PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

The Design of the PIMOS File System

Fumihide ITOH Takashi CHIKAYAMA Takeshi MORI Masaki SATO
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

Tatsuo KATO Tadashi SATO
Mitsubishi Electric Computer Systems (Tokyo) Corporation
87-1, Kawakami-cho, Totsuka-ku, Yokohama 244, Japan

Abstract

This paper describes the design and implementation of the PIMOS file system. The file system was designed for loosely-coupled multiprocessor systems, where caching is essential for reducing not only disk accesses but also for communication between processors. To provide applications with flexible load distribution, the caching scheme has to support consistency semantics under which modifications of a shared file immediately become visible on other processors. Two different caching schemes, one for data files and the other for directories, have been designed. This is necessary because they have different access patterns. Logging the modifications of directories and other essential information secures the consistency of the file system in case of system failure. Multiple log areas reduce the time required to write logs. Buddy division of blocks enables released blocks to be collected efficiently. Hierarchically organized free block maps control buddy division.

The file system has been implemented on PIM.

1 Introduction

PIMOS [Chikayama *et al.* 1988] has been developed by the Fifth Generation Computer Systems project of Japan as the operating system for PIM [Goto 1989] as a part of the parallel inference system for knowledge information processing. PIMOS has a file system which was designed to realize a robust file system optimized for loosely-coupled multiprocessor systems like PIM. This paper describes the design and implementation of this file system.

The file system for the parallel inference system should provide a bandwidth broad enough to support knowledge information processing application software running on high performance parallel computers. To allow flexible load distribution, the semantics it provides should be location-independent. That is, the contents of files should look the same to the program regardless of the processor it is running on.

File systems on external I/O systems poorly meet requests from multiprocessors, due to limited communication bandwidths. We, thus, constructed an internal file system on disks incorporated into multiprocessor systems. Distributed file systems are similar to our file system in that shared files are accessed from processors connected via a network, with some communication delay. However, the communication bandwidth of the network is much broader in our case. Also, processes using files are normally cooperate rather than compete. These considerations affect the design.

We have clarified functions essential to file systems for loosely-coupled multiprocessor systems, and then considered how to implement them. Although the system is an experimental one, we included the essential features of practical file systems in our design, such as disk access optimization. The system has been implemented as a part of PIMOS, in concurrent logic language KL1 [Ueda and Chikayama 1990].

2 Design Principles

In order to draw parallelism from loosely-coupled multiprocessor systems, centralizing loads to a small number of server processors with disks should be avoided.

The cost of communications between processors is more expensive in loosely-coupled multiprocessor systems than in tightly-coupled ones, and the cost of disk accessing is still more expensive than that of communication. Thus, both disk accessing and interprocessor communication should be reduced. This necessitates distributed caching.

Data cached in memory may be lost upon system failure. For data files, the loss is limited to files being modified at the time of the failure. Loss in a modified directory, however, may cause inconsistency in the file system, such as a deleted, nonexistent file still being registered in a directory. The loss may spread to files under the directory, even though they were not accessed at the time of the failure. Consequently, the file system needs pro-

tection against failure to preserve its consistency.

Disk access optimization is one of the primary features of practical file systems. Most of the overheads in disk accesses are seeks, so a reduction in seek cost, i.e., the number of seeks and per-seek cost, is required.

3 Design Overview

We allowed multiple servers to distribute the loads of file accesses. A caching mechanism was incorporated to reduce disk accesses and communication among processors. A logging mechanism secures the consistency of the file system against system failure. A disk area management scheme similar to that of conventional file systems reduces the seek time for disk accessing. An overview of these features is described in this section.

3.1 Multiple Servers

In order to draw parallelism from multiprocessor systems, load centralization should be avoided. File systems have inherent centralization in that a disk can be accessed only by a processor connected to it. Multiple disks connected to multiple processors with a server running on each relaxes centralization and make the system scalable.

A processor with disks can run a server, but the processor is not dedicated to it. The server processors also operate as clients when their disks are not accessed, providing better utilization of computational resources on multiprocessor systems.

3.2 Caching Mechanism

In order to reduce disk accessing and interprocessor communication, data files and directories are cached onto all processors that access them.

3.2.1 Caching of Data Files

Consistency semantics for caches of the same data on different processors has been realized. The execution of an application program on a multiprocessor system is distributed among processors. The strategies of distribution are diverse and depend upon the application. In order to distribute computation flexibly, file access result must be identical no matter which processor accesses the file. In other words, modification by another processor has to be visible immediately.

