

009462.P001Z

UNITED STATES PATENT APPLICATION

FOR

**SYSTEM AND METHODS FOR IMPLEMENTING AN ENTERPRISE-CLASS CONVERGED
COMPUTE-NETWORK-STORAGE INFRASTRUCTURE**

INVENTOR:

SANDIP AGARWALA
ABHISHEK CHATURVEDI
SHRAVAN GAONKAR
MALLIKARJUNAN MAHALINGAM
SAZZALA REDDY
SMIT SHAH
FARAZ SHIEKH
PRAVEEN VEGULLA
KRISHNA YADAPPANAVAR
JEFFREY A. ZABARSKY

PREPARED BY:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN
1279 OAKMEAD PARKWAY
SUNNYVALE, CA 94085-4040

(408) 720-8300

CERTIFICATE OF ELECTRONIC FILING

I hereby certify that this correspondence is being deposited with the United States Patent and Trademark Office via electronic filing through the United States Patent and Trademark Electronic Filing System on:

December 19, 2012

Date of Deposit

Name of Person Mailing Correspondence

/Carla Anysia Nascimento/

Signature

December 19, 2012

Date

009462.P001Z

-1-

SYSTEM AND METHODS FOR IMPLEMENTING AN ENTERPRISE-CLASS CONVERGED COMPUTE-NETWORK-STORAGE INFRASTRUCTURE

FIELD OF INVENTION

[0001] This invention relates generally to a storage system and more particularly to software defined distributed storage system.

BACKGROUND OF THE INVENTION

[0002] Enterprise storage systems currently available are proprietary storage appliances that integrate the storage controller functions and the storage media into the same physical unit. This centralized model makes it harder to independently scale the storage systems' capacity, performance and cost. Users can get tied to one expensive appliance without the flexibility of adapting it to different application requirements that may change over time. For small and medium scale enterprise, this may require huge upfront capital cost. For larger enterprise datacenters, new storage appliances are added as the storage capacity and performance requirements increase. These operate in silos and impose significant management overheads.

SUMMARY OF THE DESCRIPTION

[0003] A software defined distributed storage system that separates the storage controller functions with that of storage media is described. In an exemplary embodiment, a storage controller client dispatches an input/output request to one of a plurality of storage controller servers in a software defined distributed storage system. The software defined distributed storage system further includes a plurality of virtual nodes distributed among a plurality of physical nodes, and each of the plurality of physical nodes is hosted on one of the plurality of storage controller servers. The storage controller client receives the input/output request. The storage controller client further computes a target virtual node for the input/output request, where the target virtual node is one of the plurality of virtual nodes. Using the computed target virtual node, the storage controller client determines a target physical node that corresponds to the target virtual node, where the target physical node is one of the plurality of physical nodes. In addition, the storage controller client dispatches the input/output request to a target storage controller server that corresponds to the target physical node, wherein the target storage controller server is one of the plurality of storage controller servers.

[0004] In another embodiment, a storage controller server receives a write operation request to store data, where the write operation request is dispatched from a storage controller client to the storage controller server using a deterministic placement function. In response to the write operation request, the storage controller server writes the data to a write log in an active log partition of the storage controller server.

[0005] In a further embodiment, a storage controller server receives a read operation request, where the read operation request is dispatched from a storage controller client to the storage controller server using a deterministic placement function. In response to receiving the read operation request, the storage controller server reads the data that is indicated in the read operation request, where at least part of the data is stored in a write log that is in an active log partition of the storage controller server.

[0006] In an additional embodiment, the storage controller server flushes data from a write log to persistent storage. In this embodiment, the storage controller server determines if the write log is full. For each metadata virtual node, the storage controller server computes a new virtual node for the data of the current metadata virtual node. In addition, storage controller server reads data of the current virtual node and writes the data to the new virtual node.

[0007] In a further embodiment, the storage controller server synchronously replicates data to a plurality of other storage controller servers. In this embodiment, the storage controller server receives a write log entry and retrieves a target virtual node for the write log entry. In addition, the storage controller server determines a plurality of virtual mirror nodes for the target virtual node, wherein each of the mirror nodes is located on different ones of the plurality of controller servers. The storage controller synchronously sends the write log entry to each of the different ones of the plurality storage controller servers.

[0008] Additionally, in another embodiment, the storage controller asynchronously replicates data for persistent storage to a plurality of storage controller servers. In this embodiment, the storage controller server receives data, metadata for this data, and a target virtual node for the persistent storage. The storage controller server further determines a plurality of virtual mirror nodes for the target virtual node, wherein each of the mirror virtual nodes are located on different ones of the plurality of controller servers. The storage controller server further asynchronously sends the data and metadata to each of the different ones of the plurality storage controller servers, wherein the data and metadata are stored in a persistent storage of each of the plurality storage controller servers.

[0009] In a further embodiment, a first storage controller server determines if there is duplicative data in a distributed storage system. In this embodiment, the first storage controller server determines if a first entity is duplicated in the distributed storage system. The first storage controller server determines if the first entity is duplicated in the distributed storage system by receiving the first entity and looking up the first entity in a data deduplication table. If the first entity exists in the data deduplication table, updating the metadata for the first entity to point to

the duplicate entity. In addition, a second controller server determines if a second entity is duplicated in the distributed storage system.

[0010] In an additional embodiment, the storage controller server handles a failure of a component of the distributed storage system. In this embodiment, the storage controller server detects a component failure in the distributed storage system and determines a virtual node that is impacted by the component failure. The storage controller server further retrieves an active mirror set for the impacted virtual node. If a number of active mirror virtual nodes identified in the active mirror set is less than a threshold, the storage controller server allocates a new virtual node and recovers stored data from the active mirror set for the new virtual node.

[0011] Other methods and apparatuses are also described.

BRIEF DESCRIPTION OF THE DRAWINGS

- [0012] The present invention is illustrated by way of example and not limitation in the figures of the accompanying drawings in which like references indicate similar elements.
- [0013] Figure 1 is an illustration of one embodiment of a high-level view of StorFS system.
- [0014] Figure 2 is an illustration of one embodiment of a system including some of the components that comprises the storage controller server.
- [0015] Figure 3 is an illustration of one embodiment of an efficient distributed replication.
- [0016] Figure 4 is a flow diagram of one embodiment of a process to perform a full metadata tree walk for garbage collection.
- [0017] Figure 5 is a flow diagram of a process to perform an incremental metadata tree walk for garbage collection.
- [0018] Figure 6 is an illustration of one embodiment of a deduplication module and garbage collection module.
- [0019] Figure 7 is an illustration of one embodiment of the Block Management layer.
- [0020] Figure 8 is a block diagram of one embodiment of a write log layout.
- [0021] Figures 9A-B are flow diagrams of embodiments of processes to execute of read/write operation in the write log.
- [0022] Figure 9C is a flow diagram of one embodiment of a process to perform synchronous mirroring.
- [0023] Figure 10 is a flow diagram of one embodiment of a process to perform input/output processing.
- [0024] Figure 11A is a flow diagram of one embodiment of a process to flush write log content to the persistent storage.
- [0025] Figure 11B is a flow diagram of one embodiment of a process to asynchronously mirror data and metadata.
- [0026] Figure 11C is a flow diagram of one embodiment of a process to determine if the data is duplicated.

[0027] Figure 11D is a flow diagram of one embodiment of a process to lookup a VBA based on the content fingerprint.

[0028] Figure 12 is a flow diagram of one embodiment of a process to prepare a hint table that is used for data deduplication.

[0029] Figure 13 is a flow diagram of one embodiment of a process to handle a failure of a component.

[0030] Figure 14 is a flow diagram of one embodiment of a process to perform a distributed failure recovery.

[0031] Figure 15 illustrates one example of a typical computer system, which may be used in conjunction with the embodiments described herein.

DETAILED DESCRIPTION

[0032] A software defined distributed storage system called StorFS that separates the storage controller functions with that of storage media is described. In the following description, numerous specific details are set forth to provide thorough explanation of embodiments of the present invention. It will be apparent, however, to one skilled in the art, that embodiments of the present invention may be practiced without these specific details. In other instances, well-known components, structures, and techniques have not been shown in detail in order not to obscure the understanding of this description.

[0033] Reference in the specification to “one embodiment” or “an embodiment” means that a particular feature, structure, or characteristic described in connection with the embodiment can be included in at least one embodiment of the invention. The appearances of the phrase “in one embodiment” in various places in the specification do not necessarily all refer to the same embodiment.

[0034] In the following description and claims, the terms “coupled” and “connected,” along with their derivatives, may be used. It should be understood that these terms are not intended as synonyms for each other. “Coupled” is used to indicate that two or more elements, which may or may not be in direct physical or electrical contact with each other, co-operate or interact with each other. “Connected” is used to indicate the establishment of communication between two or more elements that are coupled with each other.

[0035] The processes depicted in the figures that follow, are performed by processing logic that comprises hardware (e.g., circuitry, dedicated logic, etc.), software (such as is run on a general-purpose computer system or a dedicated machine), or a combination of both. Although the processes are described below in terms of some sequential operations, it should be appreciated that some of the operations described may be performed in different order. Moreover, some operations may be performed in parallel rather than sequentially.

[0036] The terms “server,” “client,” and “device” are intended to refer generally to data processing systems rather than specifically to a particular form factor for the server, client, and/or device.

[0037] A software defined distributed storage system called StorFS that separates the storage controller functions with that of storage media is described. In one embodiment, this separation allows the storage controller to be co-located with the user application in the same physical units or reside separately in a dedicated set of storage units. This set of storage units functions as a single unified storage system, and balances the capacity and performance load across the distributed components. A cache accelerated distributed storage design along with efficiency and reliability features can provide a higher level of determinism in terms of quality of service and performance. The ability to converge compute and storage significantly simplifies the management and reduces both the capital cost as well as the operational cost.

[0038] A software defined distributed storage system called StorFS that separates the storage controller functions with that of storage media is described. In one embodiment, the capacity and performance of a storage system can be increased in two ways: scale-up and/or scale-out. Scaling-up refers to adding more resources (e.g. storage, compute, etc.) in the same physical storage unit. In this embodiment, there can be a limit up to which a system can be scaled up. Thus, further scaling requires adding more physical units that work in cohesion to provide increased capacity and performance. This is called scale-out and has been the motivation of many distributed storage system designs. In an ideal scale-out system, adding the resources increases the capacity and performance by the same ratio without impacting the reliability characteristics of the storage system. In practice, however, this ideal scale-out is hard to achieve. This is because previous distributed systems rely on a centralized component for synchronization of the distributed system’s various tasks and managing/storing of the distributed system metadata. In one embodiment, this metadata is data that describes the distributed system and the location of users’ data. This centralized metadata becomes the bottleneck because the centralized metadata distributed storage system is not able to scale beyond a certain limit.

[0039] In one embodiment, the StorFS system is a storage system that includes a plurality of storage nodes coupled together by an interconnection network. In this embodiment, the StorFS system provides a distributed storage of data to one or more local and/or remote applications that make use of the storage and/or storage services provided the StorFS system. In one embodiment, each of the storage nodes includes a storage controller that is used to manage storage requests of the local and/or remote applications using the storage. The storage controller can include a storage controller client (SC client), a storage controller server (SC server), or both the SC client and SC server. In one embodiment, the SC client handles and dispatches application storage requests to an SC server. The SC server handles the storage requests and can additionally perform storage services (e.g., synchronous and asynchronous replication, distributed data deduplication, distributed garbage collection, data flushing, and distributed recovery).

[0040] In one embodiment, the SC client dispatches storage requests from the applications to an SC server that stores the data referenced by the storage request. In this embodiment, the storage request includes an entity identifier that references the stored data and an offset that is an offset into that stored data. The entity identifier can be an iNode, a file descriptor, key-value pair identifier, etc. If the storage request is a write of data to the StorFS system, the SC client determines which of the plurality SC servers is to handle the write storage request. In one embodiment, the SC client computes the SC server to handle the write storage request using a deterministic placement function. In this embodiment, the deterministic placement function uses the input parameters to compute the virtual node and stripe unit that (or will) store the data referenced in the write storage request. In one embodiment, by using the deterministic placement function, the StorFS system does not need to store location metadata to determine the initial virtual node for the data in the request. Instead, the system stores metadata for the virtual node to physical node conversion. With the virtual node identification, in one embodiment, the SC client determines the physical node identification and the SC server that corresponds to this physical node.

[0041] The SC server that stores the data referenced by the write storage request, receives this request. In one embodiment, the SC server stores this data in a write log by appending the data at the end of the write log. This can lead to a fast write for the data that allows a quick write storage fulfillment for the requesting application. In addition and in one embodiment, the SC server can synchronously replicate the data referenced in the write storage request. In this embodiment, the SC server determines the mirrors for this virtual node using a dynamic table that tracks the mirrors for each virtual node. In addition, the SC server sends the data to the SC servers that are used for the mirror virtual nodes.

[0042] In one embodiment, when the write log is full, the SC server flushes the write log to persistent storage. In this embodiment, for each virtual node that stores data in the full write log, the StorFS system determines new virtual node for that data and sends the data to the SC server that corresponds to the new virtual node. The SC server stores this data in persistent storage. In addition, the StorFS system can determine if the data that is stored in the write log for that virtual node is a duplicate of some other data stored elsewhere in the StorFS system. In one embodiment, during the flushing operation, the StorFS system determines if an entity is stored elsewhere by using a deduplication hints table. In one embodiment, the hints table stores characteristics of the top-K storage entities in the StorFS system. If there is a match with the entity being flushed, the StorFS system updates the metadata for this entity, but does not flush the entity to the persistent storage. If not, the StorFS system flushes the entity to persistent storage.

[0043] In one embodiment, during a flush to persistent storage, the StorFS system can asynchronously replicate each virtual node in the write log. In this embodiment, the StorFS system determines the mirrors of the virtual node and asynchronously sends the data and metadata of that virtual node to the mirror virtual nodes. In one embodiment, the StorFS system transfers the data and metadata of the virtual node as a large bulk transfer so that this data transfer is more efficient.

[0044] In addition to the write operations of the StorFS system outlined above, the StorFS system can perform additional storage services on the stored data. In one embodiment, the StorFS

system can perform distributed failure recovery, distributed deduplication, and distributed garbage collection. In one embodiment, a virtual node can have one or more mirror nodes, where each of the mirror nodes is stored in a different fault domain. If a component of the virtual node goes down and/or becomes unresponsive, a new virtual node is allocated. The StorFS copies the data and/or metadata for the failed virtual node from the one or more mirror virtual nodes to the newly allocated virtual node. In one embodiment, the StorFS copies the data from the mirror nodes in large segment, making the restoration of that virtual node more efficient.

[0045] Furthermore, in one embodiment, the StorFS system can perform distributed data deduplication. As described above, during a flushing of data from the write log to persistent storage, the StorFS system checks to determine if the data being moved is duplicated elsewhere in the StorFS system. The StorFS system uses a hints table to determine whether data is duplicated. In one embodiment, the hints table stores statistics about a partial set of data in the StorFS system. By using a partial set of data statistics, the StorFS can do the data deduplication check efficiently. The StorFS system creates the hints table by creating a live object map of the data stored in the StorFS system. Using this map, the StorFS system determines a top-K statistics on the data stored in the StorFS system and uses these statistics in the hints table. In one embodiment, there is a global hints table that is used by each of the SC servers to determine if data is being duplicated in the StorFS system. In one embodiment, the hints table map stores a fingerprint for a VBA of content stored in the StorFS system.

[0046] In one embodiment, the StorFS system stores data in logically sequential segments. As data is deleted, these segments become unreferenced and can be reused. In one embodiment, the StorFS system walks a metadata tree to determine which of the segments are not referenced. For each unreferenced segment, the StorFS system collects the unreferenced segments and makes the segments available for future use. In addition, these segments can be used for fast distributed recovery of a vNode that is hosted by component that fails. In one embodiment, if a component of the StorFS system fails, the StorFS system determines which vNodes are affected by

component recovery. For each affected vNode, the StorFS system can either create a new mirror vNode and resync this new vNode or resync that affected vNode.

[0047] Figure 1 is an illustration of one embodiment of a high-level view of StorFS system 100. In Figure 1, the StorFS system 100 includes storage nodes 102A-C coupled by an interconnection network 116. While in one embodiment, three storage nodes 102A-C are illustrated as part of the StorFS system 100, in alternate embodiments, there can be more or less storage nodes. For example and in one embodiment, the StorFS system 100 can include up to several hundred storage nodes. In one embodiment, each storage node 102A-C consists of either a storage controller (SC) client (also called dispatcher) 108A-C, a storage controller (SC) server 110A-C, or both. The SC servers 110A-C manage their underlying storage (e.g., Hard disk drive (HDD), Solid state drives (SSD), PCIe flash, etc.) and collectively provide reliable and unified storage functionality to the SC clients. The SC client 108A-C processes input/output (I/O) requests from the applications that are local to its physical storage node and routes them to the appropriate SC servers for processing. For example and in one embodiment, SC client 108A can send an I/O request locally to SC Server 110A and/or remotely to SC Servers 110B or 110C. The system is capable of exposing many different interfaces to the application like file (e.g. NFS, CIFS), object, key-value, etc. In one embodiment, the storage node 102A-C can be server, blade server, personal computer, etc., or any other type of device capable of storing data. In one embodiment, the management server 112A-C is an agent that is used to communicate system management data and commands regarding the corresponding storage node 102A-C with the management client 114.

[0048] In one embodiment, the design of the StorFS system 100 distributes both the data and the metadata, and this system 100 does not require storing a complete global map for locating individual data blocks in our system. The responsibility of managing metadata is offloaded to each individual storage nodes 102A-C. In one embodiment, a cluster manager (CRM) resides on each SC Server 110 maintains some global metadata, which is small compared to the local metadata. In one embodiment, each logical file (or entity) is partitioned into equal sized “stripe

units”. The location of a stripe unit is determined based on a mathematical placement function Equation (1):

$$\begin{aligned} \text{Virtual_Node\#} &= \text{Hash}(\text{Entity_Id}, \text{Stripe_Unit\#}) \% \text{Total_Virtual_Nodes} \\ \text{Stripe_Unit\#} &= \frac{\text{offset}}{\text{Stripe_Unit_Size}} \% \text{Stripe_Unit_Per_Stripe} \end{aligned} \quad (1)$$

The *Entity_{Id}* is an identification of a storage entity that is to be operated upon, the *Total_Virtual_Nodes* is the total number of virtual nodes in the StorFS system 100, the offset is an offset into the storage entity, and the *Stripe_Unit_Size* is the size of each stripe unit in the StorFS system 100. The value *Stripe_Unit_Per_Stripe* is described further below. In one embodiment, the storage entity is data that is stored in the StorFS system 100. For example and in one embodiment, the storage entity could be a file, an object, key-value pair, etc. In this example, the *Entity_{Id}* can be an iNode value, a file descriptor, an object identifier, key/value identifier, etc. In one embodiment, an input to a storage operation is the *Entity_{Id}* and the *offset* (e.g., a write, read, query, create, delete, etc. operations). In this embodiment, the *Entity_{Id}* is a globally unique identification.

[0049] In one embodiment, the StorFS 100 system receives the *Entity_{Id}* and *offset* as input for each requested storage operation from an application 106A-C. In this embodiment, the StorFS system 100 uses the *offset* to compute a stripe unit number, *Stripe_Unit#*, based on the stripe unit size, *Stripe_Unit_Size*, and the number of virtual nodes that the entity can be spread across, *Stripe_Unit_Per_Stripe*. Using the stripe unit number and the entity identifier (*Entity_{Id}*), the StorFS system 100 computes the virtual node identifier. As described below, the StorFS system 100 uses a hash function to compute the virtual node identifier. With the virtual node identifier, the StorFS 100 can identify which physical node the storage entity is associated with and can route the request to the corresponding SC server 110A-C.

