CIS^{TM}: Content Immutable Storage for Trustworthy Electronic Record Keeping

Lan Huang, Windsor W. Hsu
IBM Almaden Research Center
650 Harry Rd.
San Jose, CA 95120, USA
{*lanhuang,windsor*}*@us.ibm.com*

Fengzhou Zheng *
Department of Computer Science
Princeton University
zheng@cs.princeton.edu

Abstract

As records are increasingly generated in large volumes and in electronic form, which allows easy destruction and clandestine modification, it is imperative that they be properly managed to preserve their trustworthiness, i.e., *their ability to provide irrefutable proof and accurate details of events that have occurred. The need for proper record keeping is further underscored by the recent corporate misconduct and ensuing attempts to destroy incriminating records. We critically examine the purpose and process of record keeping to establish the storage system requirements for ensuring that records are trustworthy. We refer to a storage system that meets these requirements as* content immutable storage *(CIS). CIS offers overwrite protection that is secure even against inside attacks, supports index mechanisms efficiently, allows records to be properly disposed of after they have expired, and is low in cost yet reliable. We also discuss design issues in satisfying the requirements and describe a prototype implementation of CIS. Performance results obtained on the prototype suggest that CIS performs similarly to regular storage for the record keeping workload.*

1. Introduction

Records such as electronic mail, financial statements, medical images, drug development logs, quality assurance documents and purchase orders, are valuable assets, representing much of the data on which key decisions in business operations and other critical activities are based. Records also serve as evidence of activity. To be useful, however, the records must be trustworthy, *i.e.,* accurate, credible and readily accessible. Having trustworthy records is particularly imperative in the litigious US. On average, a Fortune 500 company is the target of 125 non-frivolous lawsuits at any given time, and the damages awarded are increasing rapidly, as is the cost of electronic data discovery, which is projected to rise at a rate of 65% per year to reach $2 billion in 2006 [21]. Records are increasingly stored in electronic form, which makes them relatively easy to delete and modify without leaving much of a trace. Therefore, ensuring their accuracy and credibility is especially critical.

Furthermore, a growing proportion of the records are subject to regulations that specify how they should be managed. In the US alone, there are currently more than 10,000 such regulations [24]. The key focus of many of these regulations (*e.g.,* Sarbanes-Oxley Act [7], SEC Rule 17a-4 [20]) is to ensure that records are trustworthy. Non-compliance with these regulations could result in stiff penalties. The bad publicity of non-compliance and the ensuing investor impacts could cost an organization dearly. As information becomes more valuable to organizations coupled with headlines of corporate misdeeds, accounting scandals and securities fraud, the number and scope of such regulations are likely to grow. Worldwide, the volume of regulated records is projected to increase by 64% per year to almost 2 PB in 2006 [24].

While immutability of records is often specified as a requirement for proper record keeping, what is actually required in practice is that the records be "term-immutable", *i.e.,* immutable for a specified retention period. For example, SEC Rule 17a-4 [20] specifies a retention period of three years for email, attachments, memos, instant messaging, *etc.* It is important for an organization to properly dispose of records that are no longer useful to the organization and have passed any mandated retention periods. This is not only for storage space efficiency purpose but also for the reduction of associated costs for record keeping [21]. Proper disposition of records includes deleting the records and in some cases, "shredding" the records so

*The author performed this work when he was a summer intern at IBM Almaden Research Center, San Jose, CA in the summer of 2004.

that they cannot be recovered or discovered even with the use of data forensics.

To ensure their trustworthiness, electronic records have traditionally been stored by making irreversible changes to an optical storage medium, such as a Write-Once-Read-Many (WORM) optical disc [18], CD-R and DVD+-R. Such storage, however, does not allow expired records to be selectively disposed of. Instead, an entire disc of records is disposed of at a time as the medium is physically destroyed. Such storage also tends to lack the ability to commit small amounts of data to random locations. They rely on a driver software that sends write streams to write a large amount of data. This small write capability is crucial for maintaining trustworthy index mechanisms that can quickly locate relevant records during an inquiry, especially in view of the huge volumes of records that organizations have today. Moreover, for various reasons, improvement in optical technology has not kept pace with improvement in alternative technologies such as magnetic recording, especially, when it comes to storage density and performance.

Thus, many new storage systems (*e.g.*, [8, 11, 16, 22]) have recently been introduced to facilitate electronic record keeping. These new systems address some, but not all, of the limitations associated with traditional WORM storage. They also introduce new issues such as the adequacy of the overwrite protection. Object-based Storage (OSD) [4, 9] introduces a completely new storage interface to achieve storage security, performance scalability and autonomous management. It can be configured to meet part of the requirements for record keeping. But we aim to minimize the changes to existing block level interface to achieve maximum application portability and cost efficiency. The supporting infrastructure for OSD, including OSD controller and devices, is not openly available as of today. We would like to reuse the existing software/hardware stack for SCSI based storage system as much as possible. We critically examine the purpose and process of record keeping to establish the storage system requirements for ensuring that records are trustworthy. The key requirements are that the storage system must 1) offer overwrite protection that is secure even against inside attacks; 2) efficiently support index mechanisms; 3) allow records to be properly disposed of after they have expired; and 4) be low cost yet reliable. We refer to a block level storage system that satisfies these requirements as *content immutable storage*. In the paper, we discuss design issues in CIS and describe a prototype implementation of CIS. We also present performance results obtained on the prototype. The results indicate that for the target workload, which is record keeping, the performance of CIS is comparable to that of regular storage systems.

The rest of the paper is organized as follows. Section 2 discusses related work. In Section 3, we establish the key storage requirements for trustworthy record keeping. In Section 4, we discuss design issues in satisfying these requirements. In Section 5, we present performance optimizations. Section 6 describes our prototype and presents performance results obtained on the prototype. Section 7 provides our conclusions.

2. Related Work

To ensure their trustworthiness, electronic records have traditionally been stored by making irreversible changes to optical storage media [1], such as CD-ROM/DVD-ROM and MO (magneto-optical disc). However, optical storage has seen a declining market trend, due to its performance and storage density disadvantages as compared to magnetic disks. Optical disc's read/write speed has not kept up with the magnetic hard disk's performance improvements. To manage an optical library is orders of magnitude more difficult than managing a hard drive based archival store of the same capacity. In addition, functionality-wise, existing systems based on optical storage do not allow data to be selectively shredded and most do not support data writes at random locations.

Thus, many new WORM storage systems (*e.g.*, [19, 8, 11, 16, 22]) have recently been introduced to facilitate electronic record keeping. These systems addressed some of the limitations associated with traditional WORM storage. Critical functionalities, such as the efficient support of index mechanisms, are, however, still lacking. Moreover, these systems introduce new issues such as the adequacy of the overwrite protection.

Venti [19] presents an archiving storage system that uses a fingerprint of a data block's contents to address the block. The fingerprint is computed from the contents of the block using a secure one-way hash function (SHA1 [15]) and thereafter returned as the pointer for future reference to the block. Finding two distinct inputs that hash to the same value under SHA1 has not been considered to be computationally feasible [14]. Some recent cryptography research has found that it is possible to hack SHA0, MD5 and even SHA1 [5, 25, 12]. That means an intruder can replace the original content block with a junk content block, which happens to have the same SHA1 hash value. However, it is still effectively secure to use longer length one-way hash function such as SHA2. Since different contents will automatically lead to different addresses, the system effectively avoids overwriting or data modifications, if the block handles are safely stored for subsequent retrieval of the data, and if all the accesses go through the Venti system's address mapping layer. It is not difficult to imagine that an intruder can selectively damage the content of the blocks on the storage system through direct access.