This kind of consistency semantics is called Unix semantics [Levy and Silberschatz 1989] in distributed file systems. It was originally introduced to maintain software compatibility between distributed and conventional uniprocessor Unix systems. For the same reason, Unix semantics are indispensable to a file system for multiprocessor systems.

There is no problem in sharing a file when all the sharers merely read the file. When a file is shared in write mode, the simplest way to support Unix semantics is to omit caching and centralize all accesses to the file on the server. This method is reasonable in the environments where shared files are rarely modified. On multiprocessor systems, where processors solve problems cooperatively, modifying shared files is quite common, since distributing the computational load between processors, including file accessing, is essential for efficient execution.

Consequently, a caching mechanism is designed in which a shared file can be cached even if it can be modified and Unix semantics are preserved.

3.2.2 Caching of Directories

In order to identify a file, the file path name is analyzed using directory information. The caching of directories along with the caching of data files can be used to avoid the centralization of loads to server processors and reduce communication with those processors.

Accessing directories is quite different from accessing to data files. Data files are read and written by users, and the contents of files are no concern of the file system. On the other hand, the contents of directories form a vital part of the file system. Thus, a different caching mechanism for directory information was designed.

3.3 Logging Mechanism

Modifications of directories and other information vital for the file system are immediately logged on disk. Modifications are made to data files much more often, and writing all modifications immediately to a disk decreases performance severely. Instead, we provide a mechanism which explicitly specifies the synchronization of a particular file.

Simply writing to a disk immediately does not assure the consistency of the file system. For example, if the system fails while moving a file from one directory to another, the file may be registered in either both or none of the directories, depending on the internal movement algorithm. This inconsistency can be avoided by two-phase modification. First, any modification is written as a log to an area other than the original. Second, the original is modified when logging is complete.

If the system fails before the completion of the logging, the corresponding modification is canceled. If the system fails after completion of the logging but before the modification of the original, the original is modified using the log in a recovery procedure, validating the corresponding modification. In either case, the consistency of the file system is preserved. The system may fail while a log is being written, leaving an incomplete log. In order to detect this, we introduced a flag to indicate the end of a log that corresponds to an atomic modification

transaction.

The completion of logging can be regarded as completion of the modification. The original may be modified at any time before the log is overwritten. This means that logging does not slow down response time. Rather, it improves response time. For example, when a file is moved, two directories have to be modified. The modification of the two originals may need two seeks. Writing the log needs only one seek. Moreover, we use multiple log areas and write the log to the area closest to the current disk head position to reduce the seek time.

A log contains the disk block image after modification. Because the block corresponding to a more recent modification overrides the older modifications, only the newest constituent must be copied to the original. The more times the same information is modified, the less times the original is modified. Frequent modification of the same information, which is known to be the case in empirical studies [Ousterhout *et al.* 1985], minimizes the throughput decline caused by extra writing for logging.

Each log area is used circularly, overwriting the oldest log with a new log. In order to reduce disk accessing, the modifications of the original should be postponed as long as possible, that is, until immediately before the corresponding log is overwritten. To detect the logical tail of a log area, namely the last complete log, each log block has a number, named a log generation, which counts the incidences of overwriting the log area.

The multiplicity of log areas has caused a new problem to arise: how can the newest block be determined after a system failure. If there is only one log area, the newest log block is the closest one to the logical tail of the log area, and the log blocks are always newer than or as new as the corresponding original block. However log blocks in different areas do not show the order in which they were written. If the newest log block is overwritten after it is copied to the original block, the original block is newer than the remaining log blocks. We have solved this problem by attaching a number, named a block generation, to the log blocks and to the original blocks. The block generation counts incidences of modifying the block.

3.4 Disk Area Management

To reduce the number of seeks, the unit of area allocation to files should be made larger. Larger blocks cause lower storage utilization, as a whole large block must be allocated even for small files. Our solution is to provide two or more sizes of blocks and to allocate smaller blocks to small files.

To reduce the time per seek, a whole disk is divided into cylinder groups, and blocks of one file are allocated in the same cylinder group as much as is possible. The log areas mentioned in the previous subsection are placed in each cylinder group.