[0050] In one embodiment, each vNode is a collection of either one or more data or metadata objects. The StorFS system 100 does not store data and metadata in the same virtual node. This is because data and metadata may have different access patterns and quality of service (QoS) requirements. In one embodiment, a vNode does not span across two devices (e.g. a HDD). A

single storage disk of a storage node 102A-C may contain multiple vNodes. In one embodiment, the placement function uses that a deterministic hashing function and that has good uniformity over the total number of virtual nodes. A hashing function as known in the art can be used (e.g., Jenkins hash, murmur hash, etc.). In one embodiment, the “*Stripe_Unit_Per_Stripe*” attribute determines the number of total virtual nodes that an entity can be spread across. This enables distributing and parallelizing the workload across multiple storage nodes (e.g., multiple SC servers 110A-C). In one embodiment, the StorFS system 100 uses a two-level indexing scheme that maps the logical address (e.g. offset within a file or an object) to a virtual block address (VBA) and from the VBAs to physical block address (PBA). In one embodiment, the VBAs are prefixed by the ID of the vNode in which they are stored. This vNode identifier (ID) is used by the SC client and other StorFS system 100 components to route the I/O to the correct cluster node. The physical location on the disk is determined based on the second index, which is local to a physical node. In one embodiment, a VBA is unique across the StorFS cluster, where no two objects in the cluster will have the same VBA.

[0051] In one embodiment, the cluster manager (CRM) maintains a database of virtual node (vNode) to physical node (pNode) mapping. In this embodiment, each SC client and server caches the above mapping and computes the location of a particular data block using the above function in Equation (1). In this embodiment, the cluster manager need not be consulted for every I/O. Instead, the cluster manager is notified if there is any change in ‘vNode’ to ‘pNode’ mapping, which may happen due to node/disk failure, load balancing, etc. This allows the StorFS system to scale up and parallelize/distribute the workload to many different storage nodes. In addition, this provides a more deterministic routing behavior and quality of service. By distributing I/Os across different storage nodes, the workloads can take advantage of the caches in each of those nodes, thereby providing higher combined performance. Even if the application migrates (e.g. a virtual machine migrates in a virtualized environment), the routing logic can fetch the data from the appropriate storage nodes. Since the placement is done at the stripe unit granularity, access to data within a particular stripe unit goes to the same physical node. Access

to two different stripe units may land in different physical nodes. The striping can be configured at different level (e.g. file, volume, etc.) Depending on the application settings, the size of a stripe unit can range from a few megabytes to a few hundred megabytes. In one embodiment, this can provide a good balance between fragmentation (for sequential file access) and load distribution.

[0052] Figure 2 is a block diagram of one embodiment of a storage control server 200. In one embodiment, the Block Management Layer 220 of the storage control server is responsible for formatting, allocating, and freeing storage in fixed block sizes. This layer provides access to different kinds of storage (e.g. SSD, HDD, etc.) in the system. In one embodiment, the Distributed Object Layer 218 of the storage control server uses an API of the Block Management Layer 220 to provide a global distributed object namespace that is accessible over the network. In one embodiment, the Storage Efficiency Layer 216 reduces the size of data footprint on the physical medium using techniques like compression, deduplication, etc. The reduction is achieved without deteriorating the performance or the reliability of the data storage. In one embodiment, the File System Layer 214 provides a logical global namespace abstraction to organize and locate data in the cluster. In one embodiment, the Data Service Layer 212 provides enterprise data services like disaster recovery, fine grained policy management, snapshots/clones, etc. In one embodiment, the Write Cache 208 and the Read Cache 210 Layers provide acceleration for write and read I/O respectively using fast storage devices. In one embodiment, the Write Cache Layers 208 includes the write log as described below. In one embodiment, the Presentation Layer 206 provides an interface to access the StorFS storage using well-known standard protocols like NFS, CIFS, REST, iSCSI, etc. In one embodiment, the Cluster Manager (CRM) Layer 202 is responsible for the coordination across distributed StorFS components, delegating responsibilities and maintaining a consistent global state of the system. In one embodiment, the Fault Tolerance Layer 204 is responsible for resiliency and making sure that the data is available and consistent even after the failure of a software or hardware component (disk, server, network, etc.). In one embodiment, the Garbage Collection Layer 222 is responsible for

reclaiming dead space that result due to entities getting deleted or updated. This layer efficiently determines the storage blocks that are not used (or referenced) and makes them available for new data to be written. In one embodiment, the Storage Management Layer 224 provides a framework to configure, monitor, analyze and report on the operation of the overall StorFS cluster storage system as well as individual logical and physical entities in the cluster. In one embodiment, each of the layers mentioned above are fully distributed and each layer does not rely on any centralized components for their operations.

[0053] Figure 10 is an illustration of one embodiment of a process 1000 to perform input/output processing. In one embodiment, process 1000 is performed by the Storage Controller 104 of Figure 1. In Figure 10, process 1000 begins by receiving an input/output request from a client at block 1002. In one embodiment, the client can be an application that uses the StorFS system, such as applications 106A-D as described in Figure 1 above. In one embodiment, the input/output request can be a request to read content from the StorFS system, write content to the StorFS system, and/or request status of content stored in the StorFS system. At block 1004, process 1000 validates the input/output request. In one embodiment, validation may comprise of simple I/O attributes consistency check, permission checks, admission control, etc. If the request is not validated, at block 1006, process 1000 return an error to the requesting client (e.g., error code, throws an exception, etc.). If the request is validated, at block 1008, process 1000 computes the vNode for the input/output request using the placement function. In one embodiment, process 1000 uses Equation (1) described above to compute the vNode for the input/output request. In addition, process 1000 determines the pNode corresponding to the determined vNode. In one embodiment, process 1000 determines the corresponding pNode using the CRM routing table as described above with reference to Figure 2 above.

[0054] At block 1010, process 1000 sends the input/output request to SC server on the corresponding pNode. For example and in one embodiment, if process 1000 executing on SC client 108A on physical server 102A and the corresponding pNode is on the SC Server 110C of physical server 102C, process 1000 sends the input/output request to the SC server 110C on

physical server 102C. At block 1012, process 1000 determines if the input/output request is related to creating or deleting an entity. In one embodiment, an entity is data that is stored in the StorFS system 100. For example and in one embodiment, the entity could be a file, an object, key/value pair, etc. . If the request is not related to creating or deleting an entity, execution proceeds to block 1016 below. If the input/output request is related to creating or deleting an entity, at block 1014, process 1000 determines which of the vNodes need to be updated. In one embodiment, process 1000 determines the vNodes to be updated by computing all the vNodes that the entity can reside using the placement function (Equation (1)). At block 1018, process 1000 sends the input/output request to the SC server that corresponds to the vNodes determined at block 1014.

[0055] At block 1016, process 1000 receives an I/O request for write log processing. In one embodiment, the request for write log processing is to be processed by process 900 as described in Figure 9A above. Process 1000 determines if the write log processing is complete at block 1020. If the write log processing is not complete, at block 1022, process 1000 wait for the write log processing to complete and execution proceeds to block 1020. If the write log processing is complete, process 1000 returns.

I/O Dispatcher

[0056] In one embodiment, an application can use some type of middleware libraries to access networked storage. For example and in one embodiment, virtual machines may use the hypervisor's built-in NFS client or iSCSI client to access backend storage. In a clustered environment, the goal of this middleware is not only to provide connectivity to backend storage but also availability by failing over to other cluster nodes in case of failure. In one embodiment, the StorFS system uses an intelligent I/O dispatcher that intercepts I/O requests from the client (e.g. NFS Client running inside a hypervisor) and directs them to appropriate cluster nodes. In one embodiment, the I/O dispatching is performed by a SC Client, such as SC Client 108A-C as described in Figure 1 above. There are three modes of operation: i) I/O requests are routed to the

SC Client running on the same physical node (i.e. to 'localhost'); ii) A common FQDN (Fully Qualified Domain Name) entry for the cluster pointing to the 'localhost' or '127.0.0.1' is created in the DNS configuration file (typically /etc/hosts) of each cluster nodes; and iii) Intercepting and rerouting incoming I/O requests using software-based routing mechanism like DVS (Distributed Virtual Switch) and "Network Service Insertion" framework available in many popular virtualization platforms (e.g. VMware).

Distributed Multi-level caching

[0057] In one embodiment, volatile memory (e.g., DRAM) is used to cache the working data set of the application in storage/file systems. Depending on the applications, this working set can be huge and it may not be possible to cache the entire set in DRAM. Increasing the size of DRAM is an option, but this comes at an increased cost. In another embodiment, an SSD can provide a relatively low cost alternative to cache the data blocks. Unlike DRAM, SSDs have asymmetric read/write performance for different I/O patterns. Also, SSDs have limited write endurance, which makes it harder to use them as caching device.

[0058] In one embodiment, the StorFS system makes it easier to fully harness the capabilities of SSDs and multiple levels of caches. In one embodiment, this system of write cache and read cache hides the latency of slower HDDs from the users. In this embodiment, the StorFS system writes data sequentially on SSD because random writes incur higher latency and reduce the life of SSD. In addition to taking data blocks' recency and frequency into account, the StorFS system considers SSD's erase block size for eviction of 'cold' data that may no longer be required by the application. Furthermore, the StorFS system is able to prioritize the use of cache depending on the application requirements. The StorFS system additionally can use multi-level indexing to track and locate data blocks in large SSD caches. In one embodiment, this is important because these indexes can be large and may not fit entirely in the DRAM. This index is built of content hash of the object it stores as opposed to the logical or physical address of the object. This prevents storing duplicated content twice and increases the effective cache size significantly.

Furthermore, caches distributed across the storage nodes act as a single large unified cache and deliver increased performance as compared to a single isolated local cache.

Write Cache

[0059] In one embodiment, the StorFS system logs the incoming updates and writes from the client to fast storage like SSD, flash or NVRAM and synchronously replicates them to a remote cluster node. The StorFS system updates its in-memory data structure called a “Delta Tree” and acknowledges the completion of update/write operation. In one embodiment, the Delta Tree represents changes that have been committed but not transferred to the final persistent storage. Figures 9A-B are flow diagrams of embodiments of processes to execute of read/write operation in the write log. In addition, a ‘flusher’ moves the logged content asynchronously from the high-speed storage to its permanent location for persistence. During the ‘flush’, the in-memory Delta Tree is merged with the on-disk metadata to provide data access for future I/O requests. Figure 9A is a flow chart of one embodiment of a process 900 to log a write operation. In Figure 9A, process 900 begins by receiving an update or a write operation request from a client at block 902. In one embodiment, an update operation request may comprise deleting an entity (e.g. object, file, etc.), changing its attributes, etc. A write operation request may comprise writing data to an existing entity (e.g. file) or creating a new object. At block 904, process 900 validates the operation. In one embodiment, process 900 validates the update or write operation request by verifying permissions, admission control, space availability, etc. If the operation is not validated, at block 914, process 900 returns an error to the client (e.g., error code, throws an exception, etc.). If the operation request is validated, process 900 writes an entry to the active write log partition and to the mirror nodes at block 906. In one embodiment, a write log is the portion of the fast storage where incoming updates from the clients are first written. In one embodiment, this entry logs the update/write operation in the write log. For example and in one embodiment, process 900 writes an entry to the active partition 810A of the primary portion 802 and the active partition 810C on the mirror partition 804A as described in Figure 8 above. In one embodiment,

process 900 writes an entry to the end of the write log. For example and in one embodiment, process 900 appends the entry at the end of the write log event if the write log has unclaimed space in the middle of the log that can accommodate the entry. By appending the entry at the end, the performance of the write can be increased.

[0060] At block 908, process 900 determines if the write of the entry succeeded. If the write operation did not succeed, at block 914, process 900 returns an error to the client (e.g., error code, throws an exception, etc.). If the write operation did succeed, at block 910, process 900 updates the in-memory delta tree. At block 912, process 900 returns success to the client.

[0061] Figure 9B is a flow diagram of one embodiment of a process 950 to perform a read operation. In Figure 9B, process 950 begins by receiving a query/read operation request from an SC client at block 952. In this embodiment, the SC client is basically forwarding the request it received from an application to the right SC server. At block 954, process 950 validates the query/read operation. Validation involves checking for permissions, admission control, correctness of the request parameters, etc. If the query/read operation is invalid, at block 970, process 950 returns an error to the client (e.g., error code, throws an exception, etc.). If the query/read operation is valid, at block 956, process 950 determines if the content is in the write log. In one embodiment, process 950 determines if the content is in the write log by examining an in-memory index of the write log that keeps track of the write log contents. If the content is in the write log, process 950 reads the content from the write log at block 958. At block 962, process 950 determines if a full read of the content can be done from the write log. In one embodiment, a full read of the content can be done if all of the content for the read operation is in the write log. For example and in one embodiment, the requested content may be fully or partially stored in the write log. If a full read can be done from the write log, at block 968, process 950 returns the content to the client.

[0062] If the content is not in the write log or if the content is partially in the write log, at block 960, process 950 reads the content from persistent storage. In one embodiment, process 950 reads some or all of the content from the persistent storage of a physical server, such as the physical

server 102A-C of Figure 1 described above. In one embodiment, this read of the content can be a full read (e.g., if the content is not stored in the write log) or can be a partial read (e.g., if some of the content is in the write log). At block 964, process 950 determines if the read has succeeded. If the read has not succeeded, at block 970, process 950 returns an error to the client (e.g., error code, throws an exception, etc.). If the read succeeded, at block 966, process 950 merges the content from the write log and that from the persistent storage. At block 968, process 950 returns the content to the client.

[0063] As describe above, the StorFS system can perform synchronous replication of the data being stored in the in the write log. Figure 9C is a flow diagram of one embodiment of a process 980 to perform synchronous replication of a write log entry. In one embodiment, process 900 performs process 980 at block 906 of Figure 9A above to perform this synchronous replication. In Figure 9C, process 980 begins by receiving the write log entry at block 982. In one embodiment, the write log entry is an entry to store an entity in the write log. At block 984, process 980 gets an active mirror set (AMS) for the write log entry. In one embodiment, the AMS is a mirror vNode set for the vNode associated with the write log entry. In this embodiment, the AMS includes this vNode and one or more active vNodes that is used to replicate the data and metadata of that vNode. Process 980 prepares the write log entry at block 986. In one embodiment, the write log entry consists of vNode ID, timestamp, checksum and other metadata along with the incoming data from the SC client.

[0064] At block 988, process 980 synchronously sends the write log entry to the pNodes that correspond to the mirror vNodes. In one embodiment, process 980 determines the corresponding SC server for each mirror vNode by looking up the corresponding pNodes in a vNode to pNode mapping table via the CRM. In one embodiment, the write log entry is synchronous so that the success of the writing to the write log can be acknowledged to the requesting client. In this embodiment, a synchronous write allows that the data is reliably stored before write is acknowledged to the client. At block 990, process 980 receives an acknowledgement from each pNode. While in one embodiment, there is one SC server for each pNode, in alternate

embodiments there can be more than one SC server for each pNode. In one embodiment, the acknowledgement indicates the write log entry have been stored at the corresponding pNode.

[0065] In one embodiment, StorFS creates a logical partition for the write log 800 in the fast storage and formats it as illustrated in Figure 8. In Figure 8, the write log 800 includes an active partition 802, mirror partitions 804A-B, and a spare partition 808. In one embodiment, the primary partition 802 and each of the mirror partitions 804A-B are stored in different fault domains. In this embodiment, if the device(s) in one of the fault domains associated with one of the primary 802 or mirror 804A-B partitions goes down or is otherwise inaccessible, the data stored in the other partitions is still accessible. The 'primary' partition 802 stores the new incoming writes from the clients. The 'mirror' partitions 804A-Bs store the mirrored writes from other StorFS storage cluster nodes. The 'spare' partition 810 is used as a staging area for 'vNode' migration from one physical node to another. The 'active' portion of each partition is used for logging new writes (e.g., active partitions 810A, 810C, and/or 810E). When the 'active' partition is full, the active partition is switched to the 'passive' state. New writes are directed to the second primary portion 810B, where the second primary portion becomes the active partition. In one embodiment, the 'flusher' processes the passive portion and moves its content to the permanent location for persistence. The flusher process is further described in Figure 11A below. Upon completion, the passive portion is ready to accept new writes.

[0066] Figure 11A illustrates one embodiment of a process 1100 to process to flush write log content to the persistent storage. In one embodiment, the writelog flusher module 606 of the SC server as described in Figure 6 performs process 1100. In Figure 11A, process 1100 performs an outer processing loop (blocks 1102-1128) to process the passive portion and move the content of passive portion to persistent storage for each pNode. At block 1104, process determines if the write log is full. If the write log is not full, at block 1106, process 1000 waits for the write log to fill up. Execution proceeds to block 1102 above. If the write log is full, at block 1108, process 1100 marks the currently active write log as passive and the currently passive as active. In

addition, process 1100 marks the current Delta Tree as passive and switches to a new Delta Tree for subsequent writes.

[0067] Process 1100 perform an inner processing loop (blocks 1110-1126) to “flush” (e.g., move) the content of each vNode in the write log to the persistent storage. At block 1112, process 1000 traverses a portion of the Delta Tree for that vNode. In one embodiment, a write log may contain contents belonging to one or more vNodes. In this embodiment, each metadata vNode has its own Delta Tree. At block 1114, process 1000 fetches a portion of the on-disk metadata tree that will be potentially be updated by the traversed portion of the Delta Tree in block 1112. Process 1100 merges the changes in the traversed portion of the delta tree with the on-disk metadata tree. At block 1118, process 1100 reads the data from the write log. At block 1119, process 1100 determines if the data read is duplicated in the StorFS system. In one embodiment, process 1100 performs a distributed data deduplication on this data to determine if the data is duplicated using a deduplication hint table. Determining if the data is duplicated is further described in Figure 11C below. If the data is duplicated, execution proceeds to block 1124. If the data is not duplicated, execution proceeds to block 1120 below.

[0068] Process 1100 computes the vNode for the data and merged metadata at block 1120. Process 1100 writes and mirrors the data and merged metadata to the respective vNodes. In one embodiment, process 1100 uses the Distributed Object Layer 218 to perform to write and mirror this data as described in Figure 2 above. In one embodiment, process 1100 performs an asynchronous flushing at block 1120 and is further described below in Figure 11B. At block 1124, process 1100 determines if the traversal for the Delta Tree for that vNode is done. If the traversal is not done, execution proceeds to block 1112 above. If the traversal is done, process 1100 determines if the vNodes in the write log are done at block 1126, where the inner processing loop ends. If the vNode write log processing is not done, execution proceeds to block 1110. If the vNode write log processing is done, at block 1128, process 1100 marks the passive write log for this pNode as available. The outer processing loop ends at block 1128.

[0069] As described above, the flusher asynchronously sends data and metadata to the target vNode and the mirror nodes to replicate the data and metadata being moved to persistent storage. Figure 11B is a flow diagram of one embodiment of a process 1150 to asynchronously replicate data and metadata. In one embodiment, process 1100 performs process 1150 at block 1120 of Figure 11 above to asynchronously replicate data. In Figure 11B, process 1150 begins by receiving the data and the metadata, for each target vNode for the asynchronous replication at block 1152. In one embodiment, the asynchronous replication is on a vNode by vNode basis. In this embodiment, the asynchronous replication can include a replication of one or more vNodes. In one embodiment, each vNode may contain a few bytes or up to several megabytes or greater.

[0070] At block 1154, process 1150 determines an AMS for the target vNode. In one embodiment, the AMS for a target node includes the target vNode and one or more active replica vNodes that are used to store the replication data and metadata for the target vNode. The AMS is described further below. Process 1150 asynchronously sends the data and metadata for that vNode to the target vNode and other mirror nodes in the AMS at block 1156. In one embodiment, an asynchronous send of the data is one where the send is sent to the SC server that corresponds to the vNode receiving the data and control is returned to process 1150 before the receiving SC server completes the write. In one embodiment, the receiving SC server receives the data and metadata and stores this content in the persistent storage of the SC server. After this storing of the data, the SC server sends an acknowledgement to process 1150. At block 1158, process 1150 sends an acknowledgement that the data and metadata has been stored. In one embodiment, process 1150 completes when the process has received the acknowledgements for each of the issued writes.

