

# A Case for Continuous Data Protection at Block Level in Disk Array Storages

Weijun Xiao, *Student Member, IEEE*, Jin Ren, and Qing Yang, *Senior Member, IEEE*

**Abstract**—This paper presents a study of data storages for continuous data protection (CDP). After analyzing the existing data protection technologies, we propose a new disk array architecture that provides Timely Recovery to Any Point in time, referred to as TRAP. TRAP stores not only the data stripe upon a write to the array but also the time-stamped Exclusive ORs (XORs) of successive writes to each data block. By leveraging the XOR operations that are performed upon each block write in today's RAID4/5 controllers, TRAP does not incur noticeable performance overhead. More importantly, TRAP is able to recover data very quickly to any point in time upon data damage by tracing back the sequence and history of XORs resulting from writes. What is interesting is that the TRAP architecture is very space efficient. We have implemented a prototype of the new TRAP architecture using software at the block level and carried out extensive performance measurements using TPC-C benchmarks running on Oracle and Postgres databases, TPC-W running on a MySQL database, and file system benchmarks running on Linux and Windows systems. Our experiments demonstrated that TRAP not only is able to recover data to any point in time very quickly upon a failure but also uses less storage space than traditional daily incremental backup/snapshot. Compared to the state-of-the-art CDP technologies, TRAP saves disk storage space by one to two orders of magnitude with a simple and a fast encoding algorithm. In addition, TRAP can provide two-way data recovery with the availability of only one reference image in contrast to the one-way recovery of snapshot and incremental backup technologies.

**Index Terms**—Disk array, disk I/O, data storage, data protection and recovery, data backup.

## 1 INTRODUCTION

THE RAID architecture [1] has been the most prominent architecture advance in disk I/O systems for the past two decades. RAID1 provides 2xN data redundancy to protect data, while RAID3 through RAID5 store data in parity stripes across multiple disks to improve space efficiency and performance over RAID1. The parity of a stripe is the Exclusive OR (XOR) of all data chunks in the stripe. If a disk failed at time  $t_0$  and the system found such a failure at time  $t_1$ , the data in the failed disk can be recovered by doing the XOR among the good disks, which may finish at  $t_2$ . The recovered data is exactly the same image of the data as it was at time  $t_0$ . There are recent research results that are able to recover data from more than one disk failure [2], [3], [4], [5], improving the data reliability further.

The question to be asked is: "Can we recover data at time  $t_2$  to the data image of  $t_0$  after we found out at time  $t_1$  that data was damaged by human errors, software defects, virus attacks, power failures, or site failures?"

With the rapid advances in networked information services coupled with the maturity of disk technology, data damage and data loss caused by human errors, software defects, virus attacks, power failures, or site failures have become more dominant, accounting for 60 percent [6] to 80 percent [7] of data losses. Recent research [8], [9] has shown that data loss or data unavailability can cost up to

millions of dollars per hour in many businesses. The current RAID architecture cannot protect data from these kinds of failures because damaged data are not confined to one or two disks.

Traditional techniques protecting data from the above failures are mainly periodical (daily or weekly) backups and snapshots [10], [11], [12]. These techniques usually take a long time to recover data [13]. In addition, the data between backups are vulnerable to data loss. More importantly, a recent research study has shown that 67 percent of backup data cannot be recovered in the real world [14]. While this fact is well known, there has been no research study on why this is the case. Therefore, it remains unclear and an open question why such high percentage of data recovery failed.

This paper presents an analytical study on snapshot and backup technologies from the block-level storage point of view. Our investigation uncovers the limitations of the existing data protection technologies and provides theoretical explanations as to why so many data recoveries (over 67 percent recoveries) failed using these existing technologies. We show mathematically the data recovery capabilities and limitations of the existing technologies.

Based on our theoretical results, we propose a new storage architecture that overcomes the limitations of existing technologies. We provide a mathematical proof of the correctness of the new data protection technique. Besides being able to recover data damaged by various types of failures, our new architecture provides Timely Recovery to Any Point in time, hence named TRAP architecture. The TRAP architecture has optimal space and performance characteristics [15]. The idea of the new TRAP architecture is very simple. Instead of providing full redundancy of data in the time dimension, i.e., keeping a log of all previous versions of changed data blocks in time

• The authors are with the Department of Electrical, Computer, and Biomedical Engineering, University of Rhode Island, Kingston, RI 02881. E-mail: {wjxiao, rjin, qyang}@ele.uri.edu.

Manuscript received 11 Aug. 2007; revised 8 Aug. 2008; accepted 11 Aug. 2008; published online 15 Aug. 2008.

Recommended for acceptance by X. Zhang.

For information on obtaining reprints of this article, please send e-mail to: [tpds@computer.org](mailto:tpds@computer.org), and reference IEEECS Log Number TPDS-2008-04-0136. Digital Object Identifier no. 10.1109/TPDS.2008.154.

sequence [13], [16], [17], we compute XORs among changed data blocks along the time dimension to improve performance and space efficiency. With a simple and fast encoding scheme, the new *TRAP* architecture presents great space savings because of the *content locality* that exists in real-world applications.