Centera [8] describes a fingerprint addressed storage system conceptually similar to Venti [19], but at the object level. Similar to Venti [19], the system effectively behaves like a write-once device, provided that the fingerprints are safely kept for subsequent retrieval of the objects, and if all the accesses go through the system. Such systems also require a new interface, and the porting of applications to this new interface. SnapLock [16] provides overwrite protection as a feature of general file system. It protects the immutability of a file by marking the non-overwriting flag for each file. However, all these special marks reside on rewritable media and can be easily destroyed if SnapLock file system is bypassed. CIS aims to achieve secure term-based block immutability on the media, such that no bypassing is ever possible.

Object-based Storage (OSD) [4, 9] introduces an object level storage interface. It contains a new set of storage access and management commands with completely different formats from existing SCSI or ATA standards. Object-related meta data are maintained on the storage for each object. The new expanded interface helps to achieve improved performance, scalability, security and storage management. It is technically possible to maintain one field of the object meta data for immutability check, enforced by OSD, to achieve object immutability. This extra check currently does not exist. OSD provides *append* command to append data to an object. But the granule is in sectors. So it is not ready to support efficient indexing for record keeping purpose. More importantly, we aim to minimize the changes to existing applications and standard block level storage interfaces, *i.e.,* SCSI and ATA standards. The resulting benefit is reduced development cost and ease of customer acceptance.

Self-Securing Storage (S4) [23] addresses the issue of securing data on storage from a quite different angle from CIS. S4 efficiently maintains several versions of each block on the storage during a history window. The older versions will be recovered if a recovery after an intrusion is necessary. S4 takes a logging and recovery approach to recover damage from intrusion, while CIS completely prevents any modification to written blocks. In the context of S4, data can be overwritten. Data cannot be recovered beyond the protected history window. For trustworthy record keeping purpose, no loss of truthful data is ever allowed on CIS.

3. Storage Requirements for Trustworthy Record Keeping

The fundamental purpose of record keeping is to establish solid proof and accurate details of events that have occurred. Trustworthy records are, therefore, those that can be relied upon to achieve this purpose. The process of creating accurate records for all of the relevant events as they occur is generally trusted. This is the case especially if the records are used in the normal course of business, and are hence required for the proper functioning of the organization. Furthermore, record creation is an ongoing process for which periodic audits are effective in ensuring proper execution. An intruder to an application can create flaw data and write them to the storage successfully. This is beyond the control of a record keeping system. However, once the flawed data are written, they will be kept and used as traces together with all the other data of the same context to find out the truth. The worst thing that could happen is for a crime to be committed without any traces left behind. The basic objective of record keeping is not to prevent the writing of history, but to prevent the changing of history; in other words, changing the records after the fact. The key requirement for trustworthy record keeping is, therefore, to ensure that in an enquiry, all of the relevant records can be quickly located and retrieved in an un-tampered form. In this section, we examine what the requirements for storage systems are for trustworthy electronic records keeping and how they lead to our design choices for CIS in the next section.

3.1. Secure Immutability

The pressing need is to protect against clandestine modification, including destruction, of selected records during their storage. Typically, this means that records must be stored in some form of WORM storage. Modification of the records could result from software bugs and from user errors, such as issuing the wrong commands and replacing the wrong disks during service actions. Given our increasing reliance on electronic records, the potential gain from intentionally manipulating and altering the records is huge. Thus, more importantly, the WORM storage must be secure against intentional attacks, even inside attacks launched by disgruntled employees, company insiders, or conspiring technology experts. An adversary is likely to have the highest (*e.g.,* executive) level of support and insider access, privilege and knowledge. He can be thought of as the super system administrator. Although the adversary has physical access to the records, he cannot destroy them in a blatant fashion because it would result in severe penalties and even the presumption of guilt. His mission is to clandestinely hide or modify specific records.

3.2. Efficient Index Support

Furthermore, with the growing volume of records and the ever more stringent response time to enquiries, direct access mechanisms such as indexes increasingly must be maintained to ensure that all of the records relevant to an enquiry can be discovered and retrieved in a timely fashion,

typically within days and sometimes even within hours [6]. However, if records are accessed through an index, even if they are stored in WORM storage, they will still be vulnerable to logical modification if the index can be suitably manipulated [26]. In particular, an adversarial system administrator could update the index to logically hide or modify selected records unless the index is also properly committed to WORM storage. Note that creating an index at enquiry time is not an option because of the time required to build the index and the fact that the index could be created in such a way as to hide or modify records that are relevant to the enquiry. The WORM storage must, therefore, efficiently support index structures, which typically requires many small updates. Moreover, there is often a need to log small amounts of data on WORM storage, for instance, to maintain a non-alterable audit trail of the activity in the system. We mainly focus on how to support small updates in block level storage instead of how to build an index in this paper.

Universal Disk Format (UDF) [17, 2] is a widely used format for optical disks. It can be viewed as a form of index for files on optical storage. Writer software for optical media follows the UDF format to write meta data and file data to optical media. Meta data are tables of pointers where one looks up the location of the files. Meta data are not committed upon the writing of file data. Instead, they are written after a session is closed. This is to minimize the wastes of optical disk space for meta data write. UDF can also be readily used on CIS storage. With efficient small write support from CIS, we can effectively close the time gap between meta data commit and file data write and achieve better file integrity, without losing space efficiency.

3.3. Term-Retention and Disposition

Note that while immutability is often specified as a requirement for records, what is required in practice is that the records be "term-immutable", *i.e.,* immutable for a specified retention period. For example, SEC Rule 17a-4 [20] specifies a retention period of three years for email, attachments, memos, instant messaging, *etc.* Some regulations further require the capability to hold and preserve a record indefinitely or for some specified duration after a triggering event. Even after records have passed any mandated retention periods, if they are available, they are subject to discovery, and typically at great expense to the owning organization [21]. Thus, it is important for an organization to properly dispose of records that are no longer useful to the organization and have passed any mandated retention periods. Proper disposition of records include deleting the records. In some cases, the records have to be shredded such that they cannot be recovered or discovered even with the use of data forensics. In other words, the storage should support "term-WORM" and the ability to shred records that have expired, for example, by overwriting magnetically recorded data multiple times with specific patterns in order to completely erase remnant magnetic effects which could otherwise enable the data to be recovered [10].

3.4. Low Cost and Reliable

At a minimum, this requirement means that the records have to be reliably stored and protected from loss due to disasters, system failures, equipment obsolescence, *etc.* This is no different from what is expected of current storage systems except that the records have to be reliably stored over an extended period of time. The extended retention period, together with the large volume of records that is typical today, requires that the storage be very low cost, especially because there is a tendency in the short term to view the records largely as an overhead needed just for satisfying the current intense regulatory scrutiny.