These methods are commonly used in conventional file systems. A unique feature of the PIMOS file system is buddy division of a large block into small blocks, which reduces disk block fragmentation.

4 Implementation

4.1 Multiple Servers

The whole file system consists of logical volumes, each of which corresponds to one file system of Unix. A logical volume can occupy the whole or a part of a physical disk volume. The processor connected to the disk becomes the server of files and directories in the logical volume. Logging and disk area management in the volume is also the responsibility of the server.

4.2 Data File Caching Mechanism

4.2.1 Overview

To realize Unix semantics with reasonable efficiency on loosely-coupled multiprocessor systems, we decided to stress the performance of exclusive or read-only cases, and tried to minimize disk accesses and interprocessor communication in such cases.

The unit of caching is a block, which is also the unit of disk I/O. This simplifies management and makes caches on server processors unnecessary. A processor where caches are made is called a client, as in distributed file systems. Each client makes caches from all the servers together and swaps cached blocks by the least recently used (LRU) principle. Unix semantics is safeguarded by modifying the cache after excluding caching on other clients.

The caching mechanism is similar to that for coherent cache memory [Archibald and Baer 1986]. While a coherent memory caching scheme depends on a synchronous bus, our platform, a loosely-coupled multiprocessor system, provides only asynchronous message communication. This means that we must consider message overlaps.

A client classifies each cached block into five permanent states, according to the number of sharers and the necessity of writing back to the disk. In addition, there are three more temporary states. In the temporary states, the client is awaiting a response from the server to its request.

A server does not know the exact state of cached blocks, but only knows which clients are caching the blocks. Requests for data, replies to the requests, and other notifications needed for coherence are always transferred between the server and clients, rather than directly between clients. Cached data itself may be transferred directly between clients.

4.2.2 Cache States

The principle for keeping cache coherence is simple: allowing modification by a client only when the block is cached by no other client. To realize this, "shared" and "exclusive" cache states are defined. Permanent cached block states can be as follows:

Invalid (I) means that the client does not have the cache.

Exclusive-clean (EC) means that the client and no other clients have the unmodified cache.

Exclusive-modified (EM) means that the client and no other clients have the modified cache.

Shared-modified (SM) means that the client has the modified cache, and some other clients may or may not have cache for the same block.

Shared-unconcerned (SU) means that the client has the cache but does not know whether it was modified, and some other clients may or may not have cache for the same block.

Temporary cached block states can be as follows:

Waiting-data (WD) means that the client does not have and is waiting for the data to cache, and that the data can be shared with other clients.

Waiting-exclusive-data (WED) means that the client does not have and is waiting for the data to cache, and that the data cannot be shared with other clients, as the client is going to modify it.

Waiting-exclusion (WE) means that the client already has the cache and is waiting for the invalidation of caches on all other clients. In other words, the client is waiting to become exclusive.

4.2.3 State Transition by Client Request

A request from a user to a client is either to read or to write some blocks. Another operation needed for a cache block is swap-out, i.e., to write the data back to the disk forcibly by LRU. This request or operation is accepted only in permanent states, and is suspended in temporary states as the client is still processing the previous request.

The state transition for a request to read is shown in Figure 1(a). If the state is **I**, the client requests the data to the server, changes its state to **WD**, and waits. After a while, the server reports the pointer to the data and the state to change to. The pointer points to another client when it already has the data, or to the server when the server read the data from the disk because no clients have the data. The client reads the data, lets the user read it, and changes to **EC**, **SM**, or **SU** according to the report. If the state was originally **EC**, **EM**, **SM**, or **SU**, the client simply lets the user read the available data and stays in the same states.

The state transition for a request to write is shown in Figure 1(b). If the state is **I**, the client requests exclusive data to the server, changes to **WED**, and waits. After a while, the server reports the pointer. The client reads the data, lets the user modify it, and changes to **EM**. If the state was originally **EC** or **EM**, the client lets the user modify the data immediately, and changes to or stays in **EM**. If the state was **SM** or **SU**, the client requests the server to invalidate caches in other clients, changes to **WE**, and waits. Then, if the server reports completion of the invalidation, the client lets the user modify the data and changes to **EM**. Another client may also request the invalidation simultaneously, and its request may reach the server earlier. In this case, the server requests the invalidation of the cache, and the client abandons the cache and changes to **WED**. Eventually, after the server receives the request to invalidate from the client, the pointer to the data is reported.