We have implemented a prototype of the new *TRAP* architecture at the block level using the standard iSCSI protocol. The prototype is a software module inside an iSCSI target mountable by any iSCSI-compatible initiator. We install the *TRAP* prototype on PC-based storage servers as a block-level device driver and carry out experimental performance evaluation as compared to traditional data recovery techniques. Linux and Windows systems and three types of databases, Oracle, Postgres, and MySQL, are installed on our *TRAP* prototype implementation. Real-world benchmarks such as TPC-C, TPC-W, and file system benchmarks are used as workloads driving the *TRAP* implementation under the databases and file systems. Our measurement results show improvements of up to two orders of magnitude of the new *TRAP* architecture over existing technologies in terms of storage space efficiency. Such order-of-magnitude improvements are practically important given the exponential growth of data [18]. We have also carried out data recovery experiments by selecting any point in time in the past and recovering data to that time point. Experiments have shown that all recovery attempts are successful. The recovery time of the new *TRAP* architecture is compared with that of existing reliable storage architectures to show that the new *TRAP* architecture can recover data to any point in time very quickly.

We analyze the capabilities and limitations of existing data protection technologies in the next section. The detailed design and implementation of the new *TRAP* architecture is presented in Section 3. Section 4 discusses the system design and implementation, and Section 5 presents the experimental settings and the workload characteristics. Numerical results and discussions are presented in Section 6. Related work is discussed in Section 7. We conclude our paper in Section 8.

## 2 CAPABILITIES AND LIMITATIONS OF CURRENT DATA PROTECTION TECHNOLOGIES

Traditionally, data protection has been done using periodic backups. At the end of a business day or the end of a week, data are backed up to tapes. Depending on the importance of data, the frequency of backups varies. The higher the backup frequency, the larger the backup storage required. In order to reduce the backup volume size, technologies such as incremental backups and copy-on-write (COW) snapshots have been commonly used. Instead of making full backups every time, incremental backups and COW snapshots that only store the changed data are done more frequently in between full backups. For example, one can do daily incremental backups and weekly full backups that are stored at both the production site and the backup site.

The way incremental backup works is described as follows: Starting from the previous backup point, the storage keeps track of all changed blocks. At the backup time point, a backup volume is formed, consisting of all the latest changed data blocks. As a result, the incremental

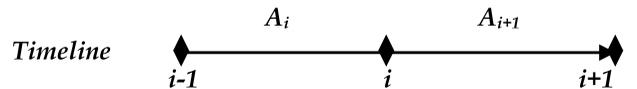


Fig. 1. A three-point timing diagram:  $i - 1$  starting point,  $i + 1$  current point, and  $i$  recovery point.

backup contains the newest data that have changed since the last backup. COW snapshots work differently from the incremental backup. At the time when a snapshot is created, a small volume is allocated as a snapshot volume with respect to the source volume. Upon the first write to a data block after the snapshot was started, the original data of the block is copied from the source volume to the snapshot volume. After copying, the write operation is performed on the block in the source volume. As a result, the data image at the time of the snapshot is preserved. Write I/Os after the first change to a block is performed as usual, i.e., only the first write to a block copies the original data to the snapshot volume. There have been many variations of COW snapshots in terms of implementation details for performance and efficiency purposes such as pointer remapping [48], redirect on writes [19], [49], etc. The main advantage of both incremental backups and COW snapshots is storage savings because only changed data are backed up.

Let us consider a data storage that consists of independent and equally sized data blocks (the specific size of a block is not significant in this discussion). Each of these data blocks is identified by a logic block address (LBA) and contains a specific data value. Let  $A$  be the entire set of LBAs of the data storage considered and  $D$  represent the set of all possible data values contained in data blocks. A binary relation  $R$  between  $A$  and  $D$  defines a mapping of addresses to their corresponding data values of the data storage. Since there is exactly one ordered pair in  $R$  with each LBA, this binary relation is a function. We refer to this function as storage data and use  $F_t$  to represent this function (storage data) from  $A$  to  $D$  at time  $t$ . And we use  $F_t(a)$  to represent the image or data value of an LBA  $a$ . That is,  $F_t$  contains a set of ordered pairs such as  $\{(a_1, d_1), (a_2, d_2) \dots\}$ , whereas  $F_t(a)$  is an image/data value of  $a$  such as  $F_t(a_1) = d_1$ . If  $A'$  is a subset of  $A$ , i.e.,  $A' \subseteq A$ , then we use  $F_t/A'$  to represent the restriction of  $F_t$  to  $A'$ . That is,  $F_t/A' = F_t \cap (A' \times D)$  [20].