4. Content Immutable Storage

To achieve secure immutability, CIS enforces immutability beneath the block level interface and securely authenticates that each command is legitimate. Thus, no one could possibly bypass the immutability firewall to tamper the written data. CIS provides variable length append command to support efficient small write for indexing. CIS's intelligence is embedded in a thin virtualization layer and a standard block level interface is exposed. Thus application porting effort is minimized. We use standard rewritable hard drives as the underlying storage media for CIS. Magnetic hard drive has better performance specifications and capacity density than optical disks. It enables us to achieve a low total cost of ownership, which is a basic requirement for record keeping. The future technology trend for hard drive, tape and optical disk also indicates that magnetic hard drive is more suitable for those record keeping tasks which require fast access and cost efficiency.

4.1. Overwrite Protection

To enforce immutability, CIS maintains state information for each block in the system. The state indicates whether a block is writable. Upon initialization, the state for each block is set to be writable. After receiving a block of data to be stored, the system checks the state of the target block to determine if that block is writable. If yes, the system writes the received data into the target block. It then updates the state of the target block to non-writable before acknowledging that the write has been successfully executed. If the state of the target block indicates that the block

is not writable, the system returns a failure to the write request. Since the immutability of the system depends on the integrity of the state information, direct access to it is restricted by storing the state information at locations on the disks which are not addressable to users. Thus, once a block has been written, it is protected from further writes.

The operations described above sound straightforward and it can be technically feasible to be realized in any level in the system stack: application software, network router, virtualization software, storage controller, or hard disk. However, implementing CIS at any level gives tradeoffs on security and cost efficiency. For security purposes, we choose to implement CIS logic in hardware or low-level firmware, both of which are deemed to be much harder to tamper with than software. For example, software can be relatively easier to replace than hardware microcode. One option is to modify commodity hard drive internals to incorporate the CIS logic. Enforcing WORM property at the disk drive level makes it more difficult to provide data protection since RAID schemes such as RAID level 5 requires rewriting the parity blocks as data is stored in the system. New storage interface OSD makes it possible to provide data protection through RAID technique beneath the OSD interface. However, as of today, OSD devices and OSD controllers are not available on the market. It is more acceptable for CIS to be built on existing SCSI interface, and take advantage of data protection feature from existing RAID controller.

A more favorable approach is to implement the CIS logic on top of a storage controller. Today's storage controllers are typically equipped with sufficient resources, providing a platform on which we could build additional functionality. It also relieves CIS of storage virtualization and RAID implementations. In addition, implementing CIS logic on controllers separates the manufacturing of low cost commodity disks from high-performance and high-functionality controllers, which are relatively price-insensitive. CIS is realized only through another thin layer of virtualization.

In general, a large and complicated system usually leads to more software bugs and opens the door to security compromises. Following the security principle of minimizing the trusted computing base, CIS was designed to be as simple as possible and to provide just the necessary functions. Therefore, CIS only aims to provide standard block level functionality, rather than file or object level services. This approach has the additional advantage that it allows more flexibility in the software components which are upstream in the system stack. For example, we can allow users to upgrade the file system without having to worry about the integrity of data stored in CIS. Having the CIS logic close to the media also gives it a better position to have complete mediation of all requests.

4.1.1. Threat Model Since CIS uses rewritable disks and the media itself does not protect its data from being overwritten, the system could be compromised if an adversary manages to access the rewritable disks directly, bypassing the CIS logic. We describe several threat cases that we cover with our binding solutions. Suppose we have CIS controller *C*, CIS disk *D* and an intruder *E* (Eve). *E* can overwrite the blocks on *D* and effectively defeat the overwrite protection of CIS controller through the following approaches.

In the first intrusion, *E* fakes itself to be the trusted CIS controller *C* and sends write commands to written blocks on *D*. *E* could achieve this by launching an interceptor between *C* and *D*. In the second case, *E* fakes itself to be a CIS disk *D*. *E* intercepts results returned from *D* and modifies the content. For example, *C* queries on the available unwritten bytes of some blocks. *E* changes the returned states on the blocks. *C* issues commands based on the wrong information and effectively overwrites data on *D*. The above two threat cases suggest that traffic both ways need to be authenticated and verified. In the third case, *E* cannot fake the identities of *C* or *D*, but it replays previous requests to overwrite existing blocks. This suggests that the timing of the operation is also an important piece of information to be protected. In the forth case, *E* replays a request to one CIS disk to another CIS disk. This way *E* can effectively overwrite the blocks on the other CIS disk. This means that the write target information needs to be encapsulated in the operations and protected.

Below we describe two solutions to cover the threat cases described above. The challenge is to achieve a secure binding of the CIS logic and the storage media, with minimal storage management overhead.

4.1.2. Physical Binding One solution is to physically bind the controller and its disks so that there is no way to directly access the disks except through the controller. In other words, the CIS logic and the disks are encapsulated in a secure physical enclosure. To cope with disk failures, the enclosure allows disks to be taken out and replaced, but only in a controlled fashion so that one cannot alter the data stored in the system by replacing disks. For example, each disk is guarded by an electronically controlled physical lock and can only be removed when the lock is released. The position of the lock is maintained through system shutdowns and power-failures so that it is not possible to remove a disk, modify the contents of the disk, and then insert the disk back into the system without the system detecting the action and rebuilding the contents of the disk. In addition, a lock is released only when the removal of the corresponding disk would not compromise the data stored in the system. For example, in a RAID level 5 array, the system could permit up to one disk to be removed from

the system at any one time. Before the lock is released, the system could attempt to remove the contents of the disk to reduce the chances for sensitive data to be leaked out of the system during routine maintenance actions.

4.1.3. Virtual Binding One limitation of physical binding is that it requires the controller and the disks to be physically bound together with a secure communication channel between the two. Virtual binding relaxes this restriction by virtually binding the two entities (WORM logic and disks), which could be physically apart, together through cryptographic means. It enables storage media mobility and flexible system capacity scaling.

The idea behind virtual binding is provided as follows. Using cryptographic methods, the disks authenticate that data requests are from a legitimate WORM controller and have not been tampered. Data access to the media from a controller is blocked until the authentication process is successful. Similarly, the controller verifies the integrity of data read from the disks.

Virtual binding logic is implemented on both the controller side and the disk side. On the disk side, it is possible to implement the virtual binding logic in individual disks or in an enclosure drawer that encapsulates the disks. We prefer to implement the virtual binding logic in disk drawers, rather than the disks themselves to avoid hardware modifications to existing hard drives. Note that when the virtual binding logic is implemented in disk drawers, the mobile granularity is a disk drawer. A disk drawer can either be loaded with disks and permanently sealed before it is shipped (*i.e.*, fail in place), or the physical binding mechanism introduced above can be used to allow dynamic disk addition and replacement.

During initialization and before a WORM controller is shipped from a trusted manufacturer, a pair of private and public keys is generated for the virtual binding logic in the controller. To prove the trustworthiness of the controller and to prevent someone from building a "fake" WORM controller, its public key is also stored in a certificate signed by the manufacturer using the manufacturer's private key. The manufacturer's public key is well-known. Alternatively, a trusted third party, such as Verisign instead of the manufacturer, can act as the certificate generator. The virtual binding module consists of a small amount of non-volatile internal memory to store its public/private keys, the certificate, and the public key of the manufacturer. Similarly, a pair of private and public keys is generated for each disk drawer. In addition, the virtual binding logic in a disk drawer maintains a user table, which contains a list of public keys of the controllers that have data access to this drawer. The virtual binding logic and its authentication information are replicated in the disk drawer to avoid having any single point of failure.

