US 20130205114A1

(19) **United States**

(12) **Patent Application Publication**

Badam et al.

(10) **Pub. No.: US 2013/0205114 A1**

(43) **Pub. Date: Aug. 8, 2013**

(54) **OBJECT-BASED MEMORY STORAGE**

(71) Applicant: **FUSION-IO**, Salt Lake City, UT (US)

(72) Inventors: **Anirudh Badam**, Princeton, NJ (US); **David Nellans**, Salt Lake City, UT (US); **Robert Wipfel**, Draper, UT (US)

(73) Assignee: **FUSION-IO**, Salt Lake City, UT (US)

(21) Appl. No.: **13/835,109**

(22) Filed: **Mar. 15, 2013**

Related U.S. Application Data

(63) Continuation of application No. 12/098,433, filed on Apr. 6, 2008, now Pat. No. 8,151,082, which is a continuation-in-part of application No. 11/952,098, filed on Dec. 6, 2007, Continuation of application No. 11/952,098, filed on Dec. 6, 2007.

(60) Provisional application No. 60/873,111, filed on Dec. 6, 2006, provisional application No. 60/974,470, filed on Sep. 22, 2007.

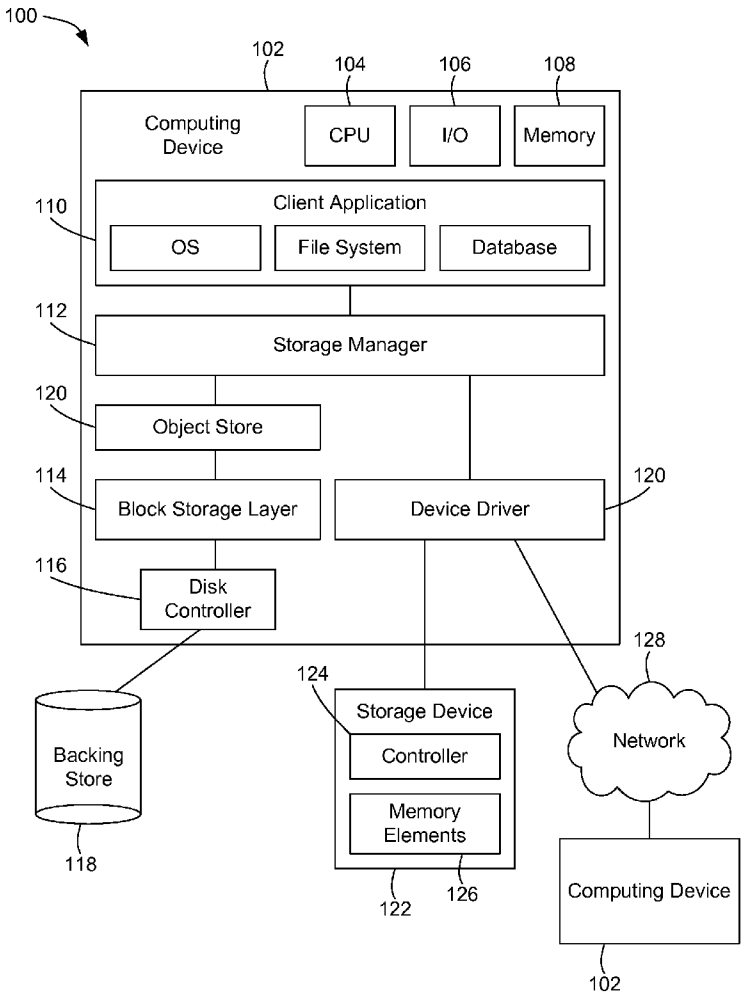
Publication Classification

(51) **Int. Cl.**
G06F 12/10 (2006.01)

(52) **U.S. Cl.**
CPC ***G06F 12/1027*** (2013.01)
USPC **711/207**

(57) **ABSTRACT**

The method includes receiving an object operation from an application at a hardware device manager. The object operation includes an object identifier. The method includes performing the object operation directly on a storage device. A physical address for the object corresponding to the object identifier is mapped directly to the object identifier in an index managed by the hardware device manager.



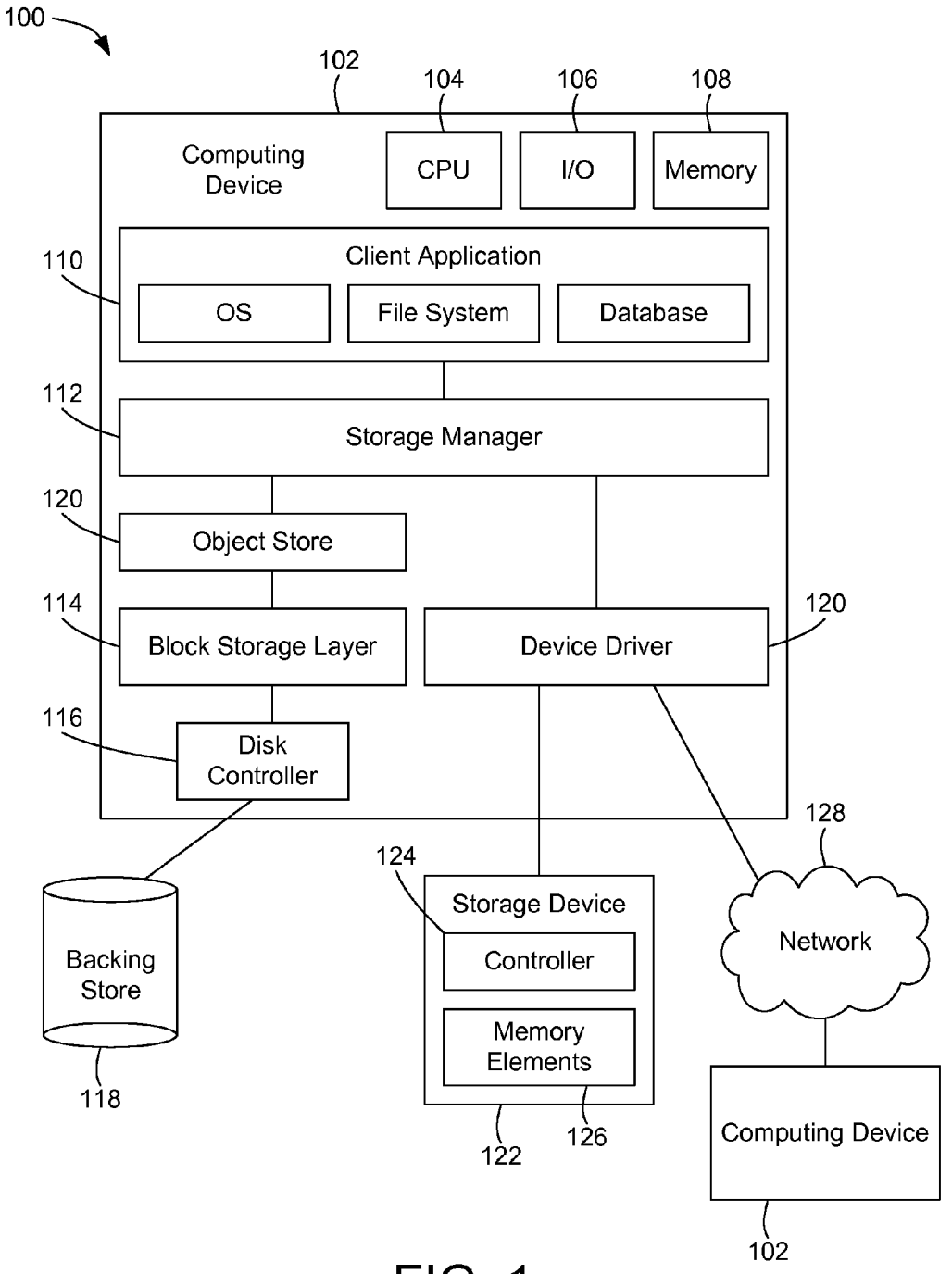

100
102
104
106
108
Computing Device
CPU
I/O
Memory
110
Client Application
OS
File System
Database
112
Storage Manager
120
Object Store
114
Block Storage Layer
Device Driver
120
116
Disk Controller
128
124
Storage Device
Controller
Memory Elements
Network
Backing Store
118
122
126
Computing Device
102

FIG. 1

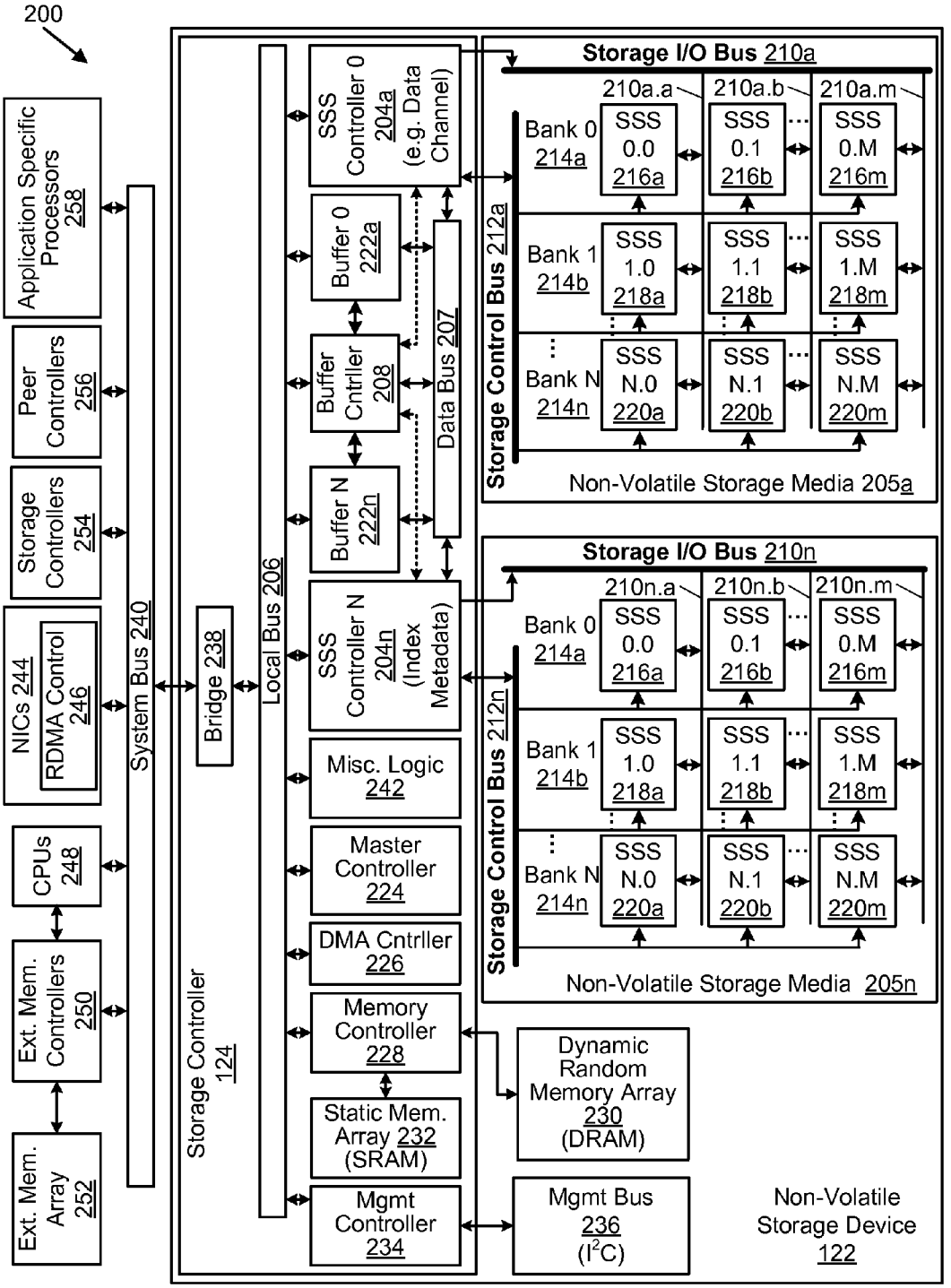

200
Application Specific Processors 258
Peer Controllers 256
Storage Controllers 254
NICs 244
RDMA Control 246
CPUs 248
Ext. Mem. Controllers 250
Ext. Mem. Array 252
System Bus 240
Bridge 238
Local Bus 206
Storage Controller 124
SSS Controller 0 204a (e.g. Data Channel)
Buffer 0 222a
Buffer Cntrller 208
Buffer N 222n
Data Bus 207
SSS Controller N 204n (Index Metadata)
Misc. Logic 242
Master Controller 224
DMA Cntrller 226
Memory Controller 228
Static Mem. Array 232 (SRAM)
Mgmt Controller 234
Storage I/O Bus 210a
Storage Control Bus 212a
210a.a
210a.b
210a.m
Bank 0 214a
Bank 1 214b
Bank N 214n
SSS 0.0 216a
SSS 0.1 216b
SSS 0.M 216m
SSS 1.0 218a
SSS 1.1 218b
SSS 1.M 218m
SSS N.0 220a
SSS N.1 220b
SSS N.M 220m
Non-Volatile Storage Media 205a
Storage I/O Bus 210n
Storage Control Bus 212n
210n.a
210n.b
210n.m
Non-Volatile Storage Media 205n
Dynamic Random Memory Array 230 (DRAM)
Mgmt Bus 236 (I2C)
Non-Volatile Storage Device 122

