NASA-CR-192906

G2591

RAMA: A FILESYSTEM FOR MASSIVELY PARALLEL COMPUTERS

IN-62-CR

158661

p. 7

Ethan L. Miller and Randy H. Katz

University of California, Berkeley
Berkeley, California

ABSTRACT

This paper describes a file system design for massively parallel computers which makes very efficient use of a few disks per processor. This overcomes the traditional I/O bottleneck of massively parallel machines by storing the data on disks within the high-speed interconnection network. In addition, the file system, called RAMA, requires little inter-node synchronization, removing another common bottleneck in parallel processor file systems. Support for a large tertiary storage system can easily be integrated into the file system; in fact, RAMA runs most efficiently when tertiary storage is used.

INTRODUCTION

Disk-based file systems are nothing new; they have been used for over thirty years. However, two developments in the past few years have changed the way computers use file systems—massive parallelism and robotically-controlled tertiary storage. It is now possible to have a multi-terabyte file system being accessed by hundreds of processors concurrently. Current disk file systems are not suited well to this environment. They make many speed, capacity, and erasability assumptions about the characteristics of the file storage medium and about the stream of requests to the file system. These assumptions held with increases in disk speed and processor speed. However, massively parallel machines may have hundreds of processors accessing a single file in different places, and will require data to come from a multi-terabyte store too large to cost-effectively fit on disk. The changes in access patterns to the file system and response times of tertiary storage media require a new approach to designing file systems.

The RAMA (Rapid Access to Massive Archive) file system differs from a standard file system in that it treats the disk as merely a cache for the tertiary storage system. Because it relies on optical disk, tape, and other mass storage devices to hold the "true" copies of each file, the disk file system may use different, more efficient methods of arranging data.

This paper describes the design of the RAMA file system. The first section provides some background on relevant file systems. Next, we detail the design of the disk-based portion of the file system. We then discuss the advantages of the system, and some possible drawbacks. We conclude with future directions for research.

PREVIOUS WORK

There have been few file systems truly designed for parallel machines. While there have been many massively parallel processors, most of them have used uniprocessor-based file systems. These computers generally perform file I/O through special I/O interfaces and employ a front-end CPU to manage the file system. This method has the major advantage that it uses well-understood uniprocessor file systems; little additional effort is needed to support a parallel processor. The disadvantage, however, is that bandwidth to the parallel processor is generally low, as there is only a single CPU managing the file system. Bandwidth is limited by this CPU's ability to handle requests and by the single channel into the parallel processor. Nonetheless, systems such as the CM-2 use this method.

Some parallel processors do use multiprocessor file systems. Generally, though, these systems make a distinction between computing nodes and file processing nodes. This is certainly a step in the right direction, as file operations are no longer limited by a single CPU. These systems, however, are often bottlenecked by centralized control. Additionally, there is often a strict division between I/O nodes and processing nodes. This unnecessarily wastes CPU cycles, as the I/O nodes are idle during periods of heavy computation and the processing nodes are idle during high I/O periods. The CM-5 is an example of this type of file system architecture [1]. The Intel iPSC/2 also uses this arrangement of computation and I/O nodes [2]. In the iPSC/2, data was distributed among many I/O nodes. However, I/O services ran *only* on I/O nodes, scaling performance by just that number. This arrangement works well if I/O needs are modest, but costs too much for a large system with thousands of computation nodes. Such a system would need hundreds or thousands of distinct I/O nodes as well, requiring additional processors and a larger interconnection network, and increasing the machine's cost.

(NASA-CR-192906) RAMA: A FILE SYSTEM FOR MASSIVELY PARALLEL COMPUTERS (California Univ.) 7 p

N93-25297

Unclas

The Bridge file system [3] is one example of a truly parallel file system. In it, each processor has a disk, distributing the file system across the entire parallel computer. Bridge showed good performance, but it required that computation move to where the data actually resided for optimal performance. This approach does not work well for supercomputing applications such as climate models. For these problems, data layout is critical, as interprocessor communication must be optimized. Additionally, this arrangement fails when a single file system must be accessed by both workstations and high-performance computers. The authors reported little speedup for "naive" use of the file system, but such use is necessary if workstations are to share a file system with a parallel computer.

