directories. Hence, instead of using the remote memory blocks for directory data, the local blocks available at the metadata server are used that provide better performance in directory operations as the file system agent at the client side need not contact any other DMemfs node for the same.

4.3 File Operations

The distributed memory file system supports functions that perform basic file operations which include: creating a file, opening a file, reading from a file, writing data into a file, seeking to a position in a file, closing a file. For every file operation called by the user application the file system agent packs the required data in a form of message that is in compliance with the developed communication protocol and sends it to the metadata server. When the request involves modification of file attributes available at the metadata server the necessary changes are made and the file system agent is informed. When the request involves reading or writing (figure 2), the metadata server processes the request and sends a message consisting of IP address of the DMemfs node to be contacted , the offset required for reading or writing and the local block number of the memory block where the read or write has to performed, back to the file system agent. Once this message is received, an independent thread is created for communication and data transfer between the client node and DMemfs node having the memory block.

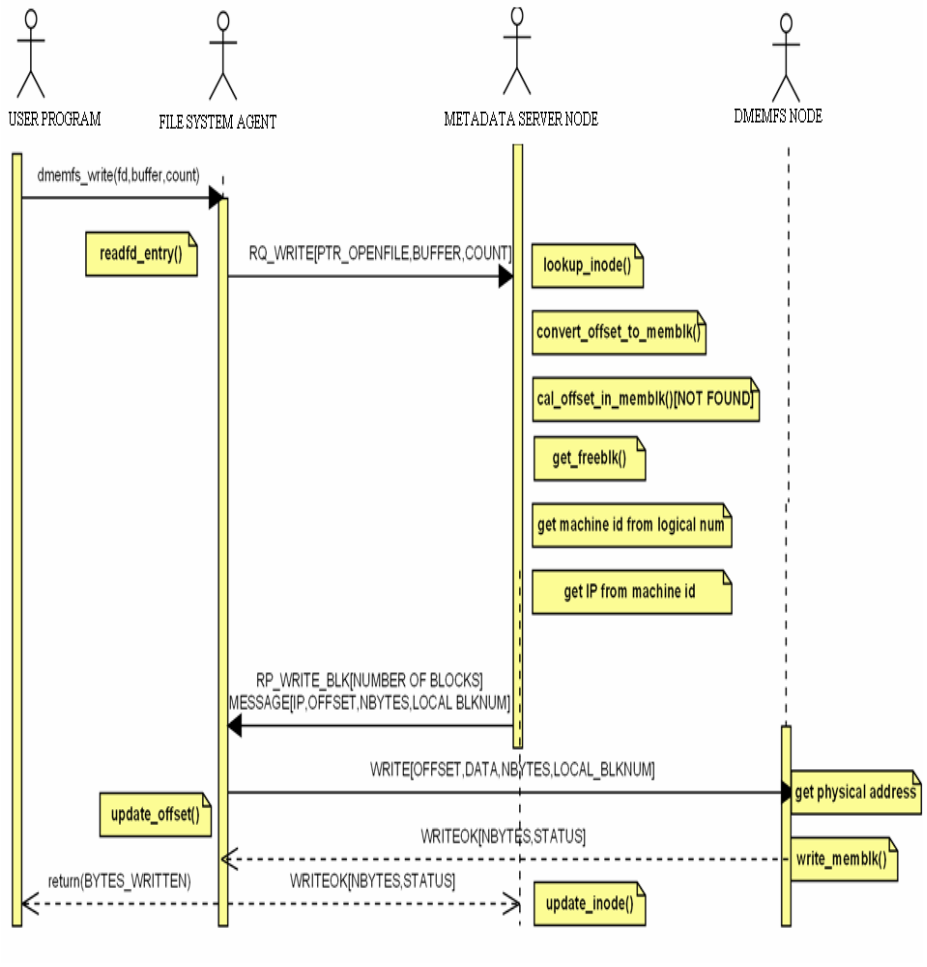


Fig. 2. Overview of write operation in distributed memory file system

5 Experimental Setup

The experiments were conducted on a cluster of 20 nodes. Each node has two quad core Intel Xeon 3.0 GHz processors with 6 MB L2 cache and 32GB physical memory. Nodes are connected by Infiniband and Gigabit Ethernet. Each node has a 7200 rpm disk of size 500 GB and a transfer rate of 3Gbps. For testing the performance of the distributed memory file system various test cases were developed that involved contiguous writing to a file, random seeking to a position in a file and reading from a file. The following sections describe few experiments and their results.