FIG. 2

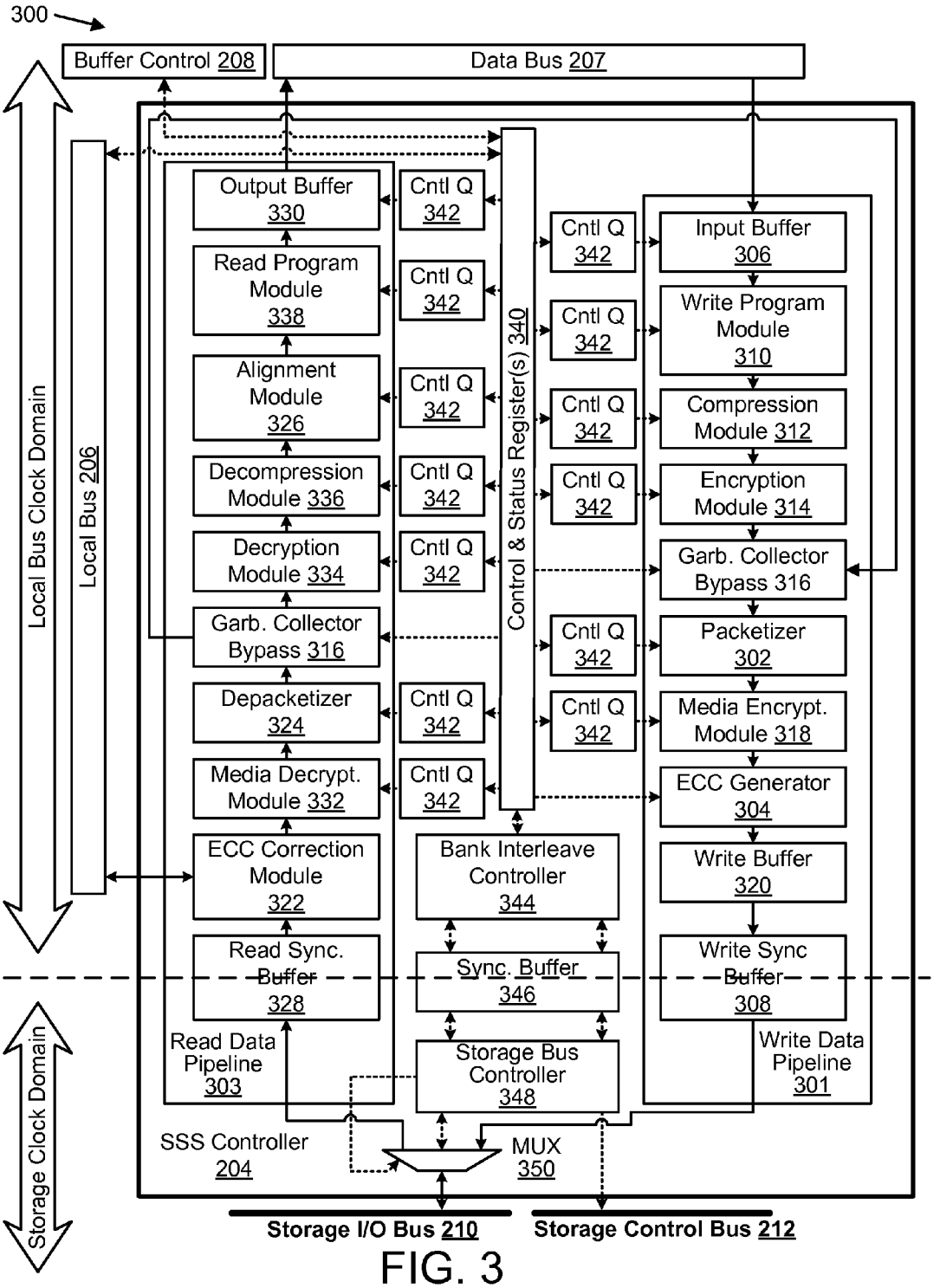

300
Buffer Control 208
Data Bus 207
Local Bus Clock Domain
Local Bus 206
Output Buffer 330
Read Program Module 338
Alignment Module 326
Decompression Module 336
Decryption Module 334
Garb. Collector Bypass 316
Depacketizer 324
Media Decrypt. Module 332
ECC Correction Module 322
Read Sync. Buffer 328
Cntl Q 342
Control & Status Register(s) 340
Input Buffer 306
Write Program Module 310
Compression Module 312
Encryption Module 314
Garb. Collector Bypass 316
Packetizer 302
Media Encrypt. Module 318
ECC Generator 304
Write Buffer 320
Write Sync Buffer 308
Bank Interleave Controller 344
Sync. Buffer 346
Storage Bus Controller 348
Read Data Pipeline 303
Write Data Pipeline 301
SSS Controller 204
MUX 350
Storage Clock Domain
Storage I/O Bus 210
Storage Control Bus 212

FIG. 3

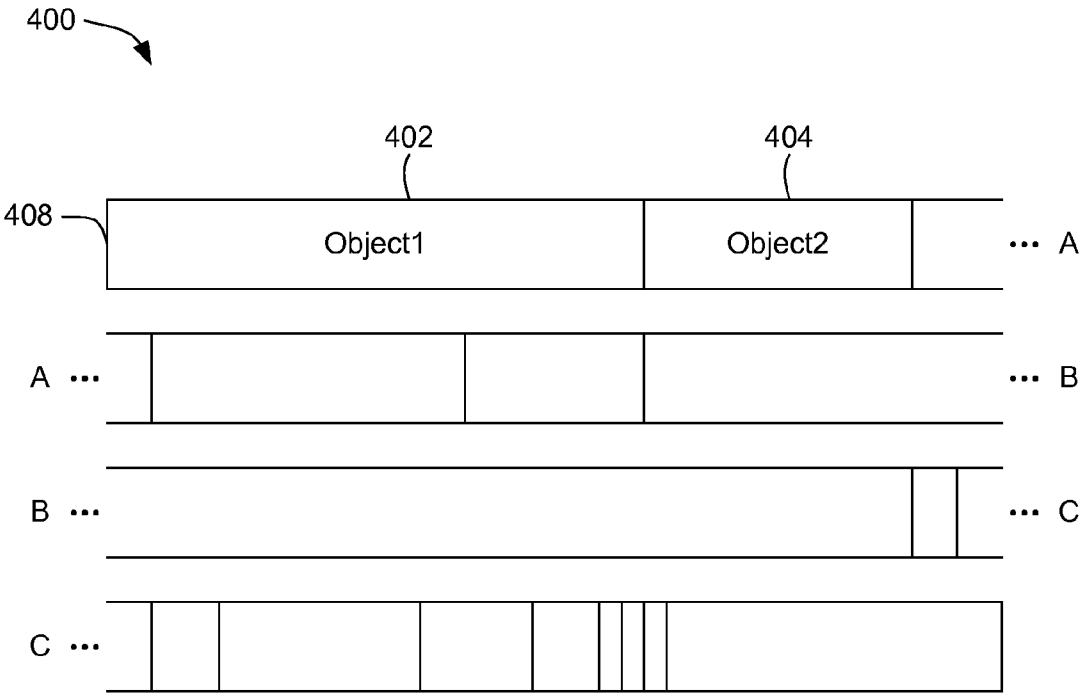
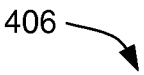


FIG. 4A



OBJECT ID	LOCATION	SIZE
Object1	Address1	Size1
Object2	Address2	Size2
Object3	Address3	Size3
Object4	Address4	Size4

FIG. 4B

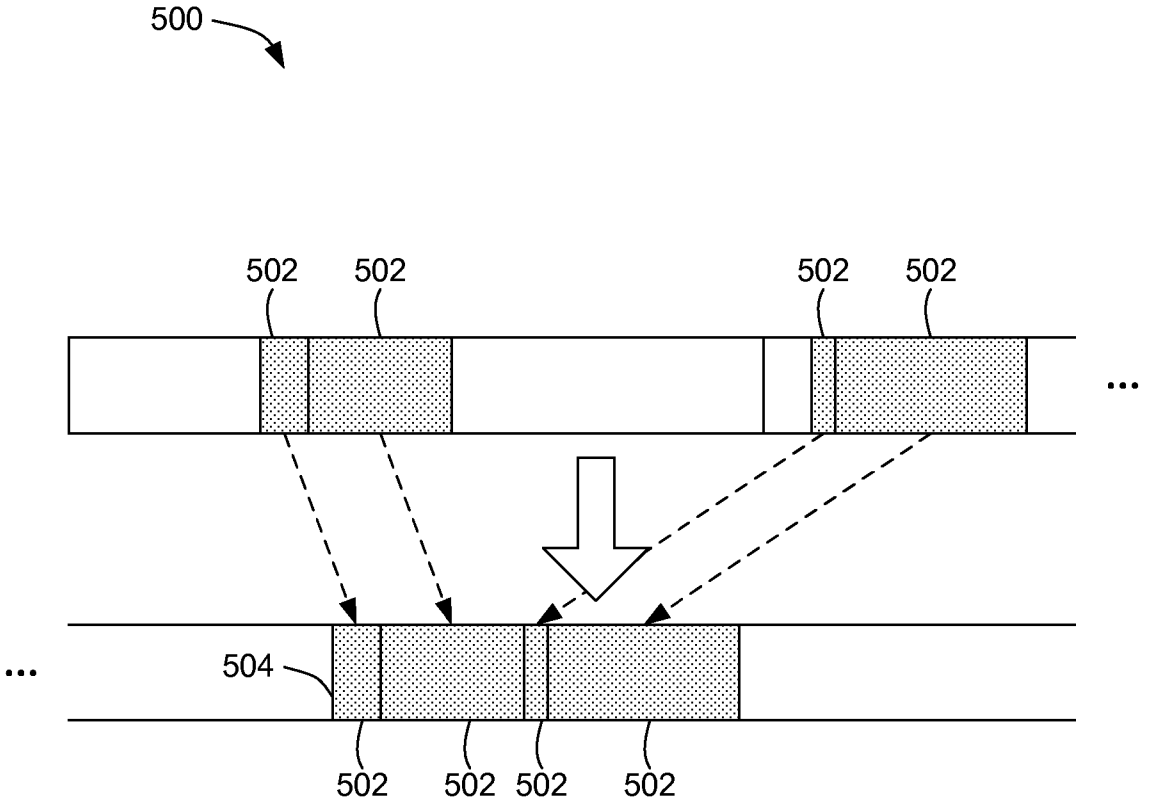


FIG. 5

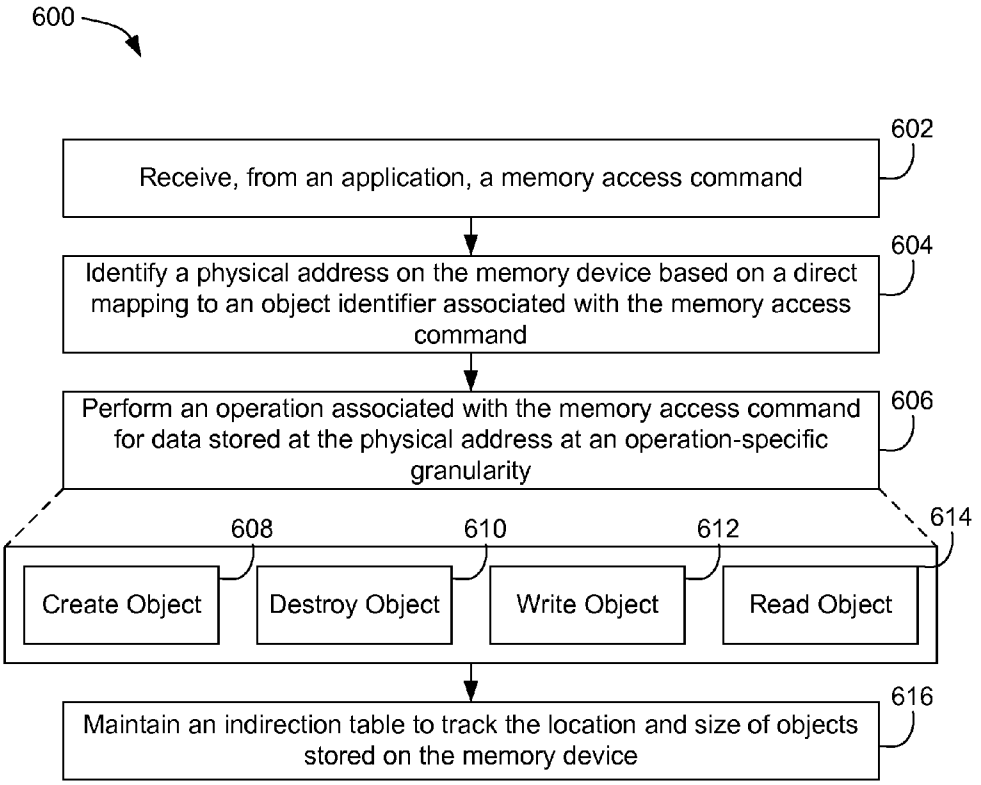

600
602
Receive, from an application, a memory access command
604
Identify a physical address on the memory device based on a direct mapping to an object identifier associated with the memory access command
606
Perform an operation associated with the memory access command for data stored at the physical address at an operation-specific granularity
614
608
610
612
Create Object
Destroy Object
Write Object
Read Object
616
Maintain an indirection table to track the location and size of objects stored on the memory device

FIG. 6

OBJECT-BASED MEMORY STORAGE

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims the benefit of priority of U.S. patent application Ser. No. 12/098,433, filed on Apr. 6, 2008 and entitled "Apparatus, System, and Method for Converting a Storage Request into an Append Data Storage Command," which is a continuation-in-part of and claims priority to U.S. patent application Ser. No. 11/952,098, filed on Dec. 6, 2007 and entitled "Apparatus, System, and Method for Servicing Object Requests within a Storage Controller" all of which are incorporated by reference herein. This application also claims the benefit of priority of U.S. patent application Ser. No. 11/952,098, filed Dec. 6, 2007 and entitled "Apparatus, System, and Method for Servicing Object Requests within a Storage Controller," which is a continuation-in-part of and claims priority to U.S. Provisional Patent Application No. 60/873,111, filed on Dec. 6, 2006 and entitled "Elemental Blade System" and U.S. Provisional Patent Application No. 60/974,470, filed Sep. 22, 2007 and entitled "Apparatus, System, and Method for Object-Oriented Solid-State Storage" all of which are incorporated by reference herein.

BACKGROUND

[0002] Many types of electronic storage media use a block-based approach for storing or accessing data on the storage media. A block-based approach can improve performance time for certain types of storage devices, such as magnetic disk drives or other drives that include mechanical movement for reading/writing/erasing data stored on the storage devices. Due to the mechanical movement, random access to the storage devices is more time expensive than sequential access. Block-based approaches typically write data to physically sequential addresses on the storage devices to reduce access time.

[0003] Block-based approaches also typically access blocks of a predetermined size, regardless of whether the data being written to or read from the storage devices fills an entire block. Additionally, applications often use objects (which may vary greatly in size) as a basis of programming. Object based programming frequently uses an object store on top of the block storage layer or other block abstraction to be able to store and track the location of each object on the storage device.

[0004] For low latency storage devices such as flash memory, which are able to handle random access to the devices quickly, block-based approaches may not be necessary to optimize the access speed of the flash memory. However, many flash memory devices still use a block-based approach, which introduces unnecessary programming and operating system resource overheads because of the need to maintain tables and to perform read-modify-writes necessary for garbage collection and for compacting live objects to minimize fragmentation of data on the storage device.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] FIG. **1** depicts a schematic diagram of one embodiment of a computing device in a network system.