[0071] As described above, the flusher uses a hint table to determine if the data being flushed is duplicated elsewhere in the StorFS system. Figure 11C is a flow diagram of one embodiment of a process 1180 to determine if the data is duplicated using a hints table. In one embodiment, process 1180 is performed by process 1100 at block 1119 to determine if the data begin flushed is duplicated. In Figure 11C, process 1180 begins by receiving the entity to be stored. As

described above, the entity can be an object, file, key-value pair, etc. Process 1180 looks up the entity in the distributed deduplication table to determine if the entity is stored elsewhere in the system at block 1184. In one embodiment, the distributed deduplication table is a hint table that stores information about some of the entities stored in the StorFS system. For example and in one embodiment, the hint table stores the top-K statistics of the entities. In one embodiment, the hint table contains the fingerprint of some of the entities that are most likely to be deduplicated. The top-K statistic is one example of creating the hint table. It stores the top K most frequently occurring entities in the StorFS storage.

[0072] At block 1186, process 1180 determines if the entity exists in the distributed deduplication table based on the lookup at block 1184. In one embodiment, process 1180 checks if the entity is duplicated at the VBA level. Determining if the entity is duplicated at the VBA level is further described in Figure 11D below. If the entity does not exist in the distributed deduplication table, process 1180 returns a status that the entity is not duplicated at block 1188. If the entity is in the distributed deduplication table, process 1180 updates the metadata for the entity at block 1190. In one embodiment, process 1180 updates the metadata tree to point to the duplicate. This duplicate may or may not reside in the same vNode.

[0073] Figure 11D is a flow diagram of one embodiment of a process 1140 to lookup a VBA based on the content fingerprint. In Figure 11D, process 1140 requests a VBA to be checked for duplication. In one embodiment, this VBA has a fingerprint available (e.g., the fingerprint is stored with the VBA, the fingerprint can be calculated on the fly, etc.). The fingerprint for this VBA is determined using a fingerprinting method as known in the art (e.g., SHA-1 fingerprint, etc.). At block 1144, process 1140 determines if the fingerprint is in the map for the hints table. In one embodiment, this map is part of the hints table that includes the top-K statistics of fingerprint as described in Figure 12. If the fingerprint for this VBA is in the map, at block 1148, process 1140 returns the VBA, which indicates that this VBA is duplicated in the StorFS system. If the fingerprint is not in the map, at block 1146, process 1140 returns NULL, indicating that this VBA is not duplicated in the StorFS system.

Log structured sequential storage layout

[0074] In one embodiment, the Log structured sequential storage layout provides an efficient use of SSD and HDD I/O bandwidth available for incoming writes. In this embodiment, both the data and metadata are written to a new disk location as opposed to rewriting them at their old location. This layout transforms the incoming random I/O traffic to large sequential writes in the backend of the physical storage media (e.g., HDD, SDD, flash, etc.). In addition, this layout not only increases the write performance, but also enables many kinds of data services, provides protection from accidental deletes, etc. This layout additionally increases the SSDs lifetime by writing in large erase cycle blocks. However, a problem with the log structured layout is defragmentation of the physical storage media. This may not be a big problem in SSDs because of their high random I/O performance, but defragmentation can cause havoc in case of HDD spindles.

[0075] As mentioned earlier, the StorFS system uses a two-level indexing scheme that maps the logical address to a virtual block address (VBA) and from the VBAs to physical block address (PBA). The system allows for objects with contiguous VBAs to be stored close to each other. This addressing scheme further enables fast lookup of objects from the underlying media and reduces defragmentation significantly. As described above, VBAs are unique across the StorFS cluster and no two objects in the cluster will have the same VBA.

[0076] Figure 7 is an illustration of one embodiment of the Block Management layer 700. In Figure 7, the detailed software stack of the Block Management Layer 700 is illustrated. The Block Storage API provides a standardized interface to create a “Block Set” 704, allocate/free blocks within a Block Set 704, perform IO to blocks, etc. This is accomplished by invoking an internal block driver API. Block drivers can provide different behaviors and, if the bottom of the driver itself calls the standard Block Storage API, drivers may be stacked one on top of another. For example, in Figure 7, block API 712D is stacked upon Block Partition Driver 712C, which is stacked upon another Block API 712B and Block File Driver 712A. For freeing an HDD block

714A, the block partition driver 712C marks the freed block as “free pending” and invokes the API provided by the block file driver 712A to return the freed block. The block file driver, in this case, does no operation (NOOP 714C) on it and returns success to the partition driver, which finally marks the block as “free acknowledged” 714D and returns back to the client that invoked the original request.

[0077] When creating/opening a Block Set, a hierarchical path can be specified prepended with the string “<driverName>:” to indicate which driver should be used. If no driver name is specified, the Block File (“file:”) driver 712A is assumed and the path is treated as a standard Unix file or device. The Block File driver splits the file into fixed sized blocks and exports a single Block Set with all physical blocks allocated with physical block numbering (0 - (numBlocks - 1)). The Block Partition driver (“part:”) 712C can be stacked on top of the Block File 712A driver to allow a single physical disk Block Set to be partitioned into multiple Block Sets.

[0078] In one embodiment, the Block Flash driver (“flash:”) 718C can be stacked between the Block Partition 718E and File 718A drivers to provide flash specific semantics such as I/O scheduling (e.g. limiting the number of concurrent writes, preventing concurrent reads/writes, etc.) and invoking “TRIM” on block free. In one embodiment, freeing a flash-based block 720A is similar to freeing an HDD-based block 714A, except that, in this embodiment, the block flash driver gets invoked for the free request, which performs a “SCSI TRIM” 720C before invoking the block file driver. In one embodiment, a TRIM releases and prepares the specified region of a flash-based device for future writes.

[0079] The segment store uses the block abstraction to build collections of ordered segments. Each segment within a Segment Store is uniquely identified by its ‘logical id’, which is a monotonically increasing 64-bit number. This provides a uniform framework to manage different kind of storage devices and abstracts out the complexity of low-level block management from the upper layers in the StorFS software stack.

[0080] In one embodiment, both the data and metadata are distributed as well as replicated, which allows for recovery in case of failure. In one embodiment, the replication is performed across different 'fault domains' so that a failure of a component in one domain does not impact the availability of components in other domains. A set of HDDs, server nodes, or a physical server rack are a few examples of fault domain. This allows the StorFS system to tolerate different kind of failures: disk failures, node failures and/or rack failures. Replication of data, however, can impose a cost on a storage system. For example, replication can impose significant overhead in terms of capacity and steady state performance.

Efficient distributed replication across different fault domains

[0081] Figure 3 is an illustration of one embodiment of an efficient distributed replication. In Figure 3, the system 300 uses a combination of synchronous and asynchronous techniques to efficiently replicate data across the cluster. In one embodiment, storage nodes of SC servers 302A-C can each receive an incoming client write 304A-C. In one embodiment, the storage node 302A receives the incoming client write with synchronous replicate module 306A. The synchronous replicate module 306A receives the write 304A and stores the contents of the write in the high-speed storage 312A (e.g., NVRAM, Flash, SSD, etc.). In one embodiment, the synchronous replicate module 306A stores the write contents at the backend of the high-speed storage 312A as described above. A flusher module 308A flushes the contents of the write to the asynchronous replication module 310A. In one embodiment, the flusher module 308A flushes the contents using process 1100 as described in Figure 11 below.

[0082] In one embodiment, the asynchronous replication module 310A stores the content to persistent storage 314A. In this embodiment, the asynchronous replication module 310A can additionally replicate this content to the other storage nodes 302B-C via the asynchronous replication modules 310B-C, respectively. These receiving asynchronous replication modules 310B-C, each store the content in the corresponding local perspective storage 314B-C, respectively.

[0083] To facilitate replication, the block storage layer generates a unique monotonically increasing logical ID for the data/metadata segment system 300 writes. The replication engine uses these logical IDs to synchronize large chunks of segments across different mirrors. This scheme eliminates the need of expensive distributed coordination, metadata comparison, and/or read-modify-write to replicate data and metadata. New incoming writes are logged and synchronously replicated in SSDs or other high-speed storage medium. Periodically, a ‘flusher’ 308A-C moves the logged content asynchronously from the high-speed storage to its permanent location for persistence.

[0084] In a typical replication based system, strong consistency is obtained by enforcing the following:

$$(R+W) > N \quad (2)$$

where ‘R’ and ‘W’ are the minimum number of replicas that must respond correctly to a ‘read’ and ‘write’ respectively, N is the total number of replicas. The value of N does not change once it has been set. This implies that if there are three replicas, R+W must always be greater than 3. In one embodiment, the StorFS system relaxes this requirement and yet provides strong consistency guarantee. For each vNode, the CRM maintains an Active Mirror Set (AMS), which defines the currently active replicas for that vNode. If a replica goes out of sync (e.g. because of failure, etc.), the replica is dropped from AMS. If a replica comes back up, the replica syncs with the corresponding replicas in the AMS and is added back to AMS upon full resync. The modified requirement for strong consistency in StorFS is:

$$(R+W) > |AMS|$$

where |AMS| is the cardinality of the AMS corresponding to a vNode. Thus the, variable sized AMS can provide greater service availability because of the less stringent requirements and without compromising strong consistency guarantees.

Efficient distributed recovery

[0085] In one embodiment, the StorFS system is designed to handle multiple failures. When such a failure happens, the StorFS system recovers from multiple sources in parallel, which significantly reduces the total recovery time. This distributed recovery is made possible in part due to our placement logic that enables the mirrors to find different location of the replicas and stream large chunk of data from those replicas. Also, if a particular replica lags behind the “primary” replica because of failures, system reboot, etc., the StorFS system can resynchronize itself with other active replicas by fetching only those segments whose logical IDs are greater than the ones it has. In this embodiment, there is no need to perform an expensive “diff-and-merge” operation across the distributed set of replicas.

[0086] Figure 13 is a flow diagram of one embodiment of a process 1300 to handle a failure of a component. In one embodiment, process 1300 is performed by the fault tolerance layer 204 of an SC server 200 as described in Figure 2 above. In one embodiment, the failure recovery is performed by each SC server in the StorFS system, which allows for a distributed failure recovery. In this embodiment, the StorFS system is not dependent on a single node for failure recovery. In Figure 13, process 1300 begins by monitoring the StorFS system at block 1302. In one embodiment, process 1300 monitors the storage cluster of the StorFS system for failures of any one of the components. A failure of the component in the StorFS system can be an entire node (e.g., a storage node) or a component of the node (e.g., disk, network, controller, etc.). At block 1304, process 300 determines if a heartbeat of component failed or if an I/O operation failed. In one embodiment, process 1300 could detect a component failure by determining that a heartbeat of component is no longer reporting. For example and in one embodiment, heartbeats are small messages that are exchanged between the storage nodes. If a storage node fails to receive heartbeat message from its peer node, it indicate a potential failure of that peer node. In another embodiment, process 1300 detects an I/O failure if a component of the SC server 200 that is performing process 1300 detects an I/O failure, such as a failure to write an entry to a write log, performing a synchronous replication, performing an asynchronous replication, sending content to persistent storage, monitoring, garbage collection, deduplication, etc.

[0087] If there is no heartbeat or I/O failure is detected, execution proceeds to block 1302. If there is a heartbeat or an I/O failure, at block 1306, process 1300 determined the failed component related to the heartbeat or I/O failure detect at block 1304. In one embodiment, the failed component could be the SC server performing process 1300 or could be a remote SC server. In one embodiment, the failed component could be failed server of hardware component (e.g., disk, network, server, etc.).

[0088] Based on the failed component, process 1300 determines the vNodes impacted by the component failure at block 1310. In one embodiment, process 1300 uses the CRM table to determined which vNodes are impacted by the component failure. In this embodiment, process 1300 determines the whether the entire pNode has failed or a component within a pNode (e.g., disk, controller, network interface, etc.). If pNode fails, all the vNodes in that pNode are impacted. If a pNode component fails, one or some of the vNodes associated with that component are impacted. In one embodiment, the CRM table includes information mapping vNodes to the pNode as well as which vNode is dependent on which component of the pNode (e.g., which disk stores which vNode, etc.). . There can be one or more vNodes impacted by the failed component. Process 1300 additionally performs a processing loop (blocks 1312-1318) to perform a distributed recovery of the vNodes that correspond to the failed component.

[0089] At block 1314, process 1300 updates the AMS of that vNode. In one embodiment, process 1300 marks that vNode as passive, which indicates that the vNode is not an active vNode of the AMS. In one embodiment, a passive vNode is a node that is not available to perform I/O operations. In one embodiment, a passive vNode attempts to catch up via the resync operation as described in Figure 14 below. In this embodiment, once a passive vNode is resynced, the vNode becomes an active vNode. In addition, process 1300 marks the vNode as a candidate for recovery. In one embodiment, recovering a vNode consists of copying data from other active vNode in the passive vNode's AMS, so that the passive vNode can be brought up to date with the current content for that vNode. Process 1300 determines if the number of active mirror vNodes in the AMS for that passive vNode is greater than zero. If the number of active node in the AMS is

greater than zero, execution proceeds to block 1302 above. In one embodiment, the passive vNode is marked for recovery and recovery of that vNode happens synchronously as described in Figure 14 below. If the number of active nodes is zero, at block 1318, process 1300 starts the StorFS recovery. In one embodiment, the StorFS recovery is further described in Figure 14 below. At block 1320, the process 1300 completes the recovery.

[0090] Figure 14 is a flow diagram of one embodiment of a process 1400 to perform a distributed failure recovery of one or more vNodes. In Figure 14, process 1400 begins by performing processing loop (blocks 1402 – 1418) to perform a distributed recovery for each vNode. At block 1404, process 1400 determines if the vNode is marked for recovery. If the vNode is not marked for recovery, execution proceeds to block 1402 above. In one embodiment, a vNode marked for recovery is a vNode that was or is associated with a failed component and the content of that vNode may be out of sync with the content that is to be stored on that vNode. For example and in one embodiment, a SC server could fail and the vNode stored on that SC server would be marked for recovery. If the vNode is marked for recovery, execution proceeds to block 1406.

[0091] At block 1406, process 1400 determines if the number of active mirrors in the AMS for this vNode is greater than the threshold. In one embodiment, if the number of active mirrors is large enough, then that vNode is not necessary for that vNode to be recovered. This means that there are enough active mirrors for the content that is stored in AMS vNode set. If the number of mirrors in the AMS for this vNode is greater than the threshold, execution proceeds to block 1402 above and this vNode is not recovered. If there are enough active vNodes, the passive vNode remains marked for recovery and recovery happens at a later time. If the number of mirrors in the AMS for this vNode is less than or equal to the threshold, process 1400 determines if the node hosting the vNode is alive. In one embodiment, process 1400 could send a heartbeat message to the node to determine if the node is alive. For example and in one embodiment, this node may have temporarily failed and restored itself and is now back up. If the node hosting

vNode is alive, execution proceeds to block 1412, where process 1400 starts the resync process for this vNode.

[0092] If the node hosting the vNode is not alive, at block 1410, process 1400 creates a new passive mirror vNode for this vNode. In one embodiment, this new passive mirror vNode is used for recovery. In one embodiment, process 1400 requests from the CRM to create the passive mirror vNode. In this embodiment, the CRM determines a placement of the new mirror vNode based on the fault domain of the failed vNode. For example and in one embodiment, the CRM places that new mirror vNode in a fault domain that is different from the fault domain of the failed vNode and other vNodes in the AMS set of the failed vNode. Process 1400 further marks this new mirror vNode as passive. In this embodiment, the new passive vNode is the node used for the resync at block 1412 below. Execution proceeds to block 1412 below.

[0093] At block 1414, process 1400 performs a resync for the passive vNode. The passive vNode to be resynced could be the original vNode marked for recovery or a new passive mirror vNode created for the resync. In one embodiment, the resync operation occurs by the passive node hosting the passive mirror vNode sending requests to one, some, or all of the active mirrors in the AMS set for the passive vNode. In one embodiment, the passive mirror fetches the unsync content by segment ID. In one embodiment, the data in the mirrors is stored by segment, which allows for efficient storage of the content for that vNode. In one embodiment, the passive vNode that is being resync may have some the content already stored (e.g., if a node that hosts the failed vNode fails and recovers, this failed vNode may have some or most of the content and is missing the most recent content). In this embodiment, the failed vNode will need the just the missing content. This missing content is identified by the segment ID. By comparing segment ID and not using “diff-and-copy” approach, the resync process is more efficient because comparison of segment IDs is a more efficient process than comparison of the actual stored content. In addition, the segment IDs are stored sequentially, which allows the resync process to determine which is the latest segment ID successfully stored on the passive vNode. Once that is determined, the resync process can copy segment ID greater than the last successful segment ID to the passive vNode.

In one embodiment, the resync process can copy content from different active mirror vNodes. By resyncing using segments, the resync process can use bulk transfers to copy content in logical segment ID order.

[0094] At block 1416, process 1400 checks if the active mirror vNodes and the passive vNode being resync have the same highest segment ID. If these vNodes do have the same highest segment ID, the resync process for this passive vNode is complete and execution proceeds to block 1418. At block 1418, process 1400 marks the passive vNode as active and adds the passive vNode into the AMS. Execution proceeds to block 1402 above, where another vNode is resynced. If these vNodes do not have the same highest segment ID, execution proceeds to block 1414 above.

Automated load balancing across the distributed system of hybrid storage/compute nodes

[0095] In one embodiment, the StorFS system distributes a storage workload across the cluster by stripping it across multiple virtual nodes. Stripping achieves capacity and performance load balancing and provides higher cumulative performance rather than being bottlenecked by one cluster node. One of the characteristics as well as the challenges in a hybrid storage/compute cluster is the constant change in the StorFS systems' behavior. For example, an application's requirements may change; workloads or resources may be added or removed to/from the cluster; and/or there can be a failure in the StorFS system (e.g. a network failure, disk failure, server failure, etc.) that may change the cluster's resource availability. The StorFS system is designed to continuously adapt to these kinds of changes by automatically balancing load across available resources. These resources are of different kinds: CPU, network, memory, storage capacity, storage bandwidth, etc.

[0096] In one embodiment, the StorFS system performs a multi-dimensional optimization across this heterogeneous set of resources. This is facilitated by estimating the requirements of each virtual node, estimating resource availability in each cluster nodes and migrating one or more virtual nodes from one physical node to the other. For example and in one embodiment, if the

network becomes a bottleneck in a physical node (pNode), one or more metadata vNodes in that pNode can be relocated to another pNode that has more network capacity available. Similarly, if a pNode runs out of disk capacity, a new data vNode can be created on other pNodes or a vNode can be relocated to another pNode with more disk capacity. In one embodiment, this migration is achieved by creating a new replica of the source vNode at the destination pNode. The vNode at the destination pNode is populated in parallel from the replicas of the source vNode. While the destination vNode is being updated, the source vNode may change due to new I/O from the client. This new I/O is be appended to the source vNode and assigned new logical sequence ID. The destination vNode will continuously sync from the source based on these sequence IDs. Once they are in sync, the vNode to pNode mapping in the Cluster Manager's routing table is updated so that I/Os are routed to the new location. This ensures that the storage service remains uninterrupted when a vNode is being moved. This load distribution happens on a continuous basis and ensures that resources are utilized optimally and each workload receives the resource it need.