Without loss of generality, let us consider three time points as shown in Fig. 1. Suppose that time point  $i - 1$  represents the original time point when data storage operation starts and time point  $i + 1$  represents the current time point. Suppose a failure occurred at some time near point  $i + 1$ . We are interested in recovering data to as it was at time point  $i$ . We use integer numbers to represent time points since all storage events occur at discrete time points with a clear sequential ordering.

**Definition 1.** Let  $A_i \subseteq A$  be a set of LBAs. We define  $A_i$  to be a write set  $i$  if it contains all LBAs whose data value has been overwritten between time point  $i - 1$  and time point  $i$ .

Looking at the diagram shown in Fig. 1, we have  $A_i$  containing all LBAs whose data values have been changed by write operations between time points  $i - 1$  and  $i$  and  $A_{i+1}$  containing all those between time point  $i$  and time point  $i + 1$ .

**Example 1.** If we have  $F_i = \{(0, 2), (1, 5), (2, 8)\}$  at time point  $i$  and  $F_{i+1} = \{(0, 4), (1, 5), (2, 0)\}$  at time point  $i + 1$  because of write operations, then we have  $A_{i+1} = \{0, 2\}$ . That is, data values at addresses 0 and 2 have been changed from 2 and 8 to 4 and 0, respectively, whereas the data value of address 1 has not been changed, since time point  $i$ .

It is possible that the overwritten value as seen at time point  $i$  is the same as the original value at time point  $i - 1$  caused by one or several write operations between time points  $i - 1$  and  $i$ . We therefore define a *substantial write set* that actually changed data values as follows:

**Definition 2.** Let  $A'_i \subseteq A_i$ . We define  $A'_i$  to be a substantial write set  $i$  if the data value of every LBA in  $A'_i$  has been changed between time point  $i - 1$  and time point  $i$ .

It should be noted here that the changed data value is generally not related to the original value because of the nature of write operations at block-level storages. That is,  $F_{i+1}(a)$  is independent of  $F_i(a)$ . Furthermore,  $F_i(a)$  is independent of  $F_i(b)$  for all  $b \in A$  and  $b \neq a$ , as stated in the beginning of this section: data blocks are independent. We believe this assumption is reasonable because block-level storages regard each data block as an independent block without any knowledge of the file systems and applications above them.

**Definition 3.** A COW snapshot as seen at time point  $i + 1$  that was started at time point  $i$  is defined as  $F_i/A_{i+1}$ , where  $A_{i+1}$  is write set  $i + 1$ .

As we know, a COW snapshot makes a copy of the original data upon the first write to the block. As a result, it keeps a set of original data of all changed blocks since the snapshot started. Consider the storage data in Example 1. Suppose the COW snapshot was started at time point  $i$ . At time point  $i + 1$ , we have the snapshot  $\{(0, 2), (2, 8)\}$ , which is  $F_i/A_{i+1}$ . That is,  $A_{i+1}$  gives all the LBAs that have been written, i.e.,  $\{0, 2\}$ , and their respective images in the snapshot should be the same as they were at time point  $i$ , i.e.,  $\{2, 8\}$ .

**Lemma 1.** If we have storage data at time point  $i + 1$  and a COW snapshot started at time point  $i$ , then we can recover data as they were at time point  $i$  as follows:

$$F_i = (F_{i+1} - F_{i+1}/A_{i+1}) \cup F_i/A_{i+1}, \quad (1)$$

where “ $-$ ” and “ $\cup$ ” are difference and union operators of sets, respectively.

The proof of this lemma is straightforward by noting that  $F_i/A_{i+1}$  is the COW snapshot as seen at time  $i + 1$  that was started at time  $i$ , and  $F_{i+1}/A_{i+1}$  are all storage data that have been changed since time point  $i$ . Equation (1) replaces all changed data with the COW snapshot that represents the original data before changes occur. This is a typical undo recovery process.

Lemma 1 gives the data recovery capability of COW snapshot technology. It is able to recover data to a previous time point provided that the most recent data is available.

This data recovery capability is very useful in practice in case of data corruption, virus attack, user errors, software bugs, and so forth. If we know that data was good at a previous time point when the snapshot was started, we can go back to that point to recover from failures caused by this type of events.

Although the COW snapshot can recover data to a previous time point as stated in Lemma 1, it has limitations. In particular, if the current data (production data) is damaged or lost because of hardware failures, OS failures, outages, or disasters, we cannot recover data to a previous time point even if we have COW snapshots and previous backup data that may be safely stored in a remote backup site. This limitation is formally stated in the following theorem.

**Theorem 1.** Suppose the storage data at time point  $i + 1$ ,  $F_{i+1}$ , is not available and the substantial write set  $A'_i$  is not empty ( $A'_i \neq \phi$ ). COW snapshots cannot recover storage data  $F_i$  as they were at time point  $i$  if  $A'_i \subseteq A_{i+1}$ .