5.1 Experiment 1

In this experiment, performance of the distributed memory file system was tested for contiguous write using various size of file system block size such as 4K, 8K, 16K,

32K, and 64K. Large chunks of data in order of kilobytes were written to a file sequentially. The experiment was conducted over Gigabit Ethernet and Infiniband. Observations for various file sizes have been plotted in figure 3 (Gigabit Ethernet) and figure 4 (Infiniband).

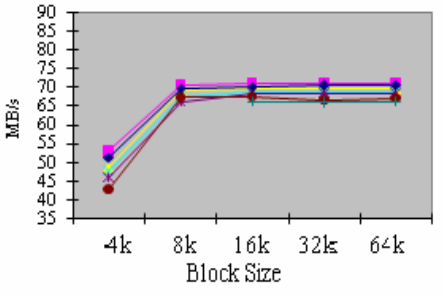


Fig. 3. Contiguous Write Performance (Gigabit Ethernet)

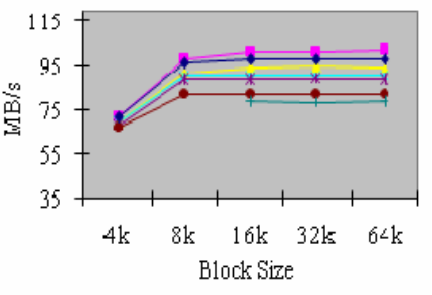
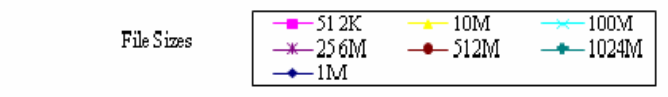


Fig. 4. Contiguous Write Performance (IP over Infiniband)



We can observe that the block size of 8K is optimum for our distributed memory file system irrespective of the underlying network bandwidth and latency.

5.2 Experiment 2

To test the file system performance for random access and read, test cases were developed that perform random seek to a position in a large file of the order of gigabytes (15GB) followed by reading chunk of data. This experiment was performed with block sizes of 8K and 16K. The comparison of performance of distributed memory file system and disk has been plotted in figure 5.

The distributed memory file system has significant performance benefits (2.3 times better) for application involving random access and read requests. This is due to the fact that seek time in distributed memory file system depends only on the time required to calculate the memory block corresponding to the seek position. Where as in disks the seek time corresponds to the amount of time required for the read/write heads to move between tracks over the surfaces of the platters which introduces a time penalty.

5.3 Experiment 3

Test cases were developed to test the file system performance for an application that writes a file contiguously in very small data chunks of the order of bytes. The comparison of performance of disk and distributed memory file system with block sizes of 8K and 16K over both the interconnects namely Gigabit Ethernet and Infiniband has been plotted in figure 6.

We observe that performance of the distributed memory file system for applications that write data to a file in small chunks in order of bytes is not so significant as compared to disk based file system as writes in the latter are performed on cache most of the time rather than disk, where as in the memory file system for every write call the additional latencies due to our implementation are involved hence, the performance gain is not so significant.