[0006] FIG. **2** depicts a schematic diagram of one embodiment of a non-volatile storage device.

[0007] FIG. **3** depicts a schematic diagram of one embodiment of a storage controller.

[0008] FIG. **4**A depicts a schematic diagram of one embodiment of a log structure in a memory device.

[0009] FIG. **4**B depicts a schematic diagram of one embodiment of a table.

[0010] FIG. **5** depicts a schematic diagram of garbage collection in a log structure on a memory device.

[0011] FIG. **6** depicts a flow chart diagram of one embodiment of a method for object based storage on a memory device.

[0012] Throughout the description, similar reference numbers may be used to identify similar elements.

DETAILED DESCRIPTION

[0013] It will be readily understood that the components of the embodiments as generally described herein and illustrated in the appended figures could be arranged and designed in a wide variety of different configurations. Thus, the following more detailed description of various embodiments, as represented in the figures, is not intended to limit the scope of the present disclosure, but is merely representative of various embodiments. While the various aspects of the embodiments are presented in drawings, the drawings are not necessarily drawn to scale unless specifically indicated.

[0014] The present invention may be embodied in other specific forms without departing from its spirit or essential characteristics. The described embodiments are to be considered in all respects only as illustrative and not restrictive. The scope of the invention is, therefore, indicated by the appended claims rather than by this detailed description. All changes which come within the meaning and range of equivalency of the claims are to be embraced within their scope.

[0015] Reference throughout this specification to features, advantages, or similar language does not imply that all of the features and advantages that may be realized with the present invention should be or are in any single embodiment of the invention. Rather, language referring to the features and advantages is understood to mean that a specific feature, advantage, or characteristic described in connection with an embodiment is included in at least one embodiment of the present invention. Thus, discussions of the features and advantages, and similar language, throughout this specification may, but do not necessarily, refer to the same embodiment.

[0016] Reference to a computer readable medium may take any physical form capable of storing machine-readable instructions, at least for a time in a non-transient state, on a digital processing apparatus. A computer readable medium may be embodied by a compact disk, digital-video disk, a blu-ray disc, a magnetic tape, a Bernoulli drive, a magnetic disk, flash memory, integrated circuits, or other digital processing apparatus memory device.

[0017] Furthermore, the described features, advantages, and characteristics of the invention may be combined in any suitable manner in one or more embodiments. One skilled in the relevant art will recognize, in light of the description herein, that the invention can be practiced without one or more of the specific features or advantages of a particular embodiment. In other instances, additional features and advantages may be recognized in certain embodiments that may not be present in all embodiments of the invention.

[0018] Reference throughout this specification to "one embodiment," "an embodiment," or similar language means that a particular feature, structure, or characteristic described in connection with the indicated embodiment is included in at

least one embodiment of the present invention. Thus, the phrases "in one embodiment," "in an embodiment," and similar language throughout this specification may, but do not necessarily, all refer to the same embodiment.

[0019] While many embodiments are described herein, at least some of the described embodiments facilitate object based storage on a storage device (also referred to herein as a memory device or a memory storage device). The storage device may be any type of memory device, including a volatile storage device or non-volatile storage device, configured to store data. Examples of non-volatile memory include flash memory, nano random access memory (nano RAM or NRAM), nanocrystal wire-based memory, silicon-oxide based sub-10 nanometer process memory, graphene memory, Silicon-Oxide-Nitride-Oxide-Silicon (SONOS), Resistive random-access memory (RRAM), programmable metallization cell (PMC), conductive-bridging RAM (CBRAM), magneto-resistive RAM (MRAM), dynamic RAM (DRAM), phase change RAM (PRAM), phase change memory or other non-volatile solid-state storage media. In other embodiments, the non-volatile memory may comprise magnetic media, optical media, or other types of non-volatile storage media. For example, in those embodiments, the non-volatile storage device may include a hard disk drive, an optical storage drive, or the like.

[0020] While the non-volatile memory is referred to herein as a "memory device," a "recording device" or a "storage device" in various embodiments, the non-volatile memory may more generally include a non-volatile recording media capable of recording data, the non-volatile recording media may be referred to as a non-volatile memory media, a non-volatile storage media, or the like. Further, the non-volatile storage device, in various embodiments, may include a non-volatile recording device, a non-volatile memory device, a non-volatile storage device, or the like. In other embodiments, the storage device may be any other storage device that may be configured to store data at a non-block based granularity. The storage device may be connected to or in communication with a backing store. The storage device may be configured to operate as a memory cache for storing data that is stored on the backing store. The storage device may be configured to operate as a backing store or main storage device. The storage device may be connected to or in communication with other similar storage devices.

[0021] The backing store may be any type of backing store, such as a hard disk drive or other type of non-volatile storage device. The access speed (or "seek time") of hard disk drives is generally limited due to the mechanical components of the drives. The access speed of memory devices such as flash devices is generally much faster than the access speed of hard disk drives. In an embodiment in which the storage device is a memory cache, at least some of the data on the backing store may be stored (or "cached") on the storage device to allow an operating system or other application to quickly access the data from the storage device rather than the backing store. In other embodiments, the storage device may not be a memory cache, but may be an additional backing store or other non-volatile storage device configured store applications and/or data associated with the computing device, and the backing store may also store applications and/or data associated with the computing device.

[0022] Many applications use object-based languages as a basis of programming. Object-based programming use objects to describe states and operations. Because so many applications use object-based programming, interactions between applications and storage devices that are block-based typically require additional programming and operating system resources, including an object store on top of the block abstraction that allows the operating system and other applications to interact with the block-based storage device. Because flash memory devices do not necessarily have to operate at a block granularity, operating the flash memory devices at a granularity that is more compatible with applications that operate at an object granularity may improve the speed and efficiency of the operating system and storage device. Specifically, a storage device that is configured to operate at a block granularity or at an object-level granularity may eliminate the need for an additional object store on top of storage layers for the storage device. Performing operations on the storage device at an operation-specific granularity includes performing the operations for the specific size of data associated with the operation. For example, the object data associated with the operation may have any size, and an object-level granularity corresponds to the size of the data (such as the object size/length), rather than to a predetermined block size.

[0023] FIG. **1** depicts a schematic diagram of one embodiment of a computing device **102** in a network system **100**. The depicted network system **100** includes various components, described in more detail below, that are capable of performing the functions and operations described herein. In one embodiment, at least some of the components of the network system **100** are implemented on the computing device **102**. For example, the functionality of one or more components of the network system may be implemented by computer program instructions stored and executed on the computing device **102**. The network system **100** may be implemented in a clustered environment or network **128** with additional computer devices. The computing device **102** may include various components, including a processor **104** (such as a CPU), input/output devices **106**, a memory device **108**, a storage manager **112**, an object store **120**, a block storage layer **114**, a device driver **122**, and a disk controller **116**. In some embodiments, the computing device **102** includes a backing store **118**. In other embodiments, the backing store **118** is not contained within the computing device **102**, but may be a standalone backing store **118** or part of another computing device **102** or system. The computing device **102** may also include or be connected to one or more storage devices **122** configured to act as a memory cache or as a primary storage device. Some or all of the components of the network system **100** may be stored on a single computing device **102** or on a network **128** of computing devices **102**, including a wired and/or wireless communication network. The network system **100** may include more or fewer components or subsystems than those depicted herein. In some embodiments, the network system **100** may be used to implement the methods described herein.

[0024] The illustrated network system **100** also includes a client application **110**. In one embodiment, the client application **110** uses object-based programming. The client application **110** may be any application that submits access requests to the backing store **118** to perform read/write/erase operations on the backing store **118**. The client application **110** may also submit access requests to the storage device **122** to perform operations on the storage device **122**. For example, the client application **110** may be an operating system (OS), a file system, a database, or some other application capable of submitting access requests to the backing store **118**. In gen-

eral, the client application **110** operates in conjunction with the storage manager **112** to access data from either the storage device **122** or the backing store **118**. In one embodiment, the storage manager **112** accesses the backing store **118** via the block storage layer **114**. The storage manager **112** may be implemented via software in some embodiments. The block storage layer **114** may be implemented in a device driver **120**, a volume manager or driver interface. The block storage layer **114** provides support to the storage manager for block-based file systems, including traditional file systems, database systems, and other software designed for magnetic disk drives. Thus, the block storage layer **114** may provide support to the storage manager **112** for the backing store **118** and other block-based storage devices. The computing device **102** may include a disk controller **116** between the block storage layer **114** and the backing store **118** to allow the block storage layer **114** to correctly locate data or specific sectors on the backing store **118**.

[0025] In one embodiment, the backing store **118** is a block-based backing store **118**, such that an object store **120** is built on top of the block storage layer **114** so that the client application **110** is able to interface with the backing store **118**. In one embodiment, the storage manager **112** accesses the storage device **122** via a device driver **120** or other driver interface. The storage device **122** may operate at a non-block granularity. The device driver **120** may interface with the storage device to perform object-based storage for the storage device **122**. In one embodiment, at least some of the operations for object-based storage are implemented at each of the storage device **122** and the device driver **120**.

[0026] The storage device **122** may include a hardware device manager, such as a controller **124**, that provides at least some of the functionality for object-based storage. In certain embodiments, the hardware device manager is a hardware storage device manager. In other embodiments, the hardware device manager is a hardware memory device manager. In certain embodiments, the hardware device manager may comprise a device driver **120** configured to control a storage device **122**. In other embodiments, the hardware device manager may comprise a device driver **120** configured to control a non-volatile memory device (not illustrated).

[0027] The controller **124** may be a device interface for interacting with the device driver **120**. The controller **124** may include hardware, firmware, device driver and/or software implementations, or any combination thereof. In one embodiment, the storage manager **112** accesses the storage device via the driver **120**. The controller **124** on the storage device **122** exposes direct access to memory elements **126** on the storage device **122** to the device driver **120**. In another embodiment, the controller **124** maintains a log structure on the storage device **122** and presents the log structure to the device driver **120**. The device driver **120** may cooperate with hardware support offered by the controller **124** corresponding to the storage device **122**. In one embodiment, the device driver **120** may provide support for any type of storage granularity, including an object-level granularity, which may include mapping physical addresses for the memory elements **126** to logical addresses or virtual addresses. Object-level granularity may reduce programming requirements or operating system resources needed to interact with the storage device **122** because object-level granularity performs the operations associated with access requests at a granularity or size for the specific data associated with the access requests. For example, an object operation may be performed directly on the storage device **122** at a granularity that corresponds to the size of the object of the object operation, rather than at a granularity that corresponds to a predetermined block size for block-based storage media. Implementing such functions at the device driver **120** also eliminates the need for a separate address translation layer at the controller **124** on the storage device **122**.

[0028] Address translation for the backing store **118** or storage device **122** may include operating independently of an existing operating system and file system to map physical addresses such as physical block addresses (PBAs) of the memory elements **126** to the logical block addresses (LBAs) in an organized structure. In other embodiments, the address translation, for example at the device driver **120** or block storage layer **114**, operates in conjunction with an existing operating system on the computing device **102** to map the physical addresses of the memory elements **126** or backing store **118** to the LBAs. The LBAs allow storage manager **112** to maintain a logical organization for the storage device while potentially storing related data in different physical locations in the storage device. The device driver **120** may also manage where data is written so that data is written to the correct locations in the storage device **122** based on where the storage device **122** has been cleaned or erased, so that subsequent access requests to the data are directed to the correct physical locations in the storage device **122**.

[0029] The storage manager **112** or an address translation layer (ATL) also may map the LBAs to PBAs on the backing store **118**, in an embodiment in which the backing store **118** is not a block-based storage device. This may allow the ATL to manage and track the data on the backing store **118**. In one embodiment, the ATL maps an LBA to a single PBA of the backing store **118**.

[0030] In another embodiment, the device driver **120** manages storing allocation information for each object within the storage device **122**. Each object corresponds to a virtual location presented by the device driver **120** to the operating system and higher level software layers. Within the storage device **122**, access requests are processed at an object-level or operation-specific granularity, rather than predetermined blocks. The granularity corresponds to a physical address (or group of physical addresses) of the memory elements **126** associated with the operation. Using this addressing approach, the device driver **120** may translate the logical address presented by the client application **110** to one or more physical addresses of the corresponding memory elements **126**. The device driver **120** or storage manager **112** may perform additional mapping if needed. In order to facilitate these mappings, the device driver **120** may manage various data structures. For example, the device driver **120** may manage a table (such as an indirection table) for tracking the location and size of each object on the storage device **122**. The device driver **120** may manage the objects in a log-structured manner. In various embodiments, the indirection table and/or log structure for the storage device **122** may be stored on the computing device **102** (for example, in memory **108**) or on the storage device **122**.

[0031] The storage device **122** may be any kind of storage device **122**. The storage device **122** may be a non-volatile storage device in which data stored on the storage device **122** persists across reboots, such that on reboot of the storage device **122**, the data may need to be invalidated for various reasons. These reasons may include, but are not limited to, changes in the data for the corresponding locations on the

backing store and/or storing information related to the ATL in volatile memory which is erased during a reboot.

[0032] In one embodiment, memory elements **126** in the storage device **122** for storing data are organized in an array or in multiple arrays. The storage device **122** may be a volatile storage device or a caching device implemented using any known caching technology. In some embodiments, the memory elements **126** are cells that are part of an integrated circuit (IC) package or chip. Each chip may include one or more die, and each die includes an array of memory elements **126**.

[0033] The storage device **122** may be used for storing data associated with the computing device **102** or other computing devices **102** connected to a network **128**. Although the computing device **102** is shown with two storage devices **122**, other embodiments of the computing device **102** may include one or more than one storage device **122**. Similarly, multiple storage devices **122** may be implemented at various locations within the nodes of the network **128**. Embodiments of the network **128** may provide dedicated or shared memory resources for one or more of the computing devices **102**, though other implementations of storage/memory resources or capacity may be used in conjunction with the network **128**.

[0034] The memory elements **126** may be operated in a variety of modes. In general, solid-state memory elements **126** can be set to different programmable states that correspond to different bits or bit combinations. In a specific example, the memory elements **126** may be operated in a single level cell (SLC) mode to store a single bit of data. In another example, the memory elements **126** may be operated in a multiple level cell (MLC) mode to store two or more bits of data. In another example, the memory elements **126** may be MLC memory elements configured to operate in an SLC mode. In other embodiments, the storage device **122** includes other types of memory elements **126**.