**Proof.** We prove this theorem by contradiction. Let us assume that COW snapshots can recover storage data  $F_i$  as they were at time point  $i$  without  $F_{i+1}$ . That is, for all  $a \in A_i$ , we can reconstruct  $F_i(a)$  from what we have available:

- Data backup made previously:  $F_{i-1}$ ,
- COW snapshot as seen at time point  $i$  that was started at time  $i - 1$ :  $F_{i-1}/A_i$ , and
- COW snapshot as seen at time point  $i + 1$  that was started at time  $i$ :  $F_i/A_{i+1}$ .

Since different data blocks are independent in our storage system, for every LBA  $a \in A_i$ , the only way to reconstruct its data value,  $F_i(a)$ , is to reconstruct it from  $F_{i-1}(a)$ ,  $F_{i-1}/A_i(a)$ , and/or  $F_i/A_{i+1}(a)$ .

Because  $A'_i \subseteq A_{i+1}$  and  $A'_i \neq \phi$ , there is an LBA that is in  $A'_i$  but not in  $A_{i+1}$ . Let  $\beta$  be such an LBA such that  $\beta \in A'_i$  but  $\beta \notin A_{i+1}$ . Now, consider the three cases:

- Since  $\beta \in A'_i$ , we have  $F_i(\beta) \neq F_{i-1}(\beta)$  by **Definition 2**.
- Because  $F_{i-1}/A_i \subseteq F_{i-1}$  and  $A'_i \subseteq A_i$ , we have  $F_{i-1}/A_i(\beta) = F_{i-1}(\beta) \neq F_i(\beta)$ .
- The fact that  $\beta \notin A_{i+1}$  implies that  $F_i/A_{i+1}(\beta)$  is undefined because  $\beta$  is not in the domain of  $F_i/A_{i+1}$ .

Furthermore,  $F_i(\beta)$  is not related in any way to  $F_{i-1}(\beta)$  because of the nature of write operations at block-level storages. As a result, it is impossible to rebuild  $F_i(\beta)$  from  $F_{i-1}(\beta)$ ,  $F_{i-1}/A_i(\beta)$  and/or  $F_i/A_{i+1}(\beta)$ , contradicting our assumption that all data blocks can be reconstructed from snapshots and storage data at time point  $i - 1$ . Therefore, COW snapshots cannot recover storage data  $F_i$ .  $\square$

**Example 2.** Consider one example with six blocks in the storage data, as shown in Fig. 2. At time point  $i - 1$ , we have  $\{(0, a_0), (1, b_0), (2, c_0), (3, d_0), (4, e_0), (5, f_0)\}$ . From time point  $i - 1$  to time point  $i$ , three blocks have been changed to  $\{(0, a_1), (1, b_1), (3, d_1)\}$ , with the substantial write set being  $\{0, 1, 3\}$ . From time point  $i$  to time point  $i + 1$ , two blocks have been changed to  $\{(3, d_2), (4, e_2)\}$ ,

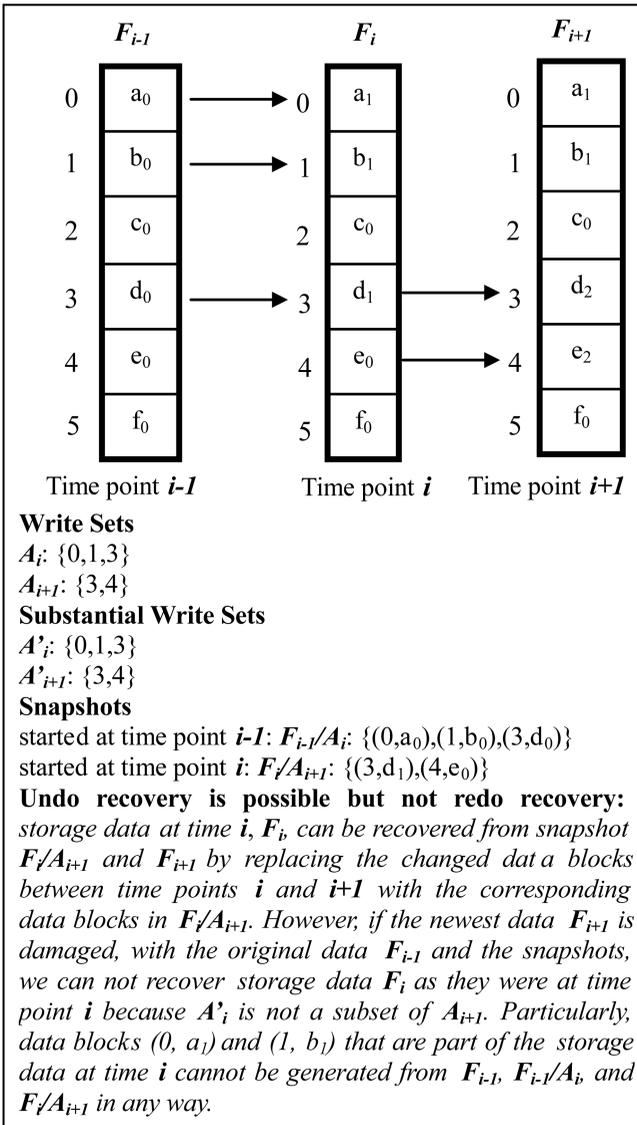


Fig. 2. Undo recovery of COW snapshots.