Since the certificate for the controller's public key cannot be altered without knowing the manufacturer's secret key, a disk drawer can check that a controller is a benevolent one from the trusted manufacturer by verifying the certificate using the public key of the manufacturer. Only public keys from legitimate controllers will be admitted and added to the disk drawer's user table. As an option, we can allow controllers to be admitted to the disk drawer's user table only during an initial registration phase.

With virtual binding, it is no longer required that the communication channel between a WORM controller and a disk drawer is secure. To prevent data traffic from being tampered, we use an encrypted content signature to certify the validity of write requests received by the disk drawer. After every fixed amount of traffic (or a fixed amount of time), the controller generates a content hash for all the bytes it sends to the disk drawer during the interval. The content is buffered and not written to the disk until it is verified through the content hash. The content hash is a secure one-way hash of the bytes it verifies and is encrypted using the private key of the controller. By decrypting the content hash and comparing it with one calculated from the bytes received, the disk drawer is able to verify whether the data requests are valid. Similarly, content hashes are generated by a disk drawer for data sent to controllers upon read requests. This is necessary because a WORM controller still relies on the state information to decide whether a block is writable, and the state information is also stored on the disk drawer. To prevent replay attacks, the encrypted content hash also includes a time-stamp (or a sequence number). To prevent another type of replay attack where an adversary records data requests from a controller to one disk drawer and then applies them to another disk drawer (assuming the controller is admitted by both disk drawers), we further include the public key of the target disk drawer in the encrypted content hash.

Our design decouples the relatively complicated authentication logic from the individual commodity hard drives. The authentication intelligence is embedded in storage controller and the disk drawer enclosure (not the disks). The disks are permanently sealed in the secure drawer. The benefit of this design is that CIS can use off-the-shelf hard drives and standard interfaces. The downside is that in case of disk failure, disk repairing becomes more complicated. In case of disk failure, RAID recovery procedure is invoked. We allow spare disks on new drawers to join an existing RAID array, whose drawer runs out of spare disks. This means a RAID array can cross disk drawers. It requires advanced disk management functions in the storage controller. Empirically, magnetic hard drive is more likely to experience media failure than optical disk. A rarely used magnetic hard drive's life is speculated to be longer than that of a drive used for online applications, although this

conclusion has not been systematically verified. For those records whose retention period is longer than the average life span of the storage media, CIS relies on a trusted migration tool to migrate the records onto newer media before the old media fails. The migration process is performed infrequently and can be carried out during system idle time. The system idle time is relatively easy to identify in an archiving system.

4.2. Block Append Capability

The ability to write small amounts of data to WORM storage is key to efficiently supporting index mechanisms and audit trails. Traditional WORM storage has a minimum write unit called sector that is typically 512 Bytes. Updating any part of an index or adding to an audit trail on such storage would involve creating a new copy of any sector that is updated, wasting both time and storage space. Furthermore, many traditional WORM storage devices lack the ability to write an arbitrary sector on the media. Instead, sectors have to be written in sequential order or a large collection of sequential sectors have to be written all at once. In such cases, the indexing has to be performed at one time on a large collection of data and once the indexing is done, new data cannot be added to the index. This means that the index is not available until after the entire collection of data is stored. As data is added over a period of time, the system would create many indices, which may need to be searched to find a particular piece of data.

By using rewritable media as the underlying storage, CIS can read a block that has already been written, add data to it, and then write it again. This means that the system can effectively append data to a block. Instead of using a bit to indicate whether a block has already been written, the system maintains state information for each block to indicate the amount of data already stored in that block. We refer to this value as the length field of the block. The length field indicates the offset into the block at which new data can be written. For a 512 Bytes block, the length field takes up a mere 10 bits.

During the initialization, the length field for all the blocks are set to zero, indicating that the corresponding blocks are empty. On a write, the system first reads the length field corresponding to the target block to check if there is enough space left in the block for the new data. If there is insufficient space, the system returns a write failure to the application. Since the failure is due to an illegitimate write from the application, the application should handle this error accordingly. Otherwise, the system reads the current contents of the target block, inserts the new data at an offset equal to the value of the length field, writes the resulting contents back to the target block, and increases the length field by the amount of new data inserted. If the length field is zero, the system skips the read of the current contents. Note that the system has to ensure that the block writes are effectively atomic. On a read of the block, the system gives an indication of the amount of data stored in the block.

As an optimization, the system permits data to be rewritten with the same data. This is useful in situations where an application keeps the last block of an object (*e.g.*, a log) in memory. Whenever the application appends data to the object, it can simply issue a write of the last block with a new byte count. For example, suppose that the system receives the following write command, write(target block *A*, data *D*, byte count *b*). The system first reads the length field, *l*, corresponding to the target block *A* to check if there is sufficient space left in *A* to store *b* bytes of additional data. If not, it returns a write failure. Otherwise, it reads the current contents of block *A*. If the first *l* bytes of the current contents are different from the first *l* bytes of *D*, the system returns a write failure. Otherwise, it writes *D* into the contents at an offset of *l* and writes the resulting contents back to block *A*. It also updates the length associated with block *A* to increase it by *b*. On receiving a read command such as read(target block *A*), the system first reads, *l*, the length field associated with the target block *A*. Next it reads the target block *A* and returns both *l* and the first *l* bytes of the contents.

4.3. Retention and Disposition Functions

As discussed earlier, it is imperative for organizations to properly dispose of records that are no longer of value and have passed any mandated retention periods. In other words, CIS should be "term-WORM" rather than WORM, *i.e.,* CIS enforces WORM during the specified retention term. To support this, CIS maintains a secure clock and an expiration time in the state information for each block which indicates when the retention period of a block will expire. CIS may optionally maintain a time stamp for each block, which would help identify spurious data and also counterfeit copies of the data. The retention period for a block can only be extended into the future but not shortened.

After a data block has passed its expiration time, CIS allows the data block to be disposed of. Following the common practice in the industry, CIS disposes of a block by overwriting the block multiple times with specific patterns so as to completely erase remnant magnetic effects which could otherwise enable the data to be recovered [10]. The overwrites can be driven by external software through the write interface, or it can be handled by CIS on receiving a *shred* command. The latter approach is preferred as it significantly reduces the IO overhead on host systems and the amount of data traffic over the IO interconnect.

More importantly, CIS can ensure that each of the multiple writes are recorded to the storage media and are not buffered somewhere in the system. Selectively shredding blocks on a disk is time-consuming, which involves rounds of disk I/O. If the disk space will never be reused and shredding efficiency is of higher priority, a de-gauss equipment can be used to securely wipe out the data on a disk. However, the disk is not usable anymore after being de-gaussed.

In addition to the *shred command*, CIS provides a *set_retention* command to set the expiration time for a block to a specified value. Depending on the requirements, CIS may also provide advanced retention functions such as allowing records to be held and preserved indefinitely or for some specified duration after a triggering event. For example, when a court case is filed, the disposition of relevant records need to be put on hold until the case is over. Some of the commands to support such functions include the *event_trigger* command, which sets an event on the specified blocks to enable even-driven retention management, and the *deletion_hold* command, which sets a deletion hold on the specified blocks. The *query* command returns all state information kept by CIS for the specified blocks.

4.4. Interface

CIS provides a standard block-level SCSI interface. It supports all the functions that a standard rewritable storage provides. Attempts to overwrite existing data using *write* commands will be failed by CIS. A vendor specific error message will be returned to the caller.