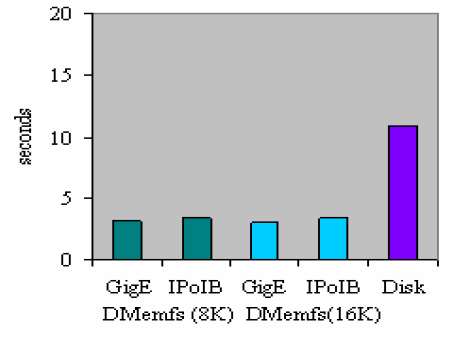


Fig. 5. Performance in experiment 2

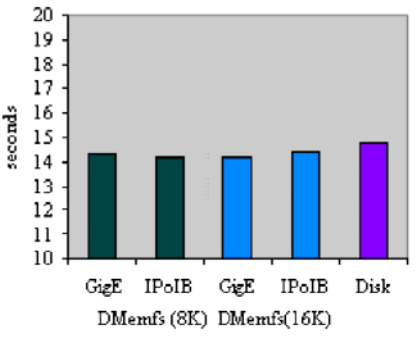


Fig. 6. Performance in experiment 3

6 Conclusion and Future Work

The Distributed Memory File System provides better performance for I/O intensive jobs in a cluster and efficiently uses the free memory available in the cluster nodes for storing file data. Further, through practical experimentation it is established that when there is random access pattern, the performance of distributed memory file system is significant (2.3 times better) as compared with that of the local disk. Our interest was also to examine whether there exists an optimal block size (analogous to the 512 bytes sector size of disk that has remained same for historical reasons) in a distributed memory file system implementation. We did observe through our experimentation that optimum size of block was 8K for optimal transfer rates. We also observed that even though the bandwidth and latencies of underlying network that interconnected memories were different, the optimum size was observed to be the same. We believe for a given network interconnect, there exists an optimal block size for a file system. Knowledge of this would help us in getting optimum throughput.

Currently the file system has been developed in user space with the client side API available as a library. In future we shall be extending this file system implementation to POSIX implementation so that its usage becomes transparent to the user. We shall be extending our research to very large file sizes and develop schemes to factor in the failure of nodes which will bring in the issues of redundancy.

Acknowledgements

We would like to express our sincere gratitude to R. S. Mundada, K. Bhatt, D. D. Sonvane, Vaibhav Kumar, Vibhuti Duggal, N. Chandorkar of Parallel Processing

Group, Computer Division, BARC for their constructive suggestions throughout the research work. We are thankful to A.G. Apte, Head, Computer Division, BARC for providing us with the opportunity to undertake this research work.

References

1. Moreira, F., Adi, J.-P., Navarro, B., Dwyer, A.O., Wright, R.: STORAGE:Ten-year Forecast of Storage Evolution. PS_WP12_HISTOR_D12-5_Storage_Ten year_Forecast_of_Storage_ Evolution (February 2006)
2. McKusick, M.K., Karels, M.J., Bostic, K.: A Pageable Memory Based File system. In: Proceedings of the Usenix Summer (June 1990)
3. Snyder, P.: Tmpfs: A Virtual Memory File System. In: Proceedings of the Autumn 1990 EUUG Conference, Nice, France, pp. 241–248 (1990)
4. Feeley, M.J., Morgan, W.E., Pighin, E.P., Karlin, A.R., Levy, H.M., Thekkath, C.A.: Implementing global memory management in a workstation cluster. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles (1995)
5. Dahlin, M., Wang, R., Anderson, T.E., Patterson, D.A.: Cooperative caching: Using remote client memory to improve file system performance. In: Operating Systems Design and Implementation, Monterey, CA, pp. 267–280. USENIX Assoc. (1994)
6. Flouris, M., Markatos, E.P.: The network Ram Disk: Using remote memory on heterogeneous NOWs. Cluster Computing, 281–293 (1999)
7. Newhall, T., Finney, S., Ganchev, K., Spiegel, M.: Nswap: A network swapping module for linux clusters. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 1160–1169. Springer, Heidelberg (2003)
8. Bach, M.J.: The Design of the Unix Operating System, Chapters (4, 5, 10)
9. Bovet, D.P., Cesati, M.: Understanding the Linux Kernel, 3rd edn. O'Reilly, Sebastopol (2005)