Office de la Propriété Intellectuelle du Canada
Un organisme d'Industrie Canada

Canadian Intellectual Property Office
An agency of Industry Canada

CA 2504322 C 2015/07/14

(11)(21) **2 504 322**

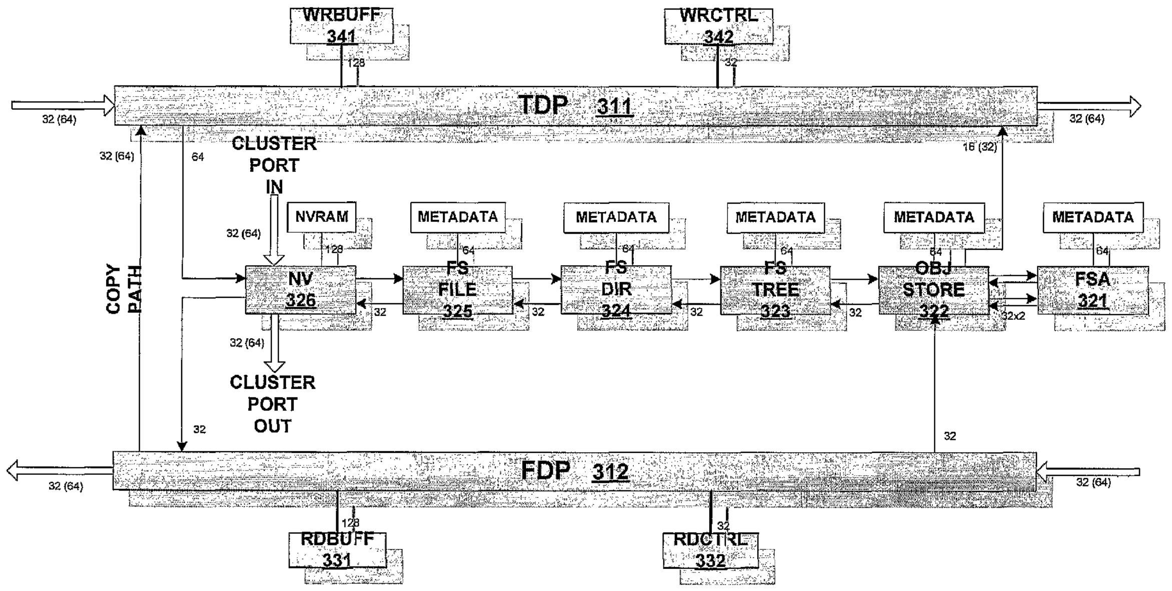
(12) **BREVET CANADIEN**
**CANADIAN PATENT**

(13) **C**

(86) **Date de dépôt PCT/PCT Filing Date:** 2003/10/30
(87) **Date publication PCT/PCT Publication Date:** 2004/05/21
(45) **Date de délivrance/Issue Date:** 2015/07/14
(85) **Entrée phase nationale/National Entry:** 2005/05/02
(86) **N° demande PCT/PCT Application No.:** US 2003/034463
(87) **N° publication PCT/PCT Publication No.:** 2004/042583
(30) **Priorité/Priority:** 2002/11/01 (US10/286,015)

(51) **Cl.Int./Int.Cl.** ***G06F 17/30*** (2006.01)
(72) **Inventeurs/Inventors:**
BARRALL, GEOFFREY S., US;
BENHAM, SIMON L., GB;
WILLIS, TREVOR E., GB;
ASTON, CHRISTOPHER J., GB
(73) **Propriétaire/Owner:**
HITACHI DATA SYSTEMS ENGINEERING UK LIMITED, ZZ
(74) **Agent:** GOWLING LAFLEUR HENDERSON LLP

(54) **Titre : SYSTEME DE FICHIERS MATERIEL**
(54) **Title: APPARATUS AND METHOD FOR HARDWARE-BASED FILE SYSTEM**



(57) **Abrégé/Abstract:**
A method and apparatus are disclosed having a non-volatile storage device and a storage processor configured to maintain, in a memory for a file system, an object structure with a first tree structure rooted by a first root node and a second tree structure rooted by a second root node. Each tree structure optionally includes a number of intermediate nodes and data blocks. Each tree structure represents a version of the file system object. The storage processor manages changes to the file system object using the first tree structure rooted by the first root node while storing the second tree structure rooted by the second root node and vice versa for a checkpoint that is used for keeping the consistency of data on the non-volatile storage device if the contents of the memory are lost.

## ABSTRACT

A method and apparatus are disclosed having a non-volatile storage device and a storage processor configured to maintain, in a memory for a file system, an object structure with a first tree structure rooted by a first root node and a second tree structure rooted by a second root node. Each tree structure optionally includes a number of intermediate nodes and data blocks. Each tree structure represents a version of the file system object. The storage processor manages changes to the file system object using the first tree structure rooted by the first root node while storing the second tree structure rooted by the second root node and vice versa for a checkpoint that is used for keeping the consistency of data on the non-volatile storage device if the contents of the memory are lost.

# Apparatus and Method for Hardware-Based File System

## Technical Field and Background Art

The present invention relates to computer file systems, and in particular to file systems that are accessed using computer hardware distinct from that associated with processors used for running computer application programs.

## Summary of the Invention

In one embodiment of the invention there is provided a file server system for accessing and utilizing a data storage system that may include magnetic storage, magneto-optical storage, or optical storage, to name but a few. The system includes a data bus arrangement, in communication with the data storage system, for providing data to be stored in the data storage system and for retrieving data from the data storage system. The system also includes a plurality of linked sub-modules, wherein the linked sub-modules as a group are in communication with a control input for receiving file service requests and a control output for responding to file service requests and process such service requests and generate responses thereto over the control output. The control input and the control output are typically distinct from the data bus arrangement. Each sub-module is configured to perform a distinct set of operations pertinent to processing of such file service requests. The system also includes a plurality of metadata memory caches. Each metadata memory cache is associated with a corresponding sub-module for storing metadata pertinent to operations of such sub-module, typically without storage of file content data.

An exemplary embodiment has the plurality of linked sub-modules arranged hierarchically.

An exemplary embodiment includes the following sub-modules: an object store sub-module for causing storage and retrieval of file system objects in the storage system, a file sub-module for managing data structure associated with file attributes, a directory sub-module for handling directory management for the file sub-module, a tree sub-module for handling directory lookups for the directory sub-module, a non-volatile storage processing sub-module with associated non-volatile storage for storing file system request data for subsequent storage in the storage system, and a free space allocation sub-module for retrieving and updating data pertinent to allocation of space in the data storage system.

Among other things, the tree sub-module manages a logical tree structure for the directory sub-module. In order to keep the tree structure substantially balanced, the directory sub-module associates each file with a randomized (or, perhaps more accurately, pseudo-randomized) value, and the tree sub-module manages a logical tree structure based upon the randomized values from the directory sub-module. Each randomized value is generated from a file name, for example, using a cyclic redundancy checksum (CRC) or other randomizing technique. The tree sub-module associates each randomized value with an index into the logical tree structure and uses the randomized values to access the logical tree structure. The tree sub-module associates each randomized value with an index into the directory table.

The non-volatile storage processing sub-module stores file system request data in the non-volatile storage at the request of a processor for recovery from a failure. The non-volatile storage processing sub-module sends an acknowledgment to the processor confirming storage of the file system request data in the non-volatile storage. The non-volatile storage processing sub-module may receive file system request data from another file server via an interface, which it stores in the non-volatile storage. The non-volatile storage processing sub-module may also send file system request data to another file server via an interface for non-volatile storage of the file system request data by the other file server.

The object store sub-module maintains a file structure for each file system object to be stored in the storage system. The file structures are typically stored in a dedicated metadata cache. File system objects typically include such things as files, directories, and file attributes. The object store sub-module effectuates storage of the file structures into the storage system at various checkpoints. Checkpoints can be initiated by an external processor or when certain events occur, for example, when a predetermined amount of time has elapsed since a last storage of the file structures into the storage system, when a portion of the non-volatile storage used for storage of the file system request data is becoming full, or when a sector cache associated with the storage system is becoming full.

In order to take a checkpoint, a checkpoint inquiry command is sent to the non-volatile storage processing sub-module to initiate storage of file structures into the storage system for a checkpoint. The checkpoint inquiry command typically includes a checkpoint number for the checkpoint. The non-volatile storage processing sub-module stores any outstanding file system requests in the non-volatile storage, optionally sends the number of file system requests to another file server via an interface for non-volatile storage of the number of file system requests by the other file server, sends the number of file system requests to the file sub-module, and subsequently sends a checkpoint command to the file sub-module (it should be noted that the storing and "mirroring" of file system requests, and the passing of file system requests to the file sub-module, occurs continuously as needed as well as during the taking of a checkpoint). The file sub-module processes any file system requests, and, upon receiving the checkpoint command from the non-volatile storage processing sub-module, waits for certain operations to complete through the remaining sub-modules and then sends a checkpoint command to the directory sub-module. The directory sub-module receives the checkpoint command from the file-sub-module and sends a checkpoint command to the tree sub-module. The tree sub-module receives the checkpoint command from the directory sub-module and sends a checkpoint command to the object store sub-module. The object store sub-module receives the checkpoint command from the tree sub-module and sends a checkpoint inquiry to the free space allocation sub-module. The free space allocation sub-module receives the checkpoint inquiry from the object store sub-module, completes any operations necessary for the checkpoint including operations initiated subsequent to receiving the checkpoint inquiry, and then sends a response to the object store sub-module. The object store sub-module then causes the file system objects to be written to the storage system, including an updated objects lists indicating any and all objects that have been modified since a last checkpoint.

In a typical embodiment of the invention, each file structure includes a plurality of nodes and at least one data block. Each node typically includes such things as pointers to other nodes, pointers to data block descriptors, and a checkpoint number indicating a checkpoint during which the node was created.

In a particular embodiment of the invention, the file structure includes at least two root nodes for storing information for a first and a second checkpoint. The storage system is logically divided into sectors, and the two root nodes are preferably stored in adjacent sectors in the storage system. Each root node typically includes such things as an object type for indicating the type of file system object (e.g., file, directory, free space object, volume descriptor object, etc.), an object length for indicating the number of data blocks associated with the file system object, a reuse count indicating the number of time the root node has been used, a pointer to a previous instantiation of the root node, a pointer to a subsequent instantiation of the root node, at least one a data block descriptor including a pointer to a data block, a checkpoint number indicating a relative time the data block was created, and an indicator to indicate whether the data block is zero or non-zero, and file attributes (enode). It should be noted that the actual object length may not be an integral number of data blocks, in which case the object length is typically rounded up to the next higher block multiple to give a count of the number of blocks used.

In addition to root nodes, the file structure may include a number of direct nodes that contain data block descriptors. A root node may include a pointer to a direct node. The file structure may also include a number of indirect nodes. Indirect nodes point to other indirect nodes or to direct nodes. A root node may include a pointer to an indirect node. It is possible for an object to have no data associated with it, in which case the object will not have any block descriptors.

In order to facilitate the creation of large empty (i.e., zero filled) files, the file structure typically includes an indicator for each data block associated with the file system object to indicate whether the data block is zero or non-zero. The file structure typically also includes an indicator for each node and data block to indicate whether each node and data block has been created. The object store sub-module creates nodes and data blocks as necessary to accommodate file system write requests and sets the indicator for each node and data block to indicate that the node or data block has been created. The object store sub-module typically creates a data block by allocating space for the data block from the free space allocation sub-module.

In order to facilitate recovery from failures, the object store sub-module typically maintains a transaction log, which it stores along with the file structure in the storage system from time to time.

Each sub-module may be implemented using dedicated hardware or a dedicated processor.

In another embodiment of the invention there is provided a clustered file server system having two or more interconnected file servers. Two file servers may be connected back-to-back, although more than two file servers are preferably interconnected through a switch. The switch provides the ability for any server to communicate with any other server. The servers then make use of this functionality to exchange file system request data amongst themselves for non-volatile storage of the file system request data, for example, in a virtual loop configuration. Typically, no modification of the switch configuration is required if one of the servers becomes unavailable, but rather, the servers realize the situation and modify the virtual loop accordingly.

In another embodiment of the invention there is provided a clustered file server system having at least three file servers and a switch. Each file server generates file system request data and includes a non-volatile storage area. The switch interconnects the file servers so that any given file server's non-volatile storage stores file system request data from a selected one of the other file servers. The switch may be configured such that the file system request data from each file server is stored in at least one other file server, for example in a virtual loop configuration. The switch is typically capable of modifying the configuration in order to bypass a file server that becomes unavailable for storing file system request data.

In another embodiment of the invention there is provided a file server having a service module for receiving and responding to file service requests over a network, a file module for servicing file service requests, and a processor in communication with the service module and the file module. The service module passes a file service request to the processor. The processor processes the file service request and passes the file service request to the file module for servicing. The file module sends a response for the file service request directly to the service module, bypassing the processor.

In another embodiment of the invention there is provided a method for managing a reusable data structure in a file system. The method involves maintaining a reuse value for the reusable data structure and changing the reuse value each time the data structure is reused. The reusable data structure is typically a root node of an object structure associated with a file system object. The reuse value is typically provided to a client for referencing the file system object. The client typically includes the reuse value when requesting access to the file system object, in which case the reuse value in the request is compared to the reuse value in the root node to determine whether the root node was reused subsequent to providing the reuse value to the client, and the request is serviced if and only if the reuse value in the request matches the reuse value in the root node.

In another embodiment of the invention there is provided a method for maintaining a file system object in a non-volatile storage at successive checkpoints. The method involves maintaining an object structure for the file system object, the object structure comprising a first tree structure rooted by a first root node and a second tree structure rooted by a second root node, each tree structure optionally including a number of intermediate nodes and a number of data blocks, each tree structure representing a version of the file system object. The method also involves alternately managing the object structure using the first tree structure rooted by the first root node while storing the second tree structure rooted by the second root node in the non-volatile storage and managing the object structure using the second tree structure rooted by the second root node while storing the first tree structure rooted by the first root node in the non-volatile storage. The method typically also involves maintaining a version number for each root node, the version number indicating the checkpoint associated with the corresponding tree structure. The non-volatile storage typically includes a plurality of sectors, and the first and second root nodes are typically stored in adjacent sectors in the non-volatile storage. The method typically also involves determining a latest valid version of the file system object based upon the version numbers of the root nodes. The method typically also involves maintaining a list of free space areas of the non-volatile storage, maintaining a list of free root nodes, allocating the root nodes for the object structure from one of the list of free space areas and the list of free root nodes, and allocating intermediate nodes and data blocks for the object structure only from the list of free space areas. The method may also involve deleting the file system object from the non-volatile storage. Deleting the file system object from the non-volatile storage typically involves adding the root nodes to the list of free root nodes and adding the intermediate nodes and data blocks to the list of free space areas.

In another embodiment of the invention there is provided a method for retaining a read-only version of an object in a file system. The method involves maintaining an object structure for the object, the object structure including at least a root node associated with a current version of the object, a number of intermediate nodes, and a number of data blocks for storing object data, wherein each node includes at least one reference to a data block or to another node in order to form a path from the root node to each data block. The method also involves storing the object structure in a non-volatile storage and making a copy of the root node for the retained version of the object. The method may also involve storing a reference to the copy of the root node in the object structure for the object. The method may also involve storing a reference to the root node in the copy of the root node. The method may also involve obtaining a reference to an earlier version of the root node from the root node and storing the reference to the earlier version of the root node in the copy of the root node and also storing a reference to the copy of the root node in the earlier version of the root node. The method may also involve storing the copy of the root node in the non-volatile storage.

The method may also involve modifying object data without modifying any intermediate nodes or data blocks associated with the retained version of the object. Modifying object data without modifying any intermediate nodes or data blocks associated with the retained version of the object typically involves making a copy of a data block, modifying the copy of the data block to form a modified copy of the data block, and forming a path from the root node to the modified copy of the data block without modifying any intermediate nodes along the path to the data block that are associated with the retained version of the object. The root node may have a reference to the data block, which is modified to refer to the modified copy of the data block rather than to the data block. Alternatively, path from the root node to the modified copy of the data block may include an intermediate node referenced by the root node that in turn has a reference to the data block, in which case a copy of the intermediate node is made, the copy of the intermediate node is modified to refer to the modified copy of the data block rather than to the data block, and the root node is modified to reference the copy of the intermediate node. Alternatively, the path from the root node to the data block may include a plurality of intermediate nodes including at least a first intermediate node referenced by the root node and a last intermediate node having a reference to the data block, in which case a copy of each intermediate node is made, the root node is modified to reference the copy of the first intermediate node, the copy of the last intermediate node is modified to reference the modified copy of the data block, and the copy of each other intermediate node is modified to reference a copy of another intermediate node in order to form a path from the root node to the modified copy of the data block.

The method may also involve deleting a data block from the object without modifying any intermediate nodes or data blocks associated with the retained version of the object. The root node may have a reference to the data block, in which case deleting the data block from the object involves removing the reference to the data block from the root node. Alternatively, the path from the root node to the data block may include an intermediate node referenced by the root node and having a reference to the data block, in which case deleting the data block from the object involves making a copy of the intermediate node, removing the reference to the data block from the copy of the intermediate node, and modifying the root node to reference the copy of the intermediate node. Alternatively, the path from the root node to the data block may include a plurality of intermediate nodes including at least a first intermediate node referenced by the root node and a last intermediate node having a reference to the data block, in which case deleting the data block from the object involves making a copy of each intermediate node, modifying the root node to reference the copy of the first intermediate node, removing the reference to the data block from the copy of the last intermediate node, and modifying the copy of each other intermediate node to reference a copy of another intermediate node in order to form a path from the root node to the copy of the last intermediate node.

The method may involve adding a new data block to the object without modifying any intermediate nodes or data blocks associated with the retained version of the object. Adding the new data block to the object may involve allocating the new data block and adding a reference to the new data block to the root node. Adding the new data block to the object may involve allocating the new data block, making a copy of an intermediate node, storing a reference to the new data block in the copy of the intermediate node, and storing a reference to the copy of the intermediate node in the root node. Adding the new data block to the object may involve allocating the new data block, allocating a new intermediate node, storing a reference to the new data block in the new intermediate node, and storing a reference to the new intermediate node in the root node. Adding the new data block may involve allocating the new data block, allocating a new intermediate node, storing a reference to the new data block in the new intermediate node, and forming a path to the new intermediate node without modifying any intermediate nodes or data blocks associated with the retained version of the object.

The method may involve maintaining a modified objects list for the retained version of the object. The modified objects lists indicates any and all intermediate nodes and data blocks added, modified, or deleted after making a copy of the root node for the retained version of the object.

The method may involve deleting the retained read-only version of the object from the file system. Deleting the retained read-only version of the object from the file system involves identifying any and all intermediate nodes and data blocks modified since retaining the read-only version of the object, identifying the copy of the root node for the retained version of the object being deleted, identifying a root node associated with an earlier retained version of the object if one exists, identifying a root node associated with a later version of the object, said later version being one of a later retained version of the object and a current version of the object, identifying any and all intermediate nodes and data blocks associated with the retained version of the object being deleted, identifying any and all intermediate nodes and data blocks that are used only by the retained version of the object being deleted, deleting from the object structure each intermediate node and data block that is used only by the retained version of the object being deleted, identifying any and all intermediate nodes and data blocks that are used by the later version of the object, adding any and all intermediate nodes and data blocks that are used by the later version of the object to a modified objects list associated with the later version of the object, determining whether the copy of the root node for the retained version of the object being deleted is only used in the retained version of the object being deleted, and deleting from the object structure the copy of the root node for the retained version being deleted if and only if the copy of the root node for the retained version of the object being deleted is only used in the retained version of the object being deleted. Identifying all intermediate nodes and data blocks that were modified in the retained read-only version of the object typically involves maintaining a list of intermediate nodes and data blocks modified since retaining the read-only version of the object. The root node for the retained read-only version of the object typically includes a reference to the root node of the earlier retained version of the object if one exists, and identifying the root node associated with the earlier retained version of the object typically involves accessing the reference to the root node of the earlier retained version of the object in the root node for the retained read-only version of the object. The root node for the retained read-only version of the object typically includes a reference to the root node of the later version of the object, and identifying the root node associated with the later version of the object typically involves accessing the reference to the root node of the later version of the object in the root node for the retained read-only version of the object. Identifying any and all intermediate nodes and data blocks that are used only by the retained version of the object being deleted typically involves, for each intermediate node and data block in the retained version of the object being deleted, identifying an equivalent intermediate node or data block in the earlier version of the object, if one exists, and in the later version of the object, if one exists; comparing the intermediate node or data block in the retained version of the object being deleted to the equivalent intermediate node or data block in both the earlier version of the object and the later version of the object; and determining that the intermediate node or data block is used only by the retained version of the object being deleted if and only if the equivalent intermediate node or data block is different in the earlier version of the object, if one exists, and in the later version of the object, if one exists. Each deleted intermediate node and data block is typically added to a list of free space areas. The root node associated with the earlier retained version of the object typically includes a reference to the copy of the root node for the retained version of the object being deleted, and deleting from the object structure the copy of the root node for the retained version being deleted typically involves replacing the reference to the copy of the root node for the retained version of the object being deleted with a reference to the root node associated with the later version of the object, if one exists, or with a null value, if one does not exist. The root node associated with the later version of the object typically includes a reference to the copy of the root node for the retained version of the object being deleted, and deleting from the object structure the copy of the root node for the retained version being deleted typically involves replacing the reference to the copy of the root node for the retained version of the object being deleted with a reference to the root node associated with the earlier version of the object, if one exists, or with a null value, if one does not exist. The deleted copy of the root node is typically added to a list of free root nodes.

In another embodiment of the invention there is provided a method for indicating the contents of a portion of an object in a file system. The method involves maintaining an object structure including a number of data blocks for the object and maintaining an indicator for each data block, each indicator having a first state for indicating that the corresponding data block is logically filled with a predetermined value and a second state for indicating that the corresponding data block contains object data. Each indicator is typically maintained in a node referencing the corresponding data block. The predetermined value is typically a zero value. The method may also involve setting an indicator to the first state to indicate that the corresponding data block is logically filled with a predetermined value without writing the predetermined value to the corresponding data block. The method may also involve writing object data into a data block and setting the indicator corresponding to the data block to the second state to indicate that the corresponding data block contains object data.

In another embodiment of the invention there is provided a method for allocating sparse objects in a file system. The method involves allocating a root node for the object and allocating additional nodes and data blocks as needed only for portions of the object that are not to be zero-filled. Each node typically includes a number of references to data blocks and/or other nodes. Each node typically includes an indicator for each reference to another node. Each indicator has a first state for indicating that the other node has been allocated and a second state for indicating that the other node has not been allocated. The indicator for each reference associated with an unallocated node is initially set to the second state. In order to write object data to a zero-filled portion of the object, additional nodes and data blocks are allocated, and, in each node having a reference to an allocated node, a reference to the allocated node is stored and the indicator for the reference to the allocated node is set to the first state.

In another embodiment of the invention there is provided a method for storing metadata associated with an object in a file system. The method involves maintaining a first object structure for the object, the object structure including at least a root node and optionally including intermediate nodes and data blocks, and storing a first portion of metadata in the root node. The method may also involve allocating a number of data blocks for storing a second portion of metadata. The method may also involve allocating a second object structure for storing a third portion of metadata, in which case a reference to the second object structure is typically stored within the first object structure, for example, within the root node of the first object structure or within the second portion of metadata.

In another embodiment of the invention there is provided an apparatus including a non-volatile storage and means for maintaining a file system object in the non-volatile storage at successive checkpoints using an object structure having two and only two root nodes for managing a current version of the object, where the means alternates between the two root nodes for managing the object at the successive checkpoints. The apparatus typically also includes means for retaining read-only versions of the object through the object structure. The apparatus typically also includes means for deleting a retained read-only version of the object from the object structure. The apparatus typically also includes means for deleting the current version of the object while at least one retained read-only version of the object exists in the object structure. The apparatus typically also includes means for reverting the current version of the object to a retained read-only version of the object.

In another embodiment of the invention there is provided a method for maintaining file system objects in a file system having a non-volatile storage. The method involves maintaining an object structure for each of a plurality of file system objects, each object structure including at least one root node and optionally including a number of intermediate nodes and a number of data blocks; maintaining a transaction log identifying any and all modified nodes; storing any and all modified intermediate nodes identified by the transaction log in the non-volatile storage; storing the transaction log in the non-volatile storage; and storing any and all modified root nodes identified by the transaction log in the non-volatile storage only after storing the transaction log in the non-volatile storage. The method may also involve determining that a failure occurred between storing the transaction log in the non-volatile storage and storing any and all modified root nodes identified by the transaction log in the non-volatile storage and, for each node identified by the transaction log, reverting to a previous version of the node stored in the non-volatile storage.

In another embodiment of the invention there is provided a method for accessing a shared resource in a distributed file system having at least a first file server that manages the shared resource and a second file server that accesses the shared resource. The method involves maintaining a cache for the shared resource by the second file server, requesting read access to the shared resource by the second file server from the first file server, providing read access to the shared resource by the first file server for the second file server, obtaining shared resource data by the second file server from the first file server, and storing the shared resource data by the second file server in the cache.

In another embodiment of the invention there is provided a method for accessing a shared resource in a distributed file system having at least a first file server that manages the shared resource and a second file server that accesses the shared resource. The method involves maintaining a cache for the shared resource by the second file server, requesting read access to the shared resource by the second file server from the first file server, denying read access to the shared resource by the first file server for the second file server, providing shared resource data to the second file server by the first file server, and omitting the shared resource data from the cache by the second file server.