[0035] Although the components of the network system **100** are shown separately, one or more components may provide some or all of the functionality of other components in the network system **100**. For example, while the storage manager **112** is depicted between the client application **110** and the block storage layer **114** and device driver **120**, some or all of the functionality of the storage manager **112** may be implemented at the client application **110** or at either the block storage layer **114** or the device driver **120**. Conversely, some or all of the functionality of the block storage layer **114** and device driver **120** may be implemented at the storage manager **112**. Alternatively, the components of the computing device **102** may be different than shown in FIG. **1** while being configured to perform the operations described herein.

[0036] In one embodiment, the controller **124** on the storage device **122** interfaces with the device driver **120** to perform the operations for storing data corresponding to access requests from the client application **110** at an object-level or operation-specific granularity. The operation-specific granularity is determined by the data associated with the access request. The data may correspond to an object associated with the client application **110**. In such an embodiment, the data is stored on the storage device **122** or accessed from the storage device **120** (or other operation associated with the access request, such as creating or destroying objects) at an object-level granularity, rather than a block granularity as with block-based storage devices.

[0037] To operate at an object-level granularity, the device driver **120** and controller **124** may be able to perform various functions associated with storing objects on the storage device **122**. For example, the functions may include object operations such as: creating an object with an associated object identifier; destroying an object stored on the storage device **122**; reading an object from the storage device **122** based on the associated object identifier and a size of the object; and writing data associated with an object already present on the storage device **122** based on the object identifier and a size of the object. The functions may include maintaining an indirection table that may be used to track the location and sizes of the objects on the storage device **122**. The functions may include performing read/write operations on the storage device **122** at an arbitrary granularity, as described in accordance with the object operations. The functions may include garbage collection on the storage device **122** to maintain the log structure of the storage device **122** by eliminating dead objects and consolidating (or compacting) live objects on the storage device **122**. Dead objects may include objects that are no longer used or referenced by the client application **110**. Live objects may include valid data for objects used or referenced by the client application **110**. In some embodiments, the storage device **122** may also include dirty objects. Dirty objects may be objects that are new, modified, or deleted, but that have not been committed to the storage device **122** or backing store **118**. To perform garbage collection, the functions may include performing read-modify-write operations on object data stored on the storage device.

[0038] FIG. **2** depicts a schematic diagram of one embodiment **200** of a non-volatile storage device **122** that includes a non-volatile storage device controller **124**. In one embodiment, the storage device controller **124** is the storage device controller of FIG. **1**. The non-volatile storage device controller **124** may include a number of storage controllers **0**-N **204***a-n*, each controlling non-volatile storage media **205**. In the depicted embodiment, two non-volatile controllers are shown: non-volatile controller **0** **204***a* and storage controller N **204***n*, and each controlling respective non-volatile storage media **205***a-n*. In the depicted embodiment, storage controller **0** **204***a* controls a data channel so that the attached non-volatile storage media **205***a* stores data. Storage controller N **204***n* controls an index metadata channel associated with the stored data and the associated non-volatile storage media **205***n* stores index metadata. In an alternate embodiment, the non-volatile storage device controller **124** includes a single non-volatile controller **204***a* with a single non-volatile storage media **205***a*. In another embodiment, there are a plurality of storage controllers **104***a-n* and associated non-volatile storage media **205***a-n*. In one embodiment, one or more non-volatile controllers **104***a*-**104***n*−1, coupled to their associated non-volatile storage media **205***a*-**110***n*−1, control data while at least one storage controller **204***n*, coupled to its associated non-volatile storage media **205***n*, controls index metadata.

[0039] In one embodiment, at least one non-volatile controller **204** is a field-programmable gate array ("FPGA") and controller functions are programmed into the FPGA. In another embodiment, the storage controller **204** includes components specifically designed as a storage controller **204**, such as an application-specific integrated circuit ("ASIC") or custom logic solution. Each storage controller **204** typically includes a write data pipeline **301** and a read data pipeline **303**, which are describe further in relation to FIG. **3**. In

another embodiment, at least one storage controller **204** is made up of a combination FPGA, ASIC, and custom logic components.

[0040] The non-volatile storage media **205** is an array of non-volatile non-volatile storage elements **216**, **218**, **220**, arranged in banks **214**, and accessed in parallel through a bi-directional storage input/output ("I/O") bus **210**. The storage I/O bus **210**, in one embodiment, is capable of unidirectional communication at any one time. For example, when data is being written to the non-volatile storage media **205**, data cannot be read from the non-volatile storage media **205**. In another embodiment, data can flow both directions simultaneously. However bi-directional, as used herein with respect to a data bus, refers to a data pathway that can have data flowing in only one direction at a time, but when data flowing one direction on the bi-directional data bus is stopped, data can flow in the opposite direction on the bi-directional data bus.

[0041] A non-volatile storage element (e.g., SSS **0.0** **216***a*) is typically configured as a chip (a package of one or more dies) or a die on a circuit board. As depicted, a non-volatile storage element (e.g., **216***a*) operates independently or semi-independently of other non-volatile storage elements (e.g., **218***a*) even if these several elements are packaged together in a chip package, a stack of chip packages, or some other package element. As depicted, a row of non-volatile storage elements **216***a*, **216***b*, **216***m* is designated as a bank **214**. As depicted, there may be "n" banks **214***a-n* and "m" non-volatile storage elements **216***a-m*, **218***a-m*, **220***a-m* per bank in an array of n×m non-volatile storage elements **216**, **218**, **220** in a non-volatile storage media **205**. Of course, different embodiments may include different values for n and m. In one embodiment, a non-volatile storage media **205***a* includes twenty non-volatile storage elements **216***a*-**216***m* per bank **214** with eight banks **214**. In one embodiment, the non-volatile storage media **205***a* includes twenty-four non-volatile storage elements **216***a*-**216***m* per bank **214** with eight banks **214**. In addition to the n×m storage elements **216***a*-**216***m*, **218***a*-**218***m*, **220***a*-**220***m*, one or more additional columns (P) may also be addressed and operated in parallel with other non-volatile storage elements **216***a*, **216***b*, **216***m* for one or more rows. The added P columns in one embodiment, store parity data for the portions of an ECC chunk (i.e., an ECC codeword) that span m storage elements for a particular bank. In one embodiment, each non-volatile storage element **216**, **218**, **220** includes single-level cell ("SLC") devices. In another embodiment, each non-volatile storage element **216**, **218**, **220** includes multi-level cell ("MLC") devices.

[0042] In one embodiment, non-volatile storage elements that share a common line **211** on the storage I/O bus **210***a* (e.g., **216***b*, **218***b*, **220***b*) are packaged together. In one embodiment, a non-volatile storage element **216**, **218**, **220** may have one or more dies per package with one or more packages stacked vertically and each die may be accessed independently. In another embodiment, a non-volatile storage element (e.g., SSS **0.0** **216***a*) may have one or more virtual dies per die and one or more dies per package and one or more packages stacked vertically and each virtual die may be accessed independently. In another embodiment, a non-volatile storage element SSS **0.0** **216***a* may have one or more virtual dies per die and one or more dies per package with some or all of the one or more dies stacked vertically and each virtual die may be accessed independently.

[0043] In one embodiment, two dies are stacked vertically with four stacks per group to form eight storage elements (e.g., SSS **0.0**-SSS **8.0**) **216***a*, **218***a* . . . **220***a*, each in a separate bank **214***a*, **214***b* . . . **214***n*. In another embodiment, 24 storage elements (e.g., SSS **0.0**-SSS **0.24**) **216***a*, **216***b*, . . . **216***m* form a logical bank **214***a* so that each of the eight logical banks has 24 storage elements (e.g., SSS**0.0**-SSS **8.24**) **216**, **218**, **220**. Data is sent to the non-volatile storage media **205** over the storage I/O bus **210** to all storage elements of a particular group of storage elements (SSS **0.0**-SSS **8.0**) **216***a*, **218***a*, **220***a*. The storage control bus **212***a* is used to select a particular bank (e.g., Bank **0** **214***a*) so that the data received over the storage I/O bus **210** connected to all banks **214** is written just to the selected bank **214***a*.

[0044] In one embodiment, the storage I/O bus **210** includes one or more independent I/O buses (**210***a.a-m* . . . **210***n.a-m*) in which the non-volatile storage elements within each column share one of the independent I/O buses that are connected to each non-volatile storage element **216**, **218**, **220** in parallel. For example, one independent I/O bus **210***a.a* of the storage I/O bus **210***a* may be physically connected to a first non-volatile storage element **216***a*, **218***a*, **220***a* of each bank **214***a-n*. A second independent I/O bus **210***a.b* of the storage I/O bus **210***b* may be physically connected to a second non-volatile storage element **216***b*, **218***b*, **220***b* of each bank **214***a-n*. Each non-volatile storage element **216***a*, **216***b*, **216***m* in a bank **214***a* (a row of non-volatile storage elements as illustrated in FIG. **2**) may be accessed simultaneously and/or in parallel. In one embodiment, where non-volatile storage elements **216**, **218**, **220** include stacked packages of dies, all packages in a particular stack are physically connected to the same independent I/O bus. As used herein, "simultaneously" also includes near simultaneous access where devices are accessed at slightly different intervals to avoid switching noise. Simultaneously is used in this context to be distinguished from a sequential or serial access in which commands and/or data are sent individually one after the other.

[0045] Typically, banks **214***a-n* are independently selected using the storage control bus **212**. In one embodiment, a bank **214** is selected using a chip enable or chip select. Where both chip select and chip enable are available, the storage control bus **212** may select one package within a stack of packages. In other embodiments, other commands are used by the storage control bus **212** to individually select one package within a stack of packages. Non-volatile storage elements **216**, **218**, **220** may also be selected through a combination of control signals and address information transmitted on storage I/O bus **210** and the storage control bus **212**.

[0046] Typically, when a packet is written to a particular location within a non-volatile storage element **216**, wherein the packet is intended to be written to a location which is specific to a particular storage element of a particular bank, a physical address is sent on the storage I/O bus **210** and is followed by the packet. The physical address contains enough information for the non-volatile storage element **216** to direct the packet to the designated location. Since all storage elements in a column of storage elements (e.g., SSS **0.0**-SSS N.**0** **216***a*, **218***a*, . . . **220***a*) are connected to the same independent I/O bus (e.g., **210***.a.a*) of the storage I/O bus **210***a*, to reach the proper location and to avoid writing the data packet to similarly addressed locations in the column of storage elements (SSS **0.0**-SSS N.**0** **216***a*, **218***a*, . . . **220***a*), the bank **214***a* that includes the non-volatile storage element SSS **0.0** **216***a* with the correct location where the data packet is to be written is

selected by the storage control bus **212***a* and other banks **214***b* . . . **214***n* of the non-volatile storage **110***a* are deselected.

[0047] Similarly, satisfying a read command on the storage I/O bus **210** requires a signal on the storage control bus **212** to select a single bank **214***a* and the appropriate location within that bank **214***a*. In one embodiment, a read command reads an entire set of addresses, and because there are multiple non-volatile storage elements **216***a*, **216***b*, . . . **216***m* in parallel in a bank **214***a*, an entire logical group of addresses is read with a read command. However, the read command may be broken into subcommands, as will be explained below with respect to bank interleave. Similarly, an entire logical group of addresses may be written to the non-volatile storage elements **216***a*, **216***b*, . . . **216***m* of a bank **214***a* in a write operation. The group of addresses read from or written to the non-volatile storage elements **216***a*, **216***b*, . . . **216***m* correspond to the particular operation and a size of the object corresponding to the operation.

[0048] An erase command may be sent out to erase a group of addresses corresponding to an object over the storage I/O bus **210** with a particular erase address to erase a particular group of addresses. Typically, storage controller **204***a* may send an erase command over the parallel paths (independent I/O buses **210***a-n.a-m*) of the storage I/O bus **210** to erase a logical group of addresses, each with a particular address to erase a particular group of addresses. Simultaneously, a particular bank (e.g., Bank **0** **214***a*) is selected over the storage control bus **212** to prevent erasure of similarly addressed locations in non-selected banks (e.g., Banks **1**-N **214***b-n*). Alternatively, no particular bank (e.g., Bank **0** **214***a*) is selected over the storage control bus **212** (or all of the banks are selected) to enable erasure of similarly addressed locations in all of the banks (Banks **1**-N **214***b-n*) in parallel. Other commands may also be sent to a particular location using a combination of the storage I/O bus **210** and the storage control bus **212**. One of skill in the art will recognize other ways to select a particular storage location using the bi-directional storage I/O bus **210** and the storage control bus **212**.

[0049] In one embodiment, packets are written sequentially to the non-volatile storage media **205**. For example, storage controller **204***a* streams packets to storage write buffers of a bank **214***a* of storage elements **216** and, when the buffers are full, the packets are programmed to a designated logical location. Storage controller **204***a* then refills the storage write buffers with packets and, when full, the packets are written to the next logical location. The next logical location may be in the same bank **214***a* or another bank (e.g., **214***b*). This process continues, logical location after logical location, typically until the addresses corresponding to an object for the particular operation are erased. In another embodiment, the streaming may continue across logical boundaries for multiple objects. The read/write/erase operations may result in operations being performed on varying numbers of non-volatile storage elements **216***a*, **216***b*, . . . **216***m* according to the objects associated with the access requests.

[0050] In a read, modify, write operation, data packets associated with requested data are located and read in a read operation. Data segments of the modified requested data that have been modified are not written to the location from which they are read. Instead, the modified data segments are again converted to data packets and then written sequentially to the next available location according to a log structure. The index entries for the respective data packets are modified to point to the packets that contain the modified data segments. The entry or entries in the index for data packets associated with the same requested data that have not been modified will include pointers to original location of the unmodified data packets. Thus, if the original requested data is maintained, for example to maintain a previous version of the requested data, the original requested data will have pointers in the index to all data packets as originally written. The new requested data may have pointers in the index to some of the original data packets and pointers to the modified data packets in the logical location that is currently being written.

[0051] In a copy operation, the index includes an entry for the original requested data mapped to a number of packets stored in the non-volatile storage media **205**. When a copy is made, a new copy of the requested data is created and a new entry is created in the index mapping the new copy of the requested data to the original packets. The new copy of the requested data is also written to the non-volatile storage media **205** with its location mapped to the new entry in the index. The new copy of the requested data packets may be used to identify the packets within the original requested data that are referenced in case changes have been made in the original requested data that have not been propagated to the copy of the requested data and the index is lost or corrupted.