Policy-aware data/metadata distribution and placement in a distributed system of hybrid storage/compute nodes

[0097] In one embodiment, the StorFS system is capable of supporting different policies at different level of granularity: at the file system level, at the individual file level, etc. These policies specify the performance, reliability, and other quality of service requirements of the applications. The system translate these high level polices to low-level system configuration. For example, the StorFS system is able to automatically select the type of storage medium (SSD v. HDD) based on application performance requirement. Similarly, based on these policies, the StorFS system can decide whether to compress or de-duplicate the data, the level of mirroring, etc.

[0098] In one embodiment, the StorFS system is capable of supporting different policies at different levels of granularity configured by the user. The granularity could be at the level of an

entire file system or an individual file or a set of files (e.g., a set of files representing a VM container). These policies specify the performance, reliability, availability and other quality of service (QoS) requirements of the applications per object. The system translates the high level policies to the low-level system configuration. Following are the policy translations:

[0099] *Performance policies:* In one embodiment, the StorFS system performs automatic placement of the data in different tiers of storage hierarchy like memory, SSD or HDD. In addition, the StorFS system can also enforce QoS to the storage I/Os based on the performance policies. In order to achieve this StorFS groups application data in "data vNodes". Each data vNode has one or more policies associated with them that is stored in the CRM database and is accessible from any nodes in the StorFS cluster. I/O and data belonging to an application are tagged with the ID of the data vNode that matches with their policy requirements. The system uses these tags to direct I/O to the physical nodes where that vNode is located.

[00100] *Availability policies:* In one embodiment, an availability policy determines the availability of StorFS storage service upon failures. StorFS implements an availability policy by placing the application data in the data vNodes that match their availability policies. Availability of the data vNodes is controlled by configuring the number of mirrors or the type of storage.

[00101] *Efficiency policies:* Based on the efficiency requirements and the system state, the StorFS system can enable or disable compression or deduplication for the users' data. On one hand, compression and deduplication reduces space usage and the number of I/Os to the storage media, but on the other hand it increases the CPU and memory overhead of the system. The StorFS system achieves a balance between the overheads and space/IO savings by tuning the compression and deduplication operations based on the user defined policies and the monitored system state.

Distributed deduplication

[00102] Deduplication refers to removing duplicate data blocks from the storage system, which reduces the space usage and hence, the cost of the StorFS system. There are many

approaches to achieve deduplication: at the file-level or at the block-level, inline vs. offline, etc. Single node deduplication systems are relatively easier to build because the metadata associated with deduplication is located in one place. Distributed dedup across multiple storage system nodes is harder because the metadata may not be local. Data block deletion requires coordination with multiple nodes to make sure that there are no local and remote references to the data block that is being deleted.

[00103] In one embodiment, the StorFS system uses multiple ways to efficiently dedup our storage system. First, instead of storing full dedup metadata, the StorFS system stores a partial Dedup table as 'hints' in fast SSD storage. In this embodiment, a matching 'hint' indicates a potential duplicate. This hint stored in a SSD not only reduces the metadata storage cost but also the lookup time. Second, the StorFS system uses a probabilistic data structure to detect the references to the data blocks. If a data block is not referenced by a node, the data block becomes candidate for removal.

[00104] Figure 6 is an illustration of one embodiment of a deduplication module 602 and garbage collection module 604. In one embodiment, the dedup analyzer generates the dedup hint table by analyzing the streams of new objects that are written to the storage. In one embodiment, the hint table consists of 'content fingerprint' to the object's VBA. The goal is to store a content fingerprint for those VBAs of those objects that are likely to be deduped. In one embodiment, one of the ways this can be achieved is by keeping track of top-K statistics for the objects. In one embodiment, the top-K statistic is a partial ordering method where the top K elements of the set are identified and ordered instead of ordering the entire set. For example, top 10 most sold cars make/model out of all types of cars are identified and sorted. In one embodiment, the writelog flusher checks the dedup table to avoid storing duplicates to the persistent storage.

[00105] In the flusher operation, the StorFS system checks to determine if the data being moved to persistent storage is duplicated elsewhere in the system using a hints table. Figure 12 is a flow diagram of one embodiment of a process 1200 to prepare a hint table that is used for data deduplication. In one embodiment, process 1200 builds that hint table on a VBA basis. In one

embodiment, the hints table is a map for a fingerprint and a VBA is the StorFS system and a reference count. In Figure 12, process 1200 receives the VBA to be analyzed at block 1202. At block 1202, process 1200 extracts a fingerprint for this VBA using a method known in the art (e.g., SHA-1 fingerprint, etc.). At block 1208, process 1200 determines if this fingerprint is in the top-K statistics. If the fingerprint is in the top-K statistics, at block 1216, process 1200 updates the reference count for this fingerprint in the top-K.

[00106] If the fingerprint is not in the top-K statistics, at block 1210, process 1200 decrements the reference count of the fingerprints in the top-K statistics. At block 1212, process 1200 determines if the reference count for an fingerprint is zero, and if so, process 1200 removes that fingerprint and replaces the zero referenced fingerprint with the new fingerprint. At block 1214, process 1200 resets the reference count for each zero referenced fingerprint to be one.

Distributed garbage collection

[00107] In one embodiment, the StorFS system writes incoming data to a new location on the persistent storage. This means that when an object or a logical offset within a file gets overwritten, the old object or file content needs to be removed (e.g. cleaned or garbage collected.) Because of this, the StorFS system needs to determine a list of valid (e.g., live) objects. Due to the presence of snapshots, clones and deduplication, multiple files, or file systems may reference the same data object. In one embodiment, the StorFS system traverses the metadata tree to compile a list of valid data objects, which form the leaf of the metadata tree. Because a cluster may contain billions of valid objects, this list can be gigantic and would not fit in the main memory. To address this issue, the StorFS system uses a space-efficient probabilistic data structure to store the list of valid objects. For example and in one embodiment, a bloom filter can be used to store the list of valid objects. The bloom filter permits membership tests with very little memory compared to the number of object entries it stores. The cleaning process involves the following steps:

- Generate the live object map: Traverse the metadata tree and populate the bloom filter
- Segment cleaner: Scan the block segments to check the number of live objects they contain and generate a segment summary statistic
- Compact segments with low utilization based on segment summary statistics

Efficient Metadata Traversal:

[00108] Metadata traversal can cause a lot of random I/Os that can lead to very high overhead in HDD. File metadata, for example, is organized in a hierarchical tree format to locate data blocks on the persistent storage. In one embodiment, the StorFS system employs a smart traversal algorithm that efficiently reads the whole metadata tree with very little random I/O to the HDD. If there is sufficient space on the faster flash drive to hold the metadata vNode, the StorFS system copies the entire metadata vNode onto the faster flash drive. The random read I/Os during the traversal are sent to faster flash drives, which have much higher random read performance compared to HDD. In addition, the StorFS system caches the top few levels of the metadata tree in the DRAM to further expedite the tree traversal.

[00109] The live object map can be generated for the complete metadata tree using the algorithm shown in Figure 4. This makes sure that all the live objects for the tree are captured in the corresponding bloom filter. Figure 4 is a flow diagram of one embodiment of a process 400 to perform a full metadata tree walk for garbage collection. In one embodiment, this walk is performed by the garbage collection module 604 as shown in Figure 6. In Figure 4, process 400 begins by performing a processing loop (blocks 402-418) to perform the metadata tree walk for each metadata vNode. At block 404, process 400 determines the number of data keys that could be potentially inserted into a Bloom Filter. In one embodiment, the number of data keys that could be inserted could be based on an estimate from the previous run of the algorithm or some information provided by metadata file tree about the number of keys in the File System.

[00110] Process 400 instantiates a bloom filter to manage the data keys at block 406. In one embodiment, this bloom filter is called the D-Bloom. At block 408, process 400 instantiates

a bloom filter to manage and clean the Metadata vNode, which are metadata keys. In one embodiment, this bloom filter is the M-Bloom and the M-Bloom filter can be up to an order of magnitude smaller than the D-Bloom. Process 400 creates a cache to store intermediate metadata blocks for filetree traversal at block 410.

[00111] At block 412, process 400 obtains the head segment ID of the Snapshot being cleaned and the head segment ID of the last cleaned snapshot. The later is referred to as the tail segment. Process 400 fetches the current list of blocks being deduplicated from the dedup module 414. In one embodiment, for each entry in the dedup list with reference count greater than zero that reside between Head Segment and Tail Segment, process 400 adds this entry to the D-Bloom list. In addition, at block 414, process 400 removes those entries from the Dedup Module whose reference count is equal to zero. For example and in one embodiment, the Dedup Module is the Dedup Module 602 as described in Figure 6 above. In one embodiment, process 400 removes these entries by sending a request to the Dedup module. At block 416, process 400 traverses the metadata tree. In one embodiment, process 400 traverses the metadata tree by for each entry that is a metadata key, adding it to the M-Bloom filter and for each entry that is a data key, adding it to the D-Bloom filter.

[00112] In one embodiment, this complete metadata traversal, however, comes at a cost. In one embodiment, instead of walking the whole tree each time, StorFS implements an approach to incrementally clean the segments written in a particular time duration (e.g. in a day.) Consider the scenario depicted above, where the full FT cleaner is executed at (t-24) hours. The segments written between times (t-24) and (t) can be of three types:

- Segments/objects that were compacted and moved by the segment cleaner.
- Segments/objects that were written because of the incoming writes from the client and are still valid.
- Segments/objects that were written because of the incoming writes from the client, but were invalidated by subsequence writes in that time window.

[00113] Figure 5 is a flow diagram of a process to perform an incremental metadata tree walk for garbage collection. In one embodiment, the garbage collection module 604 performs process 500. In Figure 5, process 500 begins by performing a processing loop (block 502-518) to incrementally walk the metadata tree for each metadata vNode. At block 504, process 500 determines the number of data keys that could be potentially inserted into a bloom filter. In one embodiment, the number of data keys that could be inserted is based on an estimate from the previous run of the algorithm or some information provided by the FT about the number of keys in the File System. At block 506, process 500 instantiates a bloom filter to manage the data keys. In one embodiment, this bloom filter is called the D-Bloom. Process 500 instantiates a bloom filter to manage and clean the Metadata vNode, which includes the metadata keys at block 508. In one embodiment, this bloom filter is called the M-bloom. In one embodiment, the size of the M-bloom is up to an order of magnitude smaller than the D-bloom bloom filter.

[00114] At block 510, process 500 creates a cache to store intermediate metadata blocks for filetree traversal. Process 500 obtains a head segment ID and the generation number of the snapshot being cleaned now at block 512. In one embodiment, the head segment ID and the generation number are recorded along with other snapshot attributes when the snapshot is written to the persistent storage. Process 500 obtains the head segment ID and the generation number from the snapshot attributes. In addition, process 500 obtains the head segment ID and the generation number of the last cleaned snapshot. In one embodiment, the head segment ID of the last cleaned snapshot is called the tail segment.

[00115] At block 514, process 500 fetches the current list of blocks being deduplicated. In one embodiment, process 500 fetches this list from the dedup module. In one embodiment, at block 514, process 500 further, for each entry in the dedup list with a reference count greater than zero that resides between Head Segment and Tail Segment, adds that entry to the D-Bloom list. In addition, process 500, for each entry with a reference count that is equal to zero, removes that entries. In one embodiment, process 500 removes the entry by sending a request to the dedup module.

[00116] Process 500 traverse those portions of the metadata tree with a generation number that is between the head generation number and the tail generation number. During the traverse, is the entry is a metadata key, process 500 adds that entry to the M-Bloom bloom filter. If the entry is data key, process 500 adds that entry to the D-Bloom bloom filter.

[00117] In one embodiment, the incremental cleaner cleanses this last category of the segments/objects. In order to achieve this, the StorFS system stores a “generation number” in the metadata tree for each object it writes. The incremental cleaner traverses those portions of the metadata tree whose generation number that is greater than the last generation number it cleaned and populated the live object map. In this embodiment, the segment cleaner cleanses those segments that were written during that time interval. In one embodiment, a system administrator can configure when an incremental full cleaners are ran. For example and in one embodiment, a system administrator may set policies to run the incremental cleaner on weekdays and full cleaner on weekends.

Segment Cleaner

[00118] In one embodiment, once the Live Object Map has been created, the segment cleaner scans the block segments to check the number of live objects they contain and generates a segment summary statistic of the utilization of the segments, (e.g. the ratio of number of the live-objects to the total number of objects in the segment.) In one embodiment, the summary statistics can be built in two ways: (1) by a complete block segment scan and (2) using and in essential statistics update. In one embodiment, the complete block segment scan operates by having the cleaner iterates over all the segments to check for valid objects. This approach has more overhead because of a large number of I/Os the approach generates.

[00119] In one embodiment, in the incremental summary statistics update, the cleaner uses the dead object hint log generated by the ‘Write Log Flusher’. The objects in this log are tested for their membership in the Live Object Map. If the test is negative, the cleaner confirms that the

object is actually dead. In addition, the cleaner updates its segment summary statistics by decrementing the utilization.

Segment Compaction

[00120] In one embodiment, compact segmentation compacts segments whose utilization drops below a certain threshold. In this embodiment, each segment can contain 4 types of entities: Live objects, dead objects, object tombstone entry, and segment tombstone entry. These entities are dealt as follows during compaction:

- Tombstone entries are discarded and need not be carried forward.
- Live objects are copy forwarded to the new segments and the corresponding index for that object is updated.
- A tombstone entry is added for each dead objects
- A segment tombstone is added for the segment being compacted.

Efficient and distributed data services

[00121] Data services like snapshot, cloning, backup, disaster recovery, etc., can be critical for any enterprise storage systems. A naïve way to implement such data services is to perform a full copy of the current storage. This is, however, very inefficient and takes lot of time and resources. More advanced storage systems implement these data services by keeping track of changes since the clones (or snapshots) were created instead of full copy. This consumes less storage resources. Single node data services are relatively easier to build, as the metadata is readily available in one central place. Building efficient data services in a distributed storage system poses significant challenges.

[00122] In one embodiment, the StorFS system uses techniques to efficiently track changes in a distributed setting. For example and in one embodiment, the StorFS is capable of performing low overhead data services at different level of granularity: from the file system level to the individual files level. As discussed above, the StorFS system stripes file and file system

data across multiple storage nodes in our cluster. Snapshotting a file, for example, involves creating a small metadata entry in the storage vNodes containing that file. This stripe process is extremely fast and require minimal amount of resources. There is no data copy involved. Furthermore, the StorFS permits creating large number of snapshots (or clones) without any deterioration in performance.

Differentiated service at different level of granularities

[00123] In one embodiment, the StorFS system is capable of providing differentiated quality of service at different level of granularities (at the file level, at the file system level, at the volume level, etc.) Users can configure the cost, performance, consistency, availability, RAID, compression, deduplication and other parameters associated with their application data. Different components in the StorFS system software stack have been designed to act in unison and translate high-level application policies to low-level system parameters that control how the data and I/O from different application need to be managed. For example and in one embodiment, the SC client or the dispatcher implements a policy driven distributed I/O scheduler for processing the incoming I/O requests. The replacement logic in the caching layer is designed to handle priorities of different application data. The replication logic can automatically determine the characteristics of replication (distributed RAID level, fault domains, etc.) based on the application availability and consistency requirements.

Dynamic QoS management for heterogeneous and concurrent storage workloads

[00124] The QoS requirements across different applications may not only vary but also change over time. Even though the underlying storage and compute resources are shared, the StorFS system is able to manage the QoS across different heterogeneous and concurrent workloads, each of which may have different requirements. The key idea is to cluster application data in to logical unit called 'vNodes'. A 'vNode' may contain either data or metadata. Data

vNode are associated with different policies. Data belonging to an application are placed in appropriate vNodes depending on the policies associated with them.

Continuous optimization and availability

[00125] As described above, storage systems are dynamic: workload may change, resource availability may change, and/or failures may happen. All of these require storage systems to continuously adapt and optimize to ensure service availability, performance and quality of service. In one embodiment, the StorFS system is designed to actively monitor the entire cluster state and adapt itself with changing conditions. Specifically, it keeps track of CPU, memory, network, IOPS, failure rate, and health of individual components. Based on these attributes, the StorFS system evaluates a multi-dimensional function to decide whether to relocate workload from one storage node to the other. In one embodiment, the relocation is done at the vNode level. In this embodiment, a VNode provides a well-defined abstraction to deal with system imbalance and resource addition/deletion and failures. When a vNode is relocated, all the traffic associated with that vNode is rerouted as well. A vNode can also be migrated on disk or storage node failure.

Intelligent prefetching

[00126] Serving I/O requests from cache is much faster than serving from disks. Therefore many storage systems prefetch data blocks when they observe some sequentiality in the I/O pattern in anticipation that those prefetched blocks are going to be accessed by the application in the near future. However, in many cases, I/O patterns are not sequential, but are still deterministic. In one embodiment, the StorFS system detects those deterministic patterns, store them as signature in its database, and at runtime, prefetches data blocks based on these signatures. This technique significantly increases the cache hit rate and reduces the overall latency. In addition, and in one embodiment, when an application is starting up (e.g., virtual machine booting.), the StorFS system uses the previously observed booting pattern to prefetch

the data blocks. This is done by capturing the sequence of recent data access, computing a signature of those accesses and looking up the signature in the signature database. If there is a match, the StorFS system prefetches the data blocks associated with that signature. If there is no match, the StorFS system tracks the subsequent data access patterns and inserts an entry in the signature database with the new signature and data access patterns.

[00127] Figure 15 shows one example of a data processing system 1500, which may be used with one embodiment of the present invention. For example, the system 1500 may be implemented including a physical server 102A-C as shown in Figure 1. Note that while Figure 15 illustrates various components of a computer system, it is not intended to represent any particular architecture or manner of interconnecting the components as such details are not germane to the present invention. It will also be appreciated that network computers and other data processing systems or other consumer electronic devices, which have fewer components or perhaps more components, may also be used with the present invention.

[00128] As shown in Figure 15, the computer system 1500, which is a form of a data processing system, includes a bus 1503 which is coupled to a microprocessor(s) 1505 and a ROM (Read Only Memory) 1507 and volatile RAM 1509 and a non-volatile memory 1511. The microprocessor 1505 may retrieve the instructions from the memories 1507, 1509, 1511 and execute the instructions to perform operations described above. The bus 1503 interconnects these various components together and also interconnects these components 1505, 1507, 1509, and 1511 to a display controller and display device 1515 and to peripheral devices such as input/output (I/O) devices which may be mice, keyboards, modems, network interfaces, printers and other devices which are well known in the art. Typically, the input/output devices 1515 are coupled to the system through input/output controllers 1517. The volatile RAM (Random Access Memory) 1509 is typically implemented as dynamic RAM (DRAM), which requires power continually in order to refresh or maintain the data in the memory.

[00129] The mass storage 1511 is typically a magnetic hard drive or a magnetic optical drive or an optical drive or a DVD RAM or a flash memory or other types of memory systems, which

maintain data (e.g. large amounts of data) even after power is removed from the system.

Typically, the mass storage 1511 will also be a random access memory although this is not required. While Figure 15 shows that the mass storage 1511 is a local device coupled directly to the rest of the components in the data processing system, it will be appreciated that the present invention may utilize a non-volatile memory which is remote from the system, such as a network storage device which is coupled to the data processing system through a network interface such as a modem, an Ethernet interface or a wireless network. The bus 1503 may include one or more buses connected to each other through various bridges, controllers and/or adapters as is well known in the art.

[00130] Portions of what was described above may be implemented with logic circuitry such as a dedicated logic circuit or with a microcontroller or other form of processing core that executes program code instructions. Thus processes taught by the discussion above may be performed with program code such as machine-executable instructions that cause a machine that executes these instructions to perform certain functions. In this context, a “machine” may be a machine that converts intermediate form (or “abstract”) instructions into processor specific instructions (e.g., an abstract execution environment such as a “process virtual machine” (e.g., a Java Virtual Machine), an interpreter, a Common Language Runtime, a high-level language virtual machine, etc.), and/or, electronic circuitry disposed on a semiconductor chip (e.g., “logic circuitry” implemented with transistors) designed to execute instructions such as a general-purpose processor and/or a special-purpose processor. Processes taught by the discussion above may also be performed by (in the alternative to a machine or in combination with a machine) electronic circuitry designed to perform the processes (or a portion thereof) without the execution of program code.