In another embodiment of the invention there is provided a file server for operation in a distributed file system having a resource shared among a plurality of file servers. The file server includes a cache for storing data associated with the shared resource and distributed lock means for controlling access to the shared resource, the distributed lock means operably coupled to selectively store shared resource data in the cache.

## Brief Description of the Drawings

The foregoing features of the invention will be more readily understood by reference to the following detailed description, taken with reference to the accompanying drawings, in which:

Fig. 1 is a block diagram of an embodiment of a file server to which various aspects of the present invention are applicable;

Fig. 2 is a block diagram of an implementation of the embodiment of Fig. 1;

Fig. 3 is a block diagram of a file system module in accordance with an embodiment of the present invention;

Fig. 4 is a block diagram showing how control flow may be used in embodiments of the present invention to permit automatic response by the file service module to a network request without intervention of software control;

Fig. 5 is a block diagram of a clustered file server arrangement embodying sector cache locking in accordance with an embodiment of the present invention;

Fig. 6 is a block diagram of a clustered file server arrangement in accordance with an embodiment of the present invention wherein non-volatile memory is mirrored in a virtual loop configuration;

Fig. 7 is a block diagram showing use of a root onode with no other onodes in accordance with the embodiment of Fig. 3;

Fig. 8 is a block diagram showing showing employment of a root onode with a direct onode;

Fig. 9 is a block diagram showing showing employment of a root onode with an indirect onode as well as direct onodes;

Fig. 10 is a block diagram illustrating use of multiple layers of indirect onodes placed between the root onode and the direct onodes;

Fig. 11 is a diagram illustrating creation of a root onode during checkpoint A in accordance with the embodiment of Fig. 3;

Fig. 12 is a diagram illustrating the effect of making further modifications, to the root onode of Fig. 11, that are written to the right hand side of the root onode;

Fig. 13 is a diagram illustrating the effect of the creation of checkpoint A, and wherein root onode of Fig. 12 has been written to disk;

Fig. 14 is a diagram illustrating the effect of the creation of checkpoint B for the same root onode;

Fig. 15 is a diagram illustrating the effect of modifying the same root onode as part of checkpoint C while checkpoint B is being created;

Fig. 16 is a diagram for the starting point of an illustration of a root onode that is part of an object structure having 2 levels of indirection;

Fig. 17 is a diagram that illustrates the structure of the object to which corresponds the root onode of Fig. 16;

Fig. 18 is a diagram that illustrates the effect of taking a checkpoint with respect to the object illustrated in Fig. 17;

Fig. 19 is a diagram that illustrates, with respect to the structure of Fig. 18, the effect of allocating a new data block 2 and updating all of the onode structures to point at this new block, before a new checkpoint has been taken;

Fig 20 is a diagram that illustrates, with respect to the structure of Fig. 19, the effect of taking checkpoint with respect to the data structure of Fig. 19;

Fig. 21 is a diagram that illustrates, with respect to the structure of Fig. 20, the effect of writing to data block 1 with the object in data overwrite mode;

Fig. 22 is a timeline showing steps in creation of a checkpoint;

Fig. 23 is a diagram that shows the structure of an exemplary object that includes four data blocks and various onodes at a checkpoint number 1;

Fig. 24 is a diagram that shows the structure of the exemplary object of Fig. 23 after a retained checkpoint is taken for a checkpoint number 2 and during modification of a data block 0 during a checkpoint number 3, specifically after a copy of the object's root onode is saved to free space and the root onode is updated to include a pointer to the saved root onode in accordance with an embodiment of the present invention;

Fig. 25 is a diagram that shows the structure of the exemplary object of Fig. 24 after a modified copy of the data block is written to free space in accordance with an embodiment of the present invention;

Fig. 26 is a diagram that shows the structure of the exemplary object of Fig. 25 after a new direct onode is created to point to the modified copy of the data block in accordance with an embodiment of the present invention;

Fig. 27 is a diagram that shows the structure of the exemplary object of Fig. 26 after a new indirect onode is created to point to the new direct onode in accordance with an embodiment of the present invention;

Fig. 28 is a diagram that shows the structure of the exemplary object of Fig. 27 after a pointer to the new indirect onode is written into the current root onode for the object in accordance with an embodiment of the present invention;

Fig. 29 is a diagram that shows the structure of the exemplary object of Fig. 28 after a retained checkpoint is taken in a checkpoint number 4 and after a data block 3 is deleted in a checkpoint number 5 in accordance with an embodiment of the present invention;

Fig. 30 is a diagram that shows the structure of the exemplary object of Fig. 29 after the retained checkpoint taken in checkpoint number 4 is deleted in accordance with an embodiment of the present invention; and

Fig. 31 is a diagram that shows the structure of the exemplary object of FIG. 30 after the current version of the object is deleted, leaving only the retained checkpoint taken in checkpoint number 2, in accordance with an embodiment of the present invention.

## Detailed Description of Specific Embodiments

Definitions. As used in this description and the accompanying claims, the following terms shall have the meanings indicated, unless the context otherwise requires:

"Data storage system" may be any suitable large data storage arrangement, including but not limited to an array of one or more magnetic or magneto-optical or optical disk drives, solid state storage devices, and magnetic tapes. For convenience, a data storage system is sometimes referred to as a "disk" or a "hard disk".

A "hardware-implemented subsystem" means a subsystem wherein major subsystem functions are performed in dedicated hardware that operates outside the immediate control of a software program. Note that such a subsystem may interact with a processor that is under software control, but the subsystem itself is not immediately controlled by software. "Major" functions are the ones most frequently used.

A "hardware-accelerated subsystem" means one wherein major subsystem functions are carried out using a dedicated processor and dedicated memory, and, additionally (or alternatively), special purpose hardware; that is, the dedicated processor and memory are distinct from any central processor unit (CPU) and memory associated with the CPU.

A "file" is a logical association of data.

"Metadata" refers to file overhead information as opposed to actual file content data.

"File content data" refers to file data devoid of file overhead information.

Pertinent to subject matter described herein commonly-owned U.S. Patent No. 8,041,735 entitled **Distributed File System and Method**, which was filed on even date herewith in the names of Francesco Lacapra, Fiorenzo Cattaneo, Simon L. Benham, Trevor E. Willis, and Christopher J. Aston.

Fig. 1 is a block diagram of an embodiment of a file server to which various aspects of the present invention are applicable. A file server of this type is described in PCT application publication number WO 01/28179 A2, published April 19, 2001, entitled "Apparatus and Method for Hardware Implementation or Acceleration of Operating System Functions"—such document, describing an invention of which co-inventors herein are also co-inventors. The present Fig. 1 corresponds generally to Fig. 3 of the foregoing PCT application. A file server 12 of Fig 1 herein has components that include a service module 13, in communication with a network 11. The service module 13 receives and responds to service requests over the network, and is in communication with a file system module 14, which translates service requests pertinent to storage access into a format appropriate for the pertinent file system protocol (and it translates from such format to generate responses to such requests). The file system module 14, in turn, is in communication with a storage module 15, which converts the output of the file system module 14 into a format permitting access to a storage system with which the storage module 15 is in communication. The storage module has a sector cache for file content data that is being read from and written to storage. As described in the foregoing PCT application, each of the various modules may be hardware implemented or hardware accelerated.

Fig. 2 is a block diagram of an implementation of the embodiment of Fig. 1. In this implementation, the service module 13, file system module 14, and storage module 15 of Fig. 1 are implemented by network interface board 21, file system board 22, and storage interface board 23 respectively. The storage interface board 23 is in communication with storage device 24, constituting the storage system for use with the embodiment. Further details concerning this implementation are set forth in U.S. Patent No. 6,826,615 entitled "Apparatus and Method for Hardware Implementation or Acceleration of Operating System Functions".

Fig. 3 is a block diagram of an embodiment of a file system module in accordance with the present invention. The file system module embodiment may be used in systems of the type described in Figs. 1 and 2. Exemplary bus widths for various interfaces are shown, although it should be noted that the present invention is in no way limited to these bus widths or to any particular bus widths.

The data flow in this embodiment is shown by upper bus 311, which is labeled TDP, for To Disk Protocol, and by lower bus 312, which is labeled FDP, for From Disk Protocol, such Protocols referring generally to communication with the storage module 15 of Fig. 1 as may be implemented, for example, by storage interface board 23 of Fig. 2. The file system module always uses a control path that is distinct from the data buses 311 and 312, and in this control path uses pointers to data that is transported over the buses 311 and 312. The buses 311 and 312 are provided with a write buffer WRBUFF and read buffer RDBUFF respectively. For back up purposes, such as onto magnetic tape, there is provided a direct data path, identified in the left portion of the drawing as COPY PATH, from bus 312 to bus 311, between the two buffers.

A series of separate sub-modules of the file system module handle the tasks associated with file system management. Each of these sub-modules typically has its own cache memory for storing metadata pertinent to the tasks of the sub-module. (Metadata refers to file overhead information as opposed to actual file content data; the file content data is handled along the buses 311 and 312 discussed previously.) These sub-modules are Free Space Allocation 321, Object Store 322, File System Tree 323, File System Directory 324, File System File 325, and Non-Volatile Storage Processing 326.

The sub-modules operate under general supervision of a processor, but are organized to handle their specialized tasks in a manner dictated by the nature of file system requests being processed. In particular, the sub-modules are hierarchically arranged, so that successively more senior sub-modules are located successively farther to the left. Each sub-module receives requests from the left, and has the job of fulfilling each request and issuing a response to the left, and, if it does not fulfill the request directly, it can in turn issue a request and send it to the right and receive a response on the right from a subordinate sub-module. A given sub-module may store a response, provided by a subordinate sub-module, locally in its associated cache to avoid resending a request for the same data. In one embodiment, these sub-modules are implemented in hardware, using suitably configured field-programmable gate arrays. Each sub-module may be implemented using a separate field-programmable gate array, or multiple sub-modules may be combined into a single field-programmable gate array (for example, the File System Tree 323 and File System Directory 324 sub-modules may be combined into a single field-programmable gate array). Alternatively, each sub-module (or combination of sub-modules) may be implemented, for example, using integrated circuitry or a dedicated processor that has been programmed for the purpose.

It can be seen that the file system embodiment provided herein is distributed in nature. This distributed nature permits keeping all of the metadata associated with the file system in cache memory that is distinct from file content cache. There are numerous benefits to this arrangement, including the ability to cache large amounts of metadata regardless of the size of the files to which they relate, increased throughput in handling file operations, and reduced processor overhead.

The processing of file system requests is delineated by a series of checkpoints that are scheduled to occur no less frequently than some user-specified interval, such as every 10 seconds. With respect to each successive checkpoint, there is stored, on disk, current file structure information that supercedes previously stored file structure information from the immediately preceding checkpoint. Checkpoints are numbered sequentially and are used to temporally group processing of file requests.

For a variety of purposes it may be useful to have knowledge of the file system structure at a selected point in time. This capability is provided by permitting user-triggered storage of file system structure data associated with the currently saved checkpoint, which is referred to hereinafter for convenience as a retained checkpoint, and is described in detail below. The retained checkpoint is essentially a read-only version of the file system structure at a particular checkpoint. Multiple retained checkpoints can be taken, and mechanisms are included for deleting a selected retained checkpoint or reverting the file system to a selected retained checkpoint (for example, to return the file system to a known state following a catastrophe).

At the heart of the file system module is the Object Store sub-module 322. In this implementation all items that are subject to storage on the hard disk, regardless of form (including, for example, files, directories, free-space allocation information, a list of objects created or modified since a last checkpoint was taken, a list of objects created or modified since a last retained checkpoint was taken, and certain file attribute information), are regarded as objects, and storage for such items is handled by the Object Store sub-module 322. The Object Store sub-module can perform the following operations with respect to an object: create, delete, write, and read. In addition, under instruction from processor, the Object Store sub-module can create a checkpoint, and can also create a retained checkpoint, delete a retained checkpoint, or revert the file system to a retained checkpoint. The Object Store sub-module tracks the physical location of data, stored on the disk, which is associated with each object, using various data structures described below. The Object Store sub-module causes disk storage requests to be sent by a communication link over the bus 311 and obtains disk storage response data by a communication link over the bus 312. If the Object Store sub-module receives a request for a read operation, the Object Store sub-module can satisfy the request directly by acting over the bus 311.

Although the storage system, with respect to which the file system embodiment herein is being used, is referred to as the "disk," it will be understood that the storage system may be any suitable large data storage arrangement, including but not limited to an array of one or more magnetic or magneto-optical or optical disk drives, solid state storage devices, and magnetic tapes.

The Free Space Allocation sub-module 321 manages data necessary for operation of the Object Store sub-module 322, and tracks the overall allocation of space on the disk as affected by the Object Store sub-module 322. On receipt of a request from the Object Store sub-module 322, the Free Space Allocation sub-module 321 provides available block numbers to the Object Store sub-module. To track free space allocation, the Free Space Allocation sub-module establishes a bit map of the disk, with a single bit indicating the free/not-free status of each block of data on the disk. This bit map is itself stored on the disk as a special object handled by the Object Store sub-module. There are two two-way paths between the Object Store and Free Space Allocation sub-modules since, on the one hand, the Object Store sub-module has two-way communication with the Free Space Allocation sub-module for purposes of management and assignment of free space on the disk, and since, on the other hand, the Free Space Allocation sub-module has two-way communication with the Object Store sub-module for purposes of retrieving and updating data for the disk free-space bit map.

The File System File sub-module 325 manages the data structure associated with file attributes, such as the file's time stamp, who owns the file, how many links there are to the file (i.e., how many names the file has), read-only status, etc. Among other things, this sub-module handles requests to create a file, create a directory, insert a file name in a parent directory, and update a parent directory. This sub-module in turn interacts with other sub-modules described below.

The File System Directory sub-module 324 handles directory management. The directory is managed as a listing files that are associated with the directory, together with associated object numbers of such files. File System Directory sub-module 324 manages the following operations of directories: create, delete, insert a file into the directory, remove an entry, look up an entry, and list contents of directory.

The File System Directory sub-module 324 works in concert with the File System Tree sub-module 323 to handle efficient directory lookups. Although a conventional tree structure is created for the directory, the branching on the tree is handled in a non-alphabetical fashion by using a pseudo-random value, such as a CRC (cyclic redundancy check sum), that is generated from a file name, rather than using the file name itself. Because the CRC tends to be random and usually unique for each file name, this approach typically forces the tree to be balanced, even if all file names happen to be similar. For this reason, when updating a directory listing with a new file name, the File System Directory sub-module 324 generates the CRC of a file name, and asks the File System Tree sub-module 323 to utilize that CRC in its index. The File System Tree sub-module associates the CRC of a file name with an index into the directory table. Thus, the sub-module performs the lookup of a CRC and returns an index.

The File System Tree sub-module 323 functions in a manner similar to the File System Directory sub-module 324, and supports the following functions: create, delete, insert a CRC into the directory, remove an entry, look up an entry. But in each case the function is with respect a CRC rather than a file.

In rare cases the CRC for two different files may be the same, and the file system module must handle such a case. To accommodate this situation, the File System Tree sub-module 324 maintains a list of all files with same CRC, and does so by having a pointer from any given file with a CRC to another file with the same CRC. (Owing to the relative uniqueness of the CRC, this is likely a short list.) The File System Tree sub-module 324 maintains the starting point of the list for any given CRC.

The Non-Volatile Storage Processing sub-module 326 interfaces with associated non-volatile storage (called NVRAM in Fig. 3) to provide a method for recovery in the event of power interruption or other event that prevents cached data—which is slated for being saved to disk— from actually being saved to disk. In particular, since, at the last checkpoint (checkpoints are discussed above near the beginning of discussion of Fig. 3), a complete set of file system structure has been stored, it is the task of the Non-Volatile Storage Processing sub-module 326 to handle storage of file system request data since the last checkpoint. In this fashion, recovery, following interruption of processing of file system request data, can be achieved by using the file system structure data from the last stored checkpoint and then reprocessing the subsequent file system requests stored in NVRAM.

In operation, the Non-Volatile Storage Processing sub-module 326, for every file system request that is received (other than a non-modifying request), is told by the processor whether to store the request in NVRAM, and, if so told, then stores in the request in NVRAM. (If this sub-module is a part of a multi-node file server system, then the request is also stored in the NVRAM of another node.) No acknowledgment of fulfillment of the request is sent back to the client until the sub-module determines that there has been storage locally in NVRAM by it (and any paired sub-module on another file server node). This approach to caching of file system requests is considerably different from prior art systems wherein a processor first writes the file system request to NVRAM and then to disk. This is approach is different because there is no processor time consumed in copying the file system request to NVRAM—the copying is performed automatically.

In order to prevent overflow of NVRAM, a checkpoint is forced to occur whenever the amount of data in NVRAM has reached a pre-determined threshold. A checkpoint is only valid until the next checkpoint has been created, at which point the earler checkpoint no longer exists.

When file server systems are clustered, non-volatile storage may be mirrored using a switch to achieve a virtual loop. Fig. 6 is a block diagram of a clustered file server arrangement in accordance with an embodiment of the present invention wherein non-volatile memory is mirrored in a virtual loop configuration. In this figure, it is assumed that five file server nodes are clustered (although this technique works with any number of server nodes, and each server node has associated a file system module, and each file system module has a Non-Volatile Storage Processing sub-module 326, designated NV_A (item 61), NV_B (item 62), NV_C (item 63), NV_D (item 64), and NV_E (item 65). Each of these sub-modules is coupled via the switch 66 to a different one of the sub-modules, to permit the coupled sub-module's associated NVRAM to retain a backup copy of the original file system request data stored in NVRAM associated with the corresponding sub-module. Couplings achieved by the switch 66 are shown in dashed lines, so that backup path 611 permits file system request data in NVRAM associated with sub-module NV_A to be backed up by NVRAM associated with sub-module NV_B. Similarly, backup path 621 permits file system request data in NVRAM associated with sub-module NV_B to be backed up by NVRAM associated with sub-module NV_C, and so on, until the last part of the loop is reached, wherein backup path 651 permits file system request data in NVRAM associated with sub-module NV_E to be backed up by NVRAM associated with sub-module NV_A. If a server node becomes non-operational, then the switch can reconfigure the loop among remaining nodes that are operational.

As described herein, a consistent file system image (termed a checkpoint) is stored on disk at regular intervals, and all file system changes that have been requested by the processor but have not yet been stored on disk in a checkpoint are stored in NVRAM by the Non-Volatile Storage Processing sub-module.

In the event of a system failure, the processor detects that the on disk file system is not "clean" and it begins the recovery procedure. Initially, the on disk file system is reverted to the state represented by the last checkpoint stored on disk. Since this is a checkpoint, it will be internally consistent. However, any changes that were requested following the taking of this checkpoint will have been lost. To complete the recovery procedure, these changes must be restored. This is possible since these changes would all have been caused by requests issued by the processor, and (as explained above) all file system changes that have been requested by the processor but have not yet been stored on disk in a checkpoint are stored in NVRAM. The lost changes can therefore be restored by repeating the sequence of file system changing operations that were requested by the processor from the time of the last checkpoint until the system failure.

In order to achieve this, the processor examines the contents of the NVRAM and extracts all the operations that were requested by the processor from the time of the last checkpoint until the system failure. It then resubmits these requests to the File System File sub-module, which satisfies the requests by making the necessary on disk changes. The File System File sub-module does not distinguish between "live" file system requests and resubmitted requests that are being issued during the recovery procedure - both are handled in an indentical fashion (with the exception that resubmitted requests are not logged in NVRAM since they already exist there).

One complication in this procedure relates to the file handles by which a file (or directory) is referenced. In normal operation, when a file is created, it is assigned a file handle. Any operations that subsequently need to refer to that file do so by means of this file handle. So, for example, the following sequence of operations might take place:

(1) Processor requests that a file be created.

(2) File System File sub-module creates file and returns handle A.

(3) Processor requests write of data to file A.

(4) File System File sub-module performs the write.

In this example, the two requests at steps (1) and (3) would be stored in NVRAM.

The complication arises because the file handle assigned by the File System File sub-module during the recovery procedure described above may differ from the file handle that was originally assigned. So, for example, the operations stored in the NVRAM might be as in the example above. However, during recovery, the file handle returned by the File System File sub-module at step (2) might be B. In this case, the write of data at step (3) using file handle A fails, since file handle A is not recognized by the File System File sub-module.

In order to overcome this problem, whenever a file is created, the processor explicitly logs the assigned handle in NVRAM (this is performed via a special request to the Non-Volatile Storage Processing sub-module). The contents of the NVRAM at recovery time therefore look like this:

(1) Processor requests that a file be created.

.... there may be more entries for other unrelated requests here ...

(2) Created file was assigned handle A.

.... there may be more entries for other unrelated requests here ...

(3) Processor requests write of data to file A.

Therefore, when a create operation is encounted during the recovery procedure, the processor searches the NVRAM to find the assigned handle. It then issues the create request and obtains a (potentially different) handle. From this point on, any references in the replayed operations to the old handle for the created file are replaced by the new handle for the created file.

For example, the recovery procedure for the example above might proceed as follows:

(1) Processor resubmits file create request.

(2) File System File sub-module creates file and returns handle B.

(3) Processor searches NVRAM for previously assigned handle and determines the previously assigned handle is handle A.

(4) Processor notes that any subsequent references in NVRAM to handle A should be replaced by handle B.

(5) Processor substitutes handle B for handle A and requests write of data to file B.

(6) File System File sub-module performs the write.

A typical embodiment utilizes an automatic response mechanism for servicing certain file system requests. Fig. 4 is a block diagram showing how control flow may be used in embodiments of the present invention to permit automatic response by the file service module to a network request without prior intervention of software control. In Fig. 4, there is shown service module 13, file system module 14, and storage module 15, as in Fig. 1, with service module 13 and file system module 14 under the control of software 41 and with storage module 15 in communication with storage arrangement 42. The connections between blocks represent control flows rather than data flows. On identification of a file service request by service module 13, the request is typically passed from the service module 13 to software control 41, for example, to handle security and other complex tasks. Then under software control 41, the request is processed by the file system module 14. On the other hand, the response to a file system request, which is not necessarily as complex, is routed from the file system module 14 directly back to the service module 13 over control flow 43 rather than being routed back through software control 41. The software control 41 is eventually informed that the request has been satisfied.

In an arrangement employing a cluster of file server nodes accessing common storage, it is necessary to deal with instances wherein multiple nodes may seek to perform conflicting tasks with respect to a common storage location. Fig. 5 is a block diagram of a clustered file server arrangement embodying sector cache locking in accordance with an embodiment of the present invention to deal with this problem. In this embodiment, file server node A (item 52) and file server node B (item 53), are both in communication with clients 51 and are configured so that each server node may access (that is, read from and write to) both disk A (item 54) and disk B (item 55). (Here, in a manner analogous to that previously discussed, the term "disk" is an arbitrary storage designator, and includes the use of several disks, e.g., or a particular region on a single disk drive, and the mode of storage is any suitable for, including but not limited to magnetic and magneto-optical.)

In this embodiment, each server node maintains a sector cache, at a sector level, of each of disk A and disk B. Under these circumstances, it is necessary to solve the problem of how to achieve cache coherency where each server node might process disk writes. This problem is addressed as follows. For a given disk, only one server node can write to the disk (although a client can write to either disk via either server node). For example, in Fig. 5, only server node A can write to disk A, and only server node B can write to disk B. Each server node runs a lock manager for the disk it writes to. The disks are split up into 32 Kbyte pages. Each page can be in one of three states: uncached, read-locked, or write-locked.

As an example, it is assumed that server node A wants to cache a disk B read. Server node A thus must first communicate with server node B, requesting a read lock for the page it wants to read. It gets the read lock, reads the data, and puts it in its sector cache. Now assume that server node B wants to write to the same page. Server node B has been informed that server node A has a read lock on this page. Server node B therefore communicates with server node A, and instructs server node A to break its read lock. Server node B then waits for a communication from server node A that the read lock has been released (whereupon server node A flushes the page from its cache). Then server node B has the write lock, and can write to the page. If server node A wants to read the page again, it requests a read lock. Server node B responds by denying the read lock but updating server node B's cache and fowarding the cached data to server node A. Server node A cannot cache this data, and will therefore be denied a read lock. A read lock can next be granted to server node A when disk B is updated from B's cache—namely at the next checkpoint. This implementation thus provides a distributed lock manager and does so in hardware.

## 1. OBJECT STORE STRUCTURES

### 1.1 Summary of Object Store Data Structures

The Object Store sub-module is used to maintain and store various types of file system objects. File system objects include file objects, directory objects, free-space allocation objects, modified checkpoint objects list objects, modified retained objects list objects, and mnode objects. File objects are created by the File System File sub-module for storage of user data and associated attributes, such as a word processor or spreadsheet files. Directory objects are created by the File System Directory sub-module for storage of directory information. Free-space allocation objects are created by the Free Space Allocation sub-module for storage of free-space allocation information. Modified checkpoint objects list objects and modified retained objects list objects (both of which are described in more detail below) are created by the Object Store sub-module for storage of information relating to checkpoints and retained checkpoints, respectively. An mnode object (which is described in more detail below) is a special object for holding excess file attributes associated with a file or directory object (i.e., file attributes that cannot fit within pre-designated areas within the file or directory object as described below, such as CIFS security attributes), and is created by the creator of the file or directory object, which includes a reference to the mnode object within the file or directory object.