CIS can be accessed through existing standard network storage protocols, including iSCSI and Fiber Channel protocols. Meanwhile, the block-append capability and retention support require additional parameters such as the retention period and bit length information to be passed to CIS devices along with block-level data requests. These new commands can be implemented using extended Command Description Block (CDB) in standard SCSI interface [3]. High level applications can invoke these new commands through ioctl calls. For example, a UDF file system can send through ioctl an *append* command to a CIS device, to commit the latest update of the index onto the media.

The use of a standard block interface, albeit extended for increased functionality, facilitates the porting of existing applications to CIS. Examples of such applications include WORM file systems (UDF), WORM object systems, backup software, or simply a data pool of blobs (binary large objects). Keeping existing standards intact, also opens the opportunity to reuse the existing software/hardware for data protection and device management, including virtualization software and RAID storage controller.

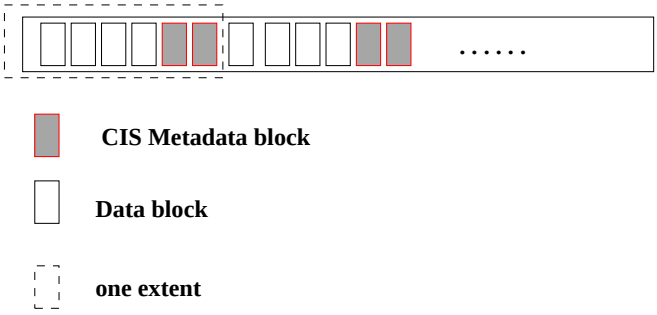

Figure 1. CIS disk layout. The disk is divided into fixed size extents. Each extent has one metadata block which tracks the status of the data blocks in this extent.

5. Performance Optimization

To achieve data immutability on rewritable media, CIS pays extra overhead to check the state information and update them for each command. For example, a *write* command could incur up to three disk I/Os in CIS. To improve CIS system throughput, we have carefully designed CIS disk layout to minimize disk head movements by co-locating state blocks and data blocks and uniformly distributing the state blocks across the disk. We also have minimized the critical section for synchronizing state information updates to reduce lock contention. Lastly, CIS aggressively merges the disk I/Os involved for each CIS command to reduce the number of I/Os sent to the devices. Our evaluation shows that, for read-intensive workloads, CIS can achieve system throughput that is comparable to a rewritable storage system. In addition, CIS performs comparably to rewritable storage systems in most common cases, and particularly for record keeping workloads where the dominant workload activities are the addition of records to many directories.

5.1. Disk Layout

CIS maintains the state information in its metadata blocks to track the status of each data block. Figure 1 illustrates the CIS disk layout on a rewritable disk. The disk can be a virtual disk or a physical disk. The disk is divided into fixed size extents. The size of an extent can be set by users according to the average write size. It is also constrained by the amount of state information that one meta block can hold. One or more contiguous physical blocks (sectors) are dedicated to metadata in each extent of CIS. In our current implementation, we use one physical block per extent for metadata.

The metadata block tracks the status of the data blocks in its associated extent. The state information that the meta

block keeps for each data block in the extent includes: length of written data, retention period, events, and deletion holds. Optionally, it can also keep some other tracing information for auditing purpose, such as last write time etc. The metadata blocks are not addressable by users. They are rewritable by CIS logic only and invisible to CIS storage users. Only data blocks are exposed to the users, and they appear as write-once blocks. If CIS maintains only retention and length states for each data block, then one metadata block of 512 bytes can hold state information for 85 data blocks. The space overhead for maintaining state information is less than 2%.

Without any optimizations, each operation that may add data to CIS involves at least three disk I/Os: *read metadata, write data*, and *write metadata*. We will illustrate this by describing how CIS handles a *write* command and an *append* command. Upon receiving a *write* command, CIS reads the metadata block(s) for the target blocks into memory. The state information is examined to determine if the write is allowed. If the target data blocks have never been written before, the write is allowed to proceed, otherwise the write operation fails. For the writes that passed the state check, CIS writes the data blocks to the target address and then writes the updated metadata blocks.

The *append* command is similarly carried out in CIS. The *append* operation allows a partial block to be written, and additional bytes to be subsequently appended to the unused portion of the data block. The meta block tracks the length of written bytes in each block. CIS reads into memory the state information and checks if enough unwritten space is available in the target data block. If there is enough space in the target block, CIS reads the data block into a temporary buffer. The bytes to be appended are spliced into the data block at the position where the last append completed. The new buffer is written to disk and the metadata block is updated and written back. In this case, four disk I/Os are needed.

One way to increase the performance of CIS is to uniformly distribute the metadata blocks over the media and among the data blocks. The benefit of such a disk layout is to minimize the disk head movements required to handle one CIS command. Since most CIS commands involve a disk read I/O for the metadata blocks, the meta blocks should be co-located with their associated data blocks. The extent size is a tunable parameter. The size of an extent should match the average I/O size of the applications. An extent size too small will cause a *write* command to span multiple extents, potentially incurring several I/Os to read the metadata for each of the extents. On the other hand, when the extent size is too big, the disk head movement between the read metadata step and the write data step is large.

5.2. Lock Contention

Another optimization in CIS is to minimize the performance overhead due to the synchronization of the meta block updates. Each meta block keeps state information for all the data blocks in an extent. Updates to a meta block need to be synchronized to ensure its data integrity and correctness. If updates to these data blocks in one extent arrive in different commands at the same time, the metadata updates essentially will be serialized and the CIS throughput may degrade. This scenario happens when small writes to different blocks of one extent occur at the same time. Discrete small size writes or appends to one extent could encounter such serialization penalty. Sequential small size writes or appends to one extent should have been coalesced by upper layers in the system. For example, a block device driver does a good job in sequential I/O coalescing. We give an example case on how a critical section is protected for a *write* command, which writes to a single block. A critical section covers the set of operations that need to sequentially executed without interruption. It is the code segment protected by a pair of lock/unlock operations. Upon the arrival of a *write* command, in the most safe or straightforward case, CIS will do the following:

1. Lock the meta block;
2. Read the meta block into memory if it is not already cached;
3. Write the data block to the disks;
4. Update the meta block in memory;
5. Write the meta block to the disks;
6. Unlock the meta block;
7. Acknowledge write success;

The above case assumes no Non-volatile memory (NVRAM) and guarantees strong data consistency. NVRAM can easily mask the contention overhead for meta block updates. However, for low cost archiving storage, we may not have NVRAM in the storage controller. Hence, here we examine how to achieve best performance through proper locking scheme rather than using extra hardware. The state in the metadata block accurately reflects what has reached media successfully. However, The critical section protected by this locking scheme involves disk I/Os which are orders of magnitude more time-consuming than in-memory operations. It potentially degrades the system throughput significantly for the scenarios with lock contention. Our goal is to minimize the duration of the critical section as much as we can.

The idea is to decouple the time-consuming disk I/O from the critical session [13]. Locks will be held at minimum duration of time to relieve lock contention. Here is the sequence of operations executed by CIS during a *write*.