The state transition for swap-out is shown in Figure 1(c). The client reports the swap-out to the server, and changes to **I**. If the state is **EM** or **SM**, the pointer to the data is also reported at the same time. The server reads the data and writes it back to the disk when it cannot make any other client **EM** or **SM**. If the state is **EC** or **SU**, writing the data back to the disk is not required, as the data is either the same as that on the disk or is cached by some other client.

4.2.4 State Transition by Server Request

A request from the server to a client is either to share, to yield, to invalidate or to synchronize the cache. It is accepted not only in permanent states but also in temporary states.

A request to share is caused by a request to read by another client. The state transition for this is shown in Figure 2(a). If the state is **EC** or **SU**, the client reports the pointer to the data and indicates that the requesting client should change to **SU**. In each case, the state of the requested client after replying is **SU**. If the state is **EM** or **SM**, there is a question of which client should take responsibility for writing back the data. In the current design, the requesting client takes it. Consequently, the requested client reports the pointer to the data and indicates that the requesting client should change to **SM**. The requested client becomes **SU**.

A client may receive a request to share in state **I**, if swap-out overlaps with the request. In this case, the server knows of the absence of the data when it receives the swap-out. The client consequently ignores the request. Moreover, the client will possibly receive the request while awaiting data after swapping it out in **WD** or **WED**. The client can also ignore the request. Fur-

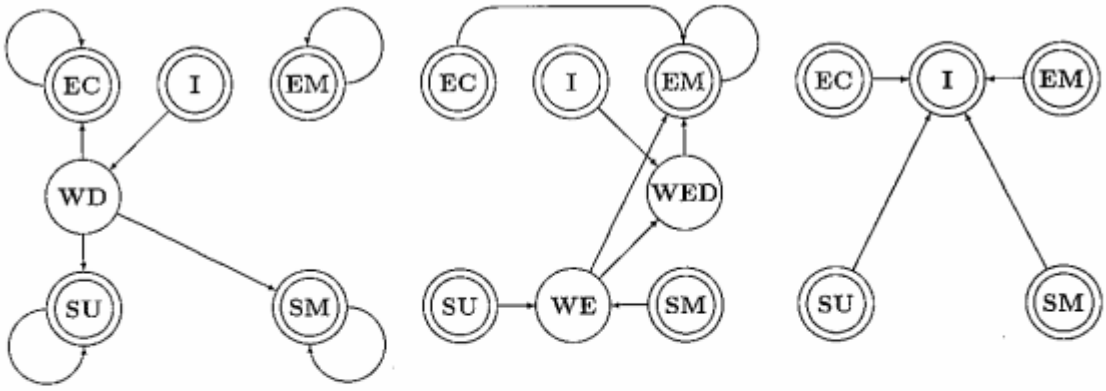

Figure 1: State transition diagrams by a request to the client

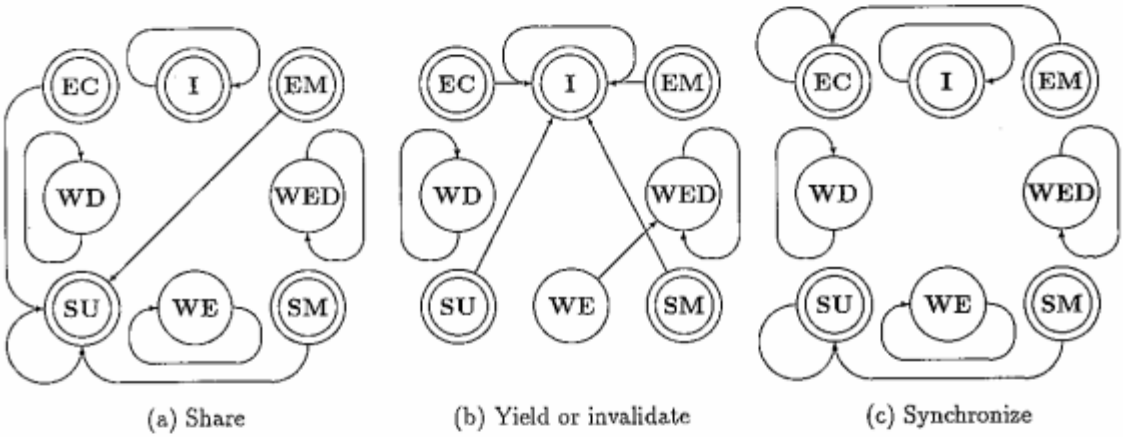

Figure 2: State transition diagrams by a request from the server

thermore, the client can receive the request in **WE** if the request to invalidate other caches overlaps with the request. In this case, the client, while waiting in **WE**, reports the pointer to the data and indicates that the requesting client should change to state **SU**. Completion of the invalidation will be reported, after the request of invalidation is received by the server.