The following is a summary of the data structures, which have been termed "onodes," employed by the Object Store sub-module to track locations of data stored on the disk. Additional details of the data structures are described later. (It should be noted that these data structures are exemplary of only one embodiment of the present invention.)

An object is made of a root onode and optionally a number of indirect and direct onodes. There are also a number of other on disk objects and structures that are used to control checkpoints and retained checkpoints. These are all described below.

There are three kinds of onodes—root, direct, and indirect. When an object (such as a file or directory, for example) is created, there is created a corresponding root onode (actually a pair of root onodes, as described below). Each root onode is stored on the disk in a given sector number of the disk and the sector number uniquely identifies the root onode and therefore also the root onode's corresponding object. In a typical embodiment, each sector is 512 bytes, so the size of a root onode is similarly limited to 512 bytes. As will become apparent, aspects of the present file structure implementation are similar to a basic Unix file structure, but traditional Unix systems have only a fixed number of indirect pointers, and when the fixed number of pointers is used, then a file size limit is reached. Additionally (among other things, traditional Unix systems use such storage techniques only for files and do not employ an object storage model in the manner of various embodiments of the present invention.

Root onodes are actually created in pairs. Since a root onode is identified by a sector number, the other member of its pair is the next higher sector number. The pair structure is employed so that one root anode of the pair is valid and complete with respect to a checkpoint. The other member of the pair is then updated, when its corresponding object is modified, to reflect the new state of the object. In normal processing, both members of the pair onode are read, and the one with the higher checkpoint number is the one that is current.

Many file system requests involve disk usage, and such usage is conveniently described with respect to blocks; a block is a specified data storage unit, and in common embodiments may range in size from 2Kbytes to 64Kbytes.

The root onode data structure includes a checkpoint number to identify under which checkpoint this version of the object has been created. Also in the root onode data structure is a parameter to identify the type of object for which the root onode is providing metadata. The object type may, for example, be any of freespace, file, or directory. In addition to object type, the root onode also has a parameter for the length of the object in blocks.

Another root onode parameter is the reuse count. A reuse count parameter is employed because if an object is deleted, it goes onto a queue of free root onodes rather than back into free space. When a given root onode is assigned to a new object, the associated root onode reuse count is incremented. The reuse count is part of the file handle associated with the object. By incrementing the reuse count each time the root onode is reused, file requests using a file handle created from an older version of the root onode can be identified and rejected.

As suggested above, the root onode also carries a series of pointers. One of these is a pointer to any immediately preceding version of the root onode. If it turns out that a retained checkpoint has been taken for the pertinent checkpoint, then there may have been stored an immediately preceding version of the root onode in question, and the pointer identifies the sector number of such an immediately preceding version of the root onode.

For the actual data to which the root onode corresponds, there is a separate pointer to each block of data associated with the root onodes's object. The location of up to 18 data blocks is stored in the root onode. For data going beyond 18 blocks, a direct onode is additionally required, in which case the root onode also has a pointer to the direct onode, which is identified in the root onode by sector number on the disk.

Like the root onode, the direct onode includes a parameter identifying the checkpoint number with respect to which the direct onode has been created. The direct onode is arranged to store the locations of up to about 60 or 61 blocks data pertinent to the object corresponding to the direct onode's root onode..

When a first direct onode is fully utilized to identify data blocks, then one or more indirect ondes are used to identify the first direct onode as well as additional direct onodes that have blocks of data corresponding to the object. In such a case the root onode has a pointer to the indirect onode, and the indirect onode has pointers to corresponding direct onodes. When an indirect onode is fully utilized, then additional intervening indirect onodes are employed as necessary. This structure permits fast identification of a part of a file, irrespective of the file's fragmentation.

The structure of direct and root onodes has another feature that, among other things, permits fast creation of large files, which typically are set to a 0 value when first created. This feature is a flag for each block pointer, in each root and direct onode, to identify whether the corresponding block has a 0 value.

There is a related feature that also facilitates the fast creation of large files. In any onode, every pointer to a block or to another onode has a bit to identify whether or not the block or onode has been actually created. In a case where the relevant blocks and onodes have not yet been created, then blocks and onodes are created as necessary to accommodate write requests, and the allocation bit is toggled accordingly. Note that creating a block requires allocation of space from the Free Space Allocation sub-module, writing the data to the block, and setting the bit flags for the pertinent onodes.

For recovery purposes, there is also stored a transaction log of all onodes that have been modified in a current checkpoint. Morever, the root onodes are not written to disk, until there is established a complete transaction log on disk of all modified root onodes. (Root onodes have this delayed write feature. Other onodes do not, and do not need to, since they are accessed only through root onodes.) In recovery mode with respect to a current invalid checkpoint, if the onode was modified in the current checkpoint, then the previous checkpoint value is used. (Note that onode contents are stored on disk along with the transaction log, as well has being maintained on the fly in metadata cache.)

Onode structure is also established, in this embodiment, in a manner to further reduce disk writes in connection with onode structure. In the end the onode structure must accommodate the storage not only of file contents but also of file attributes. File attributes include a variety of parameters, including file size, file creation time and date, file modification time and date, read-only status, and access permissions, among others. This connection takes advantage of the fact that changing the contents of a root onode can be performed frequently during a given checkpoint, since the root onode is not yet written to disk. (It will be recalled that disk writes of root onodes are delayed.) So a portion of the root onode is reserved for storage of file attributes.

More generally, the following structures for storage of file attributes are defined:

**enode** (little overhead to update, limited capacity). This structure is defined in the root onode and is 128 bytes.

**lnode** (intermediate overhead to update, and with greater capacity than the enode. The lnode is the first n bytes (typically 0-64K) of an object representing a file or directory (and which is therefore stored on disk in places pointed to by the root onode for the object). The lnode is used for such attribute information as, for CIFS purposes, a security descriptor.

**mnode** (expensive in overhead to update, near infinite capacity). This is a dedicated object for storage of data and therefore has its own storage locations on disk; the object is identified in the enode (or alternatively in the lnode).

The following provides a more detailed discussion of object storage in connection with the embodiment of Fig. 3.

## 1.2 Root Onode

Each root onode is 512 bytes in length. The following information is stored in the root onode :

- The checkpoint number with which this version of the object was created.
- The data length for this version of the object.
- The number of levels of indirection used in the runlist for this object.
- The type of the object. This is primarily used as a sanity check when a request comes in to access the object.
- A reuse count to say how many times this root onode has been used.
- A pointer to an older root onode version made for a retained checkpoint (if there is one).
- A pointer to a newer root onode version (will only be valid if this is a copy of a root onode made for a retained checkpoint).
- Up to 19 data block descriptors. Each data block descriptor includes a pointer to a data block, the checkpoint number with which the data was created, and a bit to say whether the block is zero filled.
- A single pointer to either a direct onode or an indirect onode.
- The 128 bytes of enode data for this object.
- A CRC and various sanity dwords to allow the root onode to be checked for validity.

For a given object, there are two places where the current valid version of the root onode could be stored. These are at a byte offset into the volume of either (obj_num *

512) or ((obj_num * 512) + 512). To find which one is the most up to date, they must both be read in, and the one which both passes all the validation checks and has the later creation checkpoint number is the latest valid version.

As discussed in Section 4 below, an object may include copies of root onodes that are created each time a retained checkpoint is taken. The pointer to the older root onode version and the pointer to the newer root onode version allow a doubly-liked list of root onodes to be created including the current root onode and any copies of root onodes that are created for retained checkpoints. The doubly-linked list facilitates creation and deletion of retained checkpoints.

## 1.3 Indirect Onode

The indirect onode provides a level of indirection between the root onode and the direct onode. Each indirect onode is 1 Kbyte in length. Although it is possible to pack a pair of indirect onodes into a disk block having aminimum disk block size of 2 Kbytes, each indirect onode is typically stored in a separate disk block for the sake of simplicity.

The following information is stored in the indirect onode :

- The checkpoint number with which the indirect onode was created.
- Up to 122 pointers to either indirect or direct onodes.
- A CRC and various sanity dwords to allow the indirect onode to be checked for validity.

As with the root onode currently valid indirect onodes are kept in pairs with one of the indirect onodes in the pair containing the most up to date version of the indirect onode. However, unlike the root onode there is no need to read in both of the indirect onode to work out which one is the most up to date, as the currently valid indirect onode will be pointed to directly from the current root onode.

## 1.4 Direct Onode

The direct onode provides direct pointers to data blocks on the disk. Each indirect onode is 1 Kbyte in length which means that a direct onode pair can fit into a current minimum disk block size of 2 Kbytes.

The following information is stored in the direct onode:

- The checkpoint number with which the direct onode was created.

- Up to 62 data block descriptors. Each data block descriptor includes a pointer to a data block, the checkpoint number with which the data was created, and a bit to say whether the block is zero filled.
- A CRC and various sanity dwords to allow the indirect onode to be checked for validity.

## 1.5 Dynamic Superblock

On the disk there are two dynamic superblocks – only one of which is considered to be the most up to date at any given point in time. These are used to record the state of the checkpoints on the disk.

The following information is stored in each dynamic superblock :

- The checkpoint number associated with this dynamic superblock.
- The handle of the modified checkpoint objects list object for this checkpoint.
- The object number of the modified retained objects list object from the last retained checkpoint.
- The state of this checkpoint. Possible states are WRITTEN_OBJ_LIST and CHECKPOINT_CREATED.
- A CRC and various sanity dwords to allow the indirect onode to be checked for validity.

Successive checkpoints alternate between which of the dynamic superblocks to use. When the software opens the volume it must read in both dynamic superblocks – the one with the later checkpoint number which has the volume state marked as CHECKPOINT_CREATED and passes all the sanity checks identifies the latest valid checkpoint on this volume. The OBJ_STORE OPEN_VOLUME call specifies which dynamic superblock the Object Store sub-module should use first – this will be the one which didn't specify the most up to date checkpoint.

## 1.6 Modified Checkpoint Objects List Object

At the start of each checkpoint, a modified checkpoint objects list object is created. Each time a different object is created or modified as part of this checkpoint, its object number is written to the modified checkpoint objects list object so that, when the checkpoint is created, there is an object that lists all the objects created or modified in that checkpoint.

### 1.7 Modified Retained Objects List Object

At the start of each retained checkpoint, a modified retained objects list object is created. Each time a different object is created or modified following creation of the retained checkpoint, and until the next retained checkpoint is taken, its object number is written to the modified retained objects list object.

## 2. BASIC OBJECT OPERATIONS

### 2.1 Object Creation and Deletion

When an object is first created (using a WFS API OBJ_CREATE call) it just has a root onode (actually a pair of root onodes) with no pointers to any indirect onodes, direct onodes, or data blocks.

One thing to note is that, once a disk block has been allocated as a root onode, it must never be used for anything else. This is because the handle returned for the root onode contains an object number which is the sector offset on the disk of the root onode. If the object were deleted and a client which had the handle cached then came in with another request for the file, the object store would go and read the data on the disk at the location specified by the object number. If this disk block had been reused, there is a possibility that it would look like a root onode (or actually be a new root onode) which could cause all sorts of problems.

To get around this problem, the following three things are done:

1. When a root onode is deleted its object type is set on the disk to be OBJ_TYPE_DELETED so that if a client tries to read the object in again the object store will know that the object has been deleted.

2. When objects are deleted the disk space used by their root onode is not returned to the free space allocation controller. Instead deleted root onodes are kept in a linked list of free root onodes (note that unlike data blocks it is safe to reuse these freed data blocks before a checkpoint is taken, due to the paired arrangement of the root onode). When an object is created, a free root onode is used if one is available. New disk space for the root onode is allocated only if no free root onodes are available.

3. When a root onode is first created using newly allocated free space it is given a reuse count of zero. Each time the root onode is reused for a new object the reuse count is incremented. Because the reuse count forms part of the handle returned to the client, this means that old handles referencing root onodes which have been reused will be detected as being invalid, because the reuse count will be wrong.

## 2.2 Object Data Creation

As data is created, it is first of all put into data blocks pointed to directly from the root onode. This is illustrated in the diagram of Fig. 7, showing use of a root onode with no other onodes. Note that, for the sake of simplicity in this and all the following diagrams, the root onode and direct onode are shown as having only two data pointers, and the indirect onode is shown as only having two indirect or direct onode pointers.

Once all the direct block pointers in the root onode are filled, then a direct onode A is created with a pointer from the root onode to the direct onode. Fig. 8 shows employment of a root onode with this direct onode A. Note that the root onode has multiple data block pointers but only a single pointer to either a direct or an indirect onode.

If the data in the object grows to fill all the data pointers in the direct onode, then an indirect onode B is created, as illustrated in Fig. 9. Fig. 9 shows employment of a root onode with an indirect onode as well as direct onodes. The pointer in the root onode which was pointing to the direct onode A, is changed to point at the indirect onode B, and the first pointer in the indirect onode B is set to point at the direct onode A. At the same time a new direct onode C is created, which is also pointed to from the indirect onode B. As more data is created more direct onodes are created, all of which are pointed to from the indirect onode.

Once all the direct onode pointers in the indirect onode B have been used another indirect onode D is created which is inserted between the root onode and the first indirect onode B. Another indirect onode E and direct onode F are also created to allow more data blocks to be referenced. These circumstances are shown in Fig. 10, which illustrates use of multiple layers of indirect onodes placed between the root onode and the direct onodes.

This process of adding indirect and onodes to create more levels of indirection is repeated to accommodate however much data the object contains.

It should be noted that the Inode portion of an object is handled by the Object Store sub-module as any other data portion of the object. The Object Store sub-module does not distinguish the Inode portion from the data portion, and does not automatically allocate the Inode portion. Rather, the entity that creates or modifies the object (typically sub-modules upstream from the Object Store sub-module, such as the File System File sub-module or the File System Directory sub-module) must determine how much space to leave for the Inode as data is added to the object.

### 2.3 Object Data Deletion

As data is deleted from the object and data blocks and direct and indirect onodes are no longer required they are returned to the free space allocation controller.

In accordance with one embodiment, the number of levels of indirection as the object gets smaller, until all the data in the object can be referenced via the direct block pointers in the root onode, at which point all the remaining direct and indirect onodes are freed and the indirection level will be set to zero.

### 2.4 Zero Filling

If a write to a file is done which has a start offset beyond the current end of the file, then the undefined portion of the file between the current end and the start of the new write data must be filled with zeroes. The same thing occurs if the length of the object is set to be greater than the current length.

This is particularly problematic if a file is created and then the length is set to be, say, 1GB. In a straightforward implementation this would require that the disk blocks allocated to the file actually be written to with zeroes. For a 1GB file, this would take of the order of 10 seconds. For a 1TB file, it will take of the order of 3 hours.

In embodiments of the present invention, this problem is avoided by having a bit with each data block pointer to say whether that block is zero filled. If the Object Store sub-module sees this bit set, then it knows that this block should be filled with zeroes, even though on disk it may contain something completely different. If the block is read, then Object Store sub-module will return zeroes for this block rather than its on-disk contents. If the block is written to with a write which doesn't fill the entire block, then the Object Store sub-module will first write zeroes to all of the block which isn't being written to and will reset the zero filled bit for this block.

Note that, in this case, disk blocks will be allocated for all zero filled portions of the file, although the disk blocks will not be filled with zeros.

### 2.5 Sparse Onode Structures

Once the zero filling problem has been solved, the next order problem with setting the length of an object to some very large value is the time it takes to allocate the data blocks and create the required direct and indirect onode structure. With a disk block size of 4K, a 1TB object requires approximately 4 million direct onodes as well as a lesser number of indirect onodes. This would take in the order of 40 seconds to write to disk. Also the free space allocation of all the data blocks required, and the subsequent updates to the free space bitmap, would significantly add to this time. If a checkpoint were to be taken immediately after the file creation begins, the entire system would stop servicing requests (to any volumes) for the whole of this time.

In an embodiment of the invention, this problem is solved by a twofold approach. The first aspect of the solution is not to actually allocate disk blocks for the zero filled portions of the file. This means that when the object store sees a write to a zero filled block it would first have to allocate disk space for that block and put a pointer to it in the relevant onode structure.

The second aspect builds on the first and says, in addition to not allocating the data blocks, don't create the onode structure either. To implement this aspect, each onode pointer has a bit to say whether the onode it points to is allocated or not. If not, when an operation comes along which requires that onode to be valid, only then is disk space allocated for it and the correct pointer inserted. In this way a huge zero filled object will have only a root onode, which can obviously be created very quickly.

## 3. CHECKPOINTS

### 3.1 Introduction to File System Consistency

One of the essential features of a file system is the ability to maintain file system consistency in the event of a system crash.

For embodiments of the file system herein, a checkpoint mechanism is used to maintain file system consistency, with, however, implementations differing from those of the prior art. Instead of always writing metadata to new areas of disk, as in typical prior art systems, two copies of any given piece of onode metadata are maintained, one of which is valid and the other of which may be in the process of being updated. If the system crashes while one copy is being updated, the system can revert to the other copy, which is guaranteed to be valid. For user data, the system can, on a per object basis, have the option of either always writing it to new areas on disk or overwriting the existing data to give either consistency in user-triggered file system data structure saves or higher performance and no file fragmentation. All of this is described in more detail in the following sections.

## 3.2 User Data Handling

User-data handling is considered first, as what is done with the user data affects how the metadata is handled.

It is important first to define that by "user data" it is meant anything not contained in an object's root onode, indirect onodes or direct onodes. What is user data to the object store may be metadata (such as a directory listing or a free space bitmap) to another part of the file system embodiment herein. For data such as this, it is important to make sure that the data on disk in the checkpoint is consistent in order to ensure that the on-disk file system is always consistent – even if the contents of the NVRAM are lost.

Root onodes are always written to the storage module using delayed write commands. Delayed writes are marked with a tag number, and the data associated with them is not written to disk until a tag flush is done with the correct tag number – see the section on onode handling for a description of why this is done. One problem with this is that there must be an assurance that the sector cache on the storage module never fills up with dirty root onodes as this would lock the entire system up. For the other onode structures and onode data, normal tagged writes can be used, with a different tag number to that used for root onodes. This gives the storage module the option of not having to wait for the tag flush before writing them to disk and reduces the danger of the sector cache filling up.

The onode user data can be handled in a number of different ways. Two data handling modes, namely data copy mode and data overwrite mode, and an optional third data handling mode, namely data pair mode, are discussed below. The data handling modes are selectable on a per object basis.

### Data Copy Mode

Objects using this mode guarantee that both the checkpointed metadata and user data for the object will be consistent. This mode should be used for user data which to other blocks in the system is actually metadata.

In data copy mode, when it is time to write to a data block that was previously written with an earlier checkpoint number, the following is done:

Allocate a new data block.

Copy the contents of the old block to the new block (not required if the new data fills the entire block).

Write the new data to the new block.

Update the onode data pointers to point at the new block.

Return the old block to the free space allocation controller.

The last step may seem somewhat unusual as, at this point, the old block is still part of the previous checkpoint, and there would be trouble if the Free Space Allocation sub-module then gave the block out again in response to a free space request, as the checkpointed data would then be overwritten. However, one of the requirements for the Free Space Allocation sub-module is that blocks returned to it as free space are never given out again until after a checkpoint has been taken. This makes the last step safe.

**Data Overwrite Mode**

Objects using this mode guarantee that checkpointed metadata will be consistent but not necessarily checkpointed user data. This mode could be used for all data which is true user data (i.e. file contents).

In data overwrite mode, when it is time to write to a data block that was previously written with an earlier checkpoint number, the following is done:

Write the new data to the old data block.

Note that in overwrite mode, there is only a problem with data consistency if the system crashes and the contents of the NVRAM are lost. As long as the NVRAM is functioning, the user data can be placed into a consistent state by replaying the contents of the NVRAM. This is summarised in the table below.

| Mode | NVRAM enabled | On disc file system after crash | Data written since last checkpoint after crash |
|---|---|---|---|
| Data Copy | Yes | Metadata and user data completely consistent | Completely recoverable from NVRAM |
| Data Copy | No | Metadata and user data completely consistent | Lost |
| Data Overwrite | Yes | Metadata internally completely consistent, but inconsistent with respect to the user data | Completely recoverable from NVRAM. When recovered the on disc user data and metadata are then consistent. |
| Data Overwrite | No | Metadata internally completely consistent, but | Lost |

| | | | |
|---|---|---|---|
| | | inconsistent with respect to the user data | |

**Data Pair Mode**

Considering an object such as the free space object, it will need to use data copy mode as it contains file system metadata. However, in this mode it is likely to become highly fragmented as it is constantly being updated.

For objects of this type, a mode is included whereby every data block has a pair in the same way as with the onode structures. This would allow swapping between the blocks in the pair as the data is checkpointed. Doing this would help to alleviate the problem of file fragmentation as well as eliminate the need for the Free Space Allocation sub-module to handle all the allocation and freeing of blocks as the object is modified.

Note that the free space object is particularly conducive to this sort of treatment as in normal operation it never changes size.

## 3.3 Onode Handling

As has already been explained every onode structure (root, indirect or direct onode) is actually made up of a pair of the structures. For want of better names, the individual structures will be referred to as the left hand side (LHS) and right hand side (RHS) of the pair.

Consider first of all the creation of a root onode during checkpoint A, which is illustrated in Fig. 11. When it is first created the root onode is written to the LHS of the pair. Note that, because the root onode is written to using delayed writes so although it is valid in the storage module, it will not get written to disk until a checkpoint is created.

All changes to the root onode (such as writes to the object or the deletion of the object and the creation of a new object using the same root onode) which take place before checkpoint A is created will be done on the LHS root onode.

When it is time to create checkpoint A, a tagged flush is issued which causes the LHS to be written to disk. If while the checkpoint is being taken some more modifications are made to the root onode (which will be reflected in checkpoint B) these are written to the RHS of the root onode, as shown in Fig. 12.

Once checkpoint A has been created and the root onode has been written to disk the root onode pair has the appearance illustrated in Fig. 13.

Suppose that the system begins to create checkpoint B. When the tagged flush for B is issued, the root onode will have the appearance of Fig. 14.

Suppose again that while the RHS of the root onode for checkpoint B is still being written to disk the object is modified again as part of checkpoint C. The LHS version of the root onode on disk still contains the latest valid checkpoint A, since checkpoint B has not yet been created. Delayed writes can therefore be used to update the LHS, but must ensure that none of the changes to the LHS are written to disk until checkpoint C is created. The situation while checkpoint B is being created is shown in Fig. 15.

It might appear that, once checkpoint B has been created, the system can start to write the LHS root onode for checkpoint C to disk. This is true for direct and indirect onodes but not for root onodes. The reason for this is that if the system were to crash before checkpoint C had been created, but by then the LHS of the root onode had been written to disk, then, when the object store came to read the pair of root onodes to find out which was the latest valid one, it would think that the LHS root onode associated with checkpoint C was the most up to date, which would be incorrect. For this reason, before any root onodes are written to disk, a modified checkpoint objects list is written in order to say which root onodes are going to be modified. This allows the system to recover from a crash whilst updating root onodes. This is covered further in the section on restoring a checkpoint.

Note that, for the sake of simplicity in all the following diagrams, this intermediate state where the checkpoint is in the middle of being created is ignored.

Imagine that the system starts with a root onode that looks as shown in Fig. 16.

Suppose the root onode is now extended such that there are two levels of indirection. Before taking the checkpoint, the structure will have the appearance of Fig. 17. Note that the indirect and direct onodes are all written with delayed writes before the previous checkpoint has been fully created and tagged writes once the previous checkpoint is valid on disk. The data is all written with tagged writes.

If a checkpoint is then taken, the structure will have the appearance of Fig. 18.

Now suppose a write to data block 2 is done with the object in data copy mode. In this case, a new data block 2 is allocated and all of the onode structures are updated to point at this new block. Before the checkpoint the structure will have the appearance illustrated in Fig. 19. Note that the checkpointed data has been preserved as the original, and the original data block 2 has not been modified.

And after the checkpoint the structure will have the appearance shown in Fig. 20.

Suppose now that with this new structure, data block 1 is written with the object in data overwrite mode. In this case, only the root onode (which will have a new enode) is updated as nothing else in the enode structure is changing. Note that in this mode the checkpointed data has been corrupted as the checkpointed version of block 0 has been modified. The result of this activity is illustrated in Fig. 21.

## 3.4 Storage Module Tag Numbers

During the checkpoint process a number of different storage module tag numbers are used. These are detailed in the table below.

| Tag Number | | | Used for | Storage Module Inquiry Type |
|---|---|---|---|---|
| Checkpoint N | Checkpoint N + 1 | Checkpoint N + 2 | | |
| T0 | T2 | T0 | Modified checkpoint objects list | Tagged Writes |
| D0 | D2 | D0 | Root Onodes | Delayed Writes |
| D1 | D3 | D1 | Direct & Indirect Onodes before previous checkpoint has been created | Delayed Writes |
| T1 | T3 | T1 | Direct & Indirect Onodes and Onode Data | Tagged Writes |
| T4 | T4 | T4 | Dynamic Superblock | Tagged Writes |

A given 32K storage module sector cache block can only be in one delayed write tag queue and one non delayed write tag queue. There is therefore the question of what happens if the same block is written to with different tag numbers.

The dynamic superblock is arranged such that it is the only thing in its 32K sector cache block which means that the sector cache block in which it lives can never be written to with a different tag number.