In [4], the authors describe a hashed-index file system for a computer with many diskful nodes. They note that the lack of a centralized index structure removes a major bottleneck, and found that they got nearly linear speedup as they increased the number of nodes with disks. The file system is aimed primarily at database-type workloads, however, as it does not consider sequential reads. Their idea is sound, though, and similar to the RAMA design presented in this paper.

RAMA FILE SYSTEM DESIGN

The RAMA file system is designed to provide high-bandwidth access for large files while not overly degrading performance for small files. It is thus best used for scientific computing environments, which require many files tens or hundreds of megabytes long [5]. The file system scales well to multiprocessors with hundreds of CPU nodes, each with one or more disks. Because little synchronization between file system CPUs is necessary, there will be little performance falloff as more CPUs are added. Figure 1 shows what hardware running the RAMA file system might look like.

The file system may span as many different disks and processors as desired. There are no intrinsic limitations on how large the file system may get, since RAMA keeps very little system-wide state. Each node must know how to convert a (*bitfile identifier, offset*) pair into a destination node, and nothing more. All other state information is kept locally on each node, and obtained by other nodes only when needed. A node may cache data and state that does not pertain to its local disk; the node that owns the data manages the consistency using any scheme for keeping distributed data consistent. This creates a bottleneck if many nodes share a single piece of data, but it allows many concurrent requests for different data. In particular, different nodes can share distinct parts of the same file without encountering a bottleneck as long as the parts of the file are on different disks. This scheme will be explained shortly.

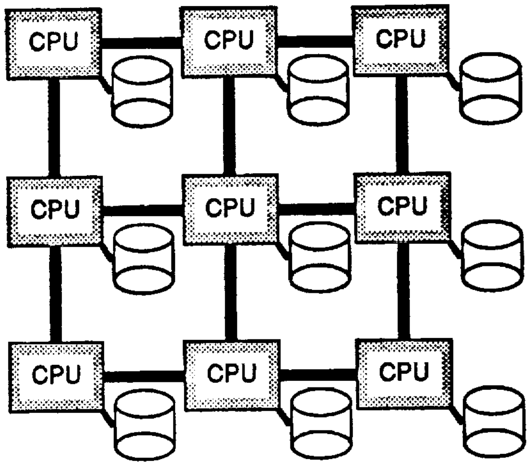

Figure 1. Typical hardware configuration for running the RAMA file system.

Tertiary storage is integrated into RAMA via user-level storage managers. Whenever a block of data is not found on disk, one of these processes is queried to find the data on tertiary storage. Clearly, this method introduces longer latency than a kernel-based storage manager would. However, latency to tertiary storage is already well over a second; the additional few milliseconds will make little difference in overall request latency. Additionally, user-level storage managers can be changed more easily to accommodate new devices and new algorithms for managing file migration, as the kernel need not be recompiled.

RAMA fits well into the Mass Storage Systems Reference Model [6]. RAMA itself acts as a bitfile server and storage server for magnetic disk. The user-level tertiary storage managers are bitfile servers for tertiary storage devices; however, they do not necessarily act as storage servers for these devices.

Data layout

In RAMA, the disk acts as a cache for tertiary storage. Unlike many file system caches, RAMA's disk is a set-associative cache. As a result, any file block can only be placed in a subset of all the disk blocks in the system. This subset is found using a hashing algorithm based solely on the file's unique identifier and the block's offset within the file. Since this is a simple algorithm that does not depend on any dynamically-changing global information, each node can have its own independent copy of the algorithm. No interprocessor synchronization is necessary to decide where on disk a needed file block is located.

This holds true regardless of the size of the file, and regardless of the size of the RAMA file system.