The actions of a client for requests to yield and to invalidate are the same, except when the pointer to the data is reported. State transition is shown in Figure 2(b). If the state is **EC**, **EM**, **SM** or **SU**, the client reports the pointer to the data or simply abandons the cache, and changes to **I**. The client can receive the requests in **I**, **WD**, or **WED** and ignore them, for the same reason as when a request to share is received. If the state is **WE**, the client reports the pointer or abandons the cache, and changes to **WED**, as described in the case of a request to write to the client.

On a request to synchronize, the server requests a client to send the data and write it back to the disk. State transition is shown in Figure 2(c). If the state is **EM** or **SM**, the client reports the pointer to the data and changes to **EC** from **EM**, or **SU** from **SM**. If the state is **EC** or **SU**, the client reports that writing back is unnecessary. If the state is **I**, **WD**, or **WED**, the swap-out has overlapped with the request. The client may ignore the request because the server will receive the swap-out message with or without the pointer to the data. If the state is **WE**, the client reports the pointer, while awaiting completion of the invalidation in **WE**.

The temporary states enable message overlaps to be dealt with efficiently.

4.3 Directory Caching Mechanism

Most accesses to directories are to analyze file path names. In order to analyze a file path name on a client, directory information is cached. The unit of caching is

one member of a directory. Each client swaps caches by LRU. The server maintains information on the directories and members that clients cache. The server also caches the disk block images of cached directories, and when a member is added or removed, it modifies the images and writes them to the disk as a log.

When a file path name is analyzed on a client, the members on the path are cached to the client one after another. If the same members appear on subsequent path name analyses, the cached information is used. When a member is added to a directory, the member is added to caches after the addition is logged by the server. These operations require communication only between the client and the server.

When a member is removed, the removal is notified to the server. The server requests the invalidation of the cache to all the clients caching the member. After the server has received acknowledgement of invalidation from all the clients, the server writes a log, thus completing the removal. Although this removal may take time, it is not expected to affect the total throughput of directory caching because frequently updated members of directories are not likely to be cached by clients other than the one that modifies them.

Information about access permission is also necessary for analyzing path names. Therefore, it is cached in the same way as directory information.

4.4 Logging Mechanism

4.4.1 Log Header

In order to manage logging, each block in the logs and the originals has a header consisting of the following items. Except for block generation, the information has no relevance in the original.

Block identifier shows the corresponding original block. It consists of a file identifier and a block offset in the file.

Log generation counts the number of times a log area is used.

Atomic modification end is a flag which shows the last block of a log corresponding to an atomic modification.

Block generation counts the number of times a block has been modified in order to identify the newest block among logs and the original.

Because the size of a header is limited, the maximum size of numbers allowed in the log generation and block generation items are limited. However, as we will discuss, three log generations, at most, can exist in a log area at any one time. This means that the cyclic use of three or slightly more generations is sufficient. Limited numbers for block generations can also be used cyclically. This assures that the newest block is always spotted in the following way.

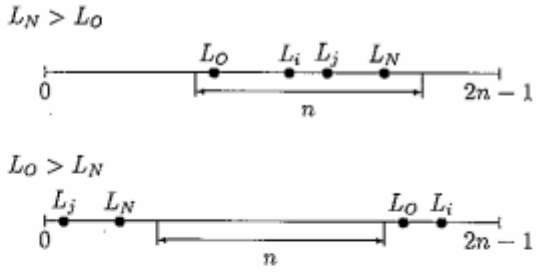

Figure 3: Distribution of block generations

Suppose that $2n$ numbers, from 0 to $2n-1$, are used for block generation. Block generation starts from 0, increases by 1 until $2n-1$ is reached, and returns to 0. We have introduced the following control: if the absolute value of the difference between the block generation of the log block to be written next, L_N, and that of the oldest existent block, L_O, is equal to n, the oldest block, whether it is in a log or is the original, is invalidated before the next log is written. Distribution of block generation under this control is shown in Figure 3. As is shown, the invariant condition is that $L_N - L_O < n$ if $L_N > L_O$, and $L_O - L_N > n$ if $L_O > L_N$.