For a given buffer if there are both root onode delayed writes and direct and indirect onode delayed writes for the same checkpoint number the buffer must end up on the root onode delayed write tag queue.

For the two delayed write inquiries the checkpoint is currently organised such that there should never be any buffers with delayed write tags from checkpoint N when starting to do delayed writes for checkpoint N + 1. If a cache block could be in two delayed write tag queues with separate dirty block bitmaps for each then the system could start to do delayed writes for the next checkpoint before the delayed write tagged flushes for the previous checkpoint have been issued. This is discussed in more detail in the section of taking a checkpoint below.

For the other two tagged write structures the way the checkpoint is currently organised there should never be any tagged buffers in the storage module from checkpoint N when the system starts doing tagged writes for checkpoint N +1. Within a checkpoint if a cache block is written to which already has a tag number assigned to it, an assurance is needed to make sure that the block ends up in the modified checkpoint objects list tag queue. This would become more complicated if the performance improvement proposed below were made to decrease the time the system is unable to process new requests while taking the checkpoint.

### 3.5 Taking a Checkpoint – The Simple Version

There are various reasons why the file system software may need to take a checkpoint.

- The half of the NVRAM being used for this checkpoint is becoming full.
- The sector cache on the storage module is becoming full.
- It is more than a previously determined period of time (typically 10 seconds) since the last time a checkpoint was taken.
- The user has requested that a retained checkpoint be taken.

There may be other times when it is necessary, desirable, or convenient to take a checkpoint.

At a system level taking a checkpoint involves the following operations on each volume which is mounted:

1. Halt all operations in the system so that the file system is in a consistent state.
2. Tag flush the modified checkpoint objects list object in the storage module.
3. Update this checkpoints dynamic superblock to say that the modified checkpoint objects list object has been written.
4. Tag flush the onode structures and onode data in the storage module.
5. Update the dynamic superblock to say that this checkpoint has now been created.

As soon as step 4 has commenced the system can begin to process new inquiries.

## 3.6 Taking a Checkpoint – Details

The description below details the actual operations required to take a checkpoint. These matters are summarized in Fig. 22. The operations are described assuming only a single volume is mounted – if there are multiple volumes then the operations in each step are repeated for each volume mounted.

1. The file system software waits until it has pushed a set of operations into the Non-Volatile Storage Processing sub-module which when completed will give a consistent file system.

2. The software then pushes a WFS_CREATE_CHECKPOINT inquiry into the Non-Volatile Storage Processing sub-module. This command includes the checkpoint number to use for the next checkpoint.

3. The Non-Volatile Storage Processing sub-module waits until all the commands prior to the checkpoint inquiry have been pushed to both the File System File sub-module and its cluster pair machine (if there is one), and it has stored all of those commands in its own NVRAM.

4. The Non-Volatile Storage Processing sub-module generates a new checkpoint command which has Non-Volatile Storage Processing sub-module as the source and the File System File sub-module as the destination. The Non-Volatile Storage Processing sub-module can then begin to process more requests from the processor, which now get stored in the other half of the NVRAM, and can begin to pass these requests on to the File System File sub-module.

5. The File System File sub-module waits until all the commands prior to the checkpoint inquiry have completed. Until this happens it can't begin to process any new commands from the Non-Volatile Storage Processing sub-module . Note that this is the point in the whole checkpoint operation where the longest delay is likely to be incurred – our current estimate being that on a busy file system this operation might take 10s of milliseconds. One option to alleviate this would be to allow the File System File sub-module to continue to process operations that won't make any modifications to the disk while waiting for outstanding operations to complete.

6. The File System File sub-module then generates a new checkpoint inquiry with the File System File sub-module as the source and the File System Directory sub-module as the destination. At this point it can begin to process new command from the Non-Volatile Storage Processing sub-module .

7. Steps 5 and 6 are then repeated for the File System Directory sub-module and File System Tree sub-module. Note that for each of these shouldn't be any need to wait for outstanding operations to complete as the wait in step 5 should have ensured that there are no outstanding operations.

8. When the Object Store sub-module receives the checkpoint command from the File System Tree sub-module it sends a create checkpoint inquiry to the Free Space Allocation sub-module. At this point it also stops processing any new inquiries from the File System Tree sub-module interface.

9. The Free Space Allocation sub-module sends back the checkpoint response when it has completed any outstanding free space inquiries and updated the appropriate bitmaps. Note that it must continue to process new free space inquiries (and wait for these to complete) while waiting to send the checkpoint response as the object inquiries it is sending to the Object Store sub-module may result in more free space inquiries being generated by the Object Store sub-module. Measures should be taken to prevent or escape from an "endless loop" situation caused by processing free space inquiries and waiting for free space inquiries to complete before sending the checkpoint response.

10. When the Object Store sub-module receives the checkpoint response from the Free Space Allocation sub-module it sends a tagged flush to the storage module to tell it to flush the modified checkpoint objects list object for this volume. The modified checkpoint objects list is an object which records the object number of all the objects which have been modified during the current checkpoint on a given volume.

11. When the TAG_FLUSH of the modified checkpoint objects list object completes the Object Store sub-module writes to the dynamic superblock for this checkpoint number with the state set to WRITTEN_OBJ_LIST and the handle of the objects list object. This needs to be written through to disk

using a tagged write followed by a tagged flush. Note that the flush of the direct and indirect onodes and onode data could be issued at the same time as this is done in order to get the checkpoint written more quickly (although this may increase the time taken to write the dynamic superblock).

12. When the dynamic superblock has been written to disk a tagged flush can be issued for all the root onodes (and for the direct and indirect onodes if this hasn't been done earlier).

13. At the same time as the tag flush is issued the Object Store sub-module can begin work on the next checkpoint. This means updating the current checkpoint number to be the one indicated in the checkpoint inquiry, switching over to using the correct tag number for all the structures in the new checkpoint and starting to process inquiries from File System Tree sub-module again. Alternatively, if the storage module cache controller is changed so that a 32K cache block could be in two delayed write tag queues (with a separate dirty block mask for each) it would be possible to begin work on the next checkpoint at the same time as the tag flush is issued in step 10. This could improve performance as there may be a significant delay between steps 10 and 12.

14. When the two tagged flushes of the onode data and onode structures completes the Object Store sub-module writes to the dynamic superblock for this checkpoint number with the state set to WRITTEN_CHECKPOINT. This needs to be written through to disk using a tagged write followed by a tagged flush.

15. When the dynamic superblock has been written to disk the checkpoint has been successfully created. The Object Store sub-module sends a checkpoint response to the File System Tree sub-module which via the File System Directory sub-module and the File System File sub-module results in a checkpoint response getting back to the Non-Volatile Storage Processing sub-module. When this sees the checkpoint response it can discard all the saved data in the NVRAM associated with this checkpoint.

16. The Non-Volatile Storage Processing sub-module then passes the response back to the processor. Only when the processor has seen the checkpoint response can it request the generation of another checkpoint.

## 3.7 Restoring a Checkpoint

When a volume is mounted, the system will normally want to go back to the last valid checkpoint.

To work out which this is the software needs to read in both of the dynamic superblocks. Both of them should be valid. The way that Object Store sub-module writes the dynamic superblock should ensure that writing a superblock cannot leave the system with a corrupted dynamic superblock on disk. Additional measures could be taken to better assure that both dynamic superblocks are valid, for example, performing two checkpoint operations before allowing any operations to be performed on the volume.

Assuming that both of the dynamic superblocks are valid the software then looks for the one with the later checkpoint number. There are two possibilities for the state of this superblock.

**WRITTEN_OBJ_LIST**

This state means that the object store had written the modified checkpoint objects list to disk but hadn't yet written out all onode structures and onode data when the system crashed. This implies that it was an unclean system shutdown and that the last valid checkpoint on disk is the one recorded in the other dynamic superblock – the state of which should be WRITTEN_CHECKPOINT.

In this state some of the root onodes on disk may have been updated as part of the creation of this checkpoint. This would be a problem when reading in this root onode as of the pair of the one written in this checkpoint would look like the latest valid one, which would be incorrect as this checkpoint wasn't complete. Note that the same problem doesn't apply to all the other onode structures as the correct one out of the pair to use is pointed to directly by the object that references it.

This problem is handled by making use of the modified checkpoint objects list. This is written to disk before any changes are made to the root onodes and provides a transaction log of which root onodes are going to be modified in the checkpoint. In the case of the checkpoint failing the software reads the modified checkpoint objects list object and goes through each of the objects it points to. For each of these it must read in the pair of root onodes and if either of them was written to in the invalid checkpoint it is invalidated.

**WRITTEN_CHECKPOINT**

This state means that the object store wasn't in the process of writing onode structures and data to disk when the system went down and the checkpoint number defined in this dynamic superblock is the last valid checkpoint on disk. This doesn't mean that the volume was shutdown cleanly, so there may still be operations in the NVRAM which need to be replayed.

## 4. RETAINED CHECKPOINTS

A checkpoint is only valid until the next checkpoint has been created, at which point the checkpoint no longer exists. Therefore, a user-triggered mechanism is provided for retaining a checkpoint such that it will remain valid and accessible (read-only) until the user chooses to delete it. As discussed above, such a checkpoint that is retained through this user-triggered mechanism is referred to herein as a retained checkpoint. The Object Store sub-module is capable of maintaining multiple retained checkpoints. As long as a retained checkpoint remains active, the onodes and data blocks that comprise the retained checkpoint cannot be modified or returned to free space. It should be noted that an onode or data block can be a component of multiple retained checkpoints, and a particular onode or data block cannot be returned to free space as long as the onode or data block is a component of at least one retained checkpoint.

### 4.1 Creating a Retained Checkpoint

A retained checkpoint is initially created on a given volume by performing the following sequence of operations :

1. Take a checkpoint.
2. Issue a command to the Object Store sub-module for the required volume to create the retained checkpoint.
3. Take another checkpoint.

When the Object Store sub-module receives the command to create the retained checkpoint, it updates a record indicating which checkpoint number the last retained checkpoint on the volume was created with. This is recorded in the dynamic superblock and gets saved to disk when the checkpoint in operation 3 is taken. It should be noted that a retained checkpoint can be taken on multiple volumes in operation 2.

## 4.2 Modifying an Object after a Retained Checkpoint

Whenever the Object Store sub-module receives a request to modify an object, it first checks the root onode object to determine the checkpoint number at which the root onode object was created. If the root onode object was created prior to creation of the last retained checkpoint, then the root onode object is part of that last retained checkpoint. In this case, the root onode object cannot be modified as described above, as this would corrupt the version of the object in the retained checkpoint. Rather, the object is modified in a special way as described by example below.

FIG. 23 shows an object structure for an exemplary object that was created at a checkpoint number 1. The object includes four data blocks, namely data block 0 (2310), data block 1 (2312), data block 2 (2314), and data block 3 (2316). A direct onode 2306 includes a pointer to data block 0 (2310) and a pointer to data block 1 (2312). A direct onode 2308 includes a pointer to data block 2 (2314) and a pointer to data block 3 (2316). An indirect onode 2304 includes a pointer to direct onode 2306 and a pointer to direct onode 2308. A root onode 2302 includes a pointer to indirect onode 2304. All onodes and all data blocks are marked with checkpoint number 1.

Suppose now that a retained checkpoint is taken at checkpoint number 2, and data block 0 (2310) is to be modified in checkpoint number 3.

In this case, the Object Store sub-module first loads the root onode 2302 for the object and realizes that the root onode 2302 (which was created at checkpoint number 1) was created prior to the last retained checkpoint being taken at checkpoint number 2. It is preferable for the most up-to-date root onode be at the sector number indicated by the object number, in order to optimize access to the most up-to-date version of the object. Therefore, before doing anything else, the Object Store sub-module saves a copy of the old root onode 2302 to free space on the disk, writes a pointer to the saved root onode into the updated root onode, and writes a pointer to the updated root onode into the saved root onode.

FIG. 24 shows the object structure after a copy of the old root onode is saved to free space on the disk. Specifically, block 2403 represents the copy of the old root onode 2302 saved to free space on the disk. A pointer to the current root onode 2402 is written into the saved root onode 2403. Block 2402 represents the updated root node with checkpoint number 3. A pointer to the saved root onode 2403 is written into the current root onode 2402.

The Object Store sub-module then traverses the object structure starting at the root onode until it reaches the descriptor for data block 0 (2310). Since data block 0 (2310) was created prior to the last retained checkpoint being taken, it cannot be modified. Instead, the Object Store sub-module writes a copy of data block 0 (2310), with the required data modifications, to free space on the disk.

FIG. 25 shows the object structure after a modified copy of data block 0 is written to free space on the disk. Specifically, block 2510 represents the modified copy of data block 0 written to free space on the disk. Block 2510 includes checkpoint number 3 (i.e., the checkpoint at which it was created).

The Object Store sub-module now needs to put a pointer to the new data block 2510 in a direct onode, but the Object Store sub-module cannot put a pointer to the new data block 2510 in the direct onode 2306 because the direct onode 2306 is a component of the retained checkpoint. The Object Store sub-module therefore creates a new direct onode with pointers to the new data block 0 (2510) and the old data block 1 (2312).

FIG. 26 shows the object structure after a new direct onode is created for the new data block. Specifically, block 2606 represents the new direct onode. Block 2606 includes checkpoint number 3 as well as pointers to the new data block 0 (2510) and the old data block 1 (2312).

The Object Store sub-module now needs to put a pointer to the new direct onode 2606 in an indirect onode, but the Object Store sub-module cannot put a pointer to the new direct onode 2606 in the indirect onode 2304 because the indirect onode 2304 is a component of the retained checkpoint. The Object Store sub-module therefore creates a new indirect onode with pointers to the new direct onode 2606 and the old direct onode 2308.

FIG. 27 shows the object structure after a new indirect onode is created for the new direct onode. Specifically, block 2704 represents the new indirect onode. Block 2704 includes checkpoint number 3 as well as pointers to the new direct onode 2606 and the old direct onode 2308.

Finally, the Object Store sub-module writes a pointer to the new indirect onode 2704 in the current version of the objects root onode 2402.

FIG. 28 shows the object structure after the pointer to the new indirect onode 2704 is written into the current version of the objects root onode 2402.

It should be noted that, after modification of data block 0 is complete, blocks 2403, 2304, 2306, and 2310 are components of the retained checkpoint but are not components of the current version of the object; blocks 2308, 2312, 2314, and 2316 are components of both the retained checkpoint and the current version of the object; and blocks 2402, 2704, 2606, and 2510 are components of the current version of the object but are not components of the retained checkpoint.

Suppose now that a retained checkpoint is taken at checkpoint number 4, and data block 3 (2316) is to be deleted in checkpoint number 5. The procedure here is similar to the procedure described above for modifying data block 0, and is described with reference to FIG. 29 which shows the object structure after deleting data block 3.

In this case, the Object Store sub-module saves a copy of the old root onode from checkpoint number 3, represented by block 2903, to free space on the disk, updates the root onode object 2902 to include checkpoint number 5, and updates various pointers in the current and saved root onodes. Specifically, saved root onode 2903 is essentially inserted into a doubly-linked list between the current root onode 2902 and the earlier saved root onode 2403. In the current root onode 2902, the pointer to an older root onode version is updated to point to the saved root onode 2903 rather than to the earlier saved root onode 2403. In the earlier saved root onode 2403, the pointer to a newer root onode version is updated to point to the newer saved root onode 2903 rather than to the current root onode 2902. In the saved root onode 2903, the pointer to a newer root onode version is updated to point to the current root onode 2902, while the pointer to an older root onode version is updated to point to the earlier saved root onode 2403.

The Object Store sub-module then traverses the object structure starting at the root onode until it reaches direct onode 2308, which includes the descriptor for data block 3 (2316). Because direct onode 2308 and data block 3 (2316) are components of an existing retained checkpoint, the Object Store sub-module cannot simply delete data block 3 (2316) and modify direct onode 2308 to remove the descriptor for data block 3 (2316). Therefore, the Object Store sub-module creates a new direct onode 2908 having checkpoint number 5 and a pointer to data block 2 (2314) but no pointer to data block 3 (2316). The Object Store sub-module also creates a new indirect onode 2904 having checkpoint number 5 and pointers to old direct onode 2606 and new direct onode 2908.

Finally, the Object Store sub-module writes a pointer to the new indirect onode 2904 into the current version of the root onode 2902.

It should be noted that, after deletion of data block 3 is complete, blocks 2903, 2403, 2304, 2704, 2306, 2308, 2310, and 2316 are components of at least one retained checkpoint but are not components of the current version of the object; blocks 2606, 2510, 2312, and 2314 are components of the current version of the object and at least one retained checkpoint; and blocks 2902, 2904, and 2908 are components of the current version of the object but are not components of any retained checkpoint.

## 4.3 Accessing a Retained Checkpoint

When the Object Store sub-module is asked to perform an operation on an object, it is passed a handle to allow it to identify the object. Among other things, this handle specifies the checkpoint number of the required object. Normally, this would be set to a value that indicates the current version of the object. However, if a different checkpoint number is specified, then the Object Store sub-module performs the operation on the requested version of the object.

The Object Store sub-module attempts to find the requested version of the object by stepping through the current and saved root onodes, using the pointer from a newer version of a root onode to an older version of a root onode, until a root onode is found having the requested checkpoint number or an earlier checkpoint number. The Object Store sub-module then traverses the object structure from that root onode. This is demonstrated by example with reference again to FIG. 29.

If the Object Store sub-module receives a request for checkpoint number 5, then the Object Store sub-module first goes to the current version of the root onode object 2902. The current root onode 2902 has checkpoint number 5, which is the requested checkpoint number. The Object Store sub-module therefore traverses the object structure from root onode 2902 to provide the requested version of the object. Specifically, root onode 2902 points to indirect onode 2904. Indirect onode 2904 points to direct onodes 2606 and 2908. Direct onode 2606 points to modified data block 0 (2510) and to data block 1 (2312). Direct onode 2908 points to data block 2 (2314). Thus, the current version of the object includes the modified data block 0 and excludes deleted data block 3.

If the Object Store sub-module receives a request for checkpoint number 4, then the Object Store sub-module first goes to the current version of the root onode object

2902. The current root onode 2902 has checkpoint number 5, which is too recent, so the Object Store sub-module follows the pointer to saved root onode 2903. The root onode 2903 has checkpoint number 3, which is earlier than the requested version of the object. The Object Store sub-module therefore traverses the object structure from root onode 2903 to provide the requested version of the object. Specifically, root onode 2903 points to indirect onode 2704. Indirect onode 2704 points to direct onodes 2606 and 2308. Direct onode 2606 points to modified data block 0 (2510) and to data block 1 (2312). Direct onode 2308 points to data block 2 (2314) and to data block 3 (2316). Thus, the retained checkpoint for checkpoint number 4 includes the modified data block 0 and also includes data block 3.

If the Object Store sub-module receives a request for checkpoint number 2, then the Object Store sub-module first goes to the current version of the root onode object 2902. The current root onode 2902 has a checkpoint number of 5, which is too recent, so the Object Store sub-module uses the pointer in root onode 2902 to access saved root onode 2903. The saved root onode 2903 has a checkpoint number of 3, which is also too recent, so the Object Store sub-module uses the pointer in root onode 2903 to access saved root onode 2403. The saved root onode 2403 has a checkpoint number of 1, which is earlier than the requested version of the object. The Object Store sub-module then traverses the object structure from saved root onode 2403 to provide the requested version of the object. Specifically, the root onode 2403 points to indirect onode 2304. Indirect onode 2304 points to direct onodes 2306 and 2308. Direct onode 2306 points to data block 0 (2310) and to data block 1 (2312). Direct onode 2308 points to data block 2 (2314) and to data block 3 (2316). Thus, the retained checkpoint for checkpoint number 2 includes the original four data blocks.

It should be noted that, if the Object Store sub-module is unable to find the requested version of an object, then the Object Store sub-module typically generates an error message. For example, with reference again to FIG. 29, if the Object Store sub-module receives a request for checkpoint number 0, then the Object Store sub-module steps through the root onodes until it reaches root onode 2403. The root onode 2403 is too recent but also does not have a pointer to an earlier root onode, so the Object Store sub-module generates an error message indicating that the requested version of the object could not be found.

It should also be noted that the retained checkpoints are not permitted to be modified, and the Object Store sub-module will only allow read operations to be performed on them.

## 4.4 Deleting a Retained Checkpoint

There are two stages to the process of deleting a retained checkpoint.

The first stage involves getting a list of all of objects that were either created or modified in the retained checkpoint that is being deleted. This is achieved by means of a special object (modified retained objects list objects) that is produced for every retained checkpoint. This object is created when either a volume is opened for the very first time, or after a retained checkpoint has been taken. Every time an object is created, or the first time an object is modified if it was created in a previous retained checkpoint, the object number is written to this object. The object number for this special object is stored in the dynamic superblock. Before creating a retained checkpoint, the software records the object number of this special object for when it later wants to delete that retained checkpoint.

The second stage of deleting the retained checkpoint involves the following sequence of operations for each object either created or modified in the retained checkpoint:

1. Lock the object so that it can't be used by another operation. This is only required if the retained checkpoint is being deleted on a live filesystem.

2. Find the root onode for the retained checkpoint, the root onode for the previous retained checkpoint (if one exists), and the root onode for either the next retained checkpoint (if one exists) or the current version of the object if the most recent retained checkpoint is being deleted and the object has not been deleted.

3. Go through the structure of the retained checkpoint being deleted and identify all the indirect and direct onodes and data blocks used by it. For each such onode and data block, determine whether the item is only used by the retained checkpoint being deleted. This can be done by finding the equivalent item in both the previous and next versions of the object. If the equivalent item is different in the previous and next versions of the object, then the item is unique to this retained checkpoint.

4. If the item is only used by the retained checkpoint being deleted, then it is no longer required, so it is returned to the free space. If the item is used by the next retained checkpoint, then the item is added to the updated retained objects list for the next

checkpoint, if it is not already in the updated retained objects list for the next retained checkpoint. By adding the item to the updated retained objects list, the Object Store sub-module will know to check if the item is still required when that retained checkpoint comes to be deleted.

5. Finally, if the root onode for this retained checkpoint is only used in this retained checkpoint, it too is no longer required and is deleted. In this case, if there is an older retained checkpoint, the pointer backwards from the next version of the root onode (if any), which previously pointed to the root onode of the retained checkpoint being deleted, is updated to point at the root onode of the previous retained checkpoint.

Note that in order to maintain file system integrity, careful attention needs to be paid to how retained checkpoint deletion ties in with the process of taking checkpoints, to make sure that checkpoints always represent a consistent view of the file system and that a crash in the middle of deleting a retained checkpoint can be recovered.

Deleting a retained checkpoint can be demonstrated by example. With reference again to FIG. 29, suppose that the retained checkpoint created with checkpoint number 4 is to be deleted. This retained checkpoint is represented by root onode 2903. The only items in the structure that are used only by this retained checkpoint are the root onode 2903 and the indirect onode 2704. These onodes are returned to free space. The root onode 2902 is updated to point at the root onode 2403 rather than to the deleted root onode 2903. FIG. 30 shows the object structure after the retained checkpoint for checkpoint number 4 is deleted.

With reference again to FIG. 30, suppose now that the current version of the object is to be deleted while the retained checkpoint for checkpoint number 2 still exists. This is similar to the case of a retained checkpoint being deleted in that there is a need to identify all of the items in the structure that are unique to the current version and return these to the free space. In this case, onodes 2904, 2606, and 2908 are used for the current version of the object but not for any remaining retained checkpoint, so these onodes are returned to free space. The current root onode 2902 is modified to indicate that it now references an object which has been deleted, but still has a pointer to at least one valid retained checkpoint. FIG. 31 shows the object structure after the current version of the object has been deleted.

When all remaining retained checkpoints for this object are deleted, the current version of the root onode 2902 is returned to the free root onode queue rather than to free space.

### 4.5 Reverting to a Retained Checkpoint

Under some conditions, it may be necessary or desirable to revert the live file system to a version represented by a retained checkpoint. Reverting the live file system to a version represented by a retained checkpoint can be accomplished in a number of different ways. Reverting the live file system to a retained checkpoint may involve such things as:

1. Copying the relevant contents of the root onode associated with the retained checkpoint into the current root onode (e.g., pointers to direct and indirect onodes and data blocks, pointer to earlier retained checkpoint, etc.).

2. Identifying the root onode associated with the preceding retained checkpoint (if one exists) and modifying the pointer in that root onode to point to the current root onode rather than to the root onode associated with the retained checkpoint to which the live file system is being reverted.

3. Deleting the root onode associated with the retained checkpoint.

4. Clearing the updated checkpoint objects list (i.e., after reverting the live file system to the version represented by the retained checkpoint, there are effectively no modified objects).

5. Deleting all objects created after the retained checkpoint, including root onodes and other objects associated with any retained checkpoints taken after the retained checkpoint to which the live file system is reverted.

Other than modifying the pointer in the root onode associated with the previous retained checkpoint, if one exists, any older retained checkpoints should remain unchanged. However, all newer retained checkpoints are effectively deleted.

### 4.6 Other Operations relating to Retained Checkpoints

All other functions normally associated with retained checkpoints can be achieved using the mechanisms described here. For instance, incremental backup can be performed by using the updated retained objects lists to work out what has changed between successive retained checkpoints.