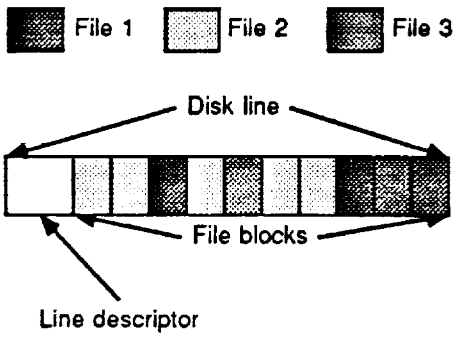

Figure 2. Structure of a disk line in RAMA.

Data blocks in RAMA are stored in *disk lines*, which consist of one or more file blocks. Each disk line has a *line descriptor* describing the data in the line's blocks, acting as a "table of contents" for the disk line. The line descriptor keeps the file identifier, block number, last access time, last modification time, and several flags for each file block in the disk line. Figure 2 shows the structure of a disk line and line descriptor. Typically, a disk line will have hundreds to a few thousand file blocks in it; performance implications of this choice will be discussed later. Since file blocks are 4-8 KB, each disk line is several megabytes long; thus, a single disk has multiple disk lines on it.

Data placement in RAMA

One major innovation in the RAMA design was motivated by a simple observation—file metadata in UNIX falls into two categories, both of which are stored in the inode and indirect blocks in standard UNIX file systems. The first type of metadata, intrinsic metadata, is information that describes the data in the file. This data includes file modification and access times, owner and protection, and file size. Because it describes the data itself and not its placement, this type of metadata must be kept with the file regardless of the medium the file is physically stored on. Positional metadata is the second type of metadata. This information tells the operating system where to find the bits contained in the file. In a UNIX system, this metadata is contained in the direct block pointers in the inode and in indirect blocks. RAMA treats and stores the two types of metadata differently, unlike UNIX systems which store both types of metadata in the inode.

In RAMA, intrinsic metadata is stored at the start of a file's data. Thus, the CPU responsible for the first few blocks of the file also manages the file's timestamps and other information about it. This arrangement has the disadvantage of slowing down directory scans such as `ls`, but has little effect during the time when a file is actually being read or written. Since high-performance computers spend far more time accessing file data than looking for the files, this is a good tradeoff for a massively parallel file system.

As already discussed, positional metadata for file blocks on disk is contained in the hashing algorithm and the line descriptors. Each file block resident on disk requires about 4 words of data in a line descriptor. If file blocks are 8 KB long, 0.2% of the disk space in the file system will be occupied by positional metadata—an acceptable overhead.

RAMA's arrangement of metadata provides several advantages for massively parallel systems. First, positional metadata is guaranteed to be near the data it describes, since disk lines occupy logically contiguous disk sectors. Thus, little or no seek time is necessary between accessing a file block's positional metadata and reading or writing the actual file data. This is not a new idea, though. The BSD 4.3 file system [7], for example, uses the same concept and keeps inodes near the files they describe.

This arrangement of metadata has another advantage. Blocks of a file are independent from each other and can be read and written by different CPUs with little interprocessor synchronization. Most file operations update little file-intrinsic metadata. Reads only change timestamps, while writes may also change file length. File length changes, however, do not require synchronization; the node responsible for keeping track of the file length simply remembers the largest size it has been told. As a result, different CPUs in a massively parallel computer may read or write two different sections of the same file without serializing their request.

Allowing each block to be accessed independently provides excellent parallelism; however, large sequential reads and writes must be broken up. Most file systems optimize data placement so large file accesses can be done with few seeks. This can be done in RAMA as well by adjusting the hashing algorithm. Instead of assigning each file block to a different disk line, the algorithm can be adjusted to map several consecutive blocks of the same file to the same line. This "sequentiality parameter" can be adjusted to trade off between faster sequential access and conflict problems. If too many consecutive blocks from a single file go to the same disk line, the line will be unable to hold much other data that hashes to the same line, degrading performance by forcing the use of tertiary storage. On the other hand, too few sequential blocks in the same line degrades performance by requiring more seeks. Simulation will determine the relationship between these two effects.