[0052] In various embodiments, the non-volatile storage device controller **124** also includes a data bus **207**, a local bus **206**, a buffer controller **208**, buffers **0**-N **222***a-n*, a master controller **224**, a direct memory access ("DMA") controller **226**, a memory controller **228**, a dynamic memory array **230**, a static random memory array **232**, a management controller **234**, a management bus **236**, a bridge **238** to a system bus **240**, and miscellaneous logic **242**, which are described below. In other embodiments, the system bus **240** is coupled to one or more network interface cards ("NICs") **244**, some of which may include remote DMA ("RDMA") controllers **246**, one or more central processing unit ("CPU") **248**, one or more external memory controllers **250** and associated external memory arrays **252**, one or more storage controllers **254**, peer controllers **256**, and application specific processors **258**, which are described below. The components **244**-**258** connected to the system bus **240** may be located in the host computing system **114** or may be other devices.

[0053] Typically, the storage controller(s) **104** communicate data to the non-volatile storage media **205** over a storage I/O bus **210**. In a typical embodiment where the non-volatile storage is arranged in banks **214** and each bank **214** includes multiple storage elements **216***a*, **216***b*, **216***m* accessed in parallel, the storage I/O bus **210** is an array of busses, one for each column of storage elements **216**, **218**, **220** spanning the banks **214**. As used herein, the term "storage I/O bus" may refer to one storage I/O bus **210** or an array of independent data busses wherein individual data busses of the array independently communicate different data relative to one another. In one embodiment, each storage I/O bus **210** accessing a column of storage elements (e.g., **216***a*, **218***a*, **220***a*) may include a logical-to-physical mapping for storage divisions (e.g., erase blocks) accessed in a column of storage elements **216***a*, **218***a*, **220***a*. This mapping (or bad block remapping) allows a logical address mapped to a physical address of a storage division to be remapped to a different storage division if the first storage division fails, partially fails, is inaccessible, or has some other problem.

[0054] Data may also be communicated to the storage controller(s) **104** from a requesting device **155** through the system bus **240**, bridge **238**, local bus **206**, buffer(s) **222**, and

finally over a data bus **207**. The data bus **207** typically is connected to one or more buffers **222***a-n* controlled with a buffer controller **208**. The buffer controller **208** typically controls transfer of data from the local bus **206** to the buffers **222** and through the data bus **207** to the pipeline input buffer **306** and output buffer **330**. The buffer controller **208** typically controls how data arriving from a requesting device can be temporarily stored in a buffer **222** and then transferred onto a data bus **207**, or vice versa, to account for different clock domains, to prevent data collisions, etc. The buffer controller **208** typically works in conjunction with the master controller **224** to coordinate data flow. As data arrives, the data will arrive on the system bus **240**, be transferred to the local bus **206** through a bridge **238**.

[0055] Typically, the data is transferred from the local bus **206** to one or more data buffers **222** as directed by the master controller **224** and the buffer controller **208**. The data then flows out of the buffer(s) **222** to the data bus **207**, through a non-volatile controller **204**, and on to the non-volatile storage media **205** such as NAND flash or other storage media. In one embodiment, data and associated out-of-band metadata ("metadata") arriving with the data is communicated using one or more data channels having one or more storage controllers **104***a*-**104***n*−1 and associated non-volatile storage media **205***a*-**110***n*−1 while at least one channel (storage controller **204***n*, non-volatile storage media **205***n*) is dedicated to in-band metadata, such as index information and other metadata generated internally to the non-volatile storage device **122**.

[0056] The local bus **206** is typically a bidirectional bus or set of busses that allows for communication of data and commands between devices internal to the non-volatile storage device controller **124** and between devices internal to the non-volatile storage device **122** and devices **244**-**258** connected to the system bus **240**. The bridge **238** facilitates communication between the local bus **206** and system bus **240**. One of skill in the art will recognize other embodiments such as ring structures or switched star configurations and functions of buses **240**, **206**, **204**, **210** and bridges **238**.

[0057] The system bus **240** is typically a bus of a host computing system **114** or other device in which the non-volatile storage device **122** is installed or connected. In one embodiment, the system bus **240** may be a PCI-e bus, a Serial Advanced Technology Attachment ("serial ATA") bus, parallel ATA, or the like. In another embodiment, the system bus **240** is an external bus such as small computer system interface ("SCSI"), FireWire, Fiber Channel, USB, PCIe-AS, or the like. The non-volatile storage device **122** may be packaged to fit internally to a device or as an externally connected device.

[0058] The non-volatile storage device controller **124** includes a master controller **224** that controls higher-level functions within the non-volatile storage device **122**. The master controller **224**, in various embodiments, controls data flow by interpreting object requests and other requests, directs creation of indexes to map object identifiers associated with data to physical locations of associated data, coordinating DMA requests, etc. Many of the functions described herein are controlled wholly or in part by the master controller **224**.

[0059] In one embodiment, the master controller **224** uses embedded controller(s). In another embodiment, the master controller **224** uses local memory such as a dynamic memory array **230** (dynamic random access memory "DRAM"), a static memory array **232** (static random access memory "SRAM"), etc. In one embodiment, the local memory is controlled using the master controller **224**. In another embodiment, the master controller **224** accesses the local memory via a memory controller **228**. In another embodiment, the master controller **224** runs a Linux server and may support various common server interfaces, such as the World Wide Web, hyper-text markup language ("HTML"), etc. In another embodiment, the master controller **224** uses a nano-processor. The master controller **224** may be constructed using programmable or standard logic, or any combination of controller types listed above. One skilled in the art will recognize many embodiments for the master controller **224**.

[0060] In one embodiment, where the storage device/non-volatile storage device controller **124** manages multiple data storage devices/non-volatile storage media **205***a-n*, the master controller **224** divides the work load among internal controllers, such as the storage controllers **104***a-n*. For example, the master controller **224** may divide an object to be written to the data storage devices (e.g., non-volatile storage media **205***a-n*) so that a portion of the object is stored on each of the attached data storage devices. This feature is a performance enhancement allowing quicker storage and access to an object. In one embodiment, the master controller **224** is implemented using an FPGA. In another embodiment, the firmware within the master controller **224** may be updated through the management bus **236**, the system bus **240** over a network connected to a NIC **244** or other device connected to the system bus **240**.

[0061] In one embodiment, the master controller **224** coordinates with NIC controllers **244** and embedded RDMA controllers **246** to deliver just-in-time RDMA transfers of data and command sets. NIC controller **244** may be hidden behind a non-transparent port to enable the use of custom drivers. Also, a driver **120** on a host computing system **114** may have access to the computer network **116** through an I/O memory driver using a standard stack API and operating in conjunction with NICs **244**.

[0062] In one embodiment, the master controller **224** is also a redundant array of independent drive ("RAID") controller. Where the data storage device/non-volatile storage device **122** is networked with one or more other data storage devices/non-volatile storage devices **102**, the master controller **224** may be a RAID controller for single tier RAID, multi-tier RAID, progressive RAID, etc. The master controller **224** also allows some objects to be stored in a RAID array and other objects to be stored without RAID. In another embodiment, the master controller **224** may be a distributed RAID controller element. In another embodiment, the master controller **224** may include many RAID, distributed RAID, and other functions as described elsewhere.

[0063] In one embodiment, the master controller **224** coordinates with single or redundant network managers (e.g., switches) to establish routing, to balance bandwidth utilization, failover, etc. In another embodiment, the master controller **224** coordinates with integrated application specific logic (via local bus **206**) and associated driver software. In another embodiment, the master controller **224** coordinates with attached application specific processors **258** or logic (via the external system bus **240**) and associated driver software. In another embodiment, the master controller **224** coordinates with remote application specific logic (via the computer network **116**) and associated driver software. In another embodi-

ment, the master controller **224** coordinates with the local bus **206** or external bus attached hard disk drive ("HDD") storage controller.

[0064] In one embodiment, the master controller **224** communicates with one or more storage controllers **254** where the storage device/non-volatile storage device **122** may appear as a storage device connected through a SCSI bus, Internet SCSI ("iSCSI"), fiber channel, etc. Meanwhile the storage device/non-volatile storage device **122** may autonomously manage objects and may appear as an object file system or distributed object file system. The master controller **224** may also be accessed by peer controllers **256** and/or application specific processors **258**.

[0065] In another embodiment, the master controller **224** coordinates with an autonomous integrated management controller to periodically validate FPGA code and/or controller software, validate FPGA code while running (reset) and/or validate controller software during power on (reset), support external reset requests, support reset requests due to watchdog timeouts, and support voltage, current, power, temperature, and other environmental measurements and setting of threshold interrupts. In another embodiment, the master controller **224** manages garbage collection to free erase blocks for reuse. In another embodiment, the master controller **224** manages wear leveling. In another embodiment, the master controller **224** allows the data storage device/non-volatile storage device **122** to be partitioned into multiple logical devices and allows partition-based media encryption. In yet another embodiment, the master controller **224** supports a storage controller **204** with advanced, multi-bit ECC correction. One of skill in the art will recognize other features and functions of a master controller **224** in a storage controller **124**, or more specifically in a non-volatile storage device **122**.

[0066] In one embodiment, the non-volatile storage device controller **124** includes a memory controller **228**, which controls a dynamic random memory array **230** and/or a static random memory array **232**. As stated above, the memory controller **228** may be independent or integrated with the master controller **224**. The memory controller **228** typically controls volatile memory of some type, such as DRAM (dynamic random memory array **230**) and SRAM (static random memory array **232**). In other examples, the memory controller **228** also controls other memory types such as electrically erasable programmable read only memory ("EEPROM"), etc. In other embodiments, the memory controller **228** controls two or more memory types and the memory controller **228** may include more than one controller. Typically, the memory controller **228** controls as much SRAM **232** as is feasible and by DRAM **230** to supplement the SRAM **232**.

[0067] In one embodiment, the object index is stored in memory **230**, **232** and then periodically off-loaded to a channel of the non-volatile storage media **205***n* or other non-volatile memory. One of skill in the art will recognize other uses and configurations of the memory controller **228**, dynamic memory array **230**, and static memory array **232**.

[0068] In one embodiment, the non-volatile storage device controller **124** includes a DMA controller **226** that controls DMA operations between the storage device/non-volatile storage device **122** and one or more external memory controllers **250** and associated external memory arrays **252** and CPUs **248**. Note that the external memory controllers **250** and external memory arrays **252** are called external because they are external to the storage device/non-volatile storage device **122**. In addition, the DMA controller **226** may also control RDMA operations with requesting devices through a NIC **244** and associated RDMA controller **246**.

[0069] In one embodiment, the non-volatile storage device controller **124** includes a management controller **234** connected to a management bus **236**. Typically, the management controller **234** manages environmental metrics and status of the storage device/non-volatile storage device **122**. The management controller **234** may monitor device temperature, fan speed, power supply settings, etc. over the management bus **236**. The management controller **234** may support the reading and programming of erasable programmable read only memory ("EEPROM") for storage of FPGA code and controller software. Typically, the management bus **236** is connected to the various components within the storage device/non-volatile storage device **122**. The management controller **234** may communicate alerts, interrupts, etc. over the local bus **206** or may include a separate connection to a system bus **240** or other bus. In one embodiment, the management bus **236** is an Inter-Integrated Circuit ("I2C") bus. One of skill in the art will recognize other related functions and uses of a management controller **234** connected to components of the storage device/non-volatile storage device **122** by a management bus **236**.

[0070] In one embodiment, the non-volatile storage device controller **124** includes miscellaneous logic **242** that may be customized for a specific application. Typically, where the non-volatile device controller **124** or master controller **224** is/are configured using a FPGA or other configurable controller, custom logic may be included based on a particular application, customer requirement, storage requirement, etc.

[0071] While the controller **124** of FIG. **2** is shown, other embodiments of a controller **124** may be used to implement the operations for object-based storage on a storage device **122**.

[0072] FIG. **3** depicts a schematic diagram of one embodiment **300** of the storage controller **204** of FIG. **2** with a write data pipeline **301**, a read data pipeline **303** and a throughput management apparatus **122** in a non-volatile storage device **122**. The embodiment **300** includes a data bus **207**, a local bus **206**, and buffer control **208**, which are substantially similar to those described in relation to the non-volatile storage device controller **124** of FIG. **2**. The write data pipeline **301** includes a packetizer **302** and an error-correcting code ("ECC") generator **304**. In other embodiments, the write data pipeline **301** includes an input buffer **306**, a write synchronization buffer **308**, a write program module **310**, a compression module **312**, an encryption module **314**, a garbage collector bypass **316** (with a portion within the read data pipeline **303**), a media encryption module **318**, and a write buffer **320**. The read data pipeline **303** includes a read synchronization buffer **328**, an ECC correction module **322**, a depacketizer **324**, an alignment module **326**, and an output buffer **330**. In other embodiments, the read data pipeline **303** may include a media decryption module **332**, a portion of the garbage collector bypass **316**, a decryption module **334**, a decompression module **336**, and a read program module **338**. The storage controller **204** may also include control and status registers **340** and control queues **342**, a bank interleave controller **344**, a synchronization buffer **346**, a storage bus controller **348**, and a multiplexer ("MUX") **350**. The components of the non-volatile controller **204** and associated write data pipeline **301** and read data pipeline **303** are described below. In other

embodiments, synchronous non-volatile storage media **205** may be used and synchronization buffers **308**, **328** may be eliminated.

[0073] The write data pipeline **301** includes a packetizer **302** that receives a data or metadata segment to be written to the non-volatile storage, either directly or indirectly through another write data pipeline **301** stage, and creates one or more packets sized for the non-volatile storage media **205**. The data or metadata segment is typically part of a data structure such as an object, but may also include an entire data structure. In another embodiment, the data segment is part of a block of data, but may also include an entire block of data. Typically, a set of data such as a data structure is received from a computer such as the host computing system **114**, or other computer or device and is transmitted to the non-volatile storage device **122** in data segments streamed to the non-volatile storage device **122**. A data segment may also be known by another name, such as data parcel, but as referenced herein includes all or a portion of a data structure or data block.

[0074] Each data structure is stored as one or more packets. Each data structure may have one or more container packets. Each packet contains a header. The header may include a header type field. Type fields may include data, attribute, metadata, data segment delimiters (multi-packet), data structures, data linkages, and the like. The header may also include information regarding the size of the packet, such as the number of bytes of data included in the packet. The length of the packet may be established by the packet type. The header may include information that establishes the relationship of the packet to a data structure. An example might be the use of an offset in a data packet header to identify the location of the data segment within the data structure. One of skill in the art will recognize other information that may be included in a header added to data by a packetizer **302** and other information that may be added to a data packet.