What is claimed is:

1. A method for maintaining a file system object in a non-volatile storage device at successive checkpoints, the method comprising:

maintaining an object structure in a memory for the file system object, the object structure comprising a first tree structure rooted by a first root node and a second tree structure rooted by a second root node, each tree structure representing a version of the file system object; and

alternately managing changes to the object structure using the first tree structure rooted by the first root node while storing the second tree structure rooted by the second root node in the non-volatile storage device for a checkpoint that is used for keeping the consistency of data on the non-volatile storage device if the contents of the memory are lost and managing changes to the object structure using the second tree structure rooted by the second root node while storing the first tree structure rooted by the first root node in the non-volatile storage device for a subsequent checkpoint.

2. The method of claim 1, further comprising: maintaining a version number for each root node, the version number indicating the checkpoint associated with the corresponding tree structure.

3. The method of claim 1, wherein the non-volatile storage device comprises a plurality of sectors, and wherein the first and second root nodes are stored in adjacent sectors in the non-volatile storage.

4. The method of claim 2, further comprising: determining a latest valid version of the file system object based upon the version numbers of the root nodes.

5. The method of claims 1-4, wherein each tree structure includes a number of intermediate nodes and a number of data blocks.

6. The method of claim 5, further comprising:

maintaining a list of free space areas of the non-volatile storage device;

maintaining a list of free root nodes;

allocating the root nodes for the object structure from one of the list of free space areas and the list of free root nodes; and

allocating intermediate nodes and data blocks for the object structure only from the list of free space areas.

7. The method of claim 6, further comprising: deleting the file system object from the non-volatile storage device.

8. The method of claim 7, wherein deleting the file system object from the non-volatile storage device comprises:

adding the root nodes to the list of free root nodes; and

adding the intermediate nodes and data blocks to the list of free space areas.

9. An apparatus comprising:

a non-volatile storage device;

means for maintaining an object structure in a memory for a file system object, the object structure comprising a first tree structure rooted by a first root node and a second tree structure rooted by a second root node, each tree structure representing a version of a file system object; and

means for alternately managing changes to the file system object using the first tree structure rooted by the first root node while storing the second tree structure rooted by the second root node in the non-volatile storage device for a checkpoint that is used for keeping the consistency of data on the non-volatile storage device if the contents of the memory are lost and managing changes to the object structure using the second tree structure rooted by the second root node while storing the first tree structure rooted by the first root node in the non-volatile storage device for a subsequent checkpoint.

10. The apparatus of claim 9, further comprising: means for retaining read-only versions of the object using the object structure.

11. The apparatus of claim 10, further comprising: means for deleting a retained read-only version of the object from the object structure.

12. The apparatus of claim 10, further comprising: means for deleting the current version of the object while at least one retained read-only version of the object exists in the object structure.

13. The apparatus of claim 10, further comprising: means for reverting the current version of the object to a retained read-only version of the object.

14. The apparatus of claims 9-13, wherein each tree structure includes a number of intermediate nodes and a number of data blocks.

15. Apparatus comprising:

a non-volatile storage device; and

a storage processor configured to maintain an object structure in a memory for a file system object, the object structure comprising a first tree structure rooted by a first root node and a second tree structure rooted by a second root node, each tree structure representing a version of the file system object, the storage processor further configured to alternately manage changes to the file system object using the first tree structure rooted by the first root node while storing the second tree structure rooted by the second root node in the non-volatile storage device for a checkpoint that is used for keeping the consistency of data on the non-volatile storage device if the contents of the memory are lost and manage changes to the object structure using the second tree structure rooted by the second root node while storing the first tree structure rooted by the first root node in the non-volatile storage device for a subsequent checkpoint.

16. The apparatus of claim 15, wherein the storage processor is hardware-implemented.

17. The apparatus of claim 15, wherein the storage processor is hardware-accelerated.

18. The apparatus of claim 15, wherein the storage processor includes a plurality of linked sub-modules including an object store sub-module configured to perform at least one maintaining the object structure and alternately managing the object using the first and second tree structures.

19. The apparatus of claim 18, wherein the object store sub-module is hardware-implemented.

20. The apparatus of claim 18, wherein the object store sub-module is hardware-accelerated.

21. The apparatus of claims 15-20, wherein each tree structure includes a number of intermediate nodes and a number of data blocks.

22. An apparatus comprising:

a first storage; and

a file server, coupled to the first storage, and comprising a second storage, and being operable to manage plurality of root nodes for an object of a file system, and being operable to manage an update for data of a first portion of the plurality of root nodes in a status in which (i) data of a second portion of the plurality of root nodes is written from the second storage of the file server to the first storage and (ii) a new checkpoint, which is used when a version of the object of file system is recovered, is taken.

23. The apparatus according to the claim 22, wherein: the new checkpoint is used when the second portion of the plurality of root nodes of the file system is recovered.

24. The apparatus according to the claim 22, wherein:

the file server is operable to manage an update for the object of the file system in another status in which (iii) data of the first portion of the plurality of root nodes is written from the second storage of the file server to the first storage and (iv) a next new checkpoint is taken.

25. The apparatus according to the claim 22, wherein:

the file server is operable to manage to change a status of the file system from the status to the another status.

26. The apparatus according to the claim 22, wherein:

the file server is operable to manage to take the new checkpoint when the amount of data in the second storage of the file server reach a threshold.

27. The apparatus according to the claim 22, wherein:

a checkpoint number of the new checkpoint identifies the version of the object of the file system.

28. The apparatus according to the claim 22, wherein:

the file server is operable to manage to recover a version of the object of the file system based on a last checkpoint managed by the file server.

29. The apparatus according to the claim 22, wherein:

the first storage is a disk, or

the second storage is a non-volatile storage.

30. The apparatus according to the claim 22, wherein:

the first storage is a disk, and
the second storage is a NVRAM.

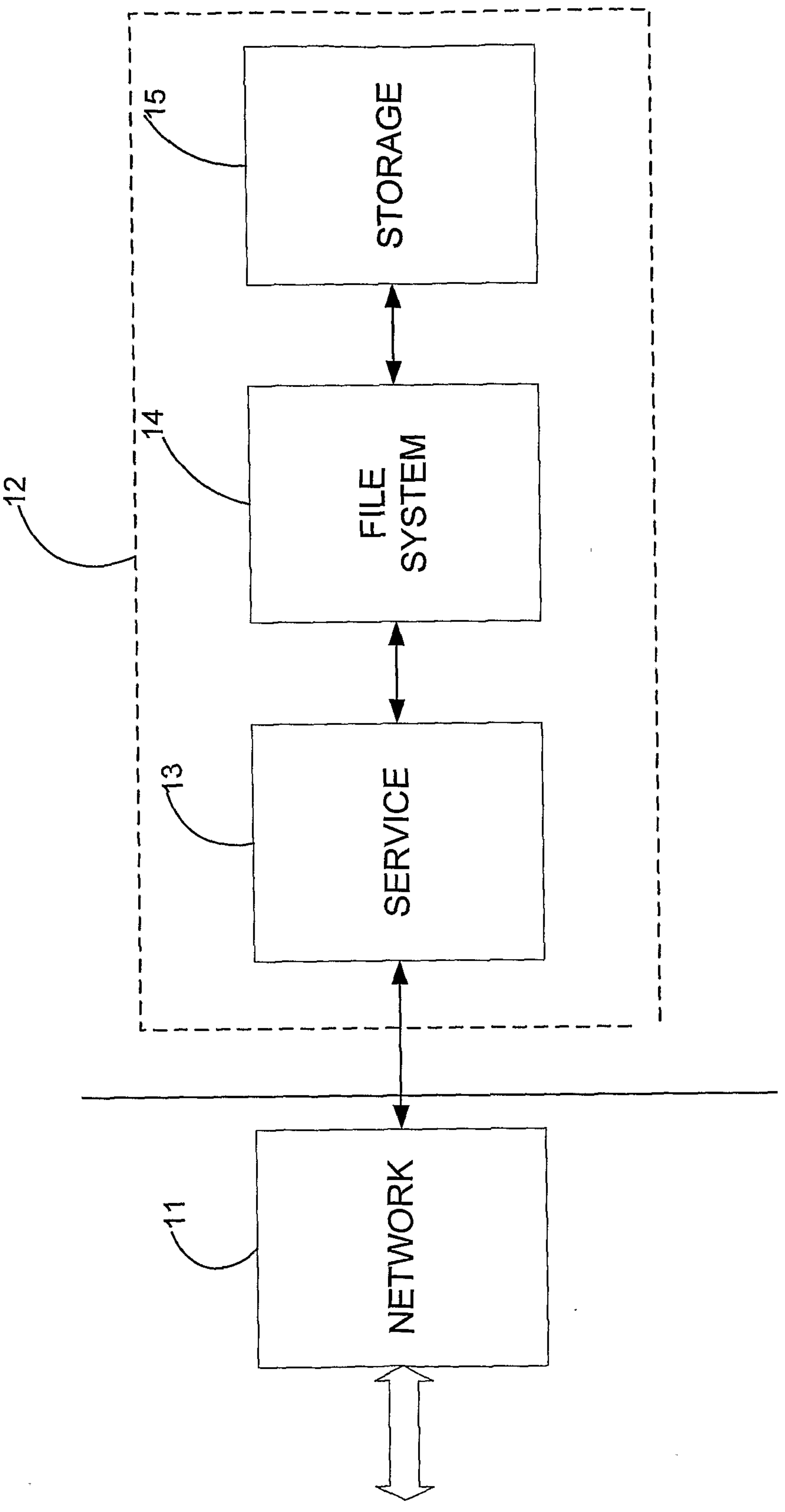


Fig. 1

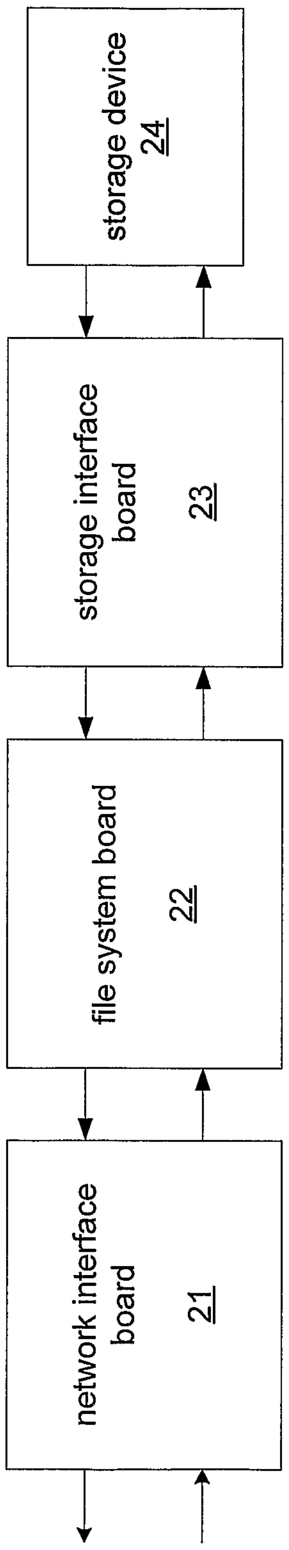

storage device 24
storage interface board 23
file system board 22
network interface board 21

Fig. 2

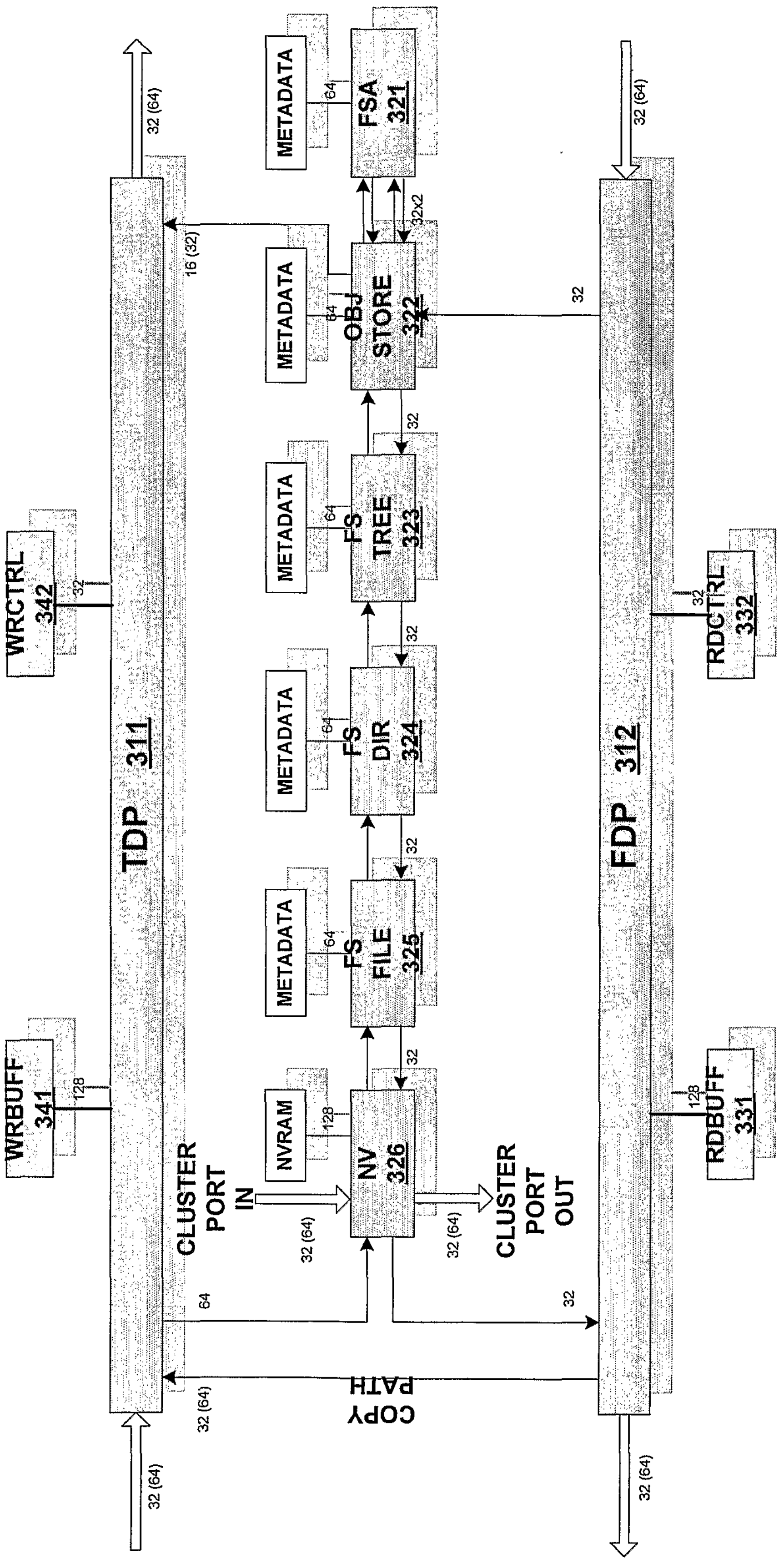

TDP 311
FDP 312
WRBUFF 341
WRCTRL 342
RDBUFF 331
RDCTRL 332
FSA 321
OBJ STORE 322
FS TREE 323
FS DIR 324
FS FILE 325
NV 326
NVRAM
METADATA
CLUSTER PORT IN
CLUSTER PORT OUT
COPY PATH
32 (64)
16 (32)
32x2
32
64
128