with the substantial write set being  $\{3, 4\}$ . By Definition 3, we have snapshot  $F_{i-1}/A_i$  as  $\{(0, a_0), (1, b_0), (3, d_0)\}$  and snapshot  $F_i/A_{i+1}$  as  $\{(3, d_1), (4, e_0)\}$ . When original data  $F_{i-1}$  is unavailable, storage data  $F_i$  can be reconstructed from COW snapshot  $F_i/A_{i+1}$  and  $F_{i+1}$  by replacing the changed blocks  $(3, d_2)$  and  $(4, e_2)$  in  $F_{i+1}$  with original data blocks  $(3, d_1)$  and  $(4, e_0)$  in  $F_i/A_{i+1}$ , respectively. If fresh data  $F_{i+1}$  is damaged, however,  $F_i$  cannot be recovered from  $F_{i-1}$  and snapshots because substantial write set  $A'_i$  is not a subset of write set  $A_{i+1}$ , as stated in Theorem 1. In this particular case, data blocks  $(0, a_1)$  and  $(1, b_1)$  cannot be rebuilt from original data  $F_{i-1}$  and snapshots in any way.

For the Incremental backup, it keeps the latest changes on data storage. It is able to recover data to a recent time point when the original storage data is available. This redo recovery can be used in practice in case of disk failures, volume crash, OS failures, outages, disasters, and so on. If we created a full data backup before the incremental

backup was started, we can reconstruct the storage data to the latest time point in the case of this type of failures.

However, incremental backups also have limitations. Particularly, if the current data gets corrupted because of virus or user errors and it happens that we do not have a prior full backup, we cannot recover data to a good time point using incremental backups and current data that are available. Similar to COW snapshots, we have the following results.

**Definition 4.** The incremental backup as seen at time point  $i$  that was started at time point  $i - 1$  is defined as  $F_i/A_i$ , where  $A_i$  is write set  $i$ .

**Lemma 2.** If we have storage data at time point  $i - 1$  and an incremental backup as seen at time point  $i$ , then we can recover data as they were at time point  $i$  as follows:

$$F_i = (F_{i-1} - F_{i-1}/A_i) \cup F_i/A_i, \quad (2)$$

where “-” and “ $\cup$ ” are difference and union operators of sets, respectively.

**Theorem 2.** Suppose the storage data at time point  $i - 1$ ,  $F_{i-1}$ , is not available and substantial write set  $A'_{i+1}$  is not empty ( $A'_{i+1} \neq \phi$ ). Incremental backups cannot recover storage data  $F_i$  as they were at time point  $i$  if  $A'_{i+1} \not\subseteq A_i$ .

The proofs of the above lemma and theorem are similar to those of COW snapshots and omitted here.

### 3 A NEW ARCHITECTURE FOR DATA PROTECTION

As we described in Section 2, snapshots cannot redo storage data to a recent time point, while incremental backups cannot undo storage data to a previous good point. The reason is that snapshots do not keep the fresh data and incremental backups do not store the original data. To overcome the limitations, a straightforward approach is to keep both versions of data every time a data change occurs. Particularly, at time point  $i$ , both snapshot  $F_{i-1}/A_i$  for the original data and incremental backup  $F_i/A_i$  for the fresh data as seen at time point  $i$  are stored at the backup volume. Similarly,  $F_i/A_{i+1}$  and  $F_{i+1}/A_{i+1}$  at time point  $i + 1$  are kept in the storage. From the previous section, one can easily find that storage data at time point  $i$ ,  $F_i$ , can be recovered by using COW snapshot  $F_i/A_{i+1}$  and fresh data  $F_{i+1}$  when storage data  $F_{i-1}$  is unavailable or by using incremental backup  $F_i/A_i$  and original data  $F_{i-1}$  when fresh data  $F_{i+1}$  is damaged or lost.

Although the above approach can recover data in two directions, it requires double the amount of storage space because two versions of changed data are stored at the backup storage. The question to be asked is: “Can we have an architecture to provide two-way recovery with the smaller size storage space?”