Consequently, after a failure, the newest block is spotted as follows. The distribution of block generations dictates that either all of the generations are in a range narrower than n or that the distribution has a gap wider than n. In the former case, the newest block is the one which has the largest generation. In the latter case, it is the one which has the largest number in the group below the gap.

In practice, the invalidation of the oldest block occurs rarely. Our current implementation allocates 24 bits for block generation. Invalidation occurs only if there are $2^{23} = 8,388,608$ modifications to the same block and, in addition, if the oldest block happens not to have been overwritten by modifications. However, even one modification every ten milliseconds during one whole day barely amounts to $100 \times 60 \times 60 \times 24 = 8,640,000$.

4.4.2 Logging Procedure

While the file system is in operation, logs are written as follows:

1. Create after-modification images of a set of blocks. Set block identifiers and block generations. Line up the blocks and set an atomic modification end flag to the log header in the last block.

2. Choose the log area in the cylinder group where the disk heads currently reside. Set log generations to the log headers. Write the log, the sequence of the blocks, to the log area.

3. Report the completion of logging.

4. Make room in the log area for the subsequent writing. In other words, if the newest blocks are in the part where the next log to the area will be written, copy them to the corresponding original blocks.

5. Invalidate the oldest blocks if necessary, i.e., if there are blocks whose next modifications will require them to be invalidated. Invalidation is performed by setting a null block identifier to the log header.

Making room and invalidating can be done at any time before the next log is written. It should be done immediately after logging to get the best response at the next logging. The size of the room is made to be the maximum size of a log corresponding to an atomic modification, or slightly more.

When a logical volume is dismounted, all the newest blocks are written to the corresponding originals.

The following tables in memory are used to control logging:

Log area table maintains the next log position and the log generation in each log area.

Log record table maintains the block identifier corresponding to each position in the log areas.

Log block table maintains, for every block that has at least one log, the position and the block generation of each log, and the block generation of the original.

4.4.3 Recovery Procedure

After a system failure, the tables for log management are recovered as follows:

1. Find out decreasing points in log generation in each log area.

2. Choose the first of the decreasing points as the tentative logical tail of each log area.

3. Find out the real logical tail of each log area by rejecting the incomplete log from the tentative logical tail.

4. Decide the logical head of each log area and recover the tables from valid log blocks.

Decreasing points in log generation show that the log blocks were logged last before system failure or were being logged at the time of the system failure. There may be more than one decreasing point if an intelligent disk drive changes the order of writing of physical blocks to promote efficiency. In this case, there is also one less increasing point than the number of decreasing points, and the decreasing and increasing points are distributed in the range of one atomic log. Taking into account the circular use of a log area, the log generation of the physical first block is usually one larger than that of the physical last block. If the two generations are equal, the physical tail of a log area is one of the decreasing points in log generation. Examples of the distribution of log generations are shown in Figure 4. There can be one, two, or three log generations in a log area.

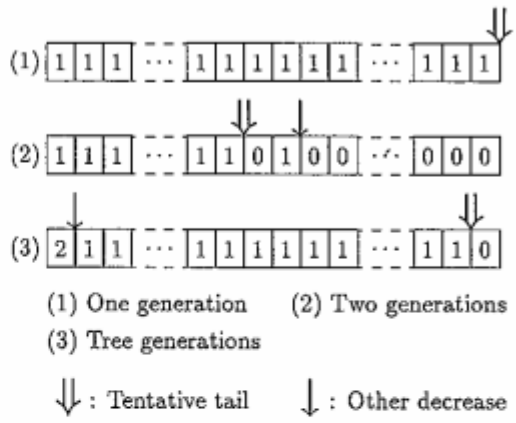

Figure 4: Distribution of log generations

If there is only one decreasing point in log generation, it becomes the tentative logical tail. If there are two or more decreasing points, the first one is selected as the tentative logical tail. The real logical tail is immediately after the last block with an atomic modification end flag before the tentative logical tail. Two tails are identical if the block immediately before the tentative logical tail has the flag.

The logical head is a certain number of blocks away from the real logical tail. The number of blocks corresponds to the room made for the next log writing. Valid log blocks consist of the blocks between the logical head and the real logical tail. After the tables are recovered, the file system can start operation.