Fig. 3

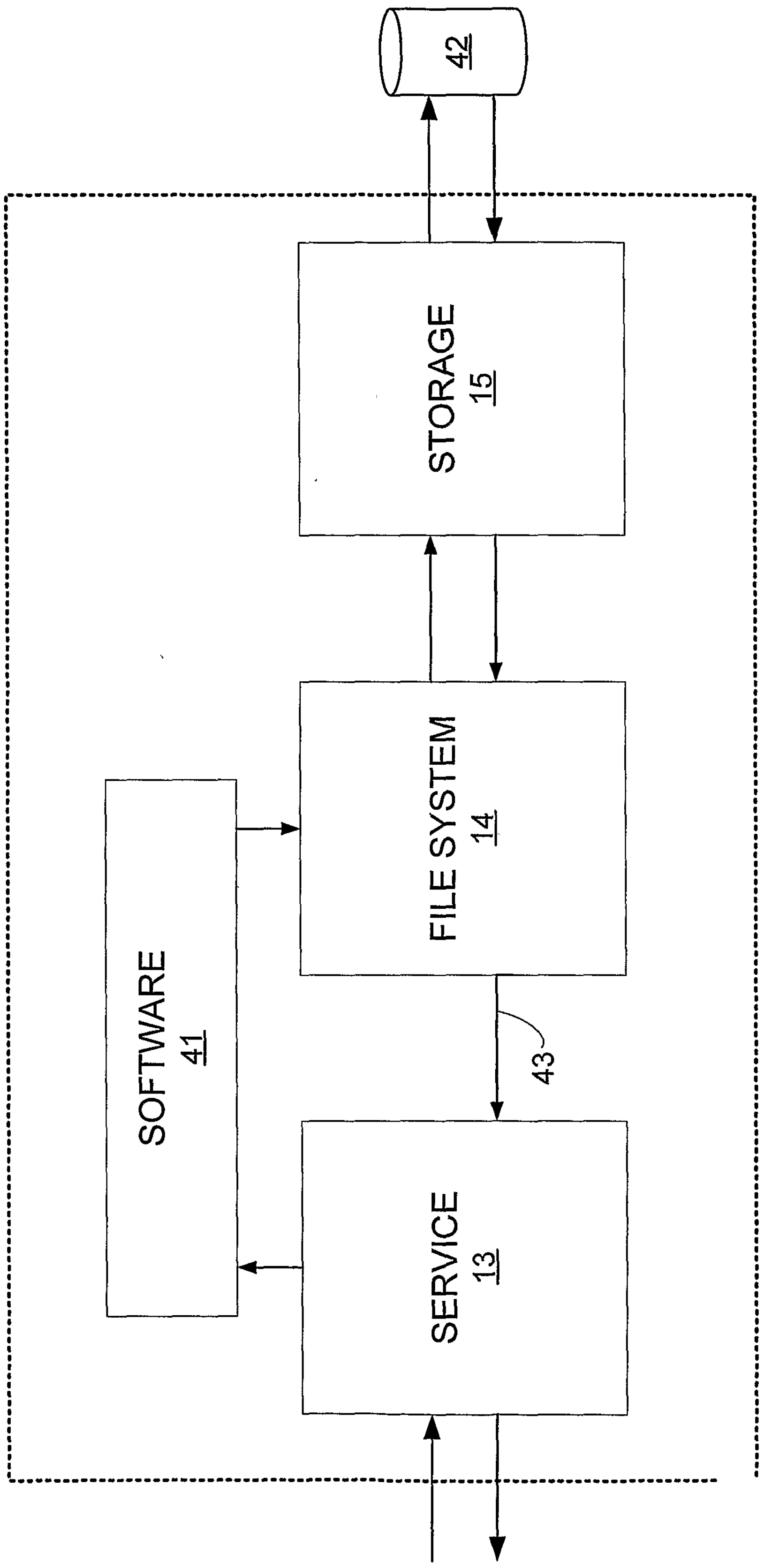


Fig. 4

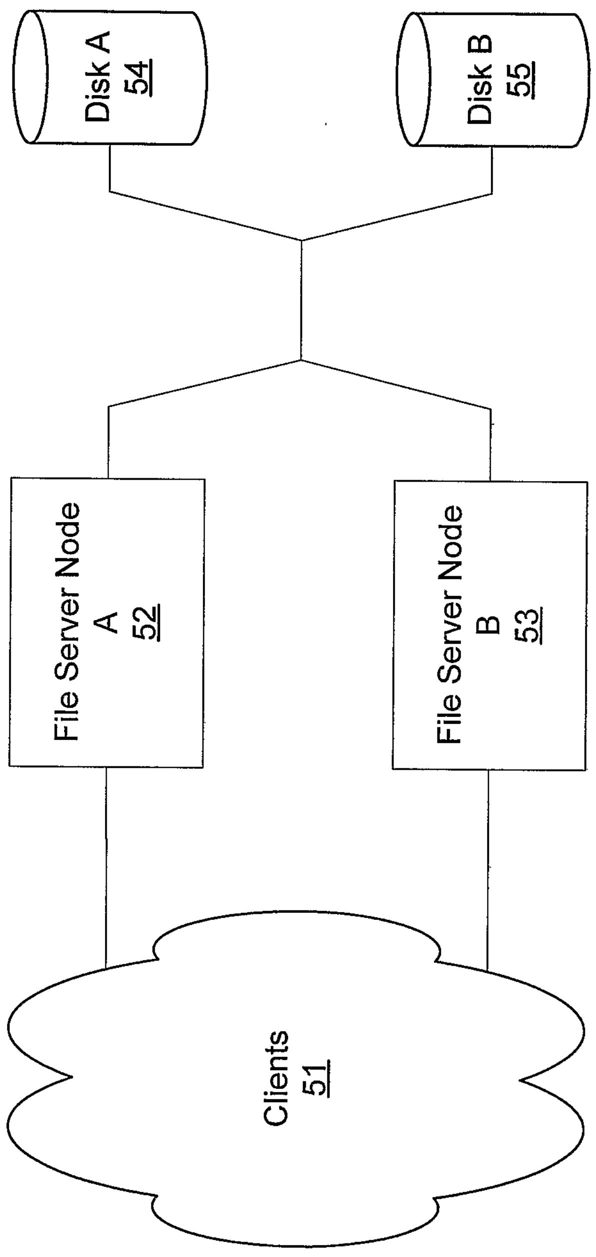

Disk A
54
Disk B
55
File Server Node
A
52
File Server Node
B
53
Clients
51

Fig. 5

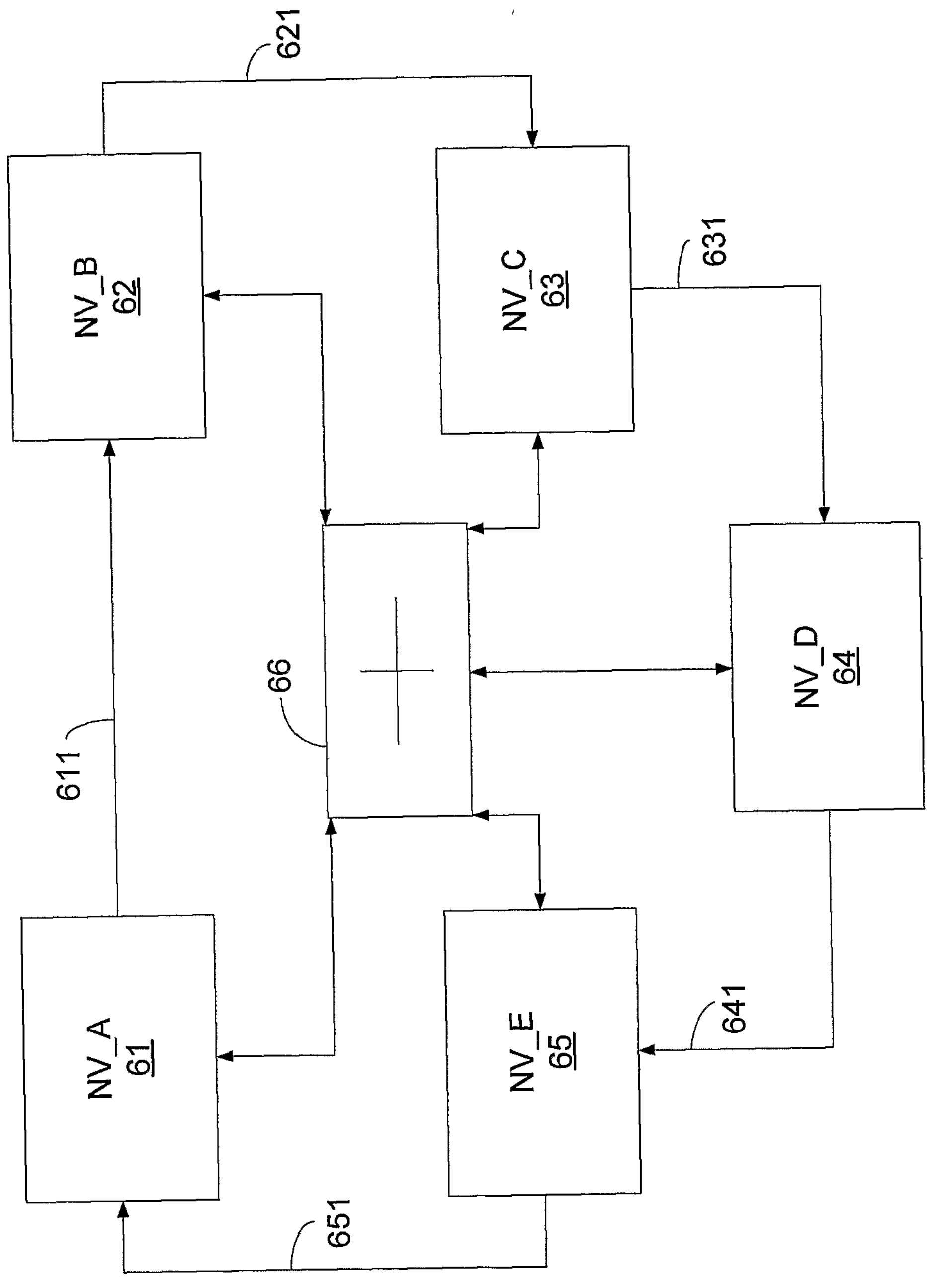


Fig. 6

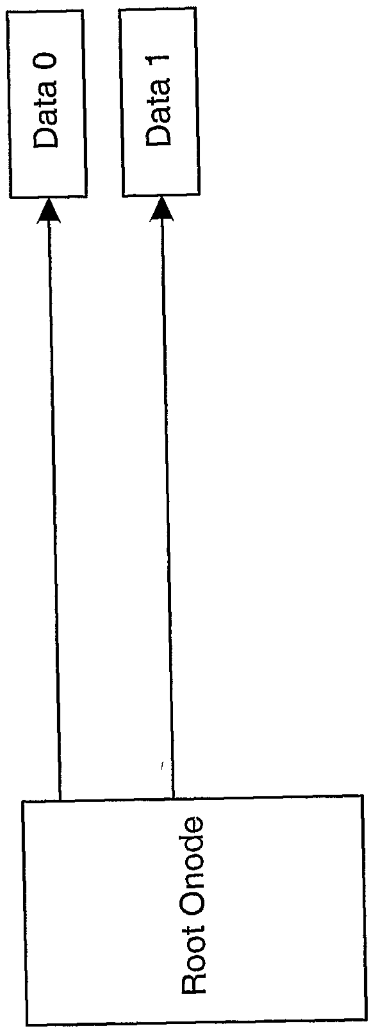


Fig. 7

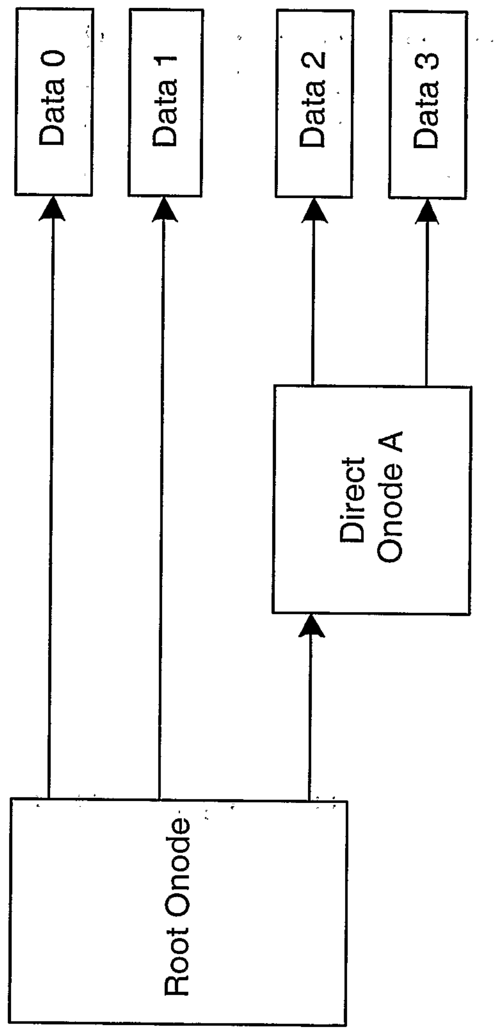


Fig. 8

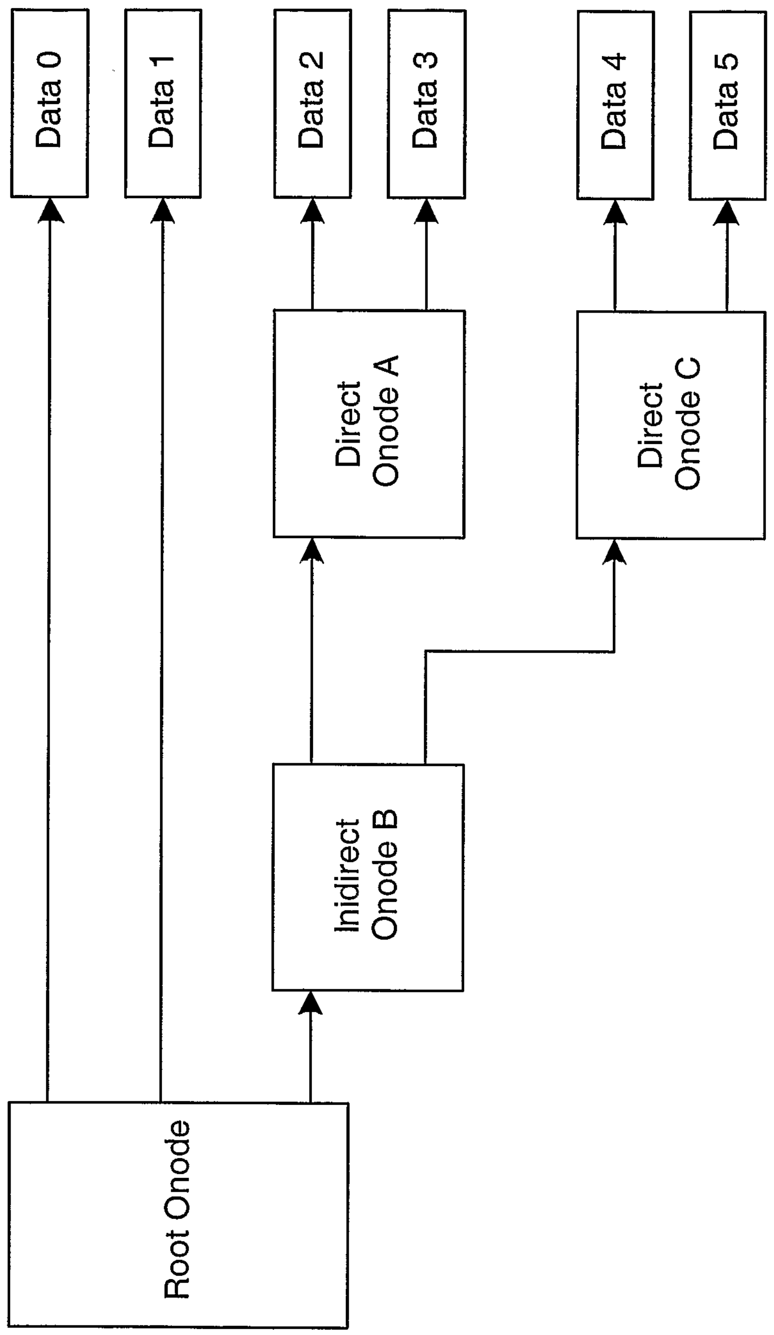

Root Onode
Inidirect Onode B
Direct Onode A
Direct Onode C
Data 0
Data 1
Data 2
Data 3
Data 4
Data 5

Fig. 9

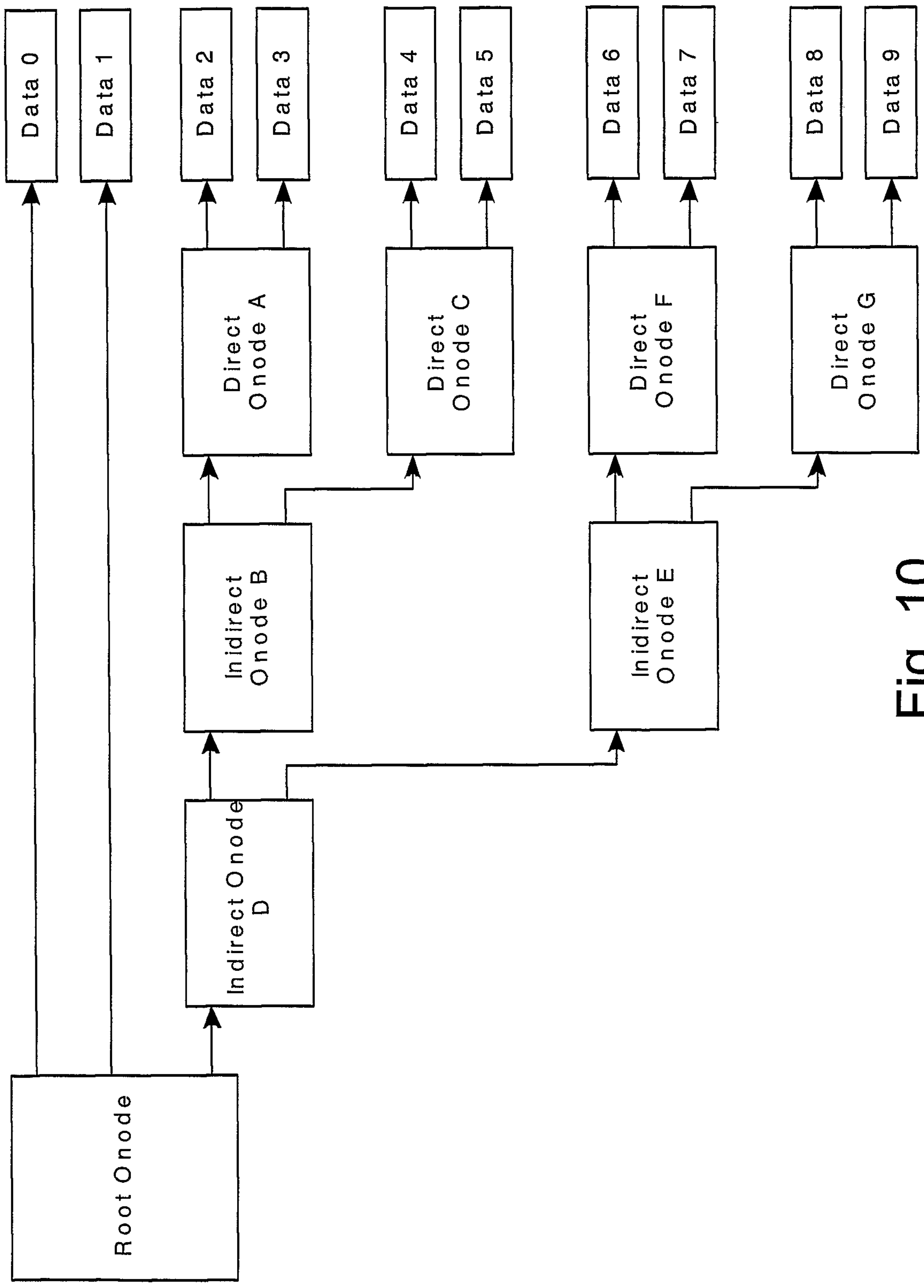

Root Onode
Indirect Onode D
Inidirect Onode B
Inidirect Onode E
Direct Onode A
Direct Onode C
Direct Onode F
Direct Onode G
Data 0
Data 1
Data 2
Data 3
Data 4
Data 5
Data 6
Data 7
Data 8
Data 9

Fig. 10

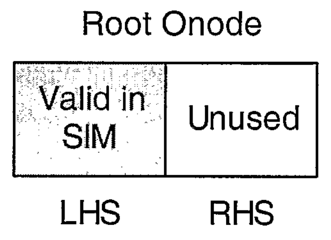


Fig. 11

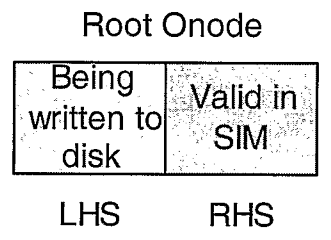


Fig. 12

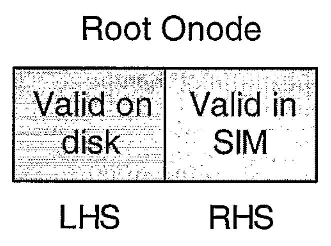


Fig. 13

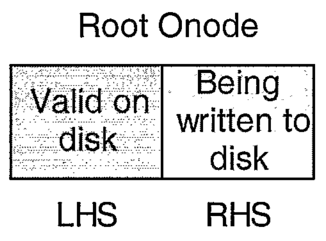


Fig. 14

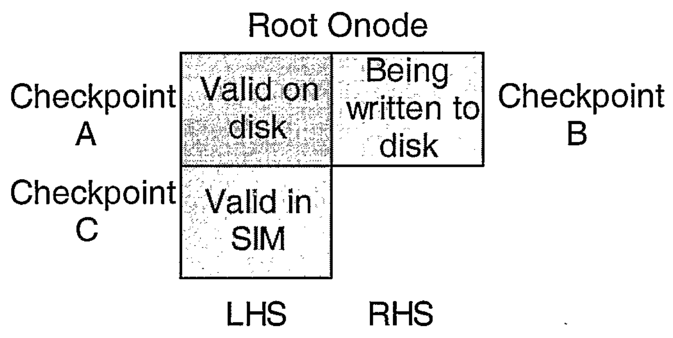

Root Onode
Checkpoint A
Valid on disk
Being written to disk
Checkpoint B
Checkpoint C
Valid in SIM
LHS
RHS

Fig. 15

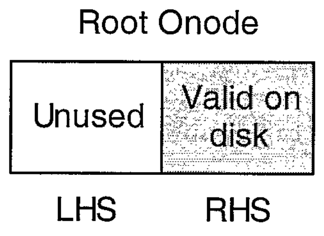


Fig. 16

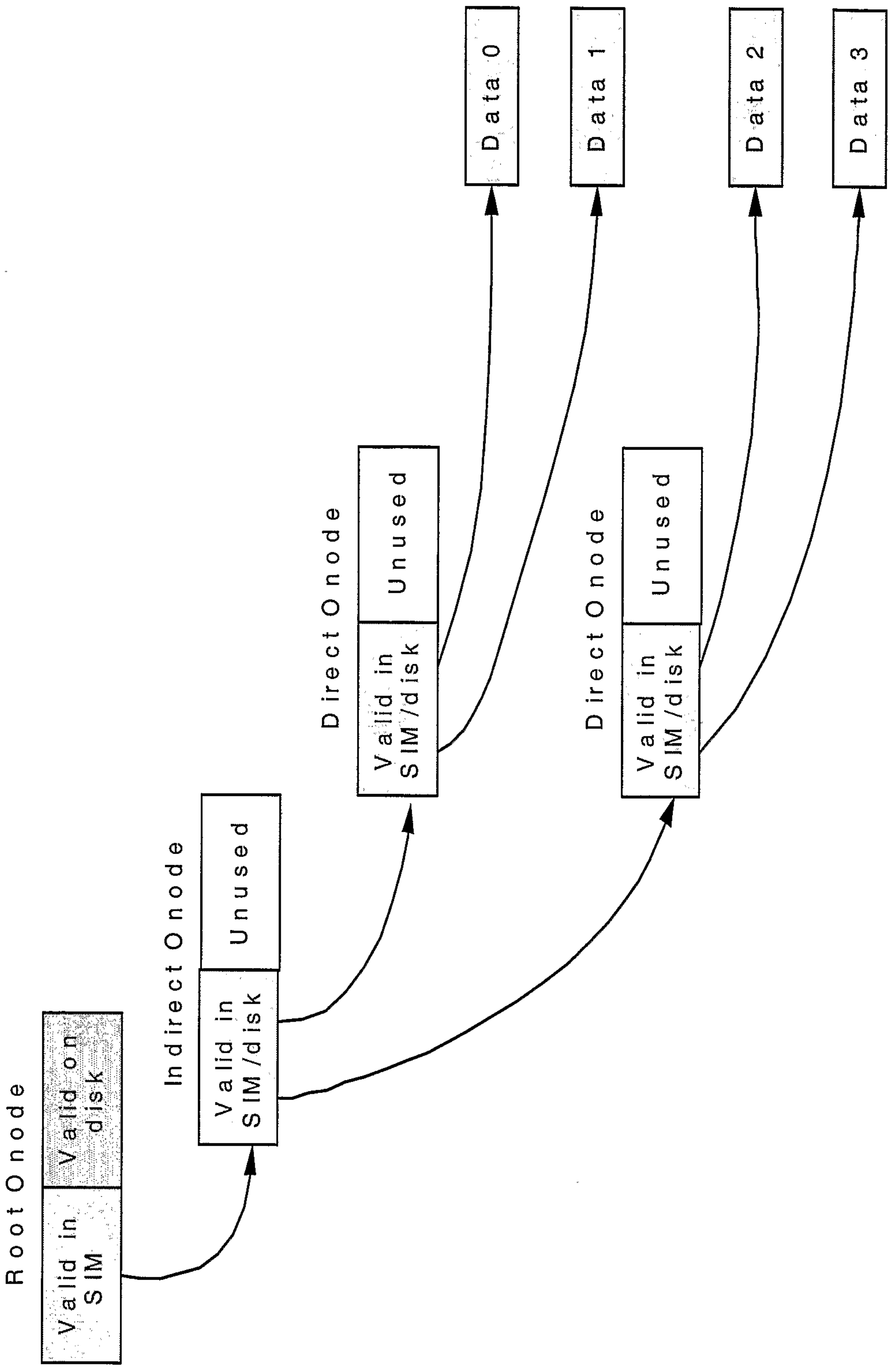

Root Onode
Valid in SIM
Valid on disk
Indirect Onode
Valid in SIM/disk
Unused
Direct Onode
Valid in SIM/disk
Unused
Direct Onode
Valid in SIM/disk
Unused
Data 0
Data 1
Data 2
Data 3

Fig. 17

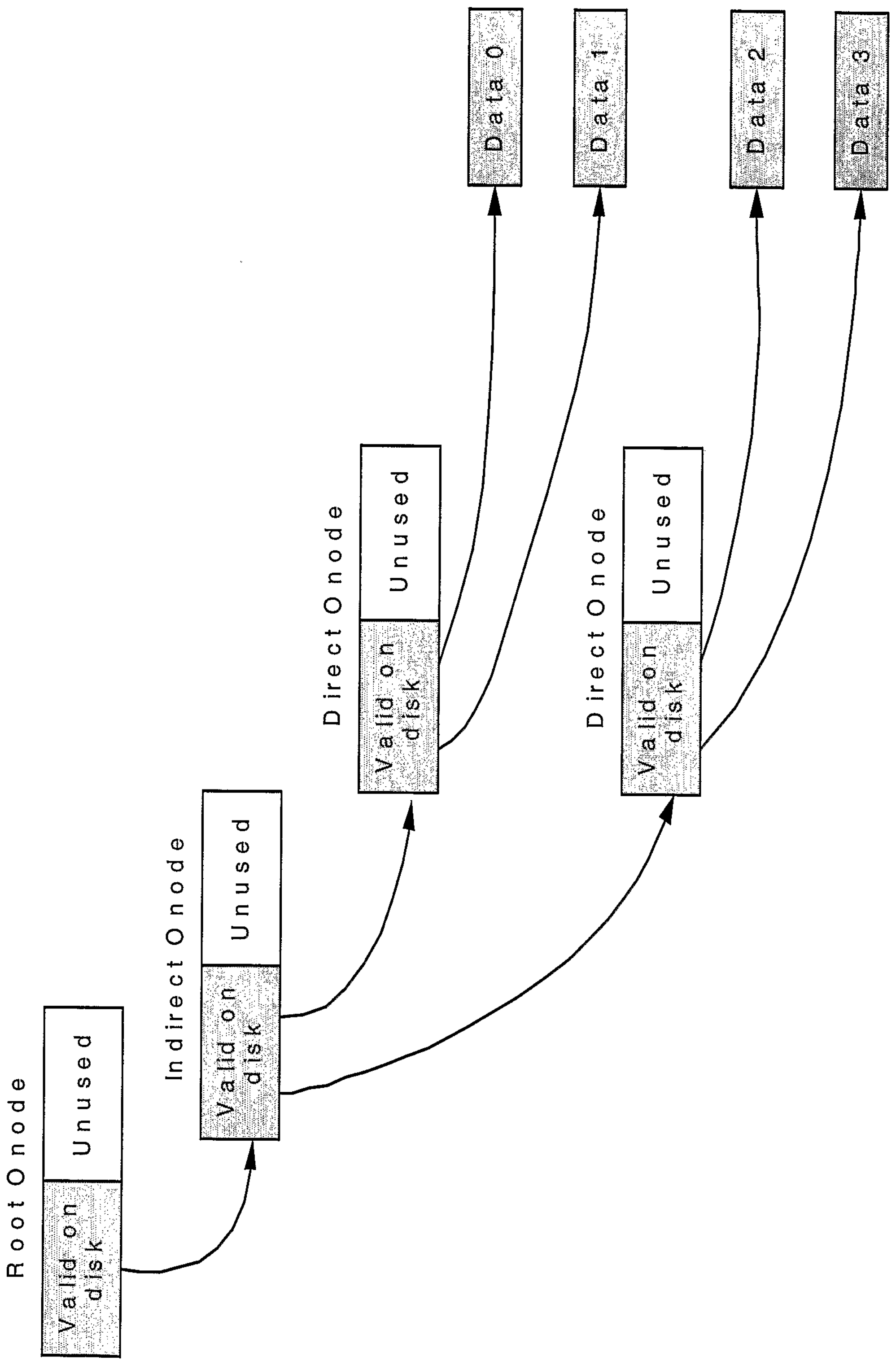

Root Onode
Valid on disk
Unused
Indirect Onode
Valid on disk
Unused
Direct Onode
Valid on disk
Unused
Direct Onode
Valid on disk
Unused
Data 0
Data 1
Data 2
Data 3

Fig. 18

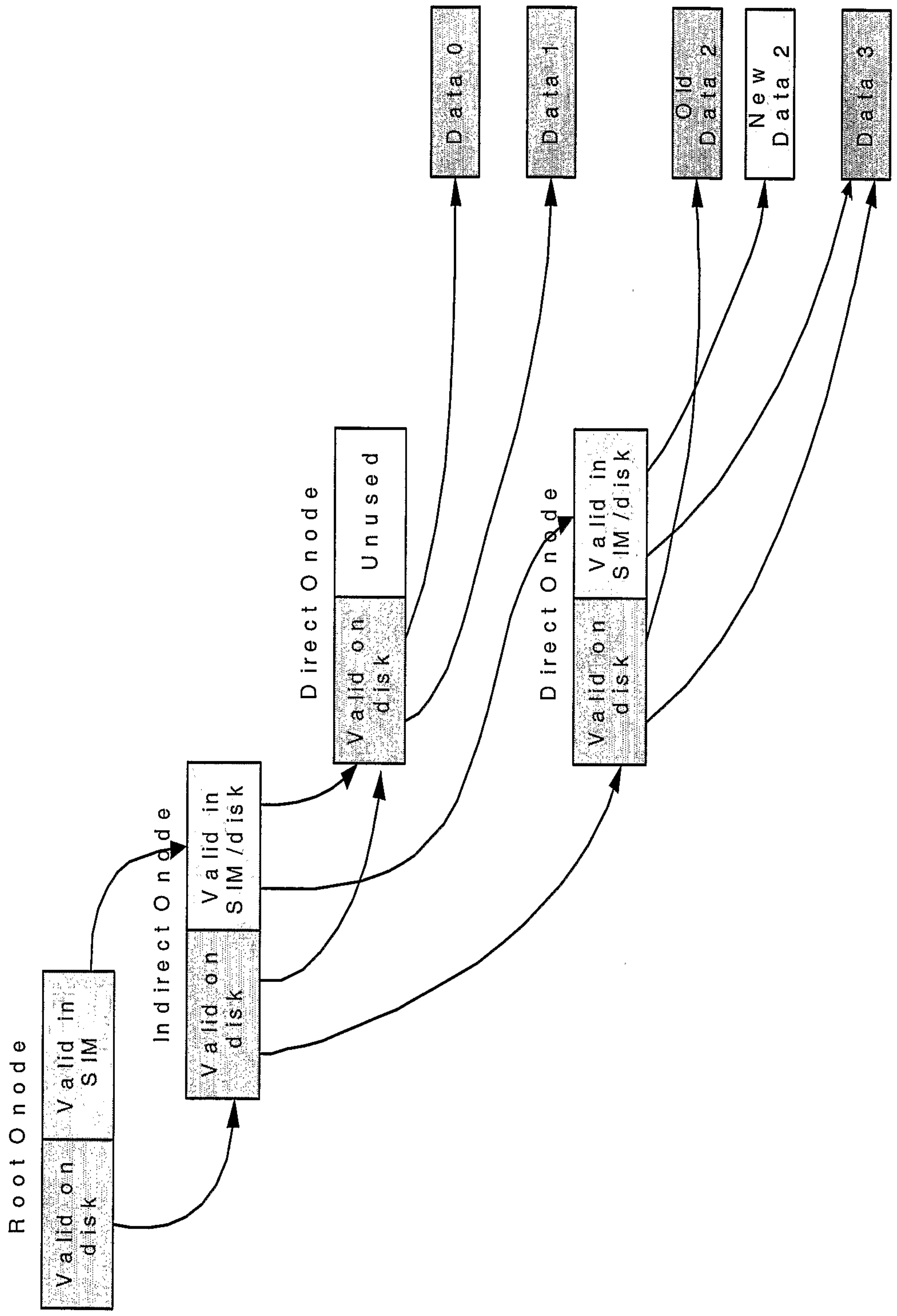

Root Onode
Valid on disk
Valid in SIM
Indirect Onode
Valid on disk
Valid in SIM/disk
Direct Onode
Valid on disk
Unused
Direct Onode
Valid on disk
Valid in SIM/disk
Data 0
Data 1
Old Data 2
New Data 2
Data 3