1. Read the meta block into memory if it is not cached;
2. *Acquire lock; Update the data block(s) in memory;*
3. *Update the meta block in memory; Release lock;*
4. Acknowledge write success to CIS user, if the write cache memory is non-volatile;
5. Write the data block to the disks;
6. Write the meta block to the disks;
7. Acknowledge write success to CIS user, if the write cache memory is volatile;

The lock protected operations are step 2 and 3 only. All disk I/O related operations are out of the critical section. The CIS user does not receive an acknowledgment of a successful write immediately after the lock release, if the memory available is volatile. Instead, the user will receive the write success acknowledgment, only after the blocks have been committed to the disks. In the case when NVRAM is not available, this procedure still benefits from the reduced lock contention, because more disk I/Os can be coalesced in step 5 and 6 than in the non-optimized case. In this optimized procedure, the critical section is very short, which involves only a few in-memory operations. Furthermore, the data integrity is not compromised, since the success acknowledgment is not returned until the data reach non-volatile media, either NVRAM or hard disks.

5.3. I/O Coalescing

CIS merges multiple disk I/Os with one CIS command to improve system throughput, even when the I/Os are not exactly contiguous. To reduce the number of disk I/O's, CIS tries to merge the writes (reads) of meta and data blocks into a single I/O. If the meta block and the starting target data block are not physically contiguous on disk, CIS will also write (read) the intervening blocks to form a contiguous sequence of blocks, provided that the intervening blocks are available in the cache. This contiguous sequence of blocks can be read or written in one disk movement, hence minimizing disk I/Os. In this case, CIS reduces the original three or four disk I/Os for each CIS command to two I/Os only, a read and a write.

A potential problem with including the intervening blocks in a write, as described, is that, if the operation fails at writing the intervening blocks, the content of the intervening blocks could be corrupted. This issue could be handled by marking intervening blocks as dirty and placing them into the NVRAM. The contents of these intervening blocks cannot be replaced in the NVRAM until the related disk operations have been completed successfully. This way, there is always a correct non-volatile copy of the data that can be used to retry the write in case a failure is encountered.

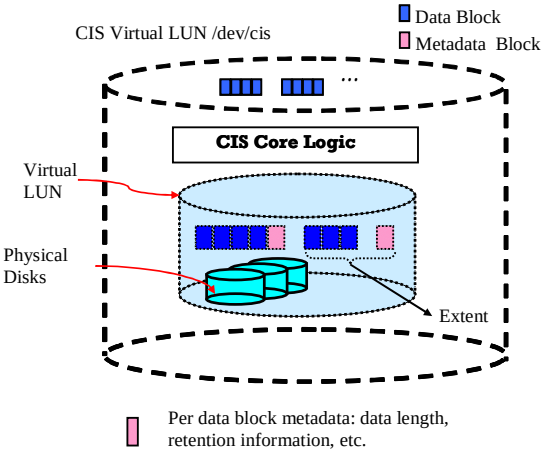

Figure 2. CIS logic is built on top of the existing virtualization layer provided by a RAID controller etc. It presents a virtual volume with term-WORM guarantees.

In some advanced disk drives, internal disk optimization can complete multiple commands in one disk head movement, even if the commands are targeted at non-contiguous block addresses. These optimizations achieve essentially the same effect as the merging of two discrete commands by CIS. In this case, CIS only needs to read or write target blocks and meta blocks, without including intervening blocks.

6. Implementation and Evaluation

Our design goal is to realize a CIS prototype that requires minimum changes to existing protocols, standards and hardware. We also aim to maximize the utilization of existing software/hardware functionalities for data protection and device management. CIS achieves these goals by paying the cost of extra meta data maintenance. However, we design CIS in a way that the overhead is close to negligible. We demonstrate the system performance of a working CIS prototype is comparable to a regular storage system under common workloads.

6.1. CIS Prototype Architecture

CIS is a very thin layer of virtualization for enforcing data immutability between applications and the disks. The underlying disk can be a virtual disk or a physical disk. Figure 2 illustrates how CIS remaps the address space of a logical volume, exposing only the data blocks to the CIS applications. The logical volume can be provided by any virtualization hardware or RAID controllers.

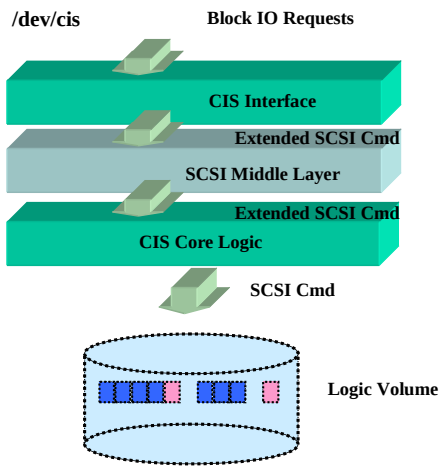

Figure 3. Software architecture of CIS prototype in Linux kernel.

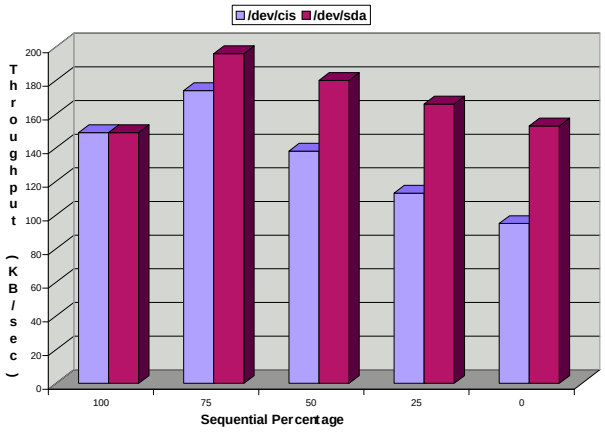

Figure 4. System throughput on CIS and rewritable media with varying percentage of sequential operations in the workload.

Our current CIS prototype is built as a block device driver in Linux 2.4.20 kernel, and is designed for portability to the embedded OS platform in a disk controller. Figure 3 shows how a CIS device driver interacts with the rest of the I/O stack of the Linux I/O subsystem. CIS presents a block device, e.g. /dev/cis. CIS interface layer implements some ioctl calls that support the CIS specific commands such as *query, append* etc. A block I/O request sent to this device will flow through the CIS interface, the SCSI middle layer and then be processed by the CIS core logic layer. CIS core logic layer checks the state information and updates state information on the disks accordingly. CIS core logic layer sends disk I/O commands to the low level devices. A small non-volatile buffer cache for caching metadata blocks is maintained by CIS core logic.

CIS provides special commands to support efficient indexing. Any file system that is aware of WORM media can be readily mounted on a CIS device, for example, a Linux UDF file system [17, 2]. In this case, Linux UDF file system utilizes only the overwrite protection feature of CIS, but not the efficient indexing support from *append* command. More advanced append-only file system or indexing software can be built on CIS.

6.2. Performance Evaluation

Our CIS prototype is composed of an Intel server with 2.4 GHZ CPU and 512 MB memory, a SATA disk array with four 7200 RPM SATA disks, and a SATA RAID controller. The default extent size for CIS is twenty 512 Bytes sectors, *i.e.*, 10 KB. The CIS device is built on top of a RAID-5 volume with three SATA disks. The CIS queue depth is ten unless otherwise noted. The purpose of the evaluation is to examine the system throughput impact due to state maintenance in CIS, compared to a standard rewritable storage system. The standard rewritable storage does not provide WORM guarantees. We evaluated the prototype with two set of traces, I/O traces for file creations and synthetic block level I/O traces. Traces are driven to raw device interfaces directly to avoid a file system caching effect. CIS read performance is the same as a rewritable storage system without WORM guarantee, since no state check overhead is added. We will focus on write-dominant traces and skip the discussion on read-only traces in the rest of the section.