Another factor in data placement is the arrangement of data blocks within a disk line. Sequential blocks from the same file might be scattered around within the disk line, which holds hundreds of data blocks. This will provide suboptimal performance, as reading sequential blocks from a file would require rotational and head switch delays. However, this can easily be remedied. Recall that only the processor directly attached to the disk actually knows where an individual file block is actually stored on disk. If the disk is idle for even a short period of time, its disk lines may be cleaned up. This process is similar to cleaning in a log-structured file system [8], but it is not necessary for operation. Unlike an LFS, RAMA can run without cleaning, as cleaning only optimizes file block placement within a disk line. There are data integrity issues involved with reorganizing a disk line; however, these may be solved by having few spare disk lines on each disk. Instead of overwriting a disk line in place, write it to a different disk line and mark the original as the new spare.

Tertiary storage and RAMA

RAMA is designed to be used in high-performance computing environments requiring many terabytes of storage. File migration to and from slower, cheaper media must be well integrated into the file system. RAMA's data layout on disk facilitates such migration.

Each block in a disk line may be in one of three states—clean, dirty, or free. Free blocks are just that. They are always available to write new data to. Clean blocks are those that have not been modified since they were written to tertiary storage. This includes both blocks that have been written back to archive and those that have been retrieved from tertiary storage. If there are no free blocks, clean blocks may be reclaimed for new data, in order of last access time. A copy of data in these blocks exists elsewhere, so it can be retrieved if it is needed later. A migration manager may change the last access time of a clean block to change the order in which valid blocks are reclaimed (this does not affect a file's last access time, however). Finally, dirty blocks are immune from overwriting. As a result, dirty blocks must be converted to clean or free blocks faster than the overall write rate. This simply means that migration from disk to tertiary storage must, on average, be faster than the rate that long-term data is created.

Migration from secondary to tertiary storage is managed by user-level processes. There may be more than one of these processes, but they will likely be coordinated to avoid duplication of effort. This is not, however, a requirement. These processes, called *migration managers*, direct the copying of files from secondary to tertiary storage. RAMA has special hooks into the file system to allow this; in particular, these processes are allowed to change the state of a file block, marking dirty blocks as clean. These managers may also adjust the modification time of a clean block so it will be more or less likely to be written over if more disk space is needed.

A typical migration manager searches through every disk line looking for dirty file blocks older than a certain time. This finds file identifiers that are good candidates for migration to a lower level of the hierarchy. This task is easily parallelizable, using one low-level migration manager for each disk. Each low-level process reads and scans all of the line descriptors on a single disk. This is not a long procedure; a 1 GB disk has less than 4 MB of line descriptors which may be read and scanned in a few seconds. The results from all of the low-level migration managers are combined by a high-level migration manager. This migration process decides which files will be sent to tertiary storage, and manages their layout on tertiary media. It also optimizes scheduling for the tertiary media readers, trying to minimize the number of media switches.

Once a file has been written to tertiary storage, its blocks become available for reuse. However, these disk blocks are not immediately freed; instead, they are marked as clean so they may be reclaimed if necessary. There is usually no reason to free blocks once they are safely stored on tertiary media, as they might satisfy a future file request. However, the blocks' modification time might be modified. The migration manager could, for example, decide to preferentially keep blocks from small files on disk. If so, it would mark clean file blocks from large files as being older than blocks of the same age from small files. This will not confuse the file system, as a whole file's modification date remains unchanged, as does the modification date for dirty blocks. Only clean blocks which need not be written to tertiary storage may have their last access dates changed.

A typical file access

To make either a read or a write request for a file in RAMA, a node first hashes the (*bitfile ID, block number*) pair to decide which disk line the data will be found in. The request is then sent to that node, and the requesting node does nothing further until the request completes and the data is ready.

Once the node with the data receives the request, it reads in the line descriptor for the disk line with the desired data (if the line descriptor is not yet cached in memory). For a file read, the line descriptor is searched for an entry that matches the desired bitfile ID and block number. If it is found, the block is read from disk and returned to the requesting node. If it is not found, a message is sent to the tertiary storage manager requesting the block.