[0075] Each packet includes a header and possibly data from the data or metadata segment. The header of each packet includes pertinent information to relate the packet to the data structure to which the packet belongs. For example, the header may include an object identifier or other data structure identifier and offset that indicate the data segment, object, data structure or data block from which the data packet was formed. The header may also include a logical address used by the storage bus controller **348** to store the packet. The header may also include information regarding the size of the packet, such as the number of bytes included in the packet. The header may also include a sequence number that identifies where the data segment belongs with respect to other packets within the data structure when reconstructing the data segment or data structure. The header may include a header type field. Type fields may include data, data structure attributes, metadata, data segment delimiters (multi-packet), data structure types, data structure linkages, and the like. One of skill in the art will recognize other information that may be included in a header added to data or metadata by a packetizer **302** and other information that may be added to a packet.

[0076] In one embodiment, the write data pipeline **301** includes an ECC generator **304** that that generates one or more error-correcting codes ("ECC") for the one or more packets received from the packetizer **302**. In other embodiments, the write data pipeline **301** does not include an ECC generator **304**. The ECC generator **304** typically uses an error-correcting algorithm to generate ECC check bits, which are stored with the one or more data packets. The ECC codes generated by the ECC generator **304** together with the one or more data packets associated with the ECC codes include an ECC chunk. The ECC data stored with the one or more data packets is used to detect and to correct errors introduced into the data through transmission and storage. In one embodiment, packets are streamed into the ECC generator **304** as un-encoded blocks of length N. A syndrome of length S is calculated, appended, and output as an encoded block of length N+S. The value of N and S are dependent upon the characteristics of the ECC algorithm, which is selected to achieve specific performance, efficiency, and robustness metrics. In one embodiment, there is no fixed relationship between the ECC blocks and the packets; the packet may include more than one ECC block; the ECC block may include more than one packet; and a first packet may end anywhere within the ECC block and a second packet may begin after the end of the first packet within the same ECC block. In one embodiment, ECC algorithms are not dynamically modified. In one embodiment, the ECC data stored with the data packets is robust enough to correct errors in more than two bits.

[0077] Beneficially, using a robust ECC algorithm allowing more than single bit correction or even double bit correction allows the life of the non-volatile storage media **205** to be extended. For example, if flash memory is used as the storage medium in the non-volatile storage media **205**, the flash memory may be written approximately 100,000 times without error per erase cycle. This usage limit may be extended using a robust ECC algorithm. Having the ECC generator **304** and corresponding ECC correction module **322** onboard the non-volatile storage device **122**, the non-volatile storage device **122** can internally correct errors and has a longer useful life than if a less robust ECC algorithm is used, such as single bit correction. However, in other embodiments the ECC generator **304** may use a less robust algorithm and may correct single-bit or double-bit errors. In another embodiment, the non-volatile storage device **110** may include less reliable storage such as multi-level cell ("MLC") flash in order to increase capacity, which storage may not be sufficiently reliable without more robust ECC algorithms.

[0078] In one embodiment, the write pipeline **301** includes an input buffer **306** that receives a data segment to be written to the non-volatile storage media **205** and stores the incoming data segments until the next stage of the write data pipeline **301**, such as the packetizer **302** (or other stage for a more complex write data pipeline **301**) is ready to process the next data segment. The input buffer **306** typically allows for discrepancies between the rate data segments are received and processed by the write data pipeline **301** using an appropriately sized data buffer. The input buffer **306** also allows the data bus **207** to transfer data to the write data pipeline **301** at rates greater than can be sustained by the write data pipeline **301** in order to improve efficiency of operation of the data bus **207**. Typically, when the write data pipeline **301** does not include an input buffer **306**, a buffering function is performed elsewhere, such as in the non-volatile storage device **122** but outside the write data pipeline **301**, in the host computing system **114**, such as within a network interface card ("NIC"), or at another device, for example when using remote direct memory access ("RDMA").

[0079] In another embodiment, the write data pipeline **301** also includes a write synchronization buffer **308** that buffers packets received from the ECC generator **304** prior to writing

the packets to the non-volatile storage media **205**. The write synchronization buffer **308** is located at a boundary between a local clock domain and a non-volatile storage clock domain and provides buffering to account for the clock domain differences. In other embodiments, synchronous non-volatile storage media **205** may be used and synchronization buffers **308** **328** may be eliminated.

[0080] In one embodiment, the write data pipeline **301** also includes a media encryption module **318** that receives the one or more packets from the packetizer **302**, either directly or indirectly, and encrypts the one or more packets using an encryption key unique to the non-volatile storage device **122** prior to sending the packets to the ECC generator **304**. Typically, the entire packet is encrypted, including the headers. In another embodiment, headers are not encrypted. In this document, encryption key is understood to mean a secret encryption key that is managed externally from a storage controller **204**.

[0081] The media encryption module **318** and corresponding media decryption module **332** provide a level of security for data stored in the non-volatile storage media **205**. For example, where data is encrypted with the media encryption module **318**, if the non-volatile storage media **205** is connected to a different storage controller **204**, non-volatile storage device **122**, or server, the contents of the non-volatile storage media **205** typically could not be read without use of the same encryption key used during the write of the data to the non-volatile storage media **205** without significant effort.

[0082] In a typical embodiment, the non-volatile storage device **122** does not store the encryption key in non-volatile storage and allows no external access to the encryption key. The encryption key is provided to the storage controller **204** during initialization. The non-volatile storage device **122** may use and store a non-secret cryptographic nonce that is used in conjunction with an encryption key. A different nonce may be stored with every packet. Data segments may be split between multiple packets with unique nonces for the purpose of improving protection by the encryption algorithm.

[0083] The encryption key may be received from a host computing system **114**, a server, key manager, or other device that manages the encryption key to be used by the storage controller **204**. In another embodiment, the non-volatile storage media **205** may have two or more partitions and the storage controller **204** behaves as though it was two or more storage controllers **104**, each operating on a single partition within the non-volatile storage media **205**. In this embodiment, a unique media encryption key may be used with each partition.

[0084] In another embodiment, the write data pipeline **301** also includes an encryption module **314** that encrypts a data or metadata segment received from the input buffer **306**, either directly or indirectly, prior sending the data segment to the packetizer **302**, the data segment encrypted using an encryption key received in conjunction with the data segment. The encryption keys used by the encryption module **314** to encrypt data may not be common to all data stored within the non-volatile storage device **122** but may vary on an per data structure basis and received in conjunction with receiving data segments as described below. For example, an encryption key for a data segment to be encrypted by the encryption module **314** may be received with the data segment or may be received as part of a command to write a data structure to which the data segment belongs. The storage device **122** may use and store a non-secret cryptographic nonce in each data structure packet that is used in conjunction with the encryption key. A different nonce may be stored with every packet. Data segments may be split between multiple packets with unique nonces for the purpose of improving protection by the encryption algorithm.

[0085] The encryption key may be received from a host computing device **102**, another computer, key manager, or other device that holds the encryption key to be used to encrypt the data segment. In one embodiment, encryption keys are transferred to the storage controller **204** from one of a non-volatile storage device **122**, host computing device **102**, computer, or other external agent, which has the ability to execute industry standard methods to securely transfer and protect private and public keys.

[0086] In one embodiment, the encryption module **314** encrypts a first packet with a first encryption key received in conjunction with the packet and encrypts a second packet with a second encryption key received in conjunction with the second packet. In another embodiment, the encryption module **314** encrypts a first packet with a first encryption key received in conjunction with the packet and passes a second data packet on to the next stage without encryption. Beneficially, the encryption module **314** included in the write data pipeline **301** of the non-volatile storage device **122** allows data structure-by-data structure or segment-by-segment data encryption without a single file system or other external system to keep track of the different encryption keys used to store corresponding data structures or data segments. Each requesting device **155** or related key manager independently manages encryption keys used to encrypt only the data structures or data segments sent by the requesting device **155**.

[0087] In one embodiment, the encryption module **314** may encrypt the one or more packets using an encryption key unique to the non-volatile storage device **122**. The encryption module **314** may perform this media encryption independently, or in addition to the encryption described above. Typically, the entire packet is encrypted, including the headers. In another embodiment, headers are not encrypted. The media encryption by the encryption module **314** provides a level of security for data stored in the non-volatile storage media **205**. For example, where data is encrypted with media encryption unique to the specific non-volatile storage device **122**, if the non-volatile storage media **205** is connected to a different storage controller **204**, non-volatile storage device **122**, or host computing system **114**, the contents of the non-volatile storage media **205** typically could not be read without use of the same encryption key used during the write of the data to the non-volatile storage media **205** without significant effort.

[0088] In another embodiment, the write data pipeline **301** includes a compression module **312** that compresses the data for metadata segment prior to sending the data segment to the packetizer **302**. The compression module **312** typically compresses a data or metadata segment using a compression routine known to those of skill in the art to reduce the storage size of the segment. For example, if a data segment includes a string of 512 zeros, the compression module **312** may replace the 512 zeros with code or token indicating the 512 zeros where the code is much more compact than the space taken by the 512 zeros.

[0089] In one embodiment, the compression module **312** compresses a first segment with a first compression routine and passes along a second segment without compression. In another embodiment, the compression module **312** compresses a first segment with a first compression routine and

compresses the second segment with a second compression routine. Having this flexibility within the non-volatile storage device **122** is beneficial so that computing devices **102** or other devices writing data to the non-volatile storage device **122** may each specify a compression routine or so that one can specify a compression routine while another specifies no compression. Selection of compression routines may also be selected according to default settings on a per data structure type or data structure class basis. For example, a first data structure of a specific data structure may be able to override default compression routine settings and a second data structure of the same data structure class and data structure type may use the default compression routine and a third data structure of the same data structure class and data structure type may use no compression.

[0090] In one embodiment, the write data pipeline **301** includes a garbage collector bypass **316** that receives data segments from the read data pipeline **303** as part of a data bypass in a garbage collection system. A garbage collection system (also referred to as a "groomer" or grooming operation) typically marks packets that are no longer valid, typically because the packet is marked for deletion or has been modified and the modified data is stored in a different location. At some point, the garbage collection system determines that a particular section (e.g., an erase block) of storage may be recovered. This determination may be due to a lack of available storage capacity, the percentage of data marked as invalid reaching a threshold, a consolidation of valid data, an error detection rate for that section of storage reaching a threshold, or improving performance based on data distribution, etc. Numerous factors may be considered by a garbage collection algorithm to determine when a section of storage is to be recovered.

[0091] Once a section of storage has been marked for recovery, valid packets in the section typically must be relocated. The garbage collector bypass **316** allows packets to be read into the read data pipeline **303** and then transferred directly to the write data pipeline **301** without being routed out of the storage controller **204**. In one embodiment, the garbage collector bypass **316** is part of an autonomous garbage collector system that operates within the non-volatile storage device **122**. This allows the non-volatile storage device **122** to manage data so that data is systematically spread throughout the non-volatile storage media **205** to improve performance, data reliability and to avoid overuse and underuse of any one location or area of the non-volatile storage media **205** and to lengthen the useful life of the non-volatile storage media **205**.

[0092] The garbage collector bypass **316** coordinates insertion of segments into the write data pipeline **106** with other segments being written by computing devices **102** or other devices. In the depicted embodiment, the garbage collector bypass **316** is before the packetizer **302** in the write data pipeline **301** and after the depacketizer **324** in the read data pipeline **303**, but may also be located elsewhere in the read and write data pipelines **106**, **108**. The garbage collector bypass **316** may be used during a flush of the write pipeline **303** to fill the remainder of the particular section of storage in order to improve the efficiency of storage within the non-volatile storage media **205** and thereby reduce the frequency of garbage collection.

[0093] Grooming may include refreshing data stored on the non-volatile storage media **205**. Data stored on the non-volatile storage media **205** may degrade over time. The storage controller **204** may include a groomer that identifies "stale" data on the non-volatile storage device **122** (data that has not been modified and/or moved for a pre-determined time), and refreshes the stale data by re-writing the data to a different storage location.

[0094] In some embodiments, the garbage collection system, groomer, and/or garbage collection bypass **316** may be temporarily disabled to allow data to be stored contiguously on physical storage locations of the non-volatile storage device **122**. Disabling the garbage collection system and/or bypass **316** may ensure that data in the write data pipeline **301** is not interleaved with other data. For example, and discussed below, garbage collection and/or the garbage collection bypass **316** may be disabled when storing data pertaining to an atomic storage request.

[0095] In some embodiments, the garbage collection and/or groomer may be restricted to a certain portion of the physical storage space of the non-volatile storage device. For example, storage metadata, such as the reverse index described below, may be periodically persisted to a non-volatile storage location. The garbage collection and/or grooming may be restricted to operating on portions of the non-volatile storage media that correspond to the persisted storage metadata.

[0096] In one embodiment, the write data pipeline **301** includes a write buffer **320** that buffers data for efficient write operations. Typically, the write buffer **320** includes enough capacity for packets to fill at least one logical page in the non-volatile storage media **205**. This allows a write operation to send an entire logical page of data to the non-volatile storage media **205** without interruption. By sizing the write buffer **320** of the write data pipeline **301** and buffers within the read data pipeline **303** to be the same capacity or larger than a storage write buffer within the non-volatile storage media **205**, writing and reading data is more efficient since a single write command may be crafted to send a full logical page of data to the non-volatile storage media **205** instead of multiple commands.

[0097] While the write buffer **320** is being filled, the non-volatile storage media **205** may be used for other read operations. This is advantageous because other non-volatile devices with a smaller write buffer or no write buffer may tie up the non-volatile storage when data is written to a storage write buffer and data flowing into the storage write buffer stalls. Read operations will be blocked until the entire storage write buffer is filled and programmed. Another approach for systems without a write buffer or a small write buffer is to flush the storage write buffer that is not full in order to enable reads.

[0098] In one embodiment, the write buffer **320** is a ping-pong buffer where one side of the buffer is filled and then designated for transfer at an appropriate time while the other side of the ping-pong buffer is being filled. In another embodiment, the write buffer **320** includes a first-in first-out ("FIFO") register with a capacity of more than a logical page of data segments. One of skill in the art will recognize other write buffer **320** configurations that allow a logical page of data to be stored prior to writing the data to the non-volatile storage media **205**.

[0099] In one embodiment, the write data pipeline **301** includes a write program module **310** with one or more user-definable functions within the write data pipeline **301**. The write program module **310** allows a user to customize the write data pipeline **301**. A user may customize the write data

pipeline **301** based on a particular data requirement or application. Where the storage controller **204** is an FPGA, the user may program the write data pipeline **301** with custom commands and functions relatively easily. A user may also use the write program module **310** to include custom functions with an ASIC, however, customizing an ASIC may be more difficult than with an FPGA. The write program module **310** may include buffers and bypass mechanisms to allow a first data segment to execute in the write program module **310** while a second data segment may continue through the write data pipeline **301**. In another embodiment, the write program module **310** may include a processor core that can be programmed through software.