Fig. 19

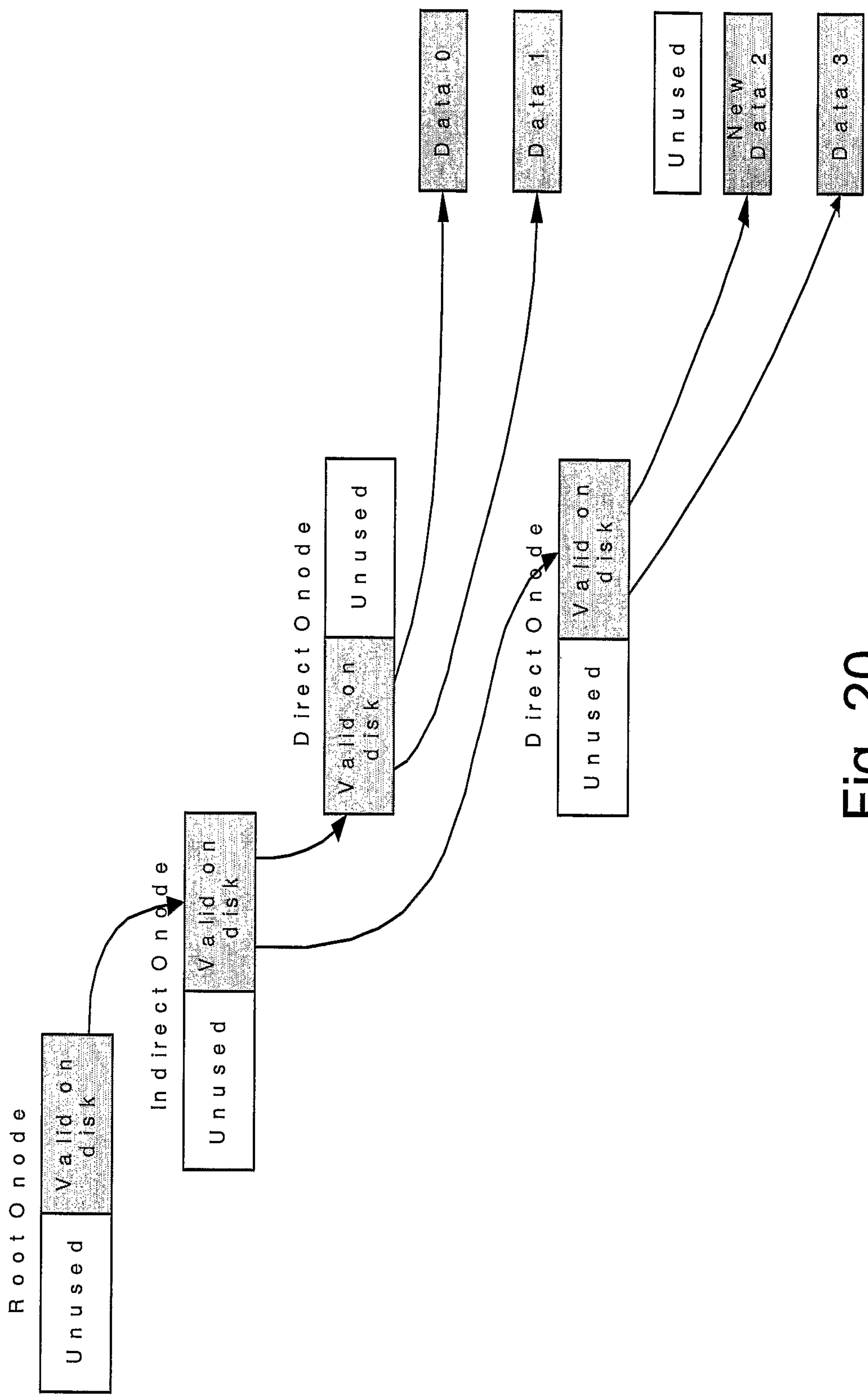

Root Onode
Unused
Valid on disk
Indirect Onode
Unused
Valid on disk
Direct Onode
Valid on disk
Unused
Direct Onode
Unused
Valid on disk
Data 0
Data 1
Unused
New Data 2
Data 3

Fig. 20

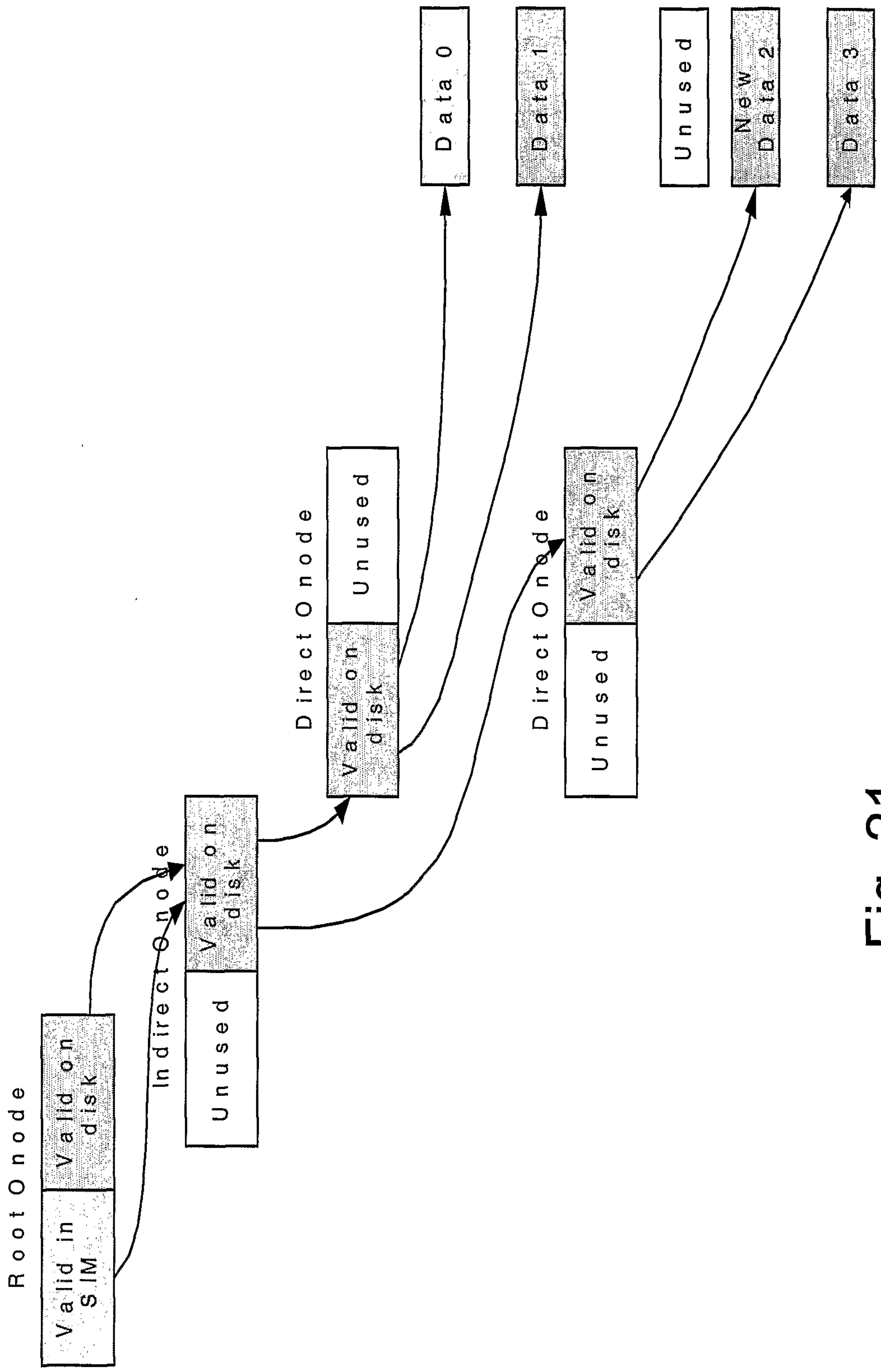
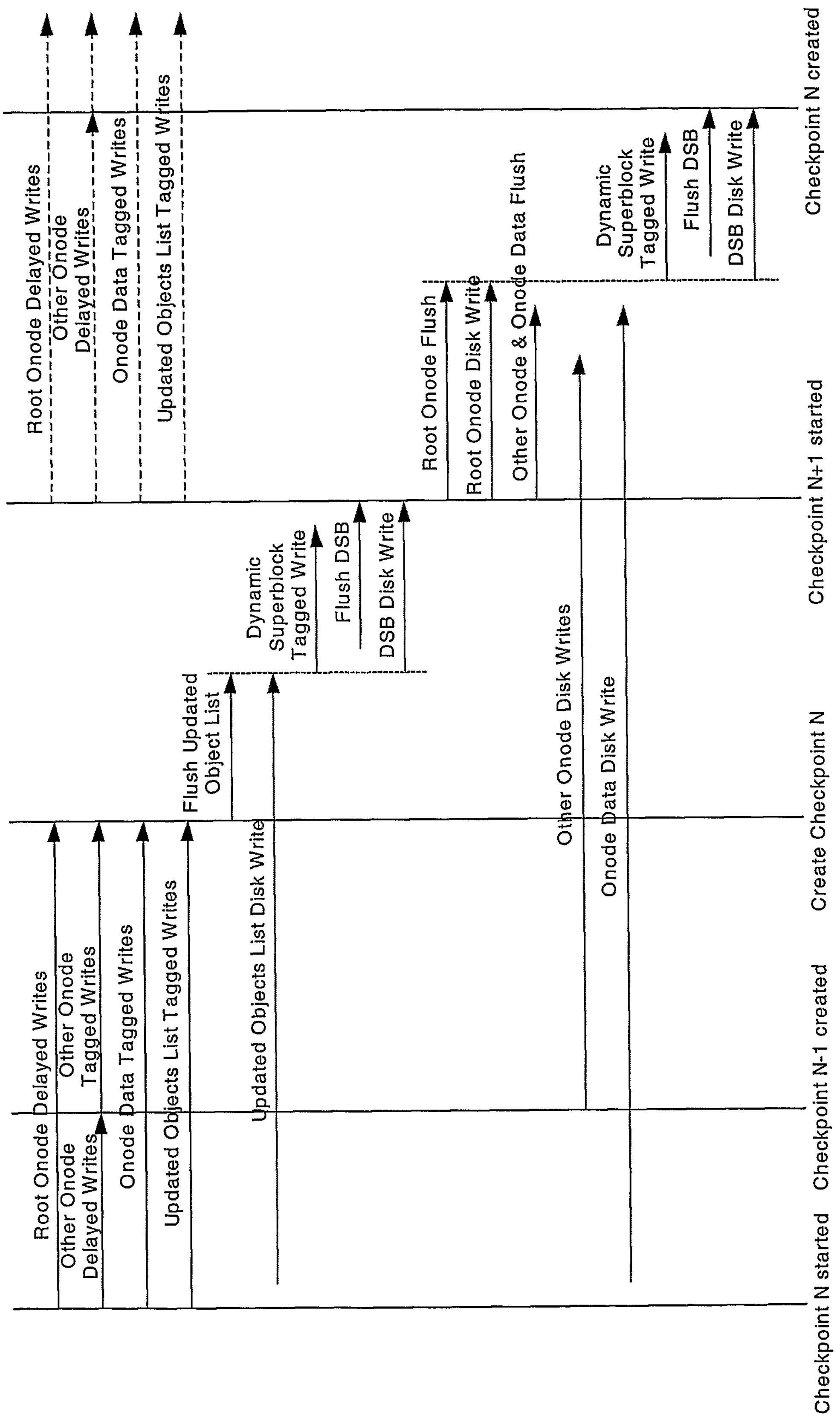

Root Onode
Valid in SIM
Valid on disk
Indirect Onode
Unused
Valid on disk
Direct Onode
Valid on disk
Unused
Direct Onode
Unused
Valid on disk
Data 0
Data 1
Unused
New Data 2
Data 3

Fig. 21

TimeLine for the Creation of Checkpoint N



Root Onode Delayed Writes
Other Onode Delayed Writes
Other Onode Tagged Writes
Onode Data Tagged Writes
Updated Objects List Tagged Writes
Updated Objects List Disk Write
Flush Updated Object List
Dynamic Superblock Tagged Write
Flush DSB
DSB Disk Write
Root Onode Delayed Writes
Other Onode Delayed Writes
Onode Data Tagged Writes
Updated Objects List Tagged Writes
Root Onode Flush
Root Onode Disk Write
Other Onode & Onode Data Flush
Other Onode Disk Writes
Onode Data Disk Write
Dynamic Superblock Tagged Write
Flush DSB
DSB Disk Write
Checkpoint N started
Checkpoint N-1 created
Create Checkpoint N
Checkpoint N+1 started
Checkpoint N created

Fig. 22

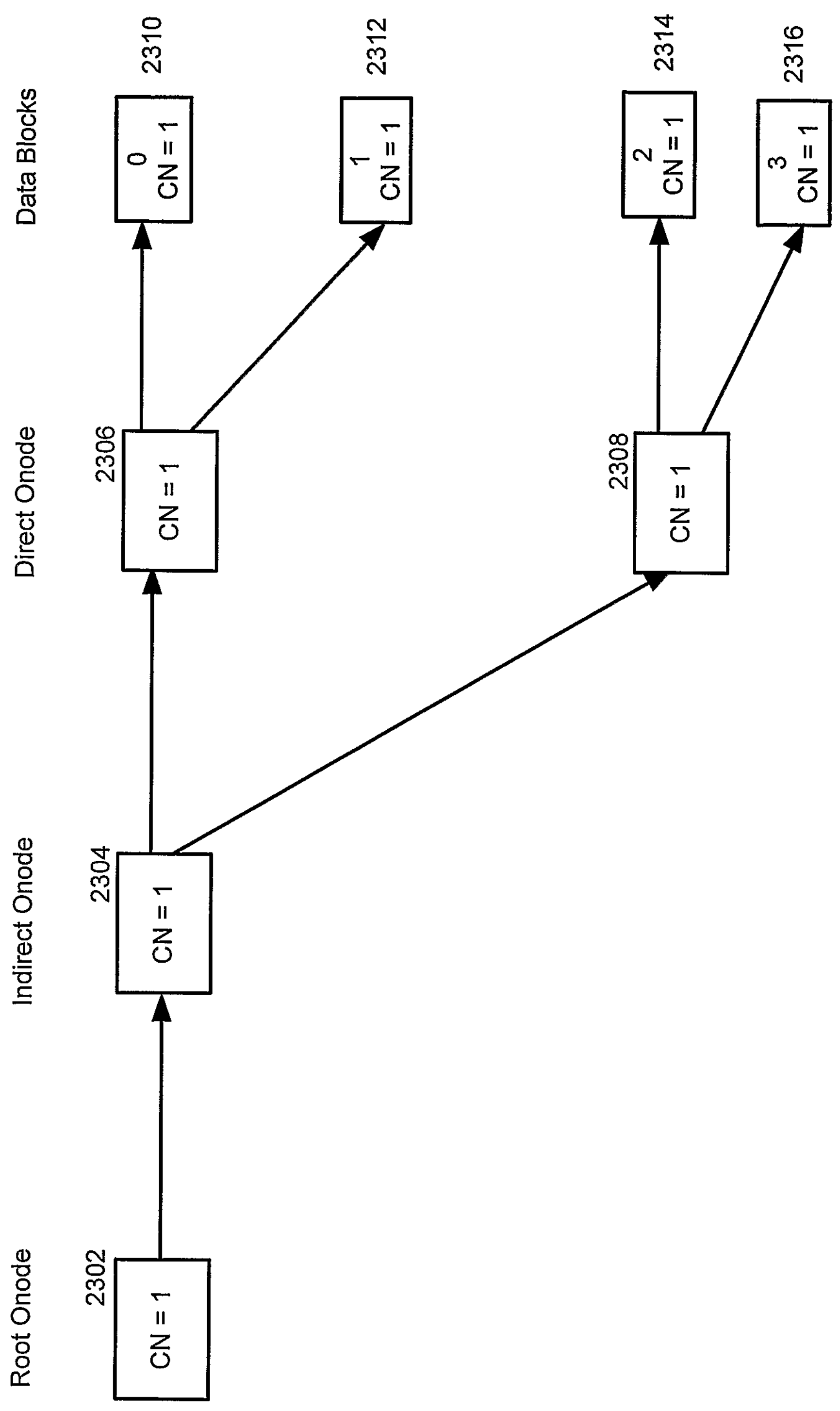

Root Onode
Indirect Onode
Direct Onode
Data Blocks
2302
CN = 1
2304
CN = 1
2306
CN = 1
2308
CN = 1
2310
0
CN = 1
2312
1
CN = 1
2314
2
CN = 1
2316
3
CN = 1

FIG. 23

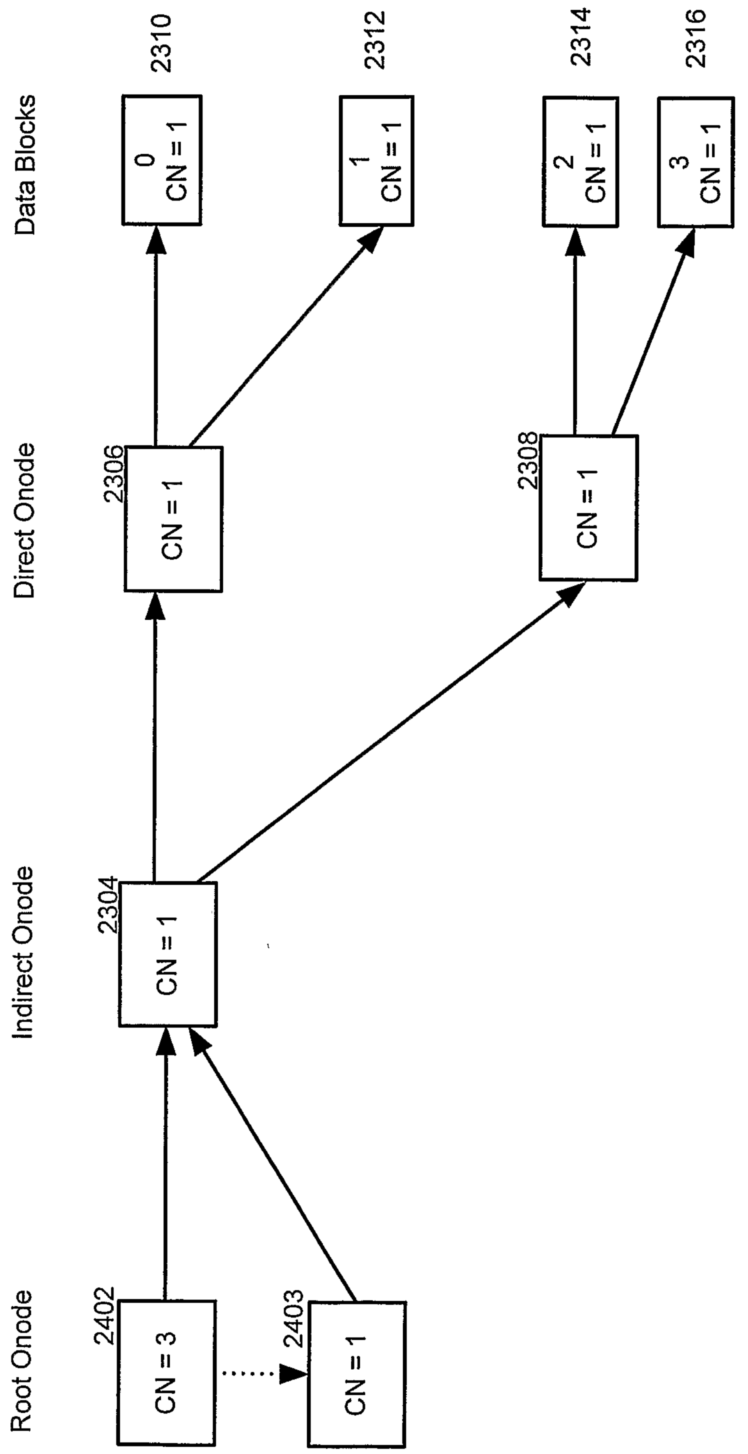

Root Onode
Indirect Onode
Direct Onode
Data Blocks
2402
CN = 3
2403
CN = 1
2304
CN = 1
2306
CN = 1
2308
CN = 1
2310
0
CN = 1
2312
1
CN = 1
2314
2
CN = 1
2316
3
CN = 1

FIG. 24

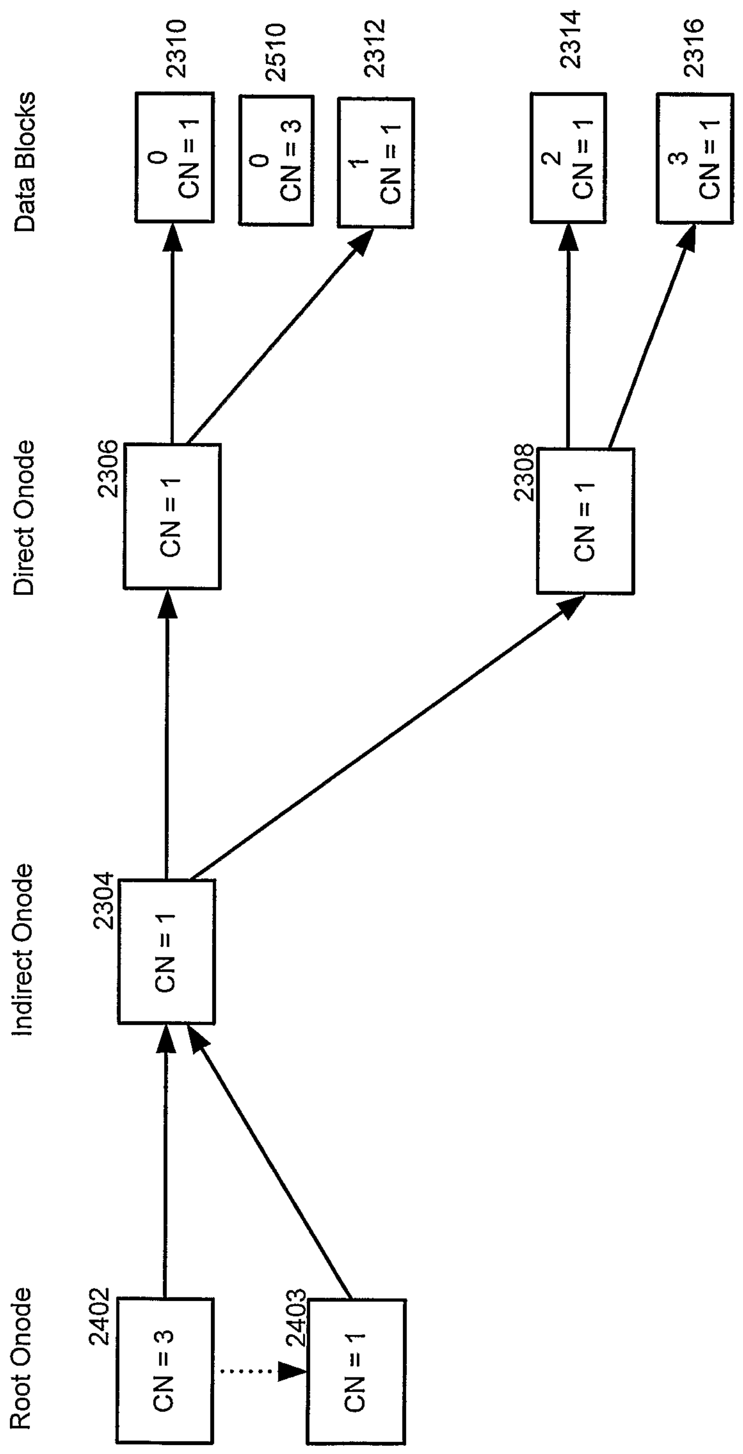

Root Onode
Indirect Onode
Direct Onode
Data Blocks
2402
CN = 3
2403
CN = 1
2304
CN = 1
2306
CN = 1
2308
CN = 1
2310
0
CN = 1
2510
0
CN = 3
2312
1
CN = 1
2314
2
CN = 1
2316
3
CN = 1