Writes follow a path similar to reads, except that the procedure for a "miss" is different. Instead of sending a message to the tertiary storage manager, the file system writes the data into the disk line and updates the line descriptor. Blocks marked free are used first; if none are available, clean blocks are used in order of desirability as set by the migration manager when the blocks were written to tertiary storage. If all of the blocks in a line are dirty, RAMA sends an emergency message to the migration manager, requesting that the line be cleaned. The request cannot finish until this is done. This last resort is similar to thrashing in a CPU cache, and exhibits poor performance.

IMPLEMENTATION ISSUES

Hashing algorithm

As [3] and [4] noted, parallel file systems without bottlenecks can achieve near-linear speedup if data is distributed well among all diskful nodes. A file system for scientific computation introduces additional problems, however. The first is sequential access. Most scientific computation involves large sequential reads and writes to the I/O system [5], so these requests must run quickly. If each block lives on a different processor-disk pair, a single half-megabyte read would need to contact 64 different nodes. Since there is a per-request overhead for each disk, this approach is inefficient. Also, it may excessively congest the interconnection network, as a single I/O sends messages to many different nodes. The disks, too, are being used inefficiently if many processes are using the file system. Instead of few large requests, disks see many small requests and spend all their time seeking to the correct locations.

To fix this problem, we adjust the hashing algorithm to keep sequential blocks from the same file in the same disk line. In the ideal case, all of these blocks could be read without any seeks. This will often not be the case, though, since the disk line will become fragmented. Even so, a disk line occupies only a few cylinders, so seeks between sequential blocks in a file are minimized. Additionally, a disk line can be reorganized to put related blocks together. As noted above, this can be done without notifying any other processors or disks. One open question, then, is how many consecutive blocks from the same file to put into a single disk line. If the number is too small, interconnection network and CPU overhead become too high. However, if the number is too high, there will be little speedup for large reads since different requests for the same file must go to the same location and be serialized. A high number may also cause pollution of a disk line, as a single file's blocks can occupy a large fraction of an entire line, crowding out other files' blocks.

Interconnection network congestion

Our initial design assumes that the interconnection network will not be the bottleneck for the file system. We feel that this is a valid assumption because a typical network will run at over 10 MB/sec, while a small inexpensive disk transfers data at only 1-2 MB/sec. However, this does not take network congestion into account. If both requests and data to fulfill those requests are evenly spread around the parallel processor, congestion will not be a constant problem. However, we will examine the effects of temporary "hot spots" in the file system. In the simulation in [4], hot spots did not present a major problem.

Data integrity and availability

As with any new file system, data integrity is a major issue. The problem is especially acute in RAMA, since a single file may be spread over many disks run by many different processors. Similarly, data availability becomes a problem when parts of a single file are stored in many different places, as the file is unavailable if any of those disks or processors is down.

Data integrity is the more important issue, as a file system must never lose data entrusted to it. RAMA must ensure that writing a new block of a file is an atomic operation. This can present a problem, since there may be two separate disk I/Os to write a file block to disk. The line descriptor must be updated, and the block itself must be written to disk. If only one operation is completed, the file system is in an inconsistent state. In a high-performance computing environment, losing the last few seconds of file I/O is not fatal, as long as the application knows that the loss occurred.

We have come up with several options for insuring that the file system remains consistent. The first writes the line descriptor in two passes, writing any file blocks between the passes. In the first pass, all blocks being written are marked as free. The new blocks themselves are then written to disk. The second pass writes the descriptor with the new table of contents. If a crash occurs before the descriptor is written the second time, all of the new file blocks are instead marked as free. This means that their data is lost, but the file system remains consistent. This method works well, but it requires three separate disk I/Os to write a file block.

We believe that a better option is to use self-identifying blocks on disk. Each block would reserve a few words to record the file identifier and block number that the data in the block corresponds to. This method has several major advantages. First, crashes no longer present any problem since the line descriptor can be rebuilt by scanning the disk line. The line descriptor is then kept only to make accessing the disk more efficient. Rebuilding the line descriptor after each crash may take some time, however. To avoid doing this, the file system assumes that all descriptors are · correct, and only rebuilds one when it finds a disagreement between a line descriptor and the self-identification for a block in its line. Another advantage for this method is that line descriptors may be written back lazily. This trades off recovery time after a crash with efficiency during normal operation. All of these benefits are countered by a few drawbacks, however. A file block is no longer the same size as a disk block, and file blocks are no longer a power of 2 bytes long. Many programs are optimized to take advantage of specific file block sizes, and it is not clear what effect changing the size will have. Another minor problem is the increased amount of metadata the file system will need. The overhead for metadata would double with a naive implementation that keeps a copy of all metadata in the file block as well. Keeping a full copy is unnecessary, though, and this overhead is only an additional 0.2% in any case.