File Size	/dev/cis	/dev/sda
16 KB	790 KB/sec	830 KB/sec
1 MB	4.66 MB/sec	4.86 MB/sec

Table 1. System throughput for CIS and rewritable storage. CIS only adds a 5% or less throughput degradation.

The first set of traces models the disk access pattern of a file creation on an archival system on CIS. The workload contains disk I/Os for file creation operations. We log the disk I/O traces of creating files spread over 4,000 directories on a WORM storage system. Each file creation involves a file data write and a directory update. The files are assigned to directories randomly. But the file blocks are allocated sequentially. So the randomness resides in directory block accesses only. The number of added bytes per file creation to the directory block is fixed and relatively small (tens of bytes). The larger the file/directory size ratio, the larger the file size. This file creation trace contains 100% writes. Table 1 shows the system throughput

of the random case for both a CIS device (/dev/cis) and a rewritable block device (/dev/sda). CIS shows 5% throughput degradation, compared to a rewritable block device, even after all the overheads to enforce data immutability are included. The amount of CIS cache is big enough to hold most frequently accessed directory blocks.

The second set of traces is synthetic block level I/O traces. We vary the percentage of I/O that is sequential and examine the system throughput differences. Figure 4 shows the system throughput in KB/sec. The trace is block level I/Os. 25% of the workload are reads and the rest are writes. I/O size is 1 KB. For sequential trace, CIS achieves the same throughput as a rewritable device. The reason is that the extra I/O overhead, due to the metadata block read, is absorbed by the CIS metadata cache and the controller read ahead cache. The extra metadata write I/O overhead is not noticeable due to I/O coalescing.With the increasing randomness of the trace, CIS's extra I/O overhead results in downgraded system throughput. CIS's throughput is only 61% of a rewritable device when the trace is random. An interesting observation on the trend of rewritable device itself, is that its throughput reaches peak when the percentage of sequential I/O is 75%, not 100%. The reason is that a sequential write following a read may invalidate the prefetched read buffers for the same blocks and make the prefetching useless. Some controller cache implementation actually invalidates all the prefetched blocks in the cache segment, besides the target block of the write operation.

Figure 5 shows the queuing effect for a mixed trace and a random trace. The workloads have 100% 1 KB writes. We vary the queuing effect by controlling the number of outstanding requests for CIS at any time. For both workloads, CIS achieves better throughput with increasing queue depth. An increasing queue depth allows more commands to be sent to the physical devices and the disk internal scheduling is more effective. Rewritable device, /dev/sda, sees the same trend. However, with locality in the workload (Figure 5(a)), CIS achieves 120% improvement in system throughput with a queue depth of ten, compared to a queue depth of one. And when the queue depth is ten, CIS's system throughput is 78% of that of /dev/sda. For the random trace, CIS can only achieve 58% of /dev/sda's system throughput. CIS improves 100% of its system throughput from the queue depth of one to the queue depth of ten. These two figures prove that a larger queue depth helps CIS to achieve better system throughput. Command queuing can effectively reduce the performance gap between CIS and a rewritable storage. For example, for a workload with 50% sequential I/Os, CIS's system throughput is only 22% lower than with a rewritable storage device.

Figure 6 shows the performance effect of each optimization we discussed in Section 5. Figure 6(a) examines the impact of the extent size on the system performance. We present three sets of results, associated with sequential, mixed and random workloads. Figure 6(b) illustrates the system throughput before and after various optimizations on sequential, mixed and random workloads. All workloads here are with 1 KB writes. As shown in Figure 6(a), for all three workloads, an increasing extent size corresponds to an increasing system throughput. Sequential workload has the biggest improvement of throughput when the extent size is increased. This can be explained by noting that with a larger extent size, more metadata will be retrieved from cache. This in turn will exceed the performance degradation due to the disk head seeking overhead. The sequential workload experiences the most metadata sharing, hence achieving the most significant performance improvement among all three workloads. However, this does not indicate that the extent size should be infinite to achieve the best system throughput. The extreme case of an infinite extent size, is similar to the case where metadata blocks are co-located on the first 100 MB of the disks. All data blocks are allocated after the first 100 MB. The performance of this extreme case is referred as *No Disk Layout Optimization* series in Figure 6(b). We can see that spreading the metadata blocks uniformly across the disk outperforms the condition when all the metadata blocks are co-located. The reason is that eventually the benefit of increasing the cache hit ratio by enlarging the extent size, will eventually be offset by the overhead of the disk head seek between the metadata area and the data area. Figure 6(b) shows that the sequential workloads experienced the most largest improvement in system throughput (776%) with all three optimizations: I/O coalescing, reduction of lock contention and uniform metadata layout. For the sequential trace, these three optimizations allow small writes to adjacent blocks to be simultaneously transferred to the device, and compensate the small write effect of RAID-5. In the case of a random trace, the disk layout optimization and the locking reduction effect are not significant. There is little lock contention with the random workload even before the optimization. Due to the write caching effect, the disk head is mainly constrained within the metadata zone. The disk operations here are mainly to serve metadata reads since data for writes are already cached. The disk head seeking overhead between the metadata area and data area is not significant. I/O coalescing achieves the most significant performance improvement in all cases among all three optimizations.

In summary, for read workload, CIS and a rewritable disk achieve the same throughput. CIS is able to achieve as good a system throughput as rewritable disks, when the metadata read I/O can be effectively reduced by a CIS metadata buffer cache. This is the case in our tests with file creation I/O traces. In those cases where metadata read

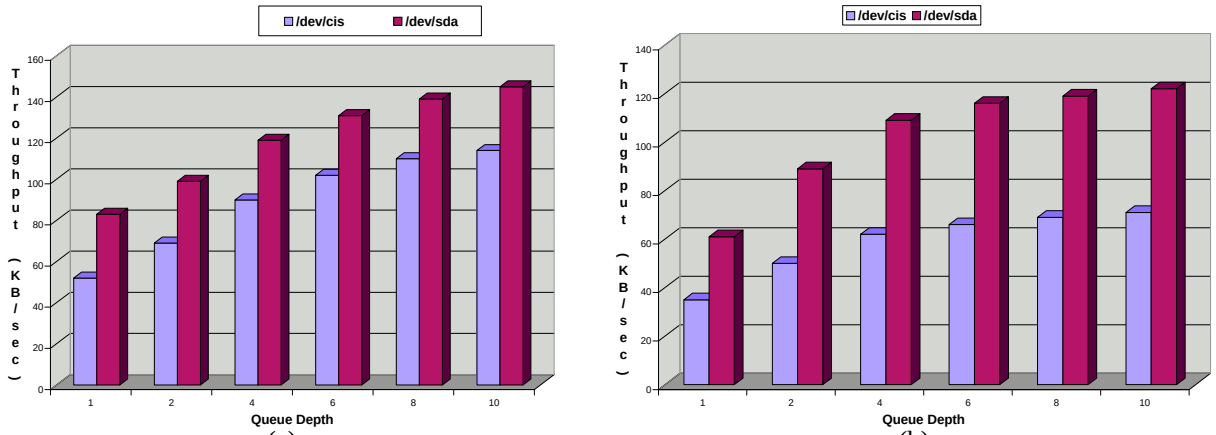

Figure 5. Queuing effect on CIS storage and rewritable storage. All operations are 1 KB writes. (a) 50% of the workload is sequential. (b) random workload.