This question motivates us to seek for a new data protection technology. The idea of our new approach is very simple. Instead of keeping all versions of a data block as it is being changed by write operations, we keep a log of parities as a result of each write on the block. Since all parities of write operations are stored at backup storage volume, our approach can provide Timely Recovery to Any Point in time

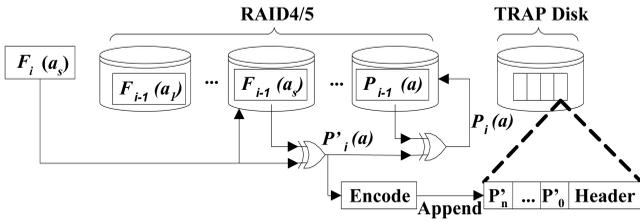


Fig. 3. Block diagram of the TRAP design.

by parity computation. Therefore, we named our approach as *TRAP*. Fig. 3 shows the basic design of the *TRAP* architecture. Suppose that at time point  $i$ , the host writes into a data block with LBA  $a_s$  that belongs to a data stripe  $a = (a_1, a_2, \dots, a_s, \dots, a_n)$ . The RAID controller performs the following operation to update its parity disk:

$$P_i(a) = F_i(a_s) \oplus F_{i-1}(a_s) \oplus P_{i-1}(a), \quad (3)$$

where  $P_i(a)$  is the new parity for the corresponding stripe,  $F_i(a_s)$  is the new data for data block  $a_s$ ,  $F_{i-1}(a_s)$  is the old data of data block  $a_s$ , and  $P_{i-1}(a)$  is the old parity of the stripe. Leveraging this computation, *TRAP* appends the first part of the above equation, i.e.,  $P'_i(a) = F_i(a_s) \oplus F_{i-1}(a_s)$ , to the parity log stored in the *TRAP* disk after a simple encoding box, as shown in Fig. 3.

Fig. 3 considers the simple case of a single block update for parity computation. When a data stripe involves multiple block modifications, we can still take advantage of parity computation for *TRAP* design. Suppose  $a_s$  and  $a_t$  are two data blocks of data stripe  $a$ .  $F_i(a_s)$  and  $F_i(a_t)$  are the new data for data block  $a_s$  and  $a_t$ , respectively. The RAID controller performs the parity computation using the follow equation:

$$P_i(a) = \left( F_i(a_s) \oplus F_{i-1}(a_s) \right) \oplus \left( F_i(a_t) \oplus F_{i-1}(a_t) \right) \oplus P_{i-1}(a). \quad (4)$$

This algorithm of parity computation is called Read-Modify-Write, which requires reading the original data values for all updated data blocks [21]. During the process of the parity computation, *TRAP* can append the first two parts of (4), which respectively reflect the exact changes of data blocks  $a_s$  and  $a_t$ , to the parity log stored in the *TRAP* disk for recovery purposes. It should be mentioned that there is another algorithm called Reconstruct-Write to compute the parity for multiple block modifications besides Read-Modify-Write. Reconstruct-Write reads the data values for all nonmodified blocks and rebuilds the parity from the fresh data blocks in one data stripe instead of reusing the old parity. This algorithm is very efficient for cases where the whole stripe or most of data blocks in a stripe need to be updated because it can reduce READ I/O operations [22]. If parity computation is done in this way, *TRAP* cannot take advantage of RAID parity computation and has to pay an additional cost for parity computation and encoding. Fortunately, this additional overhead is not

noticeable compared to disk accesses; we will discuss this overhead further in later sections.

The *TRAP* architecture makes it possible to recover data either backward, referred to as “undo,” or forward, referred to as “redo.” With traditional snapshot or backup storages, this two-way recovery is impossible, as shown in Section 2. Existing technologies can only recover data in one direction: COW snapshots can only recover data by “undo,” while incremental backups can recover data by “redo.” Being able to recover data in two directions gives a lot of practical benefits in terms of recoverability and recovery time.

Consider the parity log corresponding to a data block  $a$ , after a series of write operations, the log contains  $(P'_1(a), P'_2(a), \dots, P'_{i-1}(a), P'_i(a), \dots)$  with time points  $1, 2, \dots, i-1$ , and  $i$  associated with the parities. Suppose that we only have the data image at time point  $r$  ( $1 \leq r \leq i$ ) and all parities, and we would like to recover data backward or forward. To do a forward recovery to time point  $s$  ( $s > r$ ), for example, we perform the following computation for each data block  $a$ :

$$F_s(a) = F_r(a) \oplus P'_{r+1}(a) \oplus \dots \oplus P'_{s-1}(a) \oplus P'_s(a), \quad (5)$$

where  $F_s(a)$  denotes the data value of block  $a$  at time point  $s$  and  $F_r(a)$  denotes the data value of  $a$  at time point  $r$ . Note that

$$P'_l(a) \oplus F_{l-1}(a) = F_l(a) \oplus F_{l-1}(a) \oplus F_{l-1}(a) = F_l(a),$$

for all  $l = 1, 2, \dots, i$ . Therefore, (5) gives  $F_s(a)$  correctly, assuming that the data value,  $F_r(a)$ , exists.

The above process represents a typical redo recovery process while earlier data is available. A backward process is also possible with the parity log if the newest data is available by doing the following computation instead of (5):

$$F_s(a) = F_r(a) \oplus P'_r(a) \oplus P'_{r-1}(a) \oplus \dots \oplus P'_{s+1}(a), \quad (6)$$

where  $s < r$ . This is a typical undo process by using the newest data that is available. In order to recover data in either direction, only one reference image is needed along the time dimension because of the commutative property of XOR computation. This reference image could be the original data image, the fresh data image, or any data image in the middle. It does not need double the size of data images for two-way recovery.