[0100] Note that the write program module **310** is shown between the input buffer **306** and the compression module **312**, however, the write program module **310** could be anywhere in the write data pipeline **301** and may be distributed among the various stages **302-320**. In addition, there may be multiple write program modules **310** distributed among the various states **302-320** that are programmed and operate independently. In addition, the order of the stages **302-320** may be altered. One of skill in the art will recognize workable alterations to the order of the stages **302-320** based on particular user requirements.

[0101] The read data pipeline **303** includes an ECC correction module **322** that determines if a data error exists in ECC blocks a requested packet received from the non-volatile storage media **205** by using ECC stored with each ECC block of the requested packet. The ECC correction module **322** then corrects any errors in the requested packet if any error exists and the errors are correctable using the ECC. For example, if the ECC can detect an error in six bits but can only correct three bit errors, the ECC correction module **322** corrects ECC blocks of the requested packet with up to three bits in error. The ECC correction module **322** corrects the bits in error by changing the bits in error to the correct one or zero state so that the requested data packet is identical to when it was written to the non-volatile storage media **205** and the ECC was generated for the packet.

[0102] If the ECC correction module **322** determines that the requested packets contains more bits in error than the ECC can correct, the ECC correction module **322** cannot correct the errors in the corrupted ECC blocks of the requested packet and sends an interrupt. In one embodiment, the ECC correction module **322** sends an interrupt with a message indicating that the requested packet is in error. The message may include information that the ECC correction module **322** cannot correct the errors or the inability of the ECC correction module **322** to correct the errors may be implied. In another embodiment, the ECC correction module **322** sends the corrupted ECC blocks of the requested packet with the interrupt and/or the message.

[0103] In one embodiment, a corrupted ECC block or portion of a corrupted ECC block of the requested packet that cannot be corrected by the ECC correction module **322** is read by the master controller **224**, corrected, and returned to the ECC correction module **322** for further processing by the read data pipeline **303**. In one embodiment, a corrupted ECC block or portion of a corrupted ECC block of the requested packet is sent to the device requesting the data. The requesting device **155** may correct the ECC block or replace the data using another copy, such as a backup or mirror copy, and then may use the replacement data of the requested data packet or return it to the read data pipeline **303**. The requesting device **155** may use header information in the requested packet in order to identify data required to replace the corrupted requested packet or to replace the data structure to which the packet belongs. In another embodiment, the storage controller **204** stores data using some type of RAID and is able to recover the corrupted data. In another embodiment, the ECC correction module **322** sends an interrupt and/or message and the receiving device fails the read operation associated with the requested data packet. One of skill in the art will recognize other options and actions to be taken as a result of the ECC correction module **322** determining that one or more ECC blocks of the requested packet are corrupted and that the ECC correction module **322** cannot correct the errors.

[0104] The read data pipeline **303** includes a depacketizer **324** that receives ECC blocks of the requested packet from the ECC correction module **322**, directly or indirectly, and checks and removes one or more packet headers. The depacketizer **324** may validate the packet headers by checking packet identifiers, data length, data location, etc. within the headers. In one embodiment, the header includes a hash code that can be used to validate that the packet delivered to the read data pipeline **303** is the requested packet. The depacketizer **324** also removes the headers from the requested packet added by the packetizer **302**. The depacketizer **324** may directed to not operate on certain packets but pass these forward without modification. An example might be a container label that is requested during the course of a rebuild process where the header information is required for index reconstruction. Further examples include the transfer of packets of various types destined for use within the non-volatile storage device **122**. In another embodiment, the depacketizer **324** operation may be packet type dependent.

[0105] The read data pipeline **303** includes an alignment module **326** that receives data from the depacketizer **324** and removes unwanted data. In one embodiment, a read command sent to the non-volatile storage media **205** retrieves a packet of data. A device requesting the data may not require all data within the retrieved packet and the alignment module **326** removes the unwanted data.

[0106] The alignment module **326** re-formats the data as data segments of a data structure in a form compatible with a device requesting the data segment prior to forwarding the data segment to the next stage. Typically, as data is processed by the read data pipeline **303**, the size of data segments or packets changes at various stages. The alignment module **326** uses received data to format the data into data segments suitable to be sent to the requesting device **155** and joined to form a response. For example, data from a portion of a first data packet may be combined with data from a portion of a second data packet. If a data segment is larger than a data requested by the requesting device **155**, the alignment module **326** may discard the unwanted data.

[0107] In one embodiment, the read data pipeline **303** includes a read synchronization buffer **328** that buffers one or more requested packets read from the non-volatile storage media **205** prior to processing by the read data pipeline **303**. The read synchronization buffer **328** is at the boundary between the non-volatile storage clock domain and the local bus clock domain and provides buffering to account for the clock domain differences.

[0108] In another embodiment, the read data pipeline **303** includes an output buffer **330** that receives requested packets from the alignment module **326** and stores the packets prior to transmission to the requesting device **155**. The output buffer

330 accounts for differences between when data segments are received from stages of the read data pipeline 303 and when the data segments are transmitted to other parts of the storage controller 204 or to the requesting device 155. The output buffer 330 also allows the data bus 207 to receive data from the read data pipeline 303 at rates greater than can be sustained by the read data pipeline 303 in order to improve efficiency of operation of the data bus 207.

[0109] In one embodiment, the read data pipeline 303 includes a media decryption module 332 that receives one or more encrypted requested packets from the ECC correction module 322 and decrypts the one or more requested packets using the encryption key unique to the non-volatile storage device 122 prior to sending the one or more requested packets to the depacketizer 324. Typically, the encryption key used to decrypt data by the media decryption module 332 is identical to the encryption key used by the media encryption module 318. In another embodiment, the non-volatile storage media 205 may have two or more partitions and the storage controller 204 behaves as though it was two or more storage controllers 104 each operating on a single partition within the non-volatile storage media 205. In this embodiment, a unique media encryption key may be used with each partition.

[0110] In another embodiment, the read data pipeline 303 includes a decryption module 334 that decrypts a data segment formatted by the depacketizer 324 prior to sending the data segment to the output buffer 330. The data segment may be decrypted using an encryption key received in conjunction with the read request that initiates retrieval of the requested packet received by the read synchronization buffer 328. The decryption module 334 may decrypt a first packet with an encryption key received in conjunction with the read request for the first packet and then may decrypt a second packet with a different encryption key or may pass the second packet on to the next stage of the read data pipeline 303 without decryption. When the packet was stored with a non-secret cryptographic nonce, the nonce is used in conjunction with an encryption key to decrypt the data packet. The encryption key may be received from a host computing system 114, a client, key manager, or other device that manages the encryption key to be used by the storage controller 204.

[0111] In another embodiment, the read data pipeline 303 includes a decompression module 336 that decompresses a data segment formatted by the depacketizer 324. In one embodiment, the decompression module 336 uses compression information stored in one or both of the packet header and the container label to select a complementary routine to that used to compress the data by the compression module 312. In another embodiment, the decompression routine used by the decompression module 336 is dictated by the device requesting the data segment being decompressed. In another embodiment, the decompression module 336 selects a decompression routine according to default settings on a per data structure type or data structure class basis. A first packet of a first object may be able to override a default decompression routine and a second packet of a second data structure of the same data structure class and data structure type may use the default decompression routine and a third packet of a third data structure of the same data structure class and data structure type may use no decompression.

[0112] In another embodiment, the read data pipeline 303 includes a read program module 338 that includes one or more user-definable functions within the read data pipeline 303. The read program module 338 has similar characteristics to the write program module 310 and allows a user to provide custom functions to the read data pipeline 303. The read program module 338 may be located as shown in FIG. 3, may be located in another position within the read data pipeline 303, or may include multiple parts in multiple locations within the read data pipeline 303. Additionally, there may be multiple read program modules 338 within multiple locations within the read data pipeline 303 that operate independently. One of skill in the art will recognize other forms of a read program module 338 within a read data pipeline 303. As with the write data pipeline 301, the stages of the read data pipeline 303 may be rearranged and one of skill in the art will recognize other orders of stages within the read data pipeline 303.

[0113] The storage controller 204 includes control and status registers 340 and corresponding control queues 342. The control and status registers 340 and control queues 342 facilitate control and sequencing commands and subcommands associated with data processed in the write and read data pipelines 106, 108. For example, a data segment in the packetizer 302 may have one or more corresponding control commands or instructions in a control queue 342 associated with the ECC generator 304. As the data segment is packetized, some of the instructions or commands may be executed within the packetizer 302. Other commands or instructions may be passed to the next control queue 342 through the control and status registers 340 as the newly formed data packet created from the data segment is passed to the next stage.

[0114] Commands or instructions may be simultaneously loaded into the control queues 342 for a packet being forwarded to the write data pipeline 301 with each pipeline stage pulling the appropriate command or instruction as the respective packet is executed by that stage. Similarly, commands or instructions may be simultaneously loaded into the control queues 342 for a packet being requested from the read data pipeline 303 with each pipeline stage pulling the appropriate command or instruction as the respective packet is executed by that stage. One of skill in the art will recognize other features and functions of control and status registers 340 and control queues 342.

[0115] The storage controller 204 and or non-volatile storage device 122 may also include a bank interleave controller 344, a synchronization buffer 346, a storage bus controller 348, and a multiplexer ("MUX") 350.

[0116] FIG. 4A depicts a schematic diagram of one embodiment of a log structure 400 in a memory storage device 122. While the log structure 400 is described in conjunction with the storage device 122 of FIG. 1, the log structure 400 may be used in conjunction with any type of storage device 122 or memory device. Alternatively, the storage device 122 may be used in conjunction with any storage media in a log-structured manner or that is configured to perform operations according to an object-based storage system.

[0117] For NAND flash memory, a log structure allows program operations to be performed quickly because changes to a set of data in one location of the storage device 122 are written to a different location (that is first erased) on the same storage device 122, rather than the first location. Although not shown herein, the storage device 122 may be used in conjunction with storage media organized in a structure other than a log structure. For example, a non-log-structured storage device may include phase change memory or a write-in-place non-volatile storage media. Write-in-place storage media

may include a storage medium that is able to make changes to data stored at a location on the storage medium and write the modified data back to the same location on the storage medium. A write-out-of-place storage medium is configured to modify data stored on the storage medium and write the modified data to a different location on the storage medium, such as in a log-structured medium.

[0118] In one embodiment, the log structure **400** is configured to store objects received from a client application **110** in the order that the objects are received to be written to the storage device **122**. The size of each object **402** may be different, according to the data associated with a write operation. In one embodiment, because the log structure **400** is configured to store objects received in the order that they are received, each new object may be added to the log structure **400** at the end of the previously stored object. For example, as shown in FIG. **4**A, a first object **402** having a first size is created and stored at the beginning **408** of the log structure **400** on the storage device **122**. When a second object **404** is created, the second object **404** is placed after the first object **402** on the storage device **122**. Additional objects are then added to the log structure **400** in the order received by the storage device **122**. The objects shown in FIG. **4**A may be any type of objects, including live objects, dead objects and/or dirty objects.

[0119] Because the storage device **122** is configured to store data in a log-structured manner, the objects do not need to be stored in blocks, as with a traditional block-based storage device. Additionally, the storage device **122** is capable of retrieving (or otherwise modify) data associated with the stored objects by tracking the object identifier, location, and size of each object in a table **406**, as shown in FIG. **4**B. In one embodiment, the table **406** is stored on the storage device **122**. In another embodiment, the table **406** is stored in memory **108** on the computing device **102**. In another embodiment, the table **406** is stored in another location accessible to the storage device **122** or device driver **120**. In other embodiments, a structure other than a table may be used to store tracking information for each object on the storage device **122**. For example, a tree structure may be used to store the tracking information. Other structures configured to store such tracking information may also be used.

[0120] In one embodiment, when an object is created on the storage device **122**, the storage device **122** determines the starting address where the object will be stored. In a log structure **400**, the starting address may already be known based on where an append point (or location where the previous object ends) is currently pointing. A size of the object is used to determine the ending address for the object using the starting address. The starting address may be stored with the size of the object in the table **406**. In another embodiment, the starting and ending addresses for the object are stored in the table **406**. In one embodiment, when an object is "destroyed" or erased, the storage device **122** finds the object in the table **406** and erases the entry associated with the object from the table **406**. In another embodiment, destroyed (or dead) objects are erased from the storage device **122** during garbage collection, as described in more detail below.

[0121] In one embodiment, when a read operation is received for a specific object, the storage device **122** accesses the table **406** and determines or otherwise identifies the starting address and size associated with the object from the table **406**. The starting address and size are used to determine the ending address, and the storage device **122** reads the data stored at all of the addresses from the starting address to the ending address and returns the data to the computing device **102**. In another embodiment, the starting and ending addresses for the object are stored in the table **406**.

[0122] In one embodiment, when a write operation is received for a specific object stored on the storage device **122**, the storage device **122** accesses the table **406** and determines the starting address and size associated with the object from the table **406**. If the data of the write operation associated with the object has the same size as the object stored on the storage device **122**, the new data may then be written to the storage device **122** in a new location (for example, the append point (**504**, shown in FIG. **5**) of the log structure **400**, which may correspond to the end of the last object stored on the log structure **400**) according to the log structure **400** of the storage device **122**, the location information in the table **406** is updated with the new location of the object, and the previous size information from the table **406** is used. If the new data from the write operation does not have the same size as the object stored on the storage device **122**, the new data is written to the storage device **122** and the location and size information are updated in the table **406**.

[0123] In one example, the operations for creating, destroying, reading, and writing objects are shown below:

[0124] 1. ObjectCreate: Creates an object and returns a 64-bit object identifier (ObjID). The syntax is:

[0125] ObjID ObjectCreate(size_t size)

[0126] 2. ObjectDestroy: Destroys an object from the system. The syntax is:

[0127] void ObjectDestroy(ObjID oid)

[0128] 3. ObjectRead: Reads an object from the flash memory device into the memory of the application. The syntax is:

[0129] int ObjectRead(ObjID oid, char** buffer, size_t* size)

[0130] where ObjID is the 64-bit object identifier and buffer is the location where the object is to be placed once read from the storage device. The argument buffer is a placeholder for the actual pointer to the object in the volatile memory for an application. Such an interface allows a software developer to create objects without keeping track of the size of the objects. The object is read from the storage device, the size of the read object is placed in the size argument, and the pointer to the object is returned in buffer.