FIG. 25

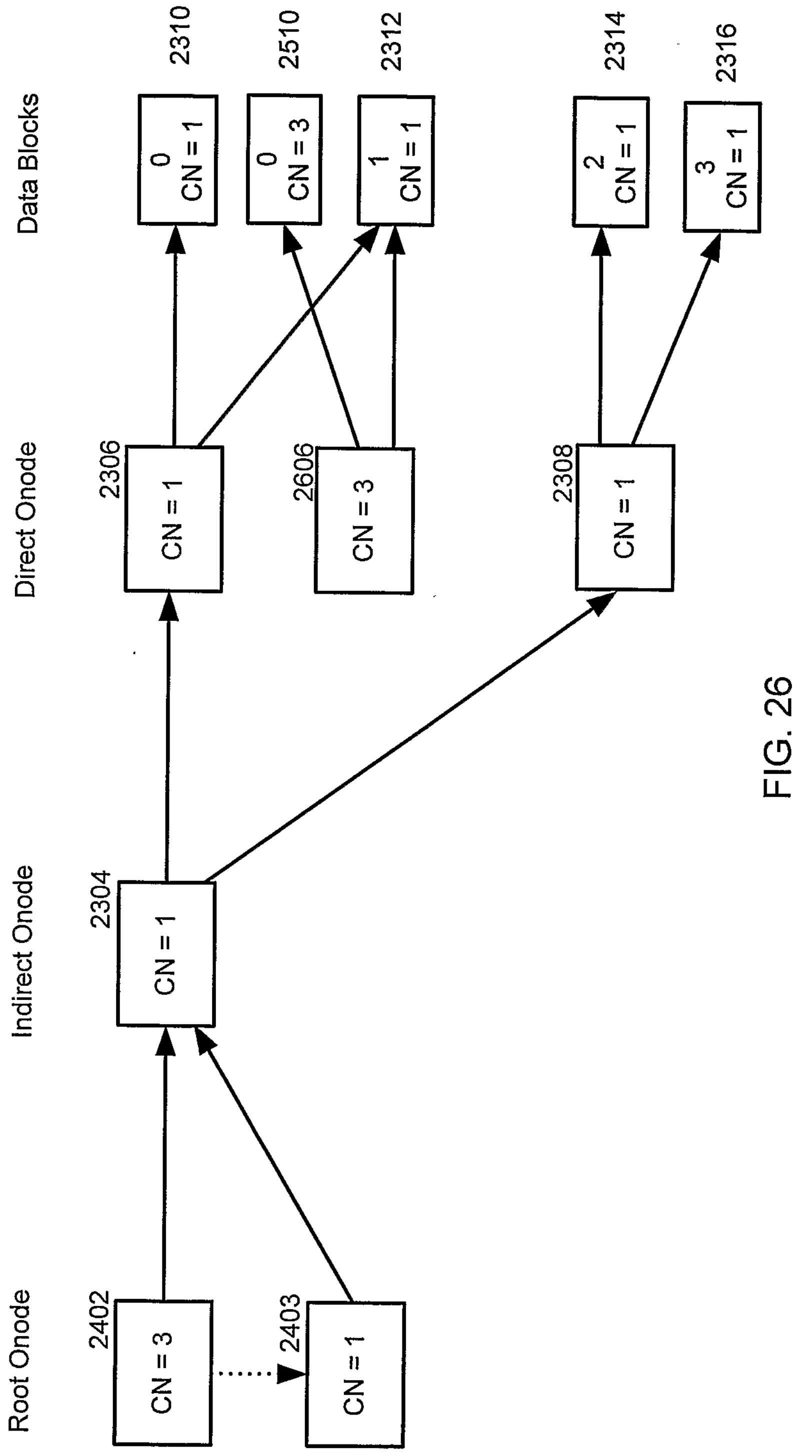

Root Onode
Indirect Onode
Direct Onode
Data Blocks
2402
CN = 3
2403
CN = 1
2304
CN = 1
2306
CN = 1
2606
CN = 3
2308
CN = 1
0
CN = 1
2310
0
CN = 3
2510
1
CN = 1
2312
2
CN = 1
2314
3
CN = 1
2316

FIG. 26

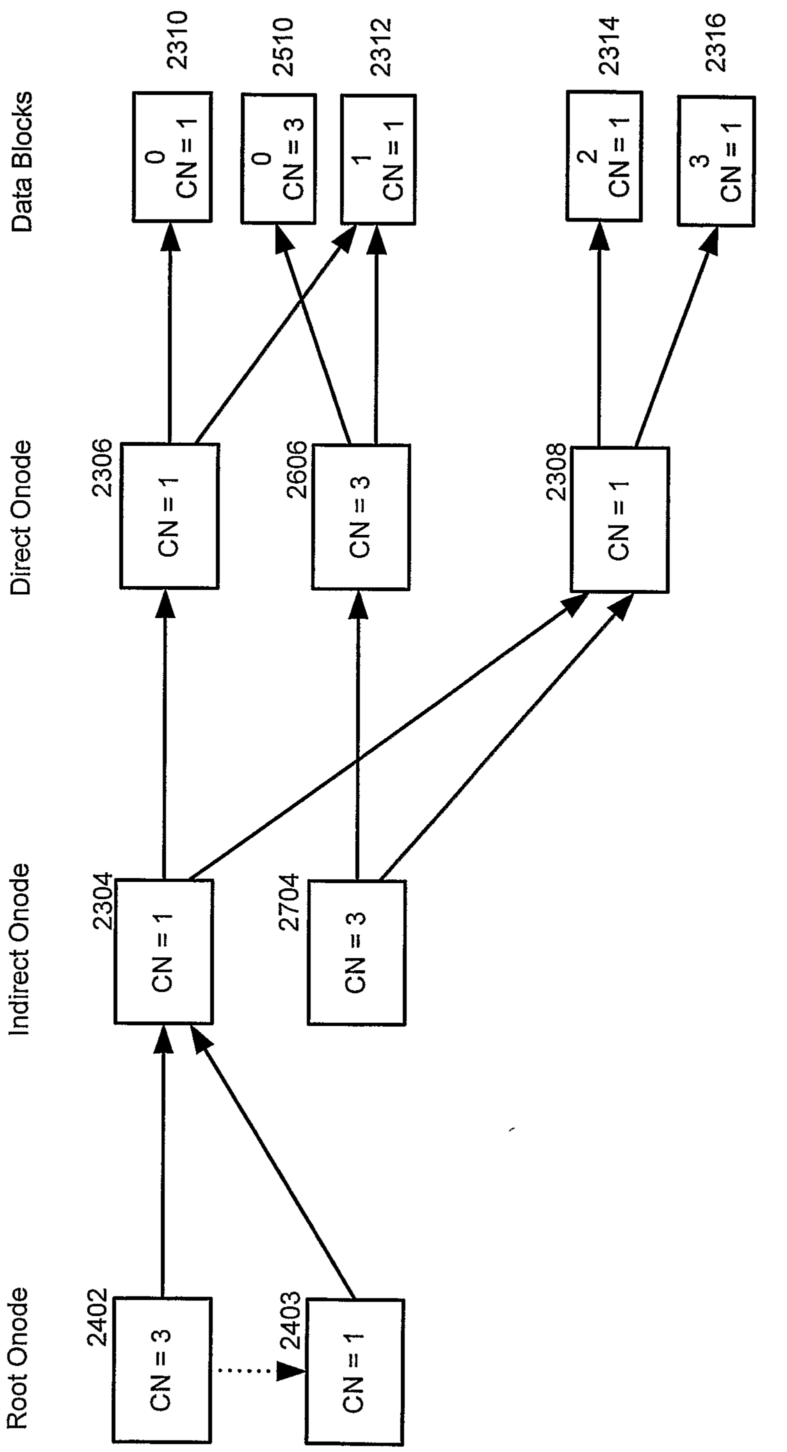

Root Onode
Indirect Onode
Direct Onode
Data Blocks
2402
CN = 3
2403
CN = 1
2304
CN = 1
2704
CN = 3
2306
CN = 1
2606
CN = 3
2308
CN = 1
2310
0
CN = 1
2510
0
CN = 3
2312
1
CN = 1
2314
2
CN = 1
2316
3
CN = 1

FIG. 27

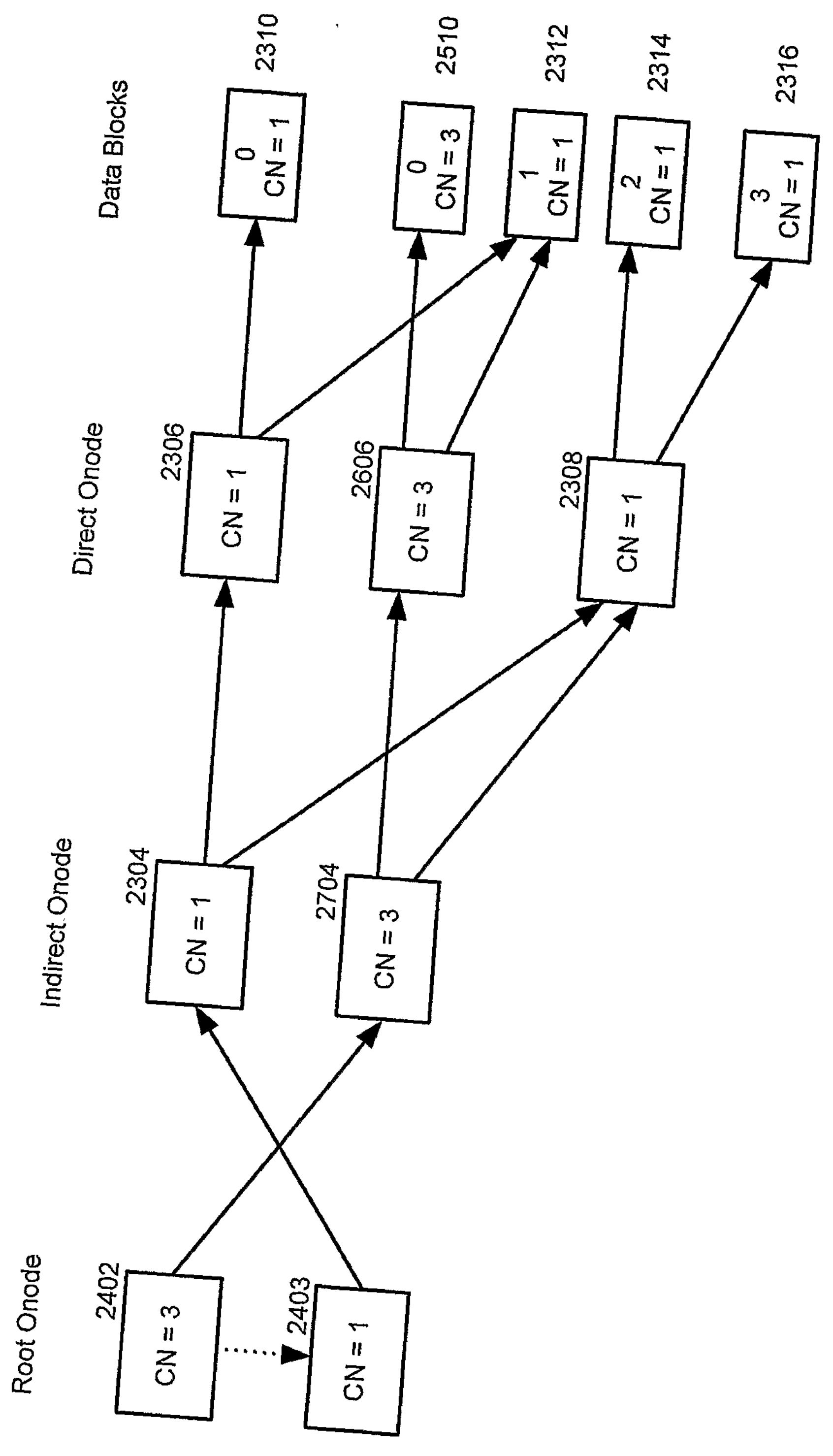
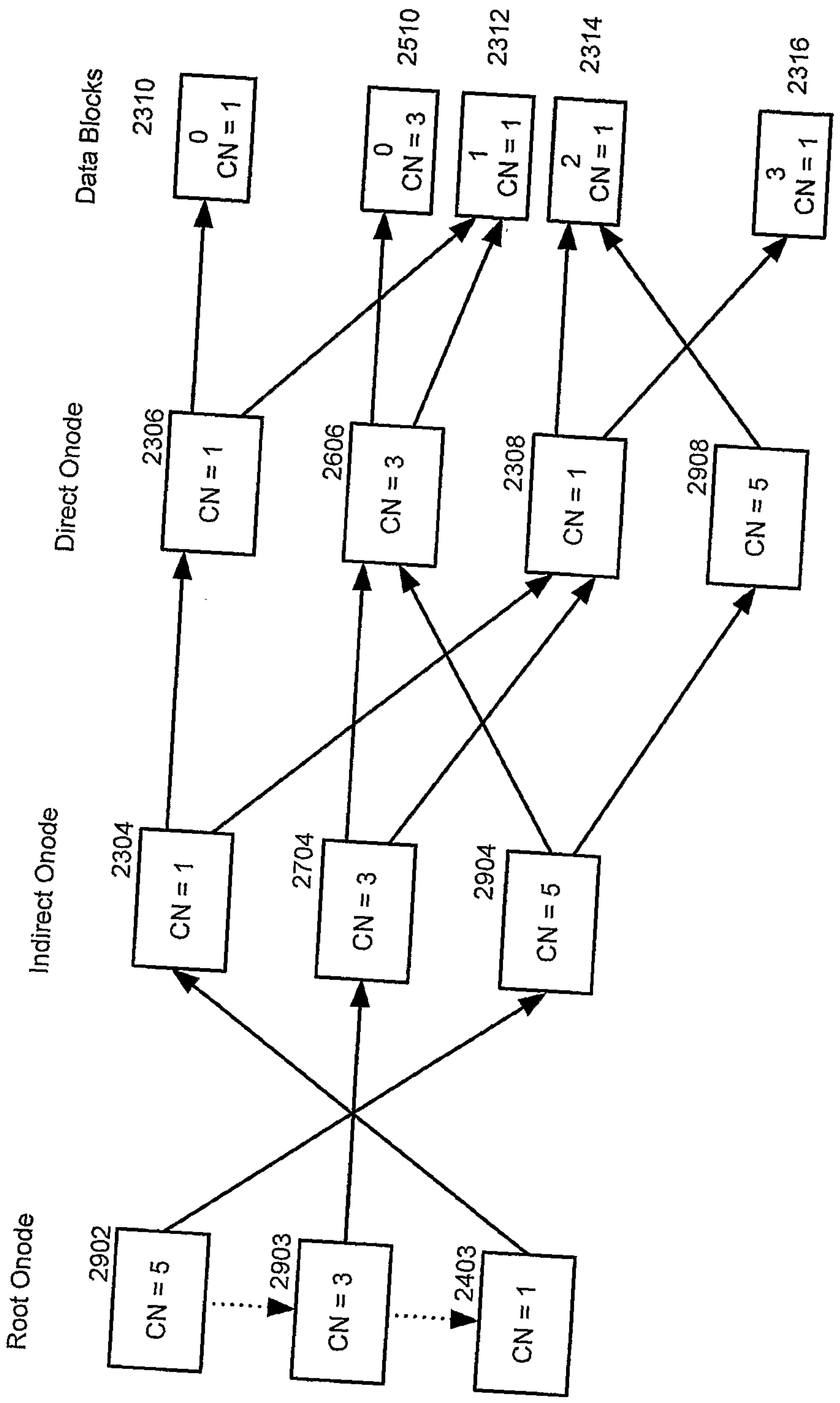

Root Onode
Indirect Onode
Direct Onode
Data Blocks
2402
CN = 3
2403
CN = 1
2304
CN = 1
2704
CN = 3
2306
CN = 1
2606
CN = 3
2308
CN = 1
2310
0
CN = 1
2510
0
CN = 3
2312
1
CN = 1
2314
2
CN = 1
2316
3
CN = 1

FIG. 28



Root Onode
Indirect Onode
Direct Onode
Data Blocks
2902
CN = 5
2903
CN = 3
2403
CN = 1
2304
CN = 1
2704
CN = 3
2904
CN = 5
2306
CN = 1
2606
CN = 3
2308
CN = 1
2908
CN = 5
2310
0
CN = 1
2510
0
CN = 3
2312
1
CN = 1
2314
2
CN = 1
2316
3
CN = 1

FIG. 29

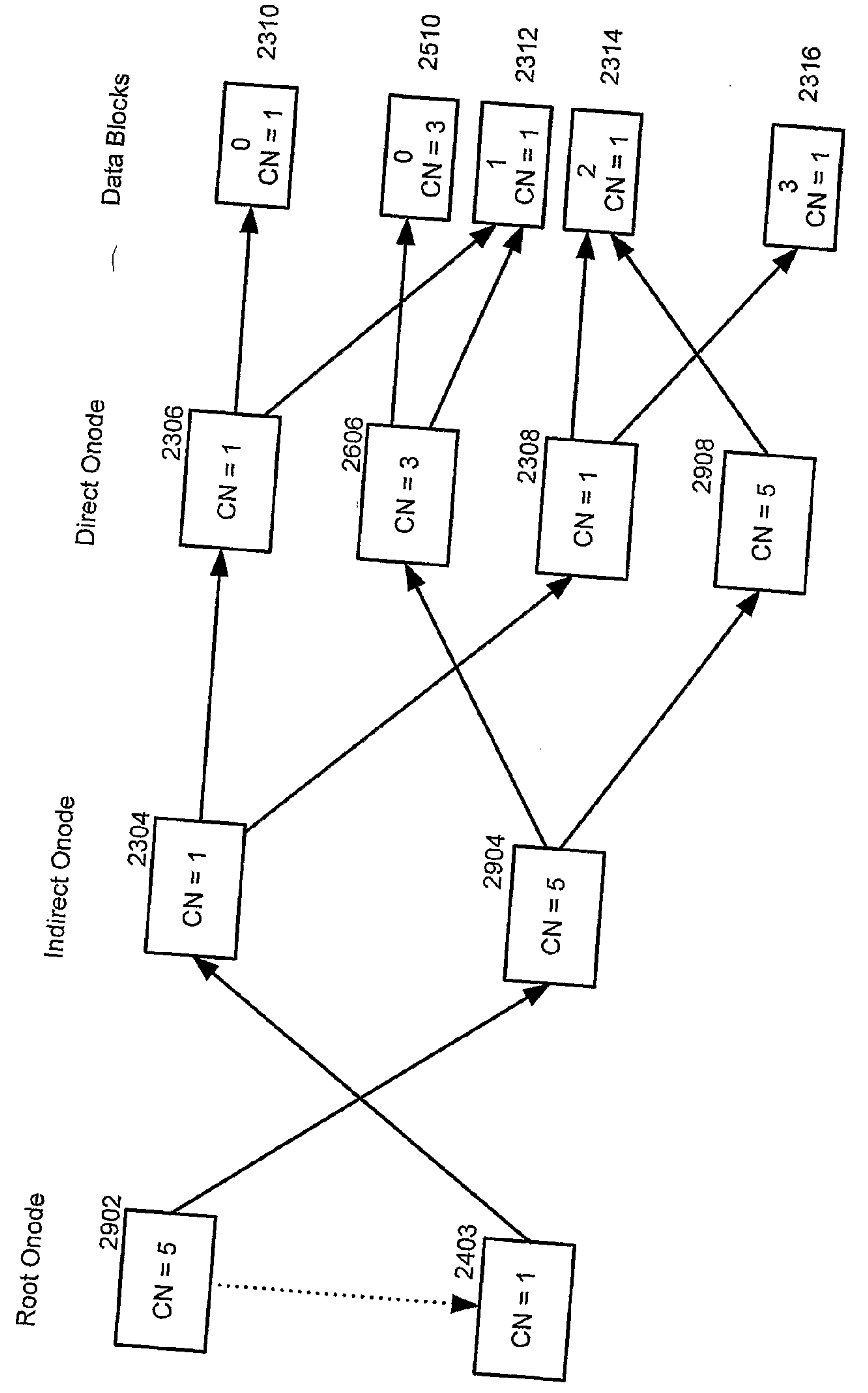

Root Onode
Indirect Onode
Direct Onode
Data Blocks
2902
CN = 5
2403
CN = 1
2304
CN = 1
2904
CN = 5
2306
CN = 1
2606
CN = 3
2308
CN = 1
2908
CN = 5
2310
0 CN = 1
2510
0 CN = 3
2312
1 CN = 1
2314
2 CN = 1
2316
3 CN = 1

FIG. 30

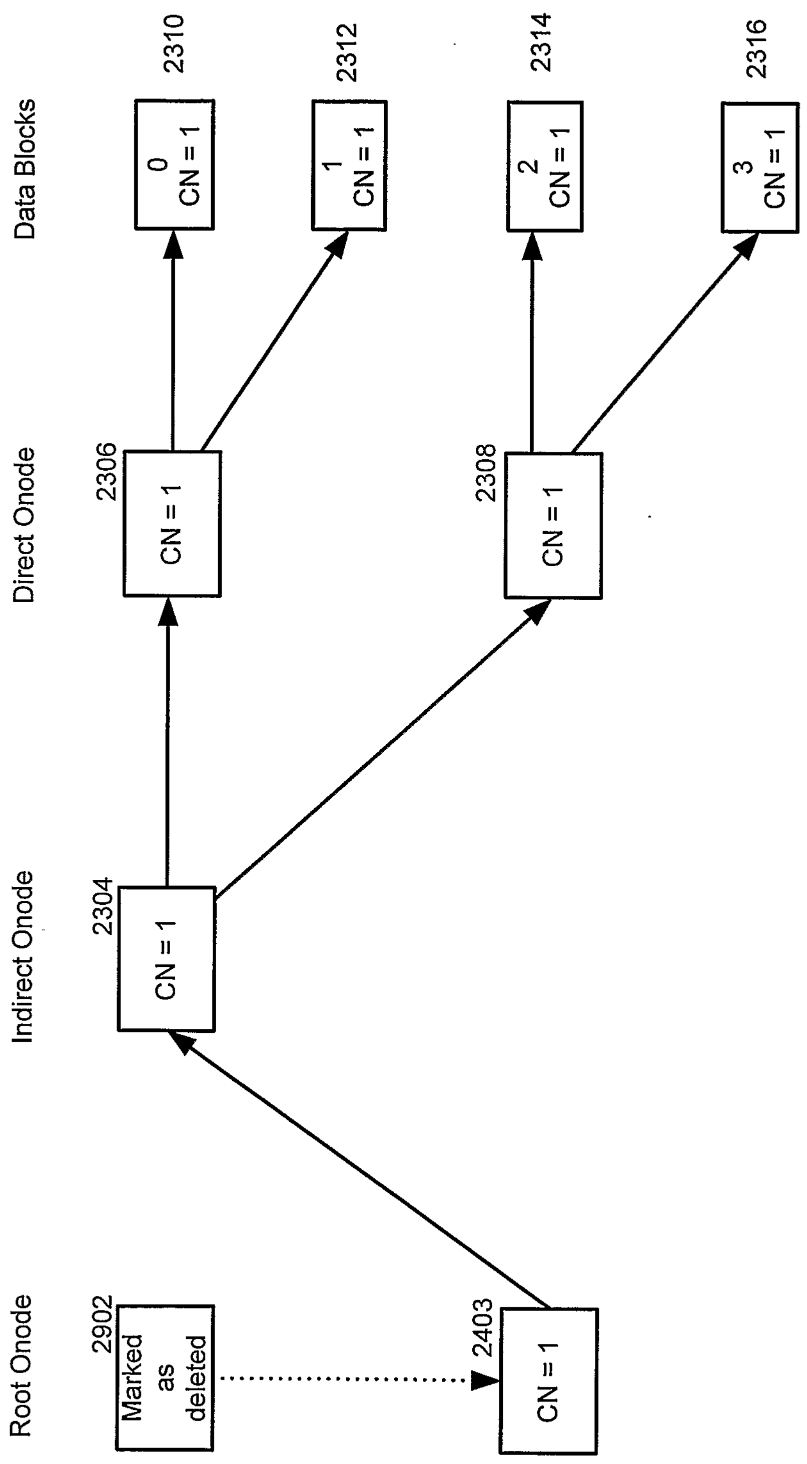
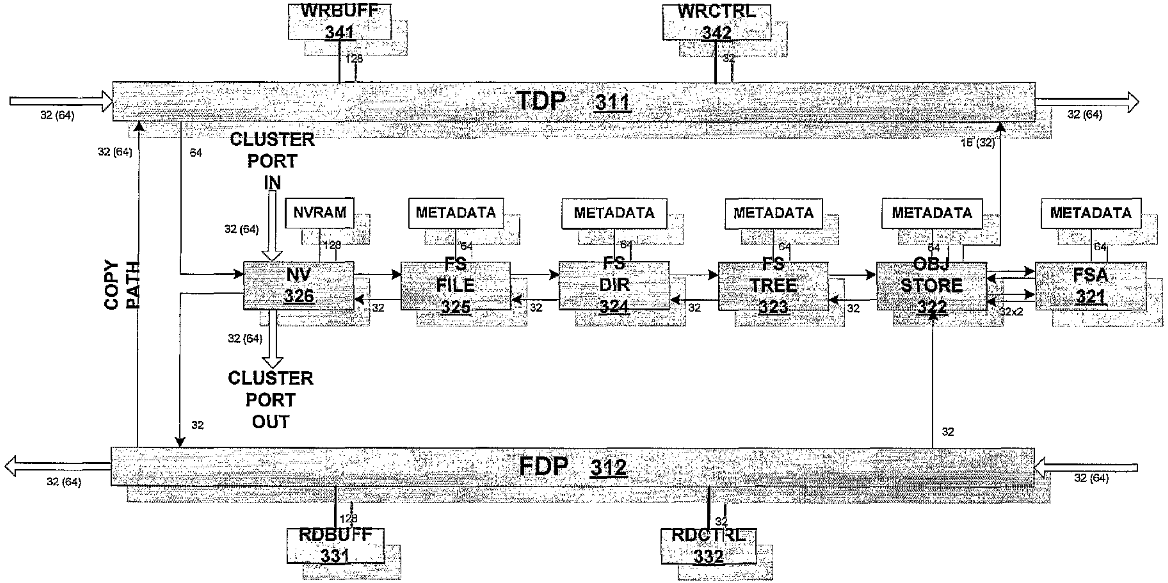

Root Onode
Indirect Onode
Direct Onode
Data Blocks
2902
Marked as deleted
2403
CN = 1
2304
CN = 1
2306
CN = 1
2308
CN = 1
2310
0 CN = 1
2312
1 CN = 1
2314
2 CN = 1
2316
3 CN = 1

FIG. 31



WRBUFF 341
WRCTRL 342
128
32
TDP 311
32 (64)
32 (64)
32 (64)
64
CLUSTER PORT IN
16 (32)
32 (64)
NVRAM
128
METADATA
64
METADATA
64
METADATA
64
METADATA
64
METADATA
64
NV 326
FS FILE 325
FS DIR 324
FS TREE 323
OBJ STORE 322
FSA 321
32
32
32
32
32x2
COPY PATH
32 (64)
CLUSTER PORT OUT
32
32
FDP 312
32 (64)
32 (64)
128
32
RDBUFF 331
RDCTRL 332