Besides being able to recover data in two directions, *TRAP* is very space efficient. Our extensive experiments have demonstrated a very strong content locality that exists in real-world applications and have shown that only 5 percent to 20 percent of bits inside a data block actually change on a write operation. The parity,  $P'_i(a)$ , reflects the exact changes at the bit level of the new write operation on the existing block. As a result, this parity block contains mostly zeros with a very small portion of bitstream that is nonzero. Therefore, it can be easily encoded to a small-size parity block to be appended to the parity log reducing the amount of storage space required to keep track of the history of writes.

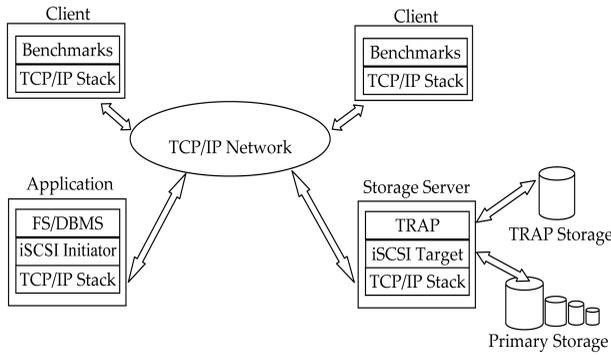


Fig. 4. System architecture of TRAP-4 implementation.

## 4 SYSTEM DESIGN AND IMPLEMENTATION

We have designed and implemented a software prototype of *TRAP*. The software prototype is a block-level device driver below a file system or database systems. As a result, our implementation is file system and application independent. Any file system or database applications can readily run on top of our *TRAP*. The prototype driver takes write requests from a file system or database system at the block level. Upon receiving a write request, *TRAP* performs a normal write into the local primary storage and at the same time performs parity computation as described above to obtain  $P'$ . The results of the parity computation are then appended to the parity log corresponding to the same LBA to be stored in the *TRAP* storage.

Our implementation is done using the standard iSCSI protocol, as shown in Fig. 4. In the iSCSI protocol, there are two communication parties, referred to as the iSCSI initiator and the iSCSI target [23]. Our *TRAP* module is implemented inside the iSCSI target as an independent module. The main functions inside the *TRAP* module include parity computation, parity encoding, and logging. The parity computation part calculates  $P'_i(a)$  as discussed above. Our implementation works on a configurable and fixed block size, referred to as the *parity block size*. The parity block size is the basic unit based on which parity computation is done. All disk writes are aligned to the fixed parity block size. As a result, a disk write request may be contained in one parity block or may go across several blocks, depending on the size and starting LBA of the write. The parity encoding part uses the open source [24] library to encode the parity before appending it to the corresponding parity log. The logging part organizes the parity log, allocates disk space, and stores the parity log in the *TRAP* disk. The *TRAP* module runs as a separate thread parallel to the normal iSCSI target thread. It communicates with the iSCSI target thread using a shared-queue data structure.

It should be mentioned that we applied a very simple encoding algorithm from *zlib* to our implementation because parties have strong content locality. Although there are better compressing and encoding algorithms available, such as motion estimation, running-length encoding [25], etc., we take a simple approach in this paper.

As shown in Fig. 4, our implementation is on top of the standard TCP/IP protocol. As a result, our *TRAP* can be set up at a remote site from the primary storage through an Internet connection. Together with a mirror storage at the

remote site, *TRAP* can protect important data from site failures or disaster events.

We have also implemented a recovery program for our *TRAP*. For a given recovery time point  $r$ , the recovery program retrieves the parity log to find maximum time point  $s$  such that  $s \leq r$  for every data block that have been changed. We then decode the parity blocks and compute XOR using either (5) or (6) to obtain the data block as it was at time point  $r$  for each block. Next, the computed data are stored in a temporary storage. Consistency check is then performed using the combination of the temporary storage and the mirror storage. The consistency check may be done several times until the storage is consistent. After consistency is checked, the data blocks in the temporary storage are stored in place in the primary storage, and the recovery process is complete.

It should be noted that a bit error in the parity log could potentially break the entire log chain, which would not be the case for traditional continuous data protection (CDP), which keeps all data blocks. There are two possible solutions to this: adding an error correcting code to each parity block or mirroring the entire parity log. Fortunately, *TRAP* uses orders of magnitude less storage, as will be evidenced in Section 6. Doubling parity log is still more efficient than traditional CDP. More research is needed to study the trade-offs regarding tolerating bit errors in parity logs.

## 5 EVALUATION METHODOLOGY

This section presents the evaluation methodology that we use to quantitatively study the performance of *TRAP* as compared to other data protection technologies. Our objective here is to evaluate three main parameters: storage space efficiency, performance impacts on applications, and recovery time.