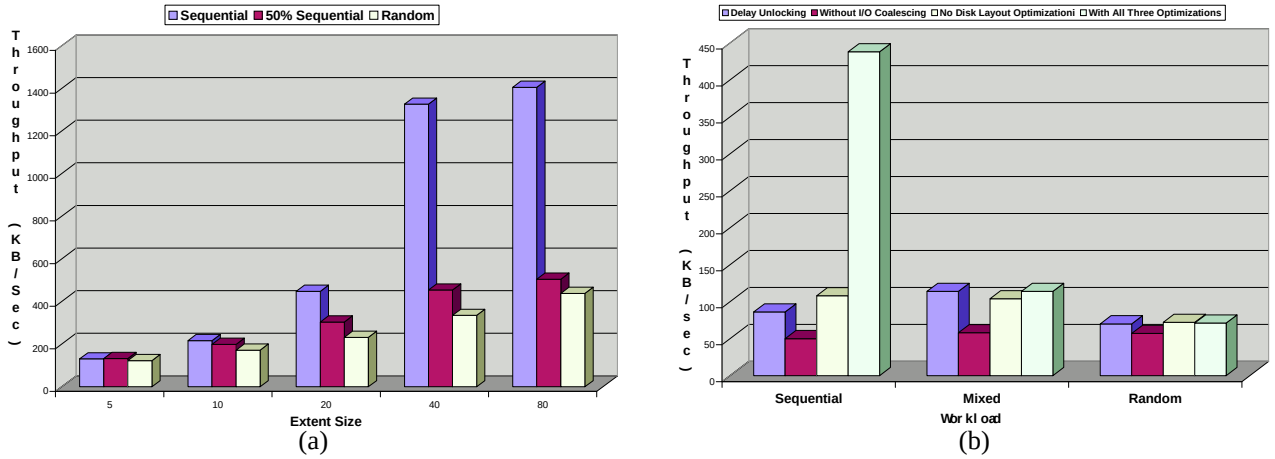

Figure 6. Performance optimization effect. 100% of the workload are 1 KB size writes. (a) System throughput on CIS storage with varying extent size. (b) System throughput on CIS storage with or without optimizations on sequential, mixed and random workload.

I/O cannot be avoided due to cache misses, CIS achieves at least half of the throughput of a rewritable disk. Command queuing can effectively amortize the performance gap between CIS and a rewritable storage. Our optimizations to further close this gap, I/O coalescing, lock contention reduction and uniform metadata layout, improve the system performance by up to 776%.

7. Conclusion

Proper record keeping is essential for the effective functioning of an organization. With our growing reliance on electronic data processing, records are increasingly generated in large volumes and in electronic form, which make them vulnerable to undetected deletion and modification. Ensuring that records are trustworthy, *i.e.*, capable of providing irrefutable proof and accurate details of events that have occurred, is therefore imperative. It is also in the best interests of the organization to properly dispose of records that have outlived their usefulness to the organization and have passed any mandated retention period.

We clearly establish the storage requirements for proper record keeping and contend that the current Write-Once-Read-Many (WORM) storage systems do not fully meet these requirements. We demonstrate a design and working prototype for CIS, which effectively meets the key storage requirements for record keeping as evidenced by the ability to 1) offer overwrite protection that is secure even against

inside attacks; 2) efficiently support index mechanisms; 3) allow records to be properly disposed of after they have expired; and 4) be low cost and yet reliable. CIS exports a standard block level SCSI interface to maximize the reuse of existing software/hardware stack. CIS securely enforces immutability check so that it can never be bypassed by intruders. It addresses the need of term-retention and record disposition. Our performance evaluation on the working prototype suggests that, for a record keeping workload, the performance of CIS is comparable to that of a regular storage system.

Acknowledgments

The authors would like to thank the shepherd, Jean Bedet, and the anonymous reviewers for their comments. The authors thank Wayne Hineman, Xiaonan Ma, and Shauchi Ong for their feedbacks on the early draft of the paper.

References

[1] Ablative Optical Disk. http://www.pegasus-ofs.com/optical.htm.

[2] Linux UDF File System. http://linux-udf.sourceforge.net.

[3] SCSI Block Command Interface Standard. http://www.t10.org.

[4] SNIA OSD Workgroup. http://www.snia.org/tech_activities/workgroups/osd/.

[5] E. Biham and R. Chen. Near-Collisions of SHA-0. In *24th Annual International Cryptology Conference*, pages 290–305, Santa Barbara, California, USA, 2004.

[6] Cohasset Associates, Inc. The role of optical storage technology. White Paper, Apr. 2003.

[7] Congress of the United States of America. Sarbanes-Oxley Act of 2002, 2002. Available at http://thomas.loc.gov.

[8] EMC Corp. EMC Centera Content Addressed Storage System, 2003. http://www.emc.com/products/systems/centera_ce.jsp.

[9] Erik Riedel. SNIA OSD Tutorial, April 2005. http://www.snia.org/education/tutorials/spr2005/storage/Object-basedStorageDevice(OSD)Basics.pdf.

[10] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *6th USENIX Security Symposium*, July 1996.

[11] IBM Corp. IBM TotalStorage DR550, 2004. http://www-1.ibm.com/servers/storage/disk/dr.

[12] A. Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In *24th Annual International Cryptology Conference*, pages 306–316, Santa Barbara, California, USA, 2004.

[13] L. Huang and T. Chiueh. Charm: An I/O-driven high-performance transaction processing system. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.

[14] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[15] Natinal Institue of Standards and Technology. FIPS 180-1, Secure Hash Standard, April 1995. US Department of Commerce.

[16] Network Appliance, Inc. SnapLock^{TM} Compliance and SnapLock Enterprise Software, 2003. http://www.netapp.com/products/filer/snaplock.html.

[17] Optical Storage Technology Association. Universal disk format specifications. http://www.otsa.org/specs/index.html.

[18] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, 1995.

[19] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of First USENIX conference on File and Storage Technologies*, 2002.

[20] Securities and Exchange Commission. SEC Interpretation: Commission Guidance to Broker-Dealers on the Use of Electronic Storage Media under the Electronic Signatures in Global and National Commerce Act of 2000 with Respect to Rule 17a-4(f), 2001. Available at http://www.sec.gov/rules/interp/34-44238.htm.

[21] Socha Consulting LLC. The 2004 Socha-Gelbmann Electronic Discovery Survey, 2004.

[22] SONY Corp. AIT-2/AIT-3 WORM Drives & Libraries, 2003. http://www.storagebysony.com/products/prod_hilite4.asp.

[23] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *4th Symposium on Operating System Design and Implementation (OSDI)*, pages 165–180, October 2000.

[24] The Enterprise Storage Group, Inc. Compliance: The effect on information management and the storage industry, May 2003.

[25] X. Wang, Y. L. Yin, and H. Yu. Collision Search Attacks on SHA1. http://theory.csail.mit.edu/ yiqun/shanote.pdf.

[26] Q. Zhu and W. W. Hsu. Fossilized index: The linchpin of trustworthy non-alterable electronic records. In *ACM SIGMOD Conference*, pages 395–406, Baltimore, Maryland, USA, 2005.