File availability is another problem that RAMA must conquer. Uniprocessor file systems spanning more than one disk may arrange disks in a RAID [9] to keep data available even when a disk has failed. It should be possible to use similar techniques for RAMA. However, it is not clear how they would be integrated into the file system, since each node may rearrange its own disks without notifying other nodes. Under the current design, then, files are unavailable when a processor or disk is down. This is a serious problem, and will be addressed by future research.

FUTURE WORK

RAMA is currently a rough file system design. We are implementing a simulator to test our ideas and refine the design. The simulator will show us whether RAMA does indeed provide a linear speedup with more processors, and whether network congestion is likely to be a problem.

Once our design has been refined by feedback from simulation results, we hope to build a real file system on a parallel processor with a disk at each node. This will allow us to benchmark our file system using real applications. Additionally, it will show us whether there are long-term problems that might not show up in simulations covering only a few days of simulated time.

There is also much work to be done integrating tertiary storage with RAMA. In this paper, we have said little about the user-level migration managers that will move data between disk and tertiary storage. Algorithms for these programs need to be developed, using data from studies such as [10] and [11]. A RAMA system with a 500 MB disk on each of 256 nodes will have a 125 GB file system; this is somewhat larger than that on most high-performance computers today. The large disk space may permit new prefetching algorithms that reduce the number of requests that stall waiting for data from tertiary storage.

CONCLUSIONS

This paper has presented the design of RAMA, a massively parallel file system to support scientific computation. RAMA's design allows it to provide high bandwidth to computationally intensive applications such as climate modeling and computational fluid dynamics while still providing efficient support for many workstations. By spreading data across disks and allowing blocks of a file to be accessed independently, RAMA provides a scalable file system for the massively parallel computers that will solve the large problems of the future.

BIBLIOGRAPHY

1. *CM-5 Technical Summary*. Thinking Machines Corporation, October 1991.

2. Paul Pierce. "A concurrent file system for a highly parallel mass storage system." In *Proceedings of the 4th Conference on Hypercubes*, 1989, pp. 155-160.

3. Peter Dibble, Michael Scott, and Carla Schlatter Ellis. "Bridge: a high-performance file system for parallel processors." In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988, pp. 154-161.

4. Amnon Barak, Bernard Galler and Yaron Farber. "A holographic file system for a multicomputer with many disk nodes." Technical Report 88-6, Dept. of Computer Science, Hebrew University of Jerusalem, May 1988.

5. Ethan Miller and Randy Katz. "Input/output behavior of supercomputing applications." In *Proceedings of Supercomputing '91*", November 1991, pp. 567-576.

6. Sam Coleman and Steve Miller. "Mass storage system reference model: Version 4." IEEE Technical Committee on Mass Storage Systems and Technology, May 1990.

7. Marshall Mckusick, William Joy, Samuel Leffler, and Robert Fabry. "A fast file system for UNIX." *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

8. Mendel Rosenblum and John Ousterhout. "The LFS storage manager." In *USENIX — Summer 1990*, June 1990, Anaheim, CA, pp. 315–324.

9. David Patterson, Garth Gibson and Randy Katz. "A case for redundant arrays of inexpensive disks (RAID)." In *Proceedings ACM SIGMOD*, June 1988, pp. 109--116.

10. Ethan Miller and Randy Katz. "An analysis of file migration in a Unix supercomputing environment." To appear in *USENIX—Winter 1993*, January 1993.

11. David Jensen and Daniel Reed. "File archive activity in a supercomputer environment." Technical Report UIUCDCS-R-91-1672, University of Illinois at Urbana-Champaign, April 1991.