### 5.1 Experimental Setup

Using our implementation described in the last section, we install our *TRAP* on a PC serving as a storage server, shown in Fig. 4. There are four PCs that are interconnected using Intel's NetStructure 10/100/1,000 Mbps 470T switch. Two of the PCs act as clients running benchmarks. One PC acts as an application server. The hardware characteristics of the four PCs are shown in Table 1.

In order to test our *TRAP* under different applications and different software environments, we set up both Linux and Windows operating systems (OSs) in our experiments. The software environments on these PCs are listed in Table 1. We install Fedora 4 (Linux Kernel 2.6.9) on one of the PCs and Microsoft Windows XP Professional on the other PCs. On the Linux machine, the UNH iSCSI implementation [26] is installed. On the Windows machines, the Microsoft iSCSI initiator [27] is installed. Since there is no iSCSI target on Windows available to us, we have developed our own iSCSI target for Windows. After installing all the OS and iSCSI software, we install our *TRAP* module on the storage server PC inside the iSCSI targets.

On top of the *TRAP* module and the OSs, we set up three different types of databases and two types of file systems. Oracle Database 10g is installed on Windows XP Professional. Postgres Database 7.1.3 is installed on Fedora 4.

TABLE 1  
Hardware and Software Environments

PC 1, 2, &3	P4 2.8GHz/256M RAM/80G+10G Hard Disks
PC 4	P4 2.4GHz/2GB RAM/200G+10G Hard Disks
OS	Windows XP Professional SP2
	Fedora 4 (Linux Kernel 2.6.9)
Databases	Oracle 10g for Microsoft Windows (32-bit)
	Postgres 7.1.3 for Linux
	MySQL 5.0 for Microsoft Windows
iSCSI	UNH iSCSI Initiator/Target 1.6
	Microsoft iSCSI Initiator 2.0
Benchmarks	TPC-C for Oracle (Hammerora)
	TPC-C for Postgres(TPCC-UVA)
	TPC-W Java Implementation
	File system micro-benchmarks
Network	Intel NetStructure 470T Switch
	Intel PRO/1000 XT Server Adapter (NIC)

MySQL 5.0 database is set up on Windows. Ext2 and NFTS are the file systems used in our experiments. To be able to run real-world Web applications, we install a Tomcat 4.1 application server for processing Web application requests issued by benchmarks.

## 5.2 Workload Characteristics

Right workloads are important for performance studies [28]. In order to have an accurate evaluation of the *TRAP* architecture, we use real-world benchmarks. The first benchmark, TPC-C, is a well-known benchmark used to model the operational end of businesses where real-time transactions are processed [29]. TPC-C simulates the execution of a set of distributed and online transactions (OLTP) for a period of between 2 and 8 hours. It is set in the context of a wholesale supplier operating on a number of warehouses and their associated sales districts. TPC-C incorporates five types of transactions with different complexities for online and deferred execution on a database system. These transactions perform the basic operations on databases such as inserts, deletes, updates, and so on. At the block storage level, these transactions will generate reads and writes that will change data blocks on disks. For the Oracle database, we use one of the TPC-C implementations developed by the Hammerora Project [30]. We build data tables for five warehouses with 25 users issuing transactional workloads to the Oracle database following the TPC-C specification. The installation of the database including all tables takes totally 3-Gbyte storage. For the Postgres database, we use the implementation from TPCC-UVA [31]. Ten warehouses with 50 users are built on the Postgres database, taking 2-Gbyte storage space. Details regarding TPC-C workload specification can be found in [29].

Our second benchmark, TPC-W, is a transactional Web benchmark developed by the Transaction Processing Performance Council that models an online bookstore [32]. The benchmark comprises a set of operations on a Web server and a back-end database system. It simulates a typical online/e-commerce application environment. Typical operations include Web browsing, shopping, and order processing. We

TABLE 2  
File System Microbenchmarks

Benchmark	Brief Description
tar	Run 5 times randomly on ext2
gcc	Compile Postgres 7.1.2 source code on ext2
zip	Compress an image directory on ext2
Latex	Make DVI and PDF files with latex source files on ext2
cp/rm/mv	Execute basic file operations (cp, rm and mv) on ext2
Linux Install	Install Redhat 8.0 on VMWare 5.0 virtual machine
XP Install	Install Windows XP system on VMWare 5.0 virtual machine
App Install	MS Office2000 and VC++ on Windows
VC++ 6.0	Compile our <i>TRAP</i> implementation codes

use the Java TPC-W implementation of the University of Wisconsin, Madison [33], and build an experimental environment. This implementation uses Tomcat 4.1 as an application server and MySQL 5.0 as a back-end database. The configured workload includes 30 emulated browsers and 10,000 items in the ITEM TABLE.

Besides benchmarks operating on databases, we have also formulated file system microbenchmarks as listed in Table