4.5 Disk Area Management

To manage the buddy division of large blocks, we use a hierarchy of free block maps in memory as shown in Figure 5. Each free block is registered as free in only one map. We also maintain the number of free blocks registered in each map.

When a free block of a certain size is required and the map of that size has enough free blocks, the map is searched. If it does not have enough free blocks to make

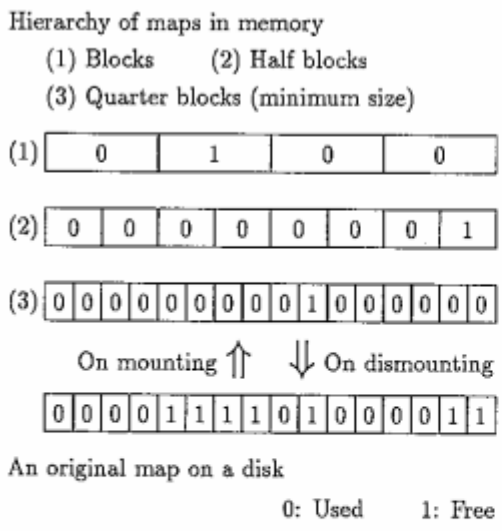

Figure 5: Hierarchy of free block maps

the search efficiently, the map for blocks of twice the size is searched. This continues until the map of the largest block size is reached.

When a block is released and the buddy of the block is free, the two blocks are united and become one free block of twice the size. Otherwise, the released block alone becomes free.

The hierarchy of maps is unfolded from the free block map on a disk whose unit is the smallest block when the logical volume is mounted. It is folded into the original map and saved on the disk when the volume is dismounted.

We use the two-step allocation method common to conventional file systems. In the free block map of the largest block size in memory, only some of the free blocks are registered as free. Another map of the largest block size is made and written to the disk where, in addition to the original used blocks, the free blocks registered as free in memory are registered as used. In this way, the map ensures that the blocks registered as free on it are free, though those registered as used are not necessarily used. Consequently, the file system can start up after a system failure, using the map of the largest block size on the disk, without a time-consuming scavenging operation.

When free blocks in memory become scarce, some are added to the map in memory, and the map entries on the disk corresponding to those blocks are changed to "used". Conversely, when free blocks in memory become surplus, some are removed from the map in memory, and the map entries on the disk corresponding to those blocks are changed to "free". The scarcity and the surplus are judged based on threshold numbers of free blocks in memory.

5 Conclusion

The design and implementation of the PIMOS file system has been described. A multiplicity of servers distributes the file system loads to them and draws out scalability from multiprocessor systems. The caching mechanism, which guarantees Unix semantics, enables applications, including file accessing, to be executed in parallel easily. The logging mechanism secures the consistency of the file system against system failure. The buddy division of free blocks suppresses fragmentation without much overhead.

We are already implementing the file system on PIM. The tuning of parameters and the evaluations of the file system are to be done in the future.

Acknowledgement

We would like to thank Mr. Masakazu Furuichi at Mitsubishi Electric Corporation and Mr. Hiroshi Yashiro at ICOT for their intensive discussions. We would also like to express our thanks to Dr. Shunichi Uchida, the manager of the research department, and Dr. Kazuhiro Fuchi, the director of the research center, both at ICOT, for their suggestions and encouragement.

References

[Archibald and Baer 1986] J. Archibald and J. L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, Vol. 4, No. 4 (1986), pp. 273-298.

[Chikayama *et al.* 1988] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. Int. Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, 1988, pp. 230-251.

[Goto 1989] A. Goto. Research and Development of the Parallel Inference Machine in FGCS Project. In M. Reeve and S. E. Zenith (Eds.), *Parallel Processing and Artificial Intelligence*, Wiley, Chichester, 1989, pp. 65-96.

[Levy and Silberschatz 1989] E. Levy and A. Silberschatz. Distributed File Systems: Concepts and Examples. TR-89-04, Department of Computer Sciences, The University of Texas at Austin, Austin, 1989.

[Ousterhout *et al.* 1985] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer and J. G. Thompson. A Trace-Driven Analysis for the UNIX 4.2 BSD File System. In *Proc. 10th ACM Symposium on Operating Systems Principles*, ACM, New York, 1985, pp. 15-24.

[Ueda and Chikayama 1990] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (1990), pp. 494-500.