[0131] 4. ObjectWrite: Writes an object from the memory of an application to the storage device. The syntax is:

[0132] int ObjectWrite(ObjID oid, char* buffer, size_t size)

[0133] The size argument allows the application to change the size of the object. Otherwise, the size argument is ignored and the previous size of the object is used.

[0134] While the above examples illustrate one embodiment of the operations for creating, destroying, reading, and writing objects on the storage device **122**, other embodiments of the operations may use other syntaxes, arguments or methods for accessing and altering data on the storage device **122**.

[0135] FIG. **5** depicts a schematic diagram **500** of garbage collection in a log structure **400** on a memory device. While the garbage collection is described in conjunction with the log structure **400** shown in FIG. **5**, the garbage collection may be used in conjunction with any storage device **122** operating in

a log-structured manner. Alternatively, the storage device **122** may use any suitable garbage collection methods.

[0136] In one embodiment, when the driver **120** or controller **124** performs garbage collection on the storage device **122**, the garbage collection is performed in a log-structured manner to maximize the lifetime of the memory elements **126** and to minimize fragmentation of data on the storage device **122**. To perform such garbage collection, the storage device **122** is configured to perform operations for objects within the log structure **400** to clean up old objects stored on the storage device **122** and reclaim unused space. These operations allow the objects to be moved within the log structure **400** and the information stored in the table **406** for the corresponding objects may be updated accordingly.

[0137] For example, in garbage collection for a group of addresses on the storage device **122**, a "read" operation retrieves live objects **502** (having valid data that has been committed to the storage device **122** or to a backing store **118** associated with the storage device **122** in a caching system) for the group of addresses from the storage device **122**, a "modify" operation adjusts the size of live objects (for example, by modifying the size or grouping the live objects together), and the "write" operation writes the modified live objects back to the storage device **122** at the current append point **504** in the log structure **400**. As shown in FIG. **5**, the live objects **502** are fragmented in the log structure **400** prior to garbage collection, and are compacted (or placed together) after garbage collection. The "write" operation may also write any dirty objects that may fit in the group of addresses for which garbage collection is being performed. After garbage collection is performed, the corresponding entries in the table **406** are updated with the new information for each object that was written to the group of addresses on the storage device **122**.

[0138] In another embodiment, a remap operation may be used to remap objects to different locations to reduce fragmentation and to manage non-volatile memories such as NAND flash memory. The remap operation may be performed on old, stale, and/or unused objects. The remap operation may be performed on dead objects so that the storage space may be erased and reclaimed. For example, during the remap operation, an object is read from the storage device **122**, the size of the object may be determined in order to set a size attribute in metadata for the object, and the object is written back to the storage device **122**. In one embodiment, because the object size is variable, the size attributes of the object may be changed (larger or smaller) before writing the object back to the storage device **122**. In one embodiment, the object size is not determined and a copy of the object in volatile memory may be used to create a new version of the object that is written to the storage device **122**. In one embodiment, writing the object back to the storage device includes storing the object at the append point **504** of the log structure **400** and invalidating the previous copy of the object. This may be done in NAND flash memory, for example, with a TRIM command that informs the storage device **122** that the previous object is no longer being used and may be erased.

[0139] FIG. **6** depicts a flow chart diagram of one embodiment of a method **600** for object based storage on a storage device **122**. Although the method **600** is shown and described with operations of the computing device **102** of FIG. **1**, other embodiments of the method **600** may be implemented with other computing devices **102**.

[0140] In one embodiment, the method **600** includes receiving **602** a memory access command to a memory storage device **122** from an application. The storage device **122** is configured to provide native support for object storage, as described herein. The application may be the client application **110** described in FIG. **1**. In one embodiment, the operations for the method **600** are performed partially or entirely at a hardware device manager, such as a driver **120** or controller **122**.

[0141] The method **600** also includes identifying **604** a physical address on the storage device **122**. The physical address is mapped directly to an object identifier associated with the memory access command according to an object store interface for the storage device **122**. In one embodiment, the physical address is mapped directly to the object identifier in an absence of a block-based translation layer or other intermediate address translation layer or intermediate mapping layer for the storage device **122**. The object store interface provides native object storage functionality on the storage device **122**. Providing direct mapping between the physical address and the object identifier allows the elimination of additional abstraction between the application and the storage device **122**. In one embodiment, the object identifier includes a virtual memory address.

[0142] The method **600** also includes performing **606** an operation associated with the memory access command for data stored at the physical address. The storage device **122** is configured to process the memory access command at an operation-specific granularity. In one embodiment, processing the memory access command includes performing an operation associated with the memory access command. The physical address has an arbitrary granularity associated with an object corresponding to the object identifier. Each operation may be performed at an object-level granularity according to the specific access command to the storage device **122**, such that access to the storage device **122** is based on the size of the object associated with the operation, rather than a predetermined block size. The storage device **122** may be configured to create objects, destroy objects, read objects, and write objects on the storage device **122**, for example, such that each operation only accesses a number of physical addresses on the storage device **122** according to the size of each object for each operation. A create object command **608** creates an object of a specific size on the storage device **122**. A destroy object command **610** destroys a particular object stored on the storage device **122**, for example, by removing a reference or pointer to the object. A write object command **612** writes data to the storage device **122** for a particular object that has been created on the storage device **122**. Writing to an object that is already written on the storage device **122** may result in the object being written to the current append point **504** of a log structure **400** on the storage device **122**. The previous version of the object may then be invalidated, for example via a TRIM command in NAND flash, and the space taken by the previous version may be reclaimed by the storage device **122**. A read object command **614** reads data corresponding to a particular object from the storage device **122**.

[0143] In one embodiment, the method **600** stores and maintain **616** an index such as a table **406** configured to track a location and a size of object data on the storage device **122**. The object data corresponds to the object identifier. The table **406** may be stored on the storage device **122**, the computing device **102**, or another storage device. The table **406** may include an entry for each of the objects stored on the storage

device **122**. The entries may include information describing the location (such as a first address) of each object and a size of each object. The location and size information may allow the storage device **122** to find and modify stored objects.

[0144] In one embodiment, the method **600** includes performing garbage collection according to a log structure **400** of the storage device **122**. In one embodiment, the garbage collection is performed by reading data from a group of physical addresses on the storage device **122** that correspond to live objects. The data for the live objects is grouped together, and written back to the group of physical addresses. In another embodiment, the garbage collection is performed by remapping data stored at a first physical address on the storage device **122** to a second physical address on the storage device **122**.

[0145] While many embodiments are described herein, some embodiments relate to a method. The method includes receiving an object operation from an application at a hardware device manager. The object operation includes an object identifier. The method includes performing the object operation directly on a storage device. A physical address for the object corresponding to the object identifier is mapped directly to the object identifier in an index managed by the hardware device manager. Other embodiments of the method are described herein. Embodiments of a device and a system are also described herein.

[0146] In one embodiment of the method, the memory access command is an object operation that is performed directly on the storage device. The object operation may be a write operation that includes writing object data to the storage device at a new physical address and mapping a physical address associated with the object identifier in the index to the new physical address. The object operation may be a read operation that includes identifying, via an index (such as a table), the physical address corresponding to the object stored on the non-volatile storage device. In one embodiment, the object operation is a delete operation that includes invalidating data corresponding to a deleted object on the storage device and removing, from the index, the object identifier corresponding to the deleted object.

[0147] An embodiment of the network system includes at least one processor coupled directly or indirectly to memory elements through a system bus such as a data, address, and/or control bus. The memory elements can include local memory employed during actual execution of the program code, bulk storage, and cache memories which provide temporary storage of at least some program code in order to reduce the number of times code must be retrieved from bulk storage during execution.

[0148] It should also be noted that at least some of the operations for the methods may be implemented using software instructions stored on a computer useable storage medium for execution by a computer. As an example, an embodiment of a computer program product includes a computer useable storage medium to store a computer readable program that, when executed on a computer, causes the computer to perform operations, as described herein.

[0149] Embodiments of the invention can take the form of an entirely hardware embodiment, an entirely software embodiment, or an embodiment containing both hardware and software elements. In one embodiment, the invention is implemented in software, which includes but is not limited to firmware, resident software, microcode, etc.

[0150] Furthermore, embodiments of the invention can take the form of a computer program product accessible from a computer-usable or computer-readable medium providing program code for use by or in connection with a computer or any instruction execution system. For the purposes of this description, a computer-usable or computer readable medium can be any apparatus that can contain, store, communicate, propagate, or transport the program for use by or in connection with the instruction execution system, apparatus, or device.

[0151] The computer-useable or computer-readable medium can be an electronic, magnetic, optical, electromagnetic, infrared, or semiconductor system (or apparatus or device), or a propagation medium. Examples of a computer-readable medium include a semiconductor or solid state memory, magnetic tape, a removable computer diskette, a random access memory (RAM), a read-only memory (ROM), a rigid magnetic disk, and an optical disk. Current examples of optical disks include a compact disk with read only memory (CD-ROM), a compact disk with read/write (CD-R/W), and a digital video disk (DVD).

[0152] Input/output or I/O devices (including but not limited to keyboards, displays, pointing devices, etc.) can be coupled to the system either directly or through intervening I/O controllers. Additionally, network adapters also may be coupled to the system to enable the data processing system to become coupled to other data processing systems or remote printers or memory devices through intervening private or public networks. Modems, cable modems, and Ethernet cards are just a few of the currently available types of network adapters.

[0153] Although the operations of the method(s) herein are shown and described in a particular order, the order of the operations of each method may be altered so that certain operations may be performed in an inverse order or so that certain operations may be performed, at least in part, concurrently with other operations. In another embodiment, instructions or sub-operations of distinct operations may be implemented in an intermittent and/or alternating manner.

[0154] In the above description, specific details of various embodiments are provided. However, some embodiments may be practiced with less than all of these specific details. In other instances, certain methods, procedures, components, structures, and/or functions are described in no more detail than to enable the various embodiments of the invention, for the sake of brevity and clarity.

[0155] Although specific embodiments of the invention have been described and illustrated, the invention is not to be limited to the specific forms or arrangements of parts so described and illustrated. The scope of the invention is to be defined by the claims appended hereto and their equivalents.

What is claimed is:

1. A method, comprising:

receiving an object operation from an application at a hardware storage device manager, wherein the object operation comprises an object identifier; and

performing the object operation directly on a storage device, wherein a physical address for the object corresponding to the object identifier is mapped directly to the object identifier in an index managed by the hardware storage device manager.

2. The method of claim **1**, wherein the physical address is mapped directly to the object identifier in an absence of a block-based translation layer for the storage device.

3. The method of claim 1, wherein the object operation is a write operation comprising:
writing object data to the storage device at a new physical address; and
mapping an address for the object identifier in the index to the new physical address.

4. The method of claim 1, wherein the object operation is a read operation comprising identifying, via the index, the physical address corresponding to the object stored on the storage device.

5. The method of claim 1, wherein the object operation is a delete operation comprising:
invalidating data corresponding to a deleted object on the storage device; and
removing, from the index, the object identifier corresponding to the deleted object.

6. The method of claim 1, wherein the storage device stores data in a log structure, wherein the physical address corresponds to an append point of the log structure.

7. The method of claim 1, wherein the index comprises a table configured to track a location and a size of object data on the storage device, wherein the object data corresponds to the object identifier.

8. The method of claim 1, wherein the physical address comprises an arbitrary granularity associated with the object corresponding to the object identifier.

9. The method of claim 1, further comprising performing garbage collection according to a log structure of the storage device by:
reading data from a group of physical addresses on the storage device, wherein the data corresponds to live objects comprising valid data stored on the storage device;
grouping the data for the live objects together; and
writing the grouped data back to the storage device.

10. The method of claim 1, further comprising performing garbage collection by remapping data stored at a first physical address on the storage device to a second physical address on the storage device.

11. The method of claim 10, wherein the storage device is configured to create objects, destroy objects, read objects, and write objects on the storage device.

12. The method of claim 1, wherein the object identifier comprises a virtual storage address.

13. A non-volatile memory device, comprising:
a plurality of memory elements configured to store data, wherein the memory elements comprise corresponding physical addresses;
a hardware device manager configured to:
receive an object operation comprising an identifier corresponding to an object stored on the non-volatile memory device; and
performing the object operation directly on the non-volatile memory device by:
writing object data to one or more physical addresses on the non-volatile memory device; and
mapping, in an index, an address for the identifier corresponding to the object directly to the one or more physical addresses on the non-volatile memory device.

14. The device of claim 13, wherein the one or more physical addresses correlate directly to the identifier in an absence of a block-based translation layer for the non-volatile memory device.

15. The device of claim 13, wherein the index comprises a table configured to track a location and a size of object data on the non-volatile memory device, wherein the object data corresponds to the identifier.

16. The device of claim 13, wherein the one or more physical addresses comprise an arbitrary granularity associated with the object.

17. The device of claim 13, wherein the hardware device manager is further configured to perform garbage collection according to a log structure of the non-volatile memory device by:
reading data from a group of physical addresses on the non-volatile memory device, wherein the data corresponds to live objects comprising valid data stored on the non-volatile memory device;
compacting the data for the live objects together; and
writing the compacted data back to the non-volatile memory device.

18. The device of claim 13, wherein the hardware device manager is further configured to perform garbage collection by remapping data stored at a first physical address on the non-volatile memory device to a second physical address on the non-volatile memory device.

19. The device of claim 13, wherein the hardware device manager is further configured to provide native support for object storage, wherein the hardware device manager is configured to create objects, destroy objects, read objects, and write objects stored on the non-volatile memory device.

20. The device of claim 19, wherein the object operation for object data stored at a location on the non-volatile memory device comprises a place holder argument for a pointer to at least one physical address of the location, wherein the pointer is output based on a size of the object data.

21. A system, comprising:
a driver interface configured to communicate with a recording device interface of a non-volatile recording device, wherein the driver interface is configured to:
submit an object operation associated with an application, wherein the object operation is associated with an object identifier;
performing a read operation directly on the non-volatile recording device by identifying an address for the object identifier from an index, wherein the address is mapped directly to one or more physical addresses on the non-volatile recording device, wherein the one or more physical addresses correspond to object data for the object identifier.

22. The system of claim 21, wherein the one or more physical addresses are mapped directly to the object identifier in an absence of a block-based translation layer for the storage device.

23. The system of claim 21, wherein the index comprises a table configured to store a size and the one or more physical addresses of the object data on the storage device.

* * * * *