[00131] The present invention also relates to an apparatus for performing the operations described herein. This apparatus may be specially constructed for the required purpose, or it may comprise a general-purpose computer selectively activated or reconfigured by a computer program stored in the computer. Such a computer program may be stored in a computer readable

storage medium, such as, but is not limited to, any type of disk including floppy disks, optical disks, CD-ROMs, and magnetic-optical disks, read-only memories (ROMs), RAMs, EPROMs, EEPROMs, magnetic or optical cards, or any type of media suitable for storing electronic instructions, and each coupled to a computer system bus.

[00132] A machine readable medium includes any mechanism for storing or transmitting information in a form readable by a machine (e.g., a computer). For example, a machine readable medium includes read only memory (“ROM”); random access memory (“RAM”); magnetic disk storage media; optical storage media; flash memory devices; etc.

[00133] An article of manufacture may be used to store program code. An article of manufacture that stores program code may be embodied as, but is not limited to, one or more memories (e.g., one or more flash memories, random access memories (static, dynamic or other)), optical disks, CD-ROMs, DVD ROMs, EPROMs, EEPROMs, magnetic or optical cards or other type of machine-readable media suitable for storing electronic instructions. Program code may also be downloaded from a remote computer (e.g., a server) to a requesting computer (e.g., a client) by way of data signals embodied in a propagation medium (e.g., via a communication link (e.g., a network connection)).

[00134] The preceding detailed descriptions are presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These algorithmic descriptions and representations are the tools used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of operations leading to a desired result. The operations are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

[00135] It should be kept in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the above discussion, it is appreciated that throughout the description, discussions utilizing terms such as “receiving,” “determining,” “transmitting,” “computing,” “sending,” “forwarding,” “dispatching,” “detecting,” “performing,” “scheduling,” “communicating,” “reading,” “writing,” “transferring,” “updating,” “returning,” “merging,” “appending,” “fetching,” or the like, refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

[00136] The processes and displays presented herein are not inherently related to any particular computer or other apparatus. Various general-purpose systems may be used with programs in accordance with the teachings herein, or it may prove convenient to construct a more specialized apparatus to perform the operations described. The required structure for a variety of these systems will be evident from the description below. In addition, the present invention is not described with reference to any particular programming language. It will be appreciated that a variety of programming languages may be used to implement the teachings of the invention as described herein.

[00137] The foregoing discussion merely describes some exemplary embodiments of the present invention. One skilled in the art will readily recognize from such discussion, the accompanying drawings and the claims that various modifications can be made without departing from the spirit and scope of the invention.

CLAIMS

What is claimed is:

1. A non-transitory machine-readable medium having executable instructions to cause one or more processing units to perform a method to dispatch an input/output request in a software defined distributed storage system, the method comprising:
 - receiving, at a storage controller client, the input/output request, wherein the software defined distributed storage system includes the storage controller client, a plurality of storage controller servers, a plurality of virtual nodes distributed among a plurality of physical nodes, and each of the plurality of physical nodes is hosted on one of the plurality of storage controller servers;
 - computing a target virtual node for the input/output request, wherein the target virtual node is one of the plurality of virtual nodes;
 - determining a target physical node that corresponds to the target virtual node, wherein the target physical node is one of the plurality of physical nodes; and
 - dispatching the input/output request to a target storage controller server that corresponds to the target physical node, wherein the target storage controller server is one of the plurality of storage controller servers.
2. The non-transitory machine-readable medium of claim 1, wherein the input/output request is selected from the group consisting of a write operation request, a read operation request, and a query operation request.
3. The non-transitory machine-readable medium of claim 1, wherein the computing of the target virtual node is computed using a deterministic placement function.
4. The non-transitory machine-readable medium of claim 1, wherein the storage controller client and the target storage controller are on the same device.

5. The non-transitory machine-readable medium of claim 1, wherein the storage controller client and the target storage controller are on different devices that are coupled together by a network.
6. The non-transitory machine-readable medium of claim 1, wherein the determining a target physical node comprises:
- performing a lookup with the target virtual node for the target physical node using a storage routing table.
7. The non-transitory machine-readable medium of claim 1, wherein the dispatching comprises:
- scheduling a dispatch of the input/output request to the target using a policy based on a characteristic of the input/output request; and
 - transmitting the input/output request to the target storage controller server based on the scheduling.
8. The non-transitory machine-readable medium of claim 7, wherein the policy is selected from the group consisting of a Quality of Service policy, a Differentiated Services policy, and a load balancing policy.

9. A non-transitory machine-readable medium having executable instructions to cause one or more processing units to perform a method to write data in a software defined distributed storage system, the method comprising:

receiving, at one of a plurality of storage controller servers, a write operation request to store data, wherein the write operation request is dispatched from a storage controller client to the one of the plurality of storage controller servers using a deterministic placement function, and the software defined distributed storage system includes the storage controller client, and the plurality of storage controller servers; and

in response to receiving the write operation request, writing the data to a write log in an active log partition of the one of the plurality of storage controller servers.

10. The non-transitory machine-readable medium of claim 9, wherein the method further comprises:

writing the data to one or more mirror log partitions.

11. The non-transitory machine-readable medium of claim 10, wherein the mirror log partitions are on different ones of the plurality of storage controller servers than that storage controller that stores the active log partition.

12. The non-transitory machine-readable medium of claim 9, wherein the writing of the data in the active log partition comprises:

appending the data to the active log partition.

13. The non-transitory machine-readable medium of claim 9, wherein the write log is stored on a fast physical storage device.

14. The non-transitory machine-readable medium of claim 9, wherein the active log partition is stored in a fast storage of the one of a plurality of storage controller servers.

15. The non-transitory machine-readable medium of claim 14, wherein the fast storage is selected from the group consisting of a solid-state drive and flash memory.

16. The non-transitory machine-readable medium of claim 9, wherein the method further comprises:

if the active log partition is full after the writing of the data, transferring the data to persistent storage.

17. The non-transitory machine-readable medium of claim 9, wherein the method further comprises:

validating the write operation request.

18. The non-transitory machine-readable medium of claim 9, wherein the method further comprises:

updating a delta tree for the write log, wherein the delta tree represents changes to the one of the plurality of storage controller servers that has been committed to that storage controller server but not transferred to a persistent storage.

19. A non-transitory machine-readable medium having executable instructions to cause one or more processing units to perform a method to process a read operation request in a software defined distributed storage system, the method comprising:

receiving, at one of a plurality of storage controller servers, the read operation request, wherein the write operation request is dispatched from a storage controller client to the storage controller server using a deterministic placement function, and the software defined distributed storage system includes the storage controller client and the plurality of storage controller servers; and

in response to receiving the read operation request,

reading data indicated in the read operation request, wherein at least part of the data is stored in a write log that is in an active log partition of the one of the plurality of storage controller servers.

20. The non-transitory machine-readable medium of claim 19, wherein the method further comprises:

returning the data to the storage controller client.

21. The non-transitory machine-readable medium of claim 19, wherein another part of the data is stored in persistent storage.

22. The non-transitory machine-readable medium of claim 21, wherein the reading the data comprises:

reading the part of the data that resides in the write log;

reading the another part of the data that is in persistent storage; and

merging the part of the data and the another part of the data.

23. The non-transitory machine-readable medium of claim 19, wherein an entirety of the data is stored in the write log.

24. The non-transitory machine-readable medium of claim 23, wherein the reading of the data comprises:

reading the entirety of the data from the write log.

25. The non-transitory machine-readable medium of claim 19, wherein the active log partition is stored in a fast storage of the one of a plurality of storage controller servers.

26. The non-transitory machine-readable medium of claim 25, wherein the fast storage is selected from the group consisting of a solid-state device and flash memory.

27. The non-transitory machine-readable medium of claim 19, wherein the read operation is a read data operation to read content stored in the storage controller server.

28. The non-transitory machine-readable medium of claim 25, wherein the read operation is a query operation to query a status about content stored in the storage controller server.

29. A non-transitory machine-readable medium having executable instructions to cause one or more processing units to perform a method to flush data stored in a write log to persistent storage, the method comprising:

determining if the write log is full; and

if the write log is full,

for each current virtual node in the write log,

computing a new first virtual node for data of the current virtual node,

reading data of the current virtual node, and

writing the data to the new first virtual node.

30. The non-transitory machine-readable medium of claim 29, wherein the for each current virtual node in the write log further includes:

if that current virtual node includes metadata,

computing a new second virtual node for the metadata of the current virtual node,

reading the metadata for the current virtual node, and

writing the metadata to the new second virtual node.

31. The non-transitory machine-readable medium of claim 30, wherein the reading the metadata comprises:

fetching a portion of an on-disk metadata tree that is relevant to the current virtual node;

and

merging the portion of the on-disk metadata tree with a portion of a delta tree that corresponds to the current virtual node.

32. A non-transitory machine-readable medium having executable instructions to cause one or more processing units to perform a method to synchronously replicate data to a plurality of storage controller servers, the method comprising:
- receiving a write log entry on one of the plurality of storage controller servers;
 - retrieving a target virtual node for the write log entry;
 - determining a plurality of virtual mirror nodes for the target virtual node, wherein each of the mirror nodes is located on different ones of the plurality of controller servers; and
 - synchronously sending the write log entry to each of the different ones of the plurality of storage controller servers.
33. The non-transitory machine-readable medium of claim 32, further comprising:
- writing the write log entry to the target virtual node.
34. The non-transitory machine-readable medium of claim 32, wherein the write log entry include virtual node identifier, a timestamp, a checksum, and data to be stored.
35. The non-transitory machine-readable medium of claim 32, wherein a virtual node is a selected from the group consisting of a collection of data objects and a collection of metadata objects.
36. The non-transitory machine-readable medium of claim 35, wherein a data object is selected from the group consisting of a file, an object, and a key-value pair.
37. The non-transitory machine-readable medium of claim 32, wherein the determining the plurality of virtual mirror nodes for the virtual node comprises:
- retrieving the plurality of virtual mirror nodes from an active mirror set for the virtual node.
38. The non-transitory machine-readable medium of claim 32, wherein the active mirror set is a set of one or more active virtual nodes that are used to mirror data stored in the virtual node.

39. The non-transitory machine-readable medium of claim 37, wherein an active mirror virtual node is a mirror node that stores the same data as stored on the target virtual node.
40. The non-transitory machine-readable medium of claim 32, further comprising:
receiving an acknowledgement for a writing of the write log entry from the each of the different ones of the plurality storage controller servers.

41. A non-transitory machine-readable medium having executable instructions to cause one or more processing units to perform a method to asynchronously replicate data for persistent storage to a plurality of storage controller servers, the method comprising:

receiving data, metadata, and target virtual node on one of the plurality of storage controller servers;

determining a plurality of virtual mirror nodes for the target virtual node, wherein each of the mirror virtual nodes is located on different ones of the plurality of controller servers; and

asynchronously sending the data and metadata to each of the different ones of the plurality storage controller servers, wherein the data and metadata are stored in a persistent storage of a respective plurality of storage controller servers.

42. The non-transitory machine-readable medium of claim 41, wherein a virtual node is a selected from the group consisting of a collection of data objects and a collection of metadata objects.

42. The non-transitory machine-readable medium of claim 42, wherein a data object is selected from the group consisting of a file, an object, and a key-value pair.

43. The non-transitory machine-readable medium of claim 41, wherein the determining the plurality of virtual mirror nodes for the virtual node comprises:

retrieving the plurality of virtual mirror nodes from an active mirror set for the virtual node.

44. The non-transitory machine-readable medium of claim 41, wherein the active mirror set is a set of one or more active virtual nodes that is used to mirror data stored in the virtual node.

45. The non-transitory machine-readable medium of claim 44, wherein an active mirror virtual node is a mirror node that stores the same data as stored on the target virtual node.

46. A non-transitory machine-readable medium having executable instructions to cause one or more processing units to perform a method to determine duplicative data in a distributed storage system, the method comprising:

determining, at a first one of a plurality of storage controller servers, if a first entity is duplicated in the distributed storage system, wherein a second entity is deduplicated by a second one of the plurality storage servers in the distributed storage system, the distributed storage system includes the plurality of storage controller servers, and the determining if the first entity is duplicated includes

receiving, at the first one of a plurality of storage controller servers, the first entity to be stored in the distributed storage system,
looking up the first entity in a data deduplication table,
if the first entity exists in the data deduplication table,
updating metadata for the first entity to indicate a vNode that stores a duplicate of the first entity.

47. The non-transitory machine-readable medium of claim 46, wherein the data deduplication table includes hints about which entities are stored in the distributed storage system.

48. The non-transitory machine-readable medium of claim 47, wherein each of the hints is a fingerprint of an entity that is stored in the distributed storage system.

49. The non-transitory machine-readable medium of claim 48, wherein the looking up the first entity comprises:

computing a fingerprint for the first entity; and
looking up the fingerprint in the data deduplication table.

50. The non-transitory machine-readable medium of claim 46, wherein the data deduplication table includes hints about which of a top-K entities are stored in the distributed storage system.

51. The non-transitory machine-readable medium of claim 50, wherein the top-K entities are less than a total number of entities stored in the distributed storage system.

52. The non-transitory machine-readable medium of claim 46, wherein the data deduplication table is globally available to the plurality of storage controller servers for data deduplication determinations.

53. The non-transitory machine-readable medium of claim 46, further comprising:
building the data deduplication table.

54. The non-transitory machine-readable medium of claim 53, wherein the building of the data deduplication table comprises:

for each of a plurality of stored entities,

computing a current fingerprint for that stored entity,

if the current fingerprint is in data deduplication table,

incrementing a reference count for this fingerprint, and

if the current fingerprint is not in the data deduplication table,

decrementing reference counts of the fingerprints in the data deduplication
table,

removing zero reference count fingerprints,

adding the current fingerprint to the data deduplication table, and

setting a reference count for the current fingerprint to be one.

55. A non-transitory machine-readable medium having executable instructions to cause one or more processing units to perform a method to handle a failure of a component of the distributed storage system, the method comprising:

detecting, by one of a plurality of storage controller servers, a component failure in the distributed storage system;

determining a virtual node that is impacted by the component failure;

retrieving an active mirror set for the impacted virtual node;

if a number of active mirror virtual nodes identified in the active mirror set is less than a threshold,

allocating a new virtual node, and

recovering stored data from the active mirror set for the new virtual node.

56. The non-transitory machine-readable medium of claim 55, wherein the active mirror set is a plurality of active virtual nodes that are used to mirror data stored in the virtual node.

57. The non-transitory machine-readable medium of claim 56, wherein each of the plurality of active virtual nodes is in a different fault domain, wherein a failure of a component for one of the plurality of active virtual node does not affect others of the plurality of active virtual nodes.

58. The non-transitory machine-readable medium of claim 57, wherein the new virtual node is allocated in a fault domain different from one of the fault domains used by the plurality of active virtual nodes.

59. The non-transitory machine-readable medium of claim 55, wherein an active virtual node is a virtual node that includes an up to date stored data for the active mirror set.

60. The non-transitory machine-readable medium of claim 55, wherein the recovering data comprises:

copying the stored data from an active virtual node of the active mirror set to the new virtual node, wherein after the copying, the active virtual node and the new virtual node have the same stored data.

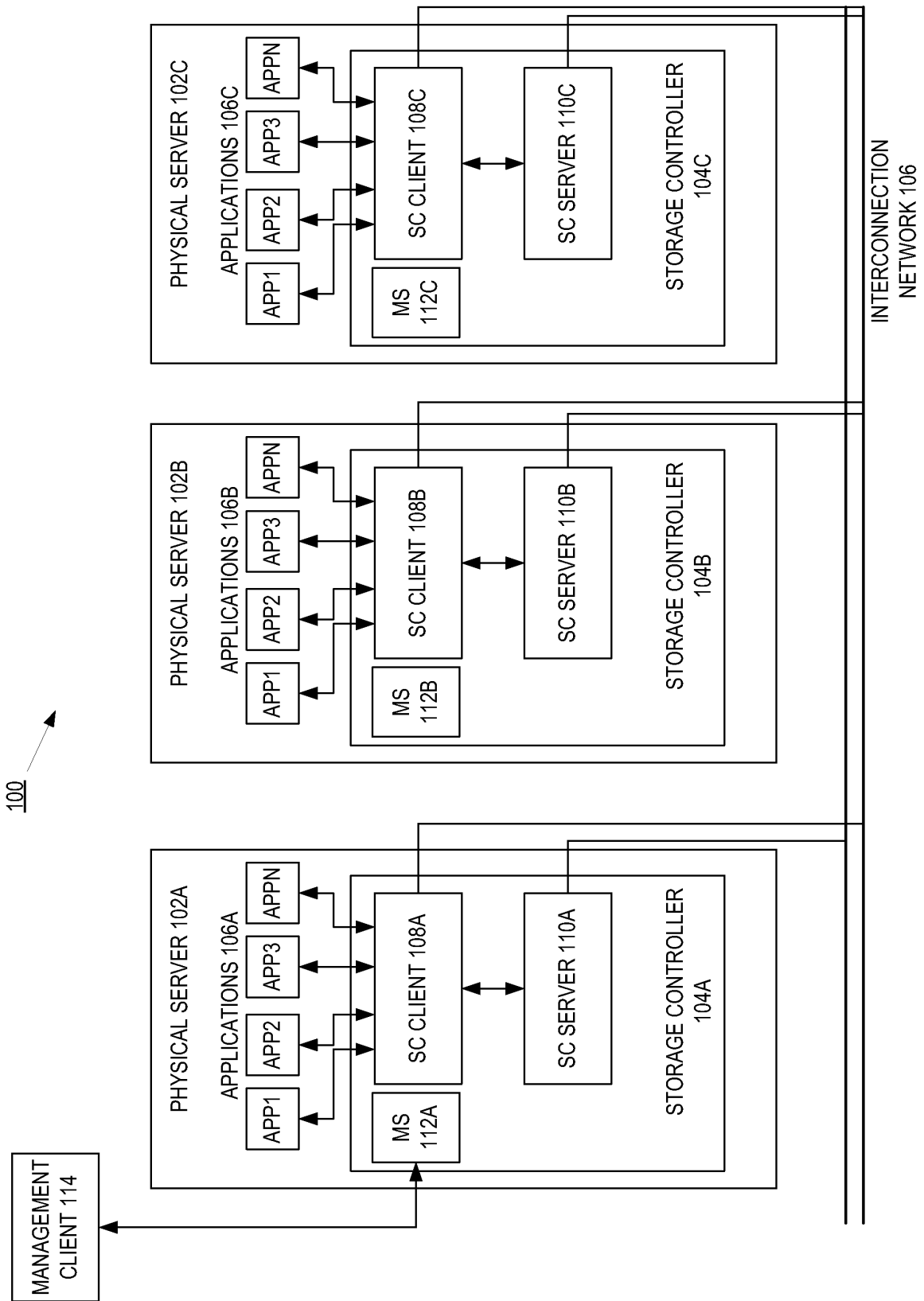
61. The non-transitory machine-readable medium of claim 55, wherein the recovering data comprises:

copying the stored data from a plurality of active virtual nodes of the active mirror set to the new virtual node, wherein after the copying, the plurality of active virtual nodes and the new virtual node each have the same stored data.

62. The non-transitory machine-readable medium of claim 61, wherein data stored in the active mirror set is stored with a generation identifier and the recovering data uses the generation identifier.

ABSTRACT OF THE DISCLOSURE

A software defined distributed storage system that separates the storage controller functions with that of storage media is described. In an exemplary embodiment, a storage controller client dispatches an input/output request to one of a plurality of storage controller servers in a software defined distributed storage system. The software defined distributed storage system further includes a plurality of virtual nodes distributed among a plurality of physical nodes, and each of the plurality of physical nodes is hosted on one of the plurality of storage controller servers. The storage controller client receives the input/output request. The storage controller client further computes a target virtual node for the input/output request, where the target virtual node is one of the plurality of virtual nodes. Using the computed target virtual node, the storage controller client determines a target physical node that corresponds to the target virtual node, where the target physical node is one of the plurality of physical nodes. In addition, the storage controller client dispatches the input/output request to a target storage controller that corresponds to the target physical node, wherein the target storage controller server is one of the plurality of storage controller servers.



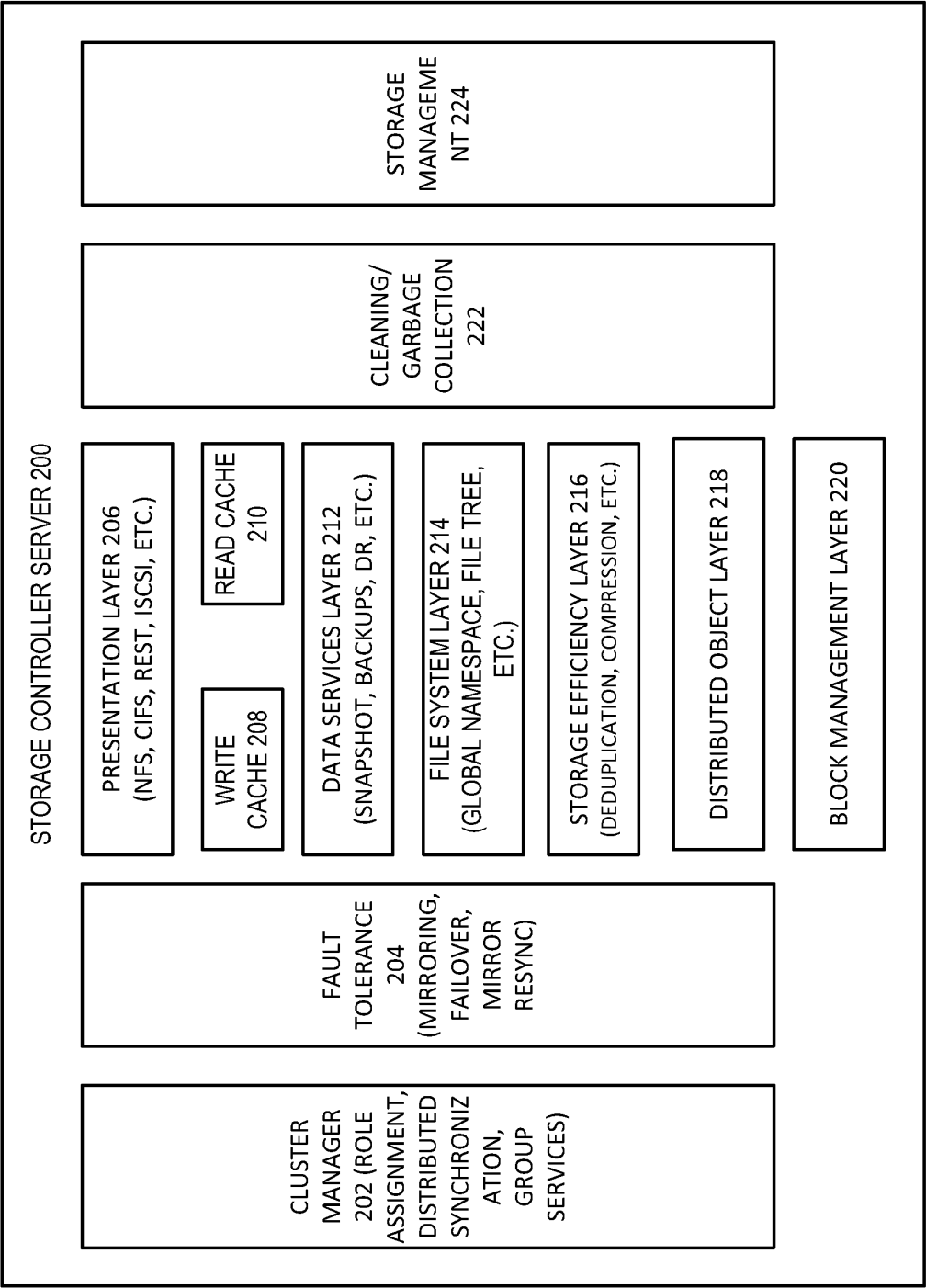


FIGURE 2

300

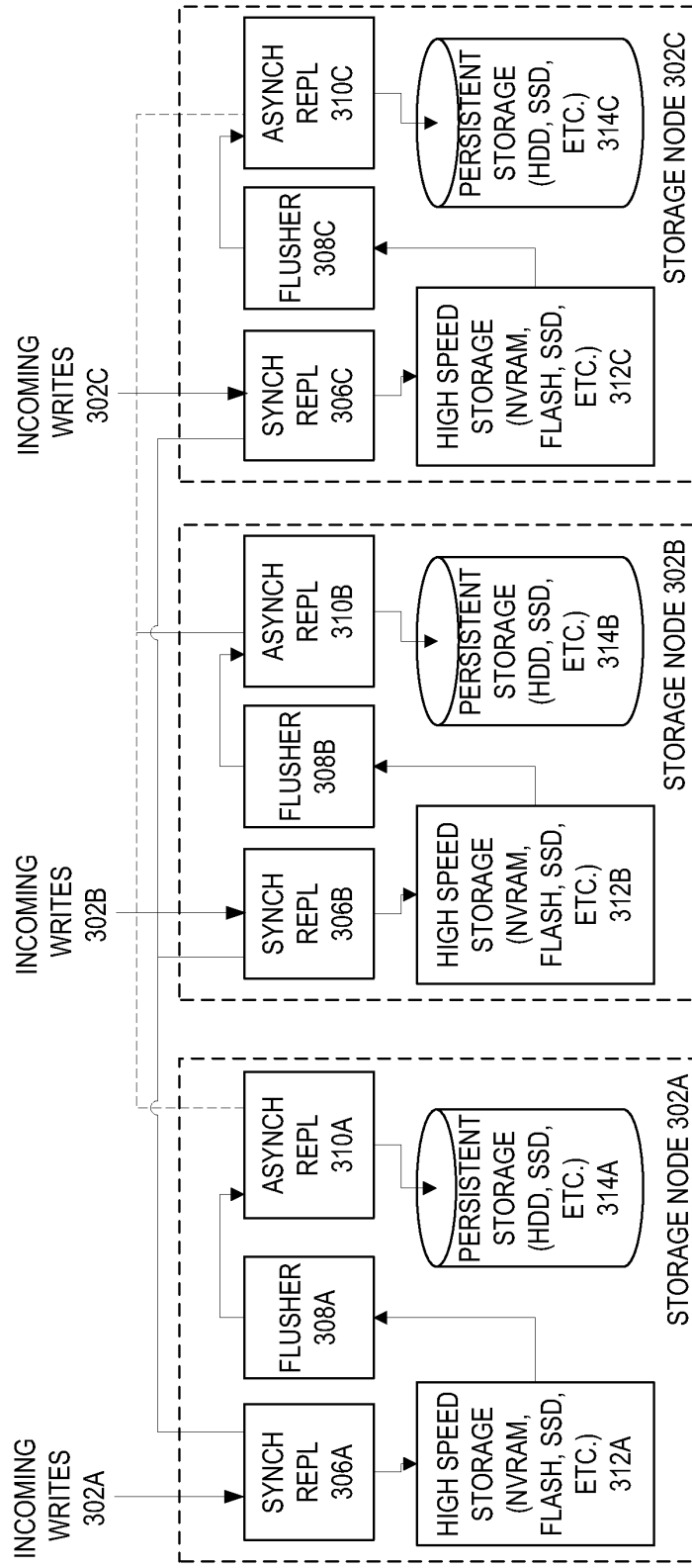
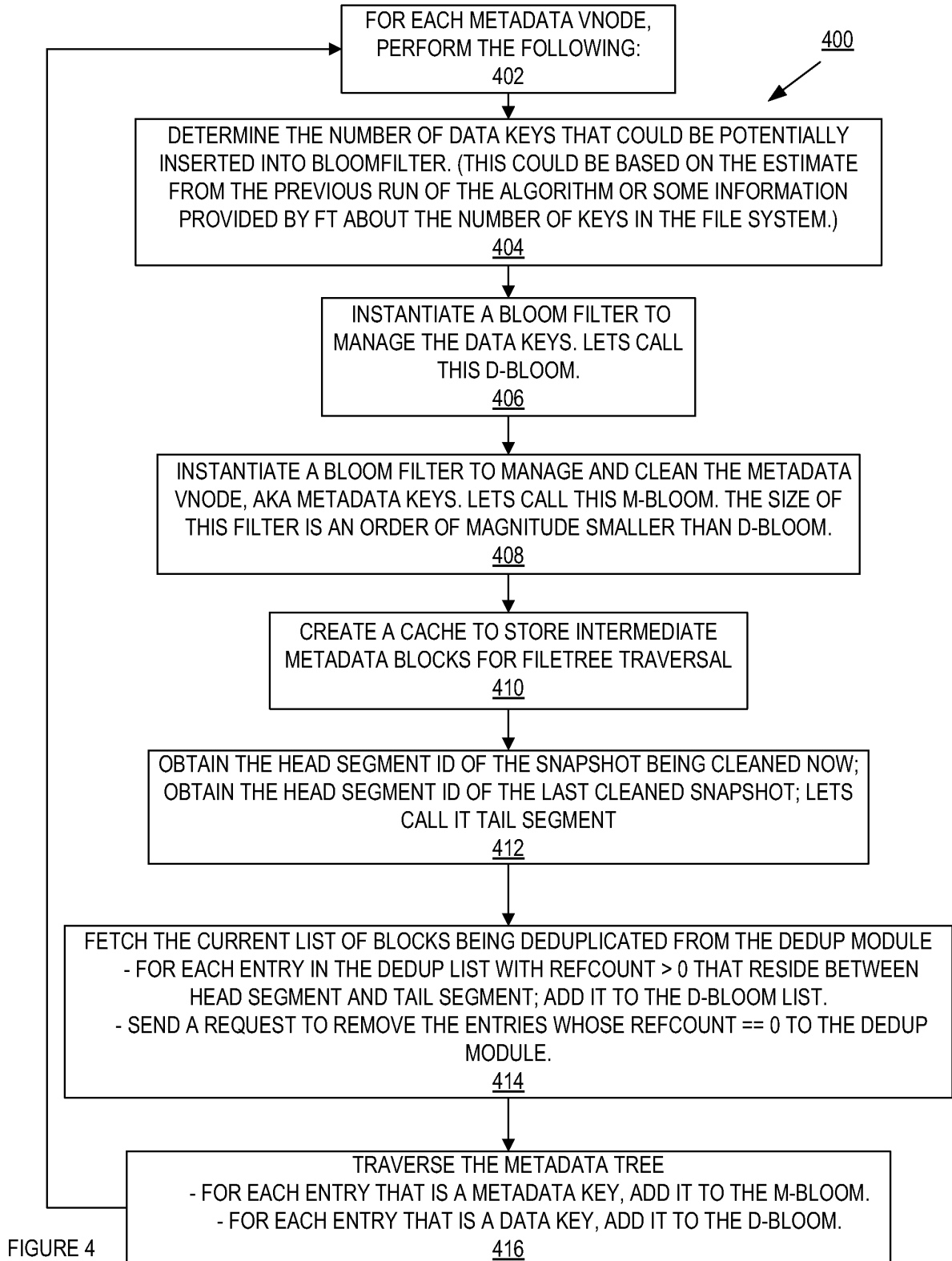


FIGURE 3



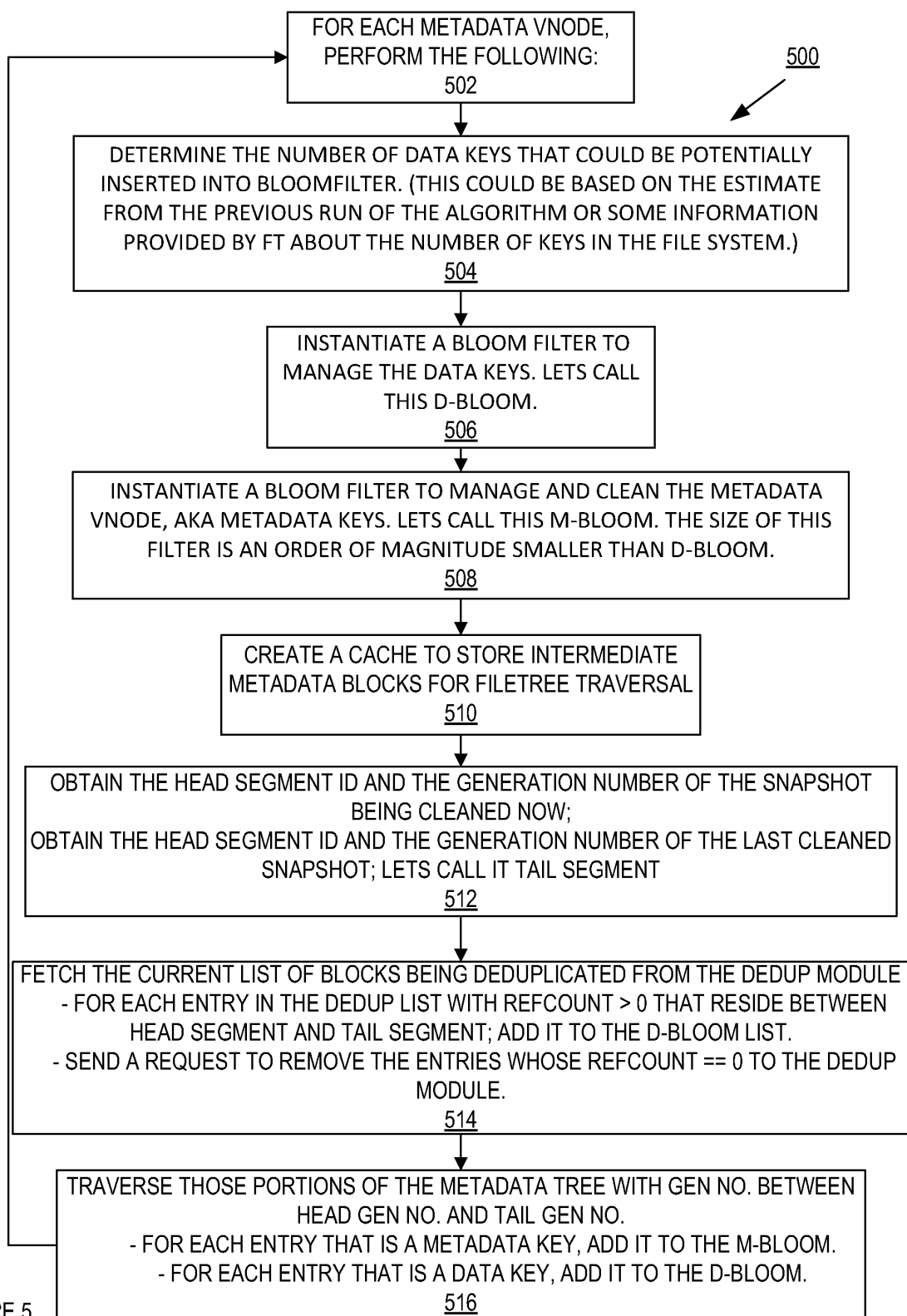


FIGURE 5

600 →

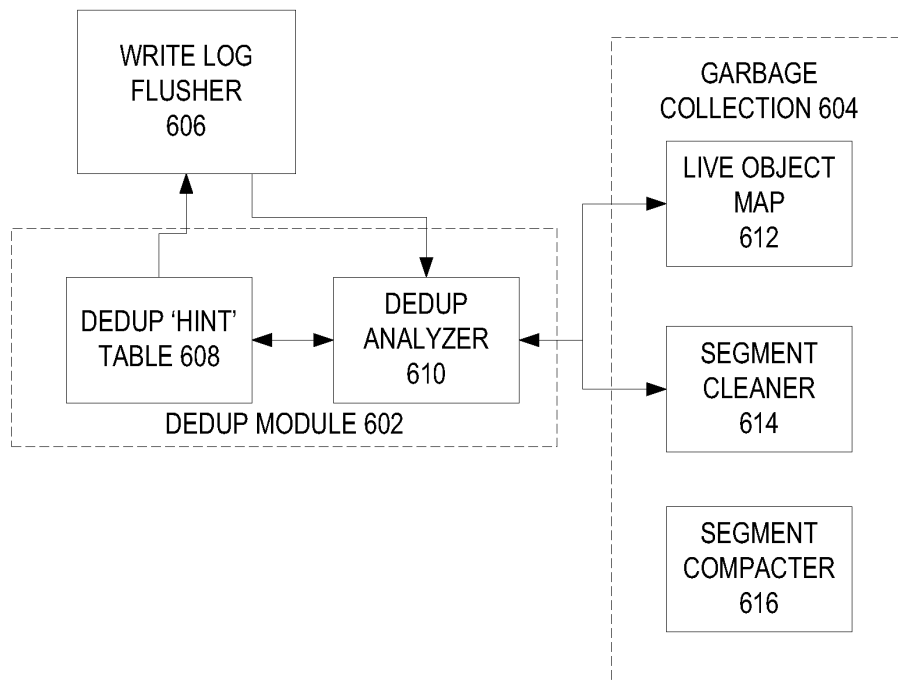
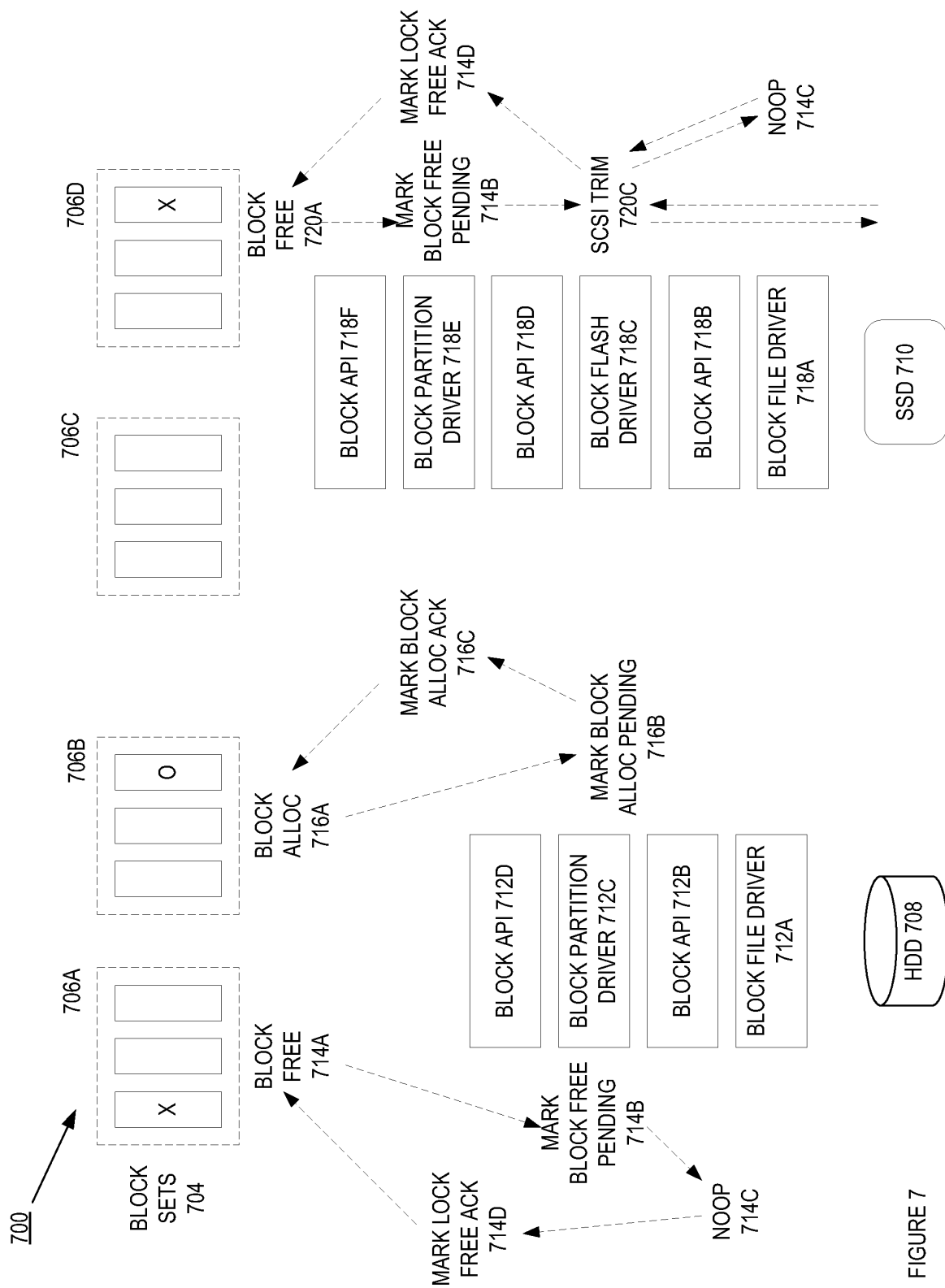


FIGURE 6



800 ↗

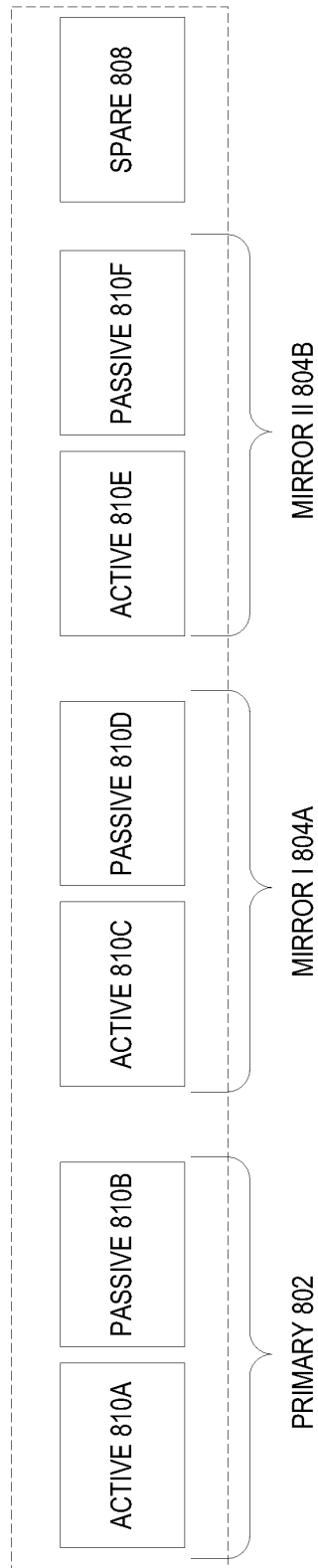
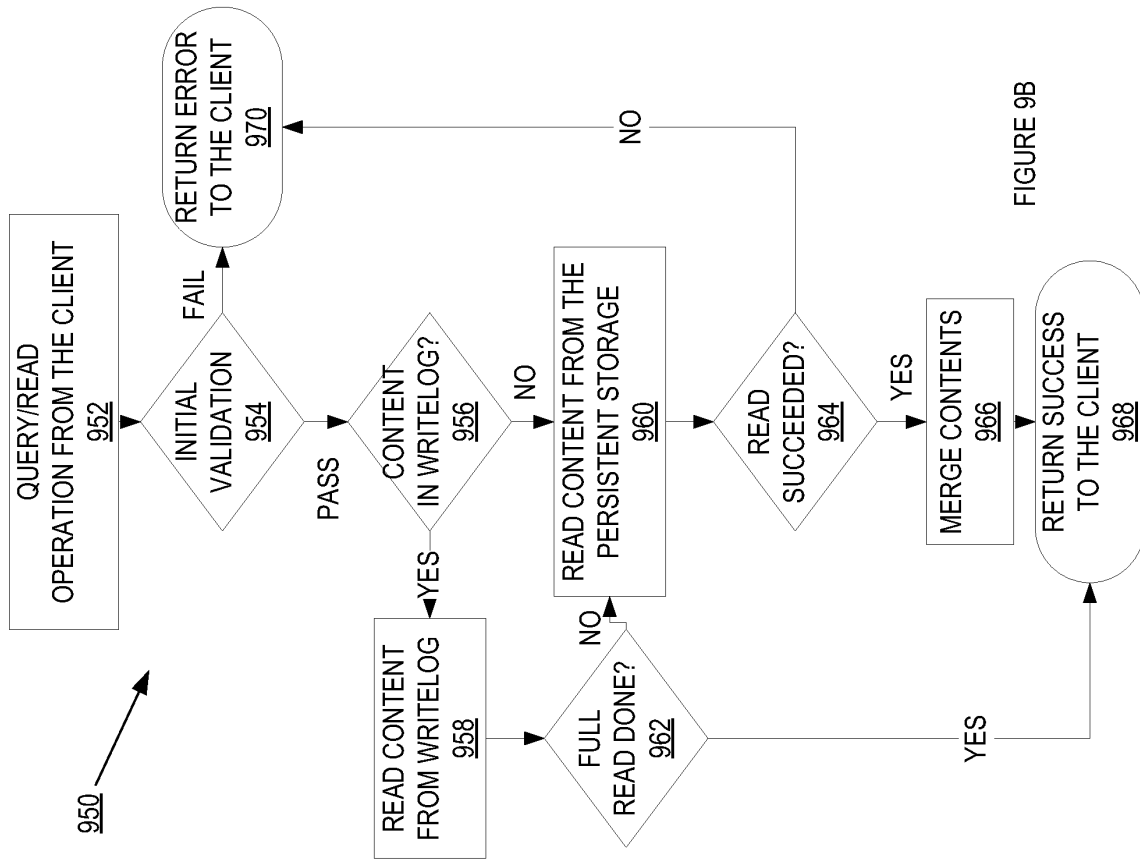
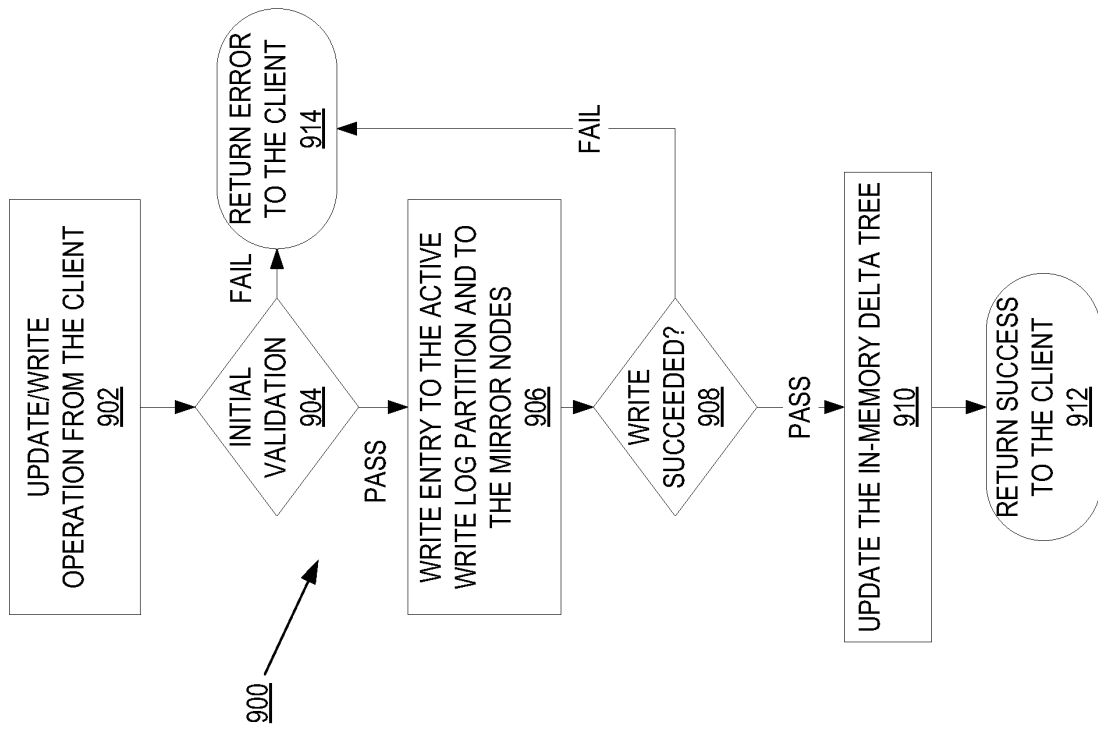


FIGURE 8



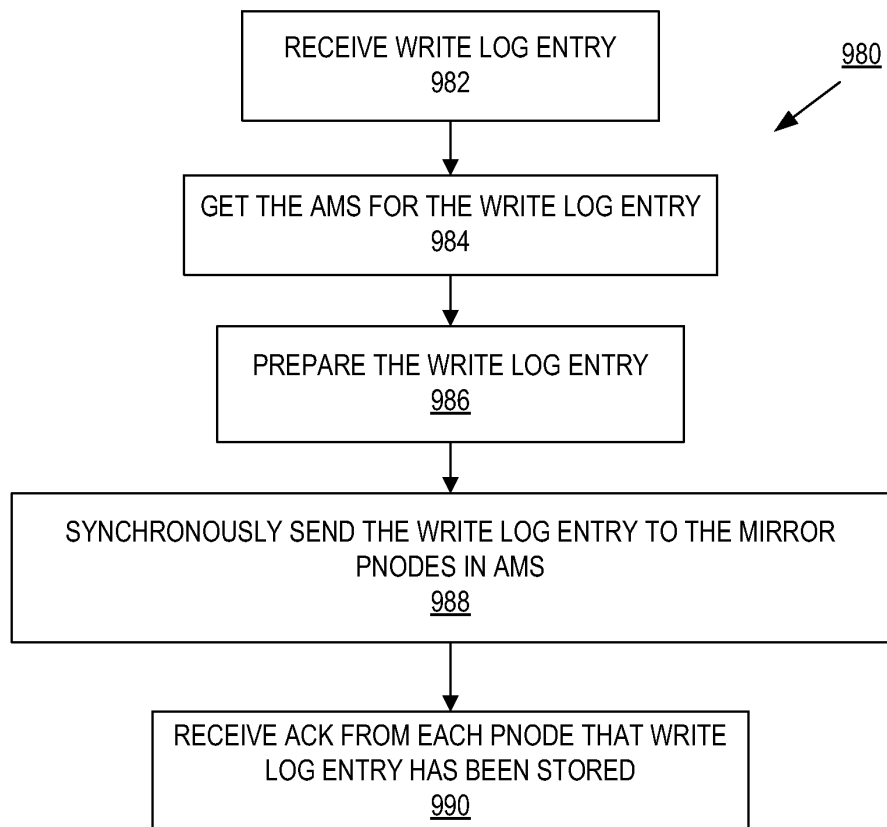


FIGURE 9C

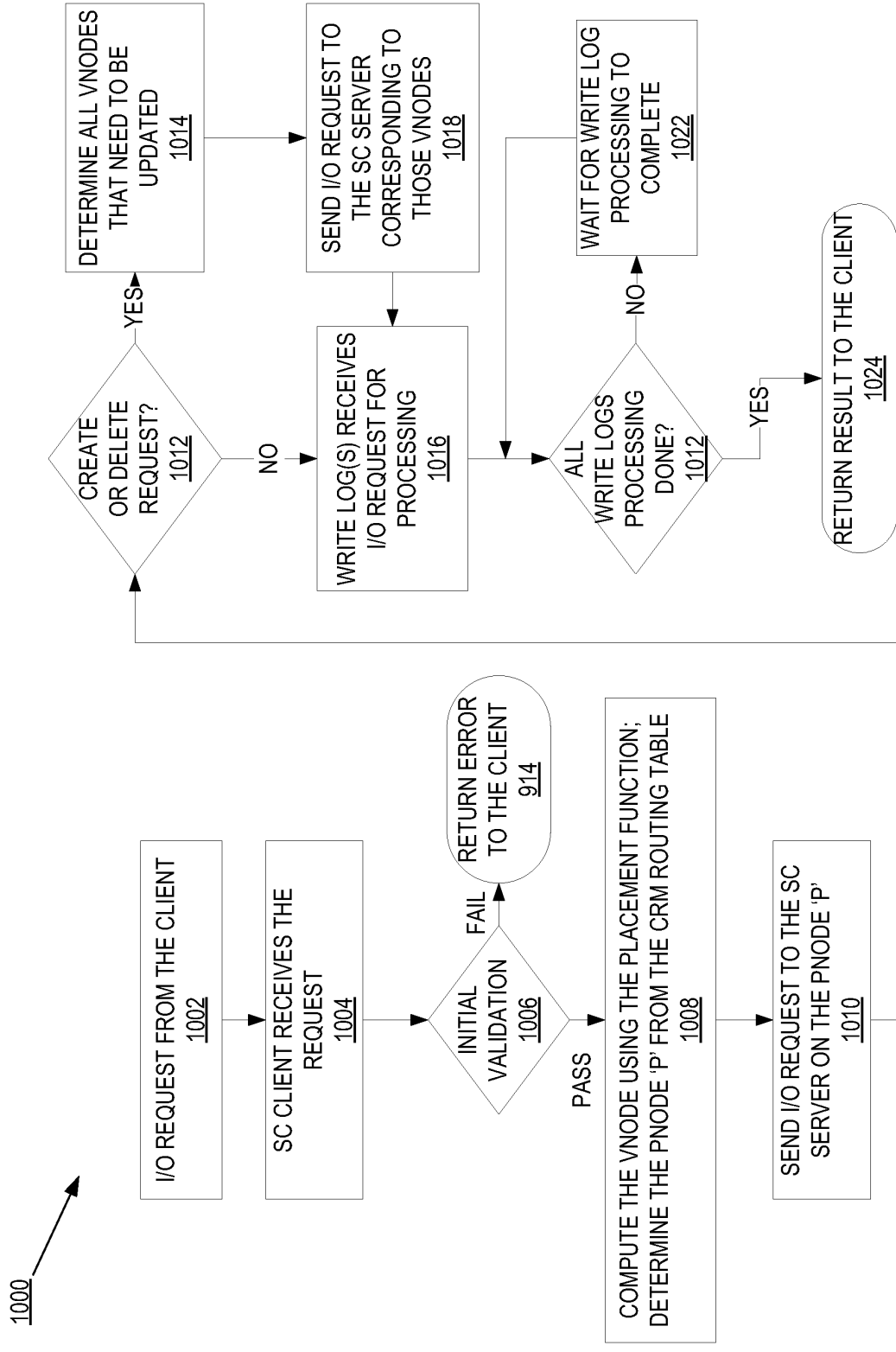
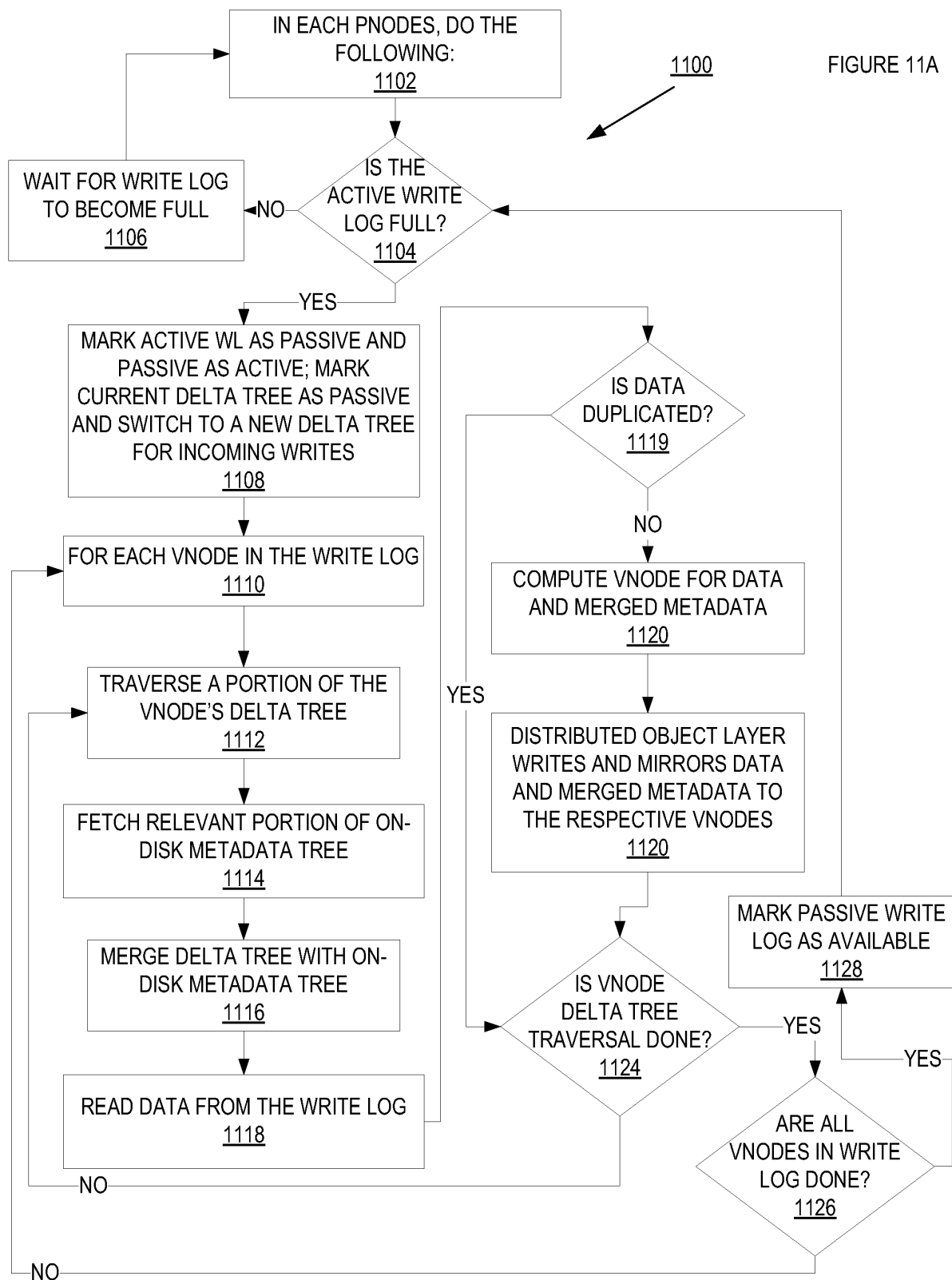


FIGURE 10



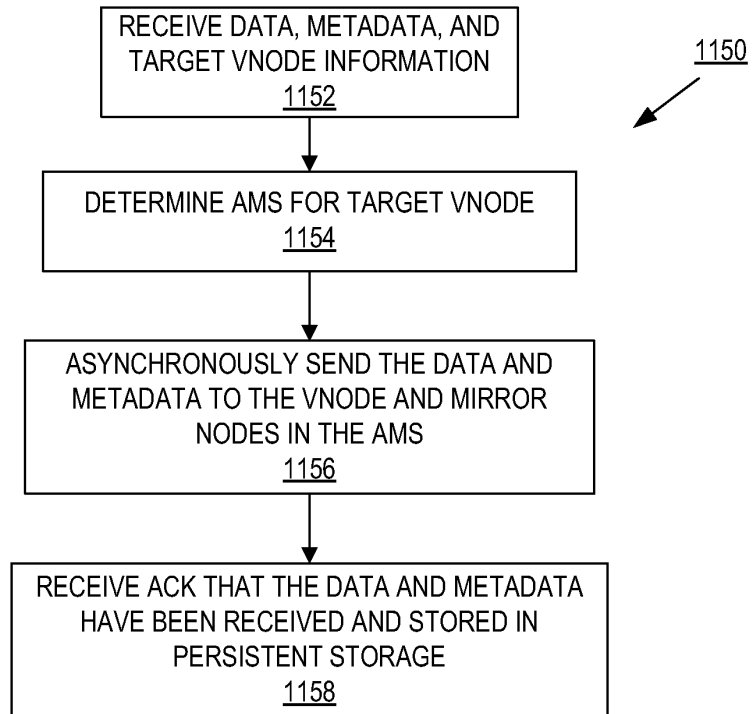


FIGURE 11B

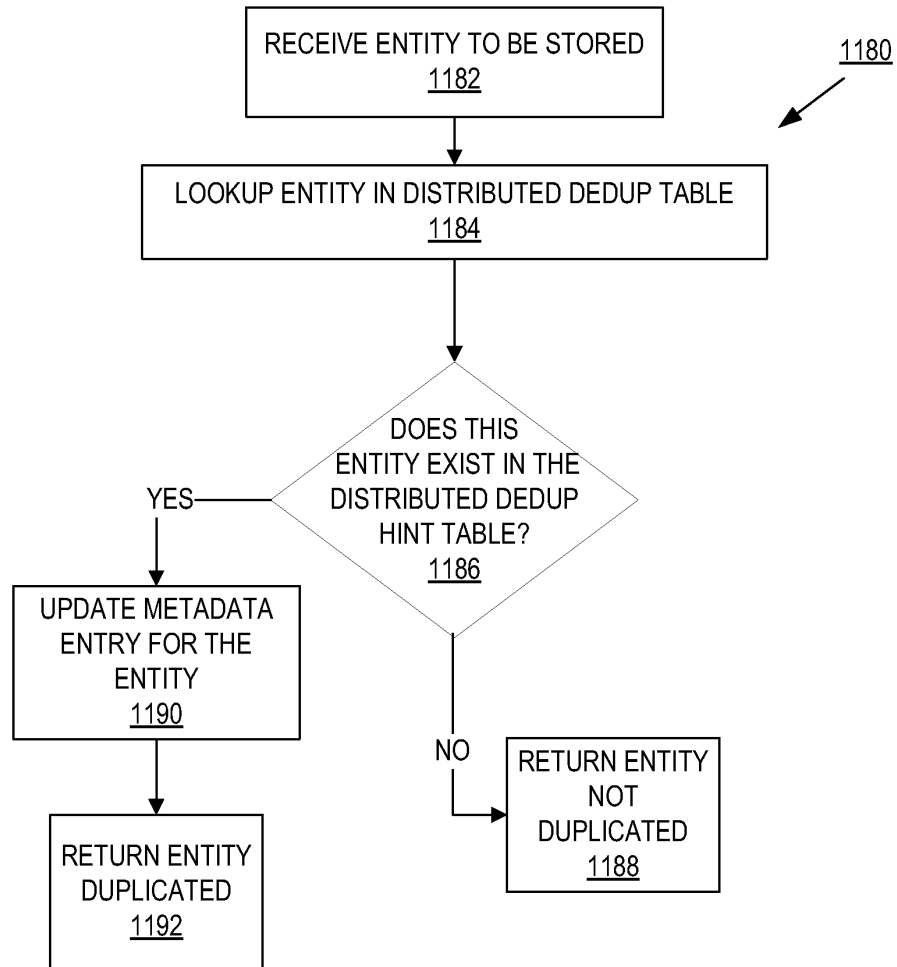


FIGURE 11C

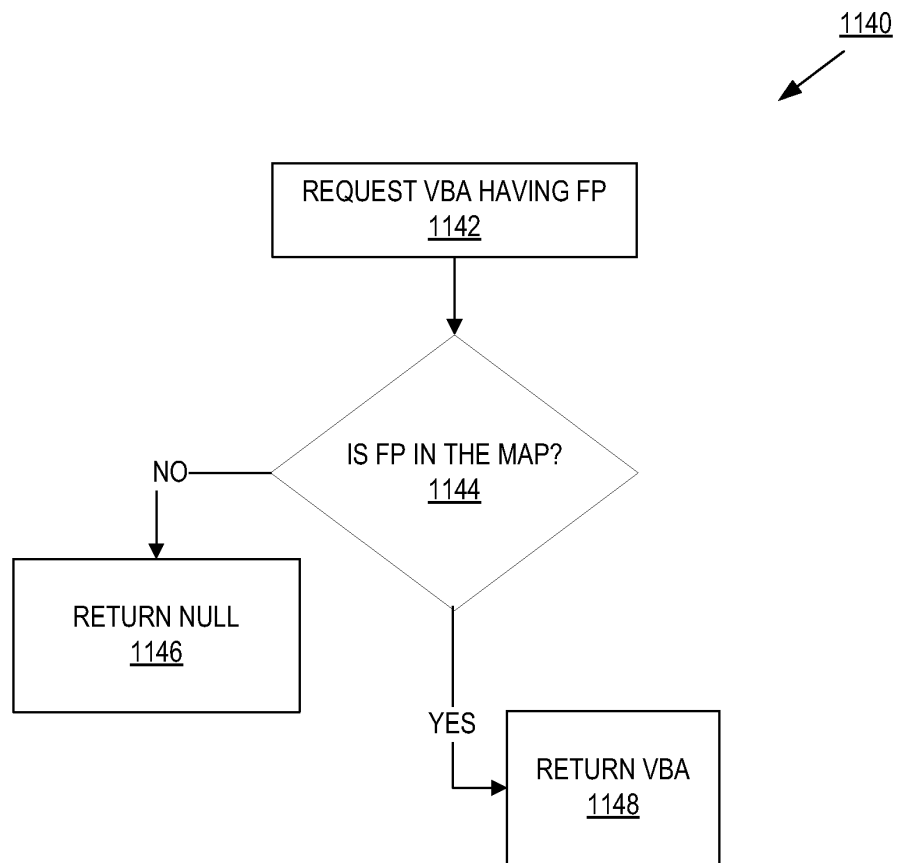


FIGURE 11D

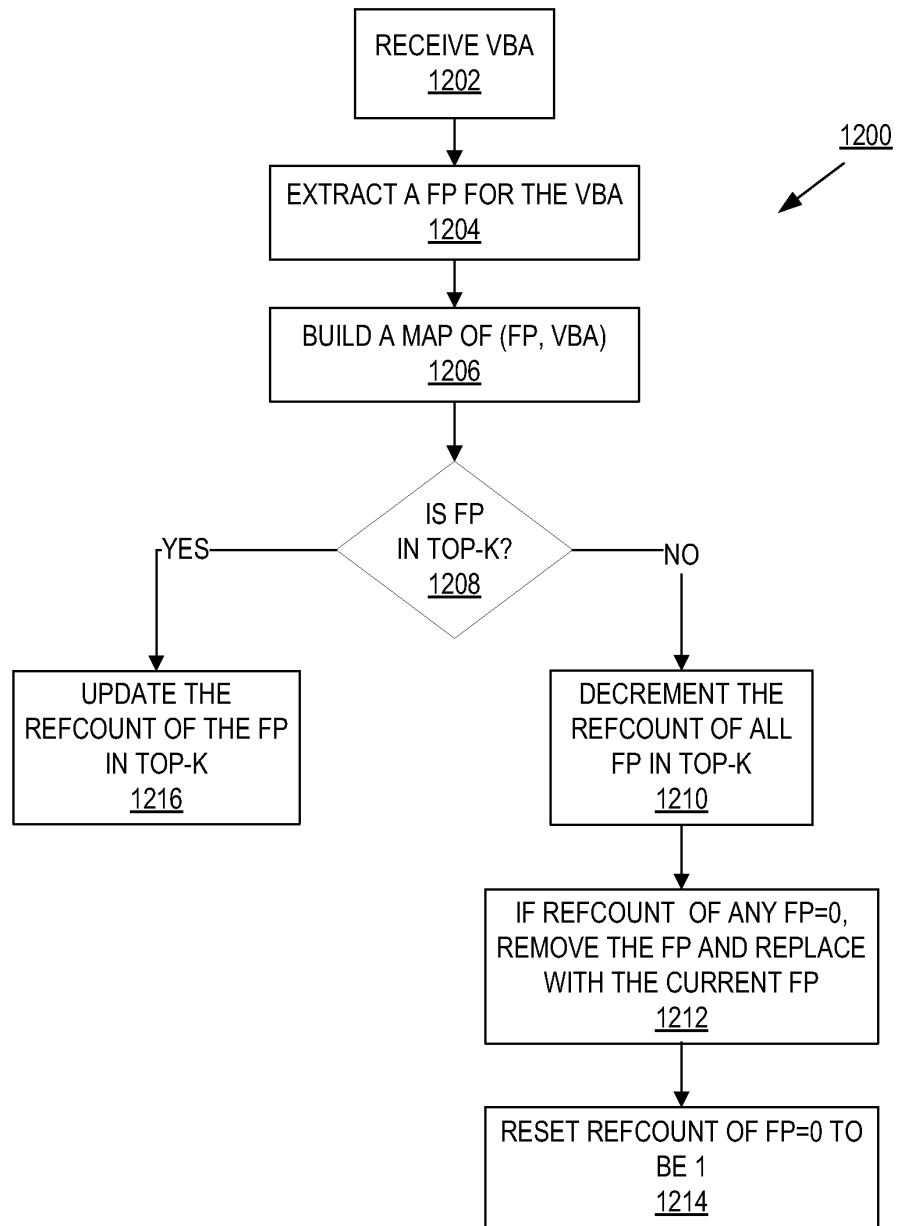


FIGURE 12

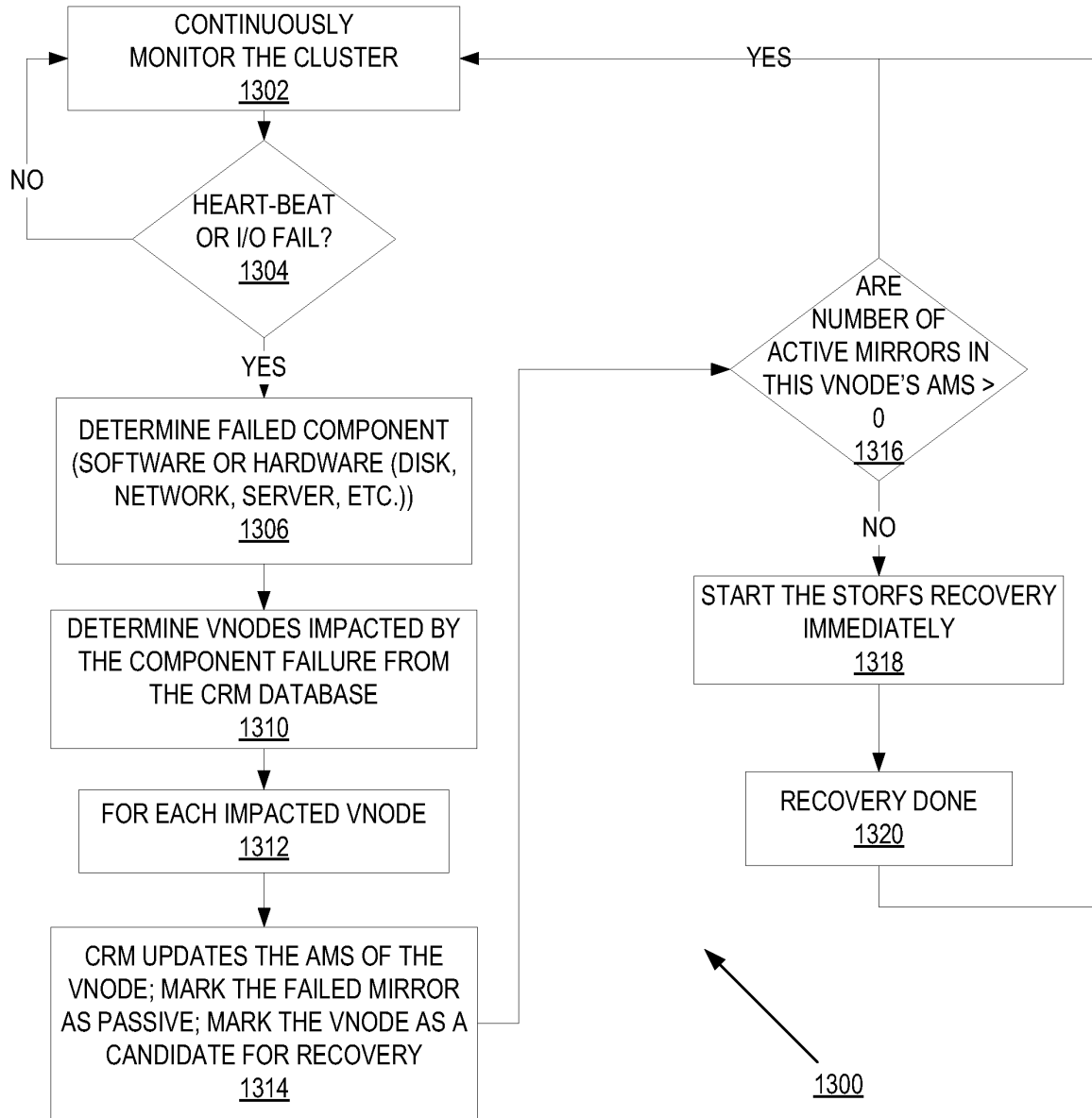
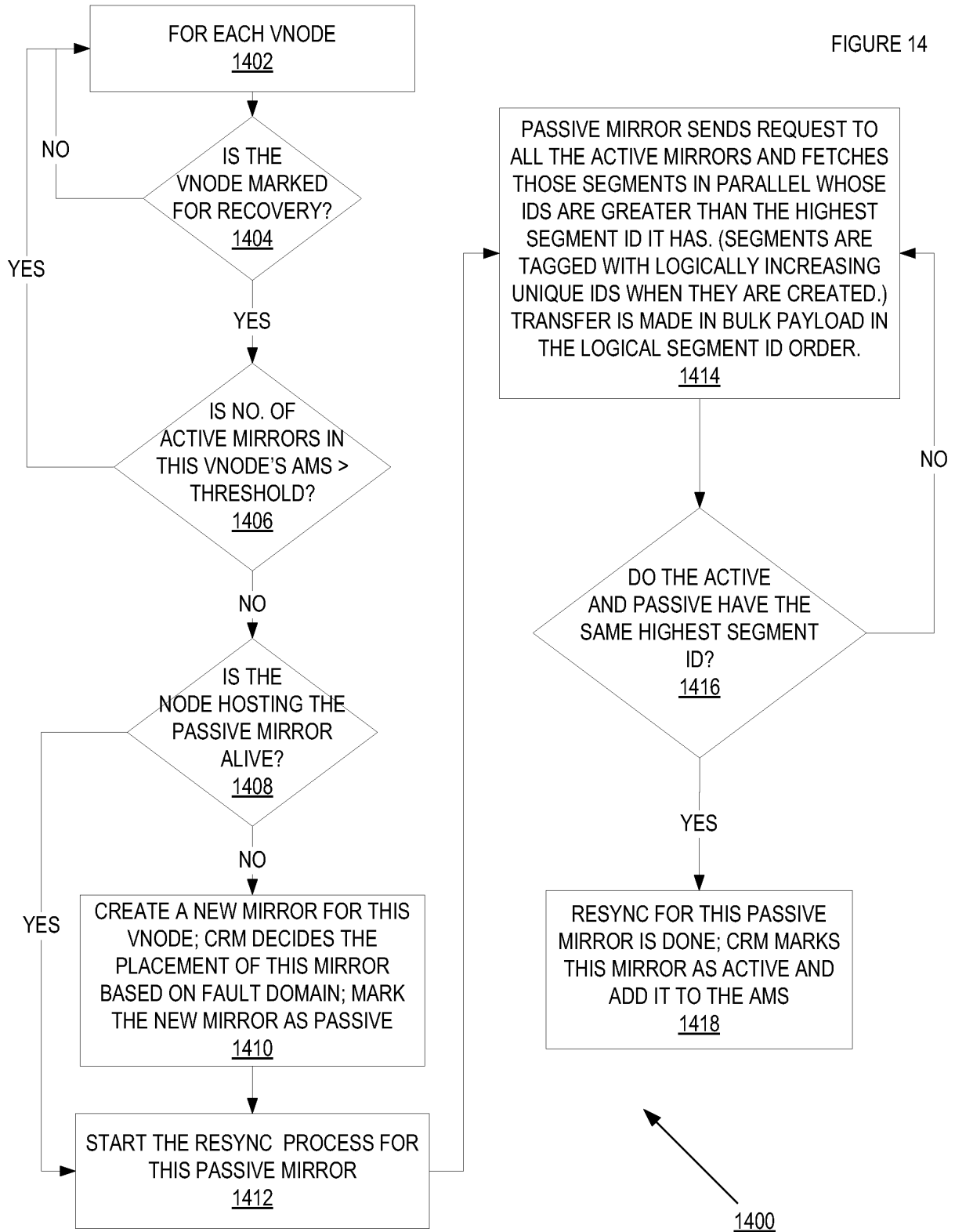


FIGURE 13

FIGURE 14



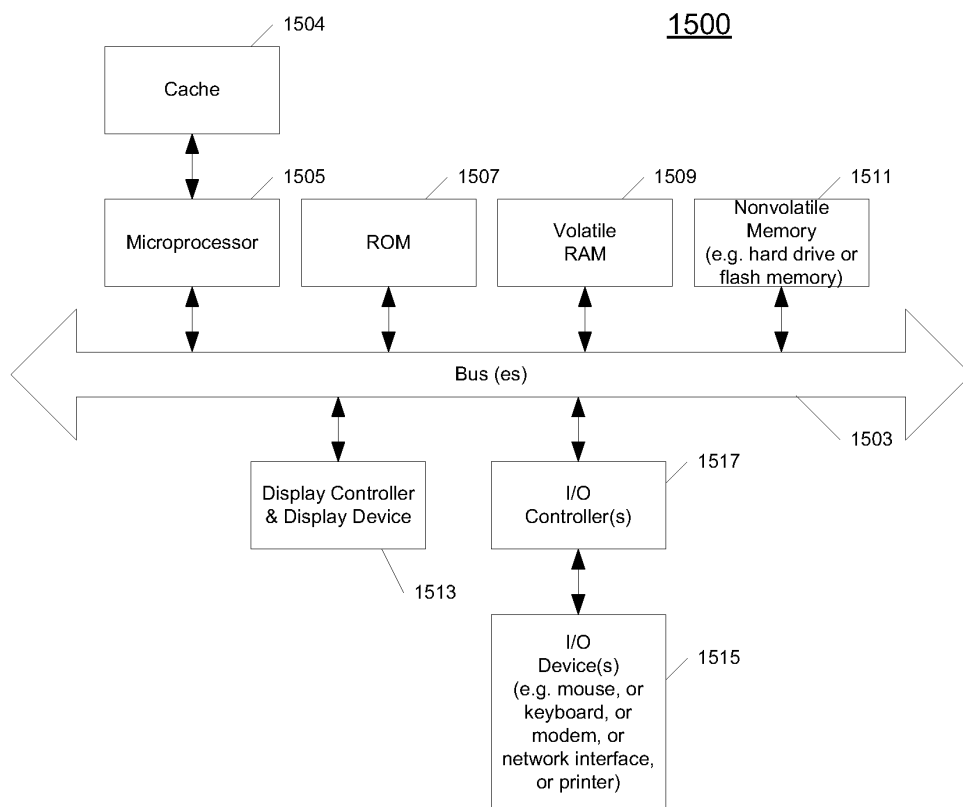


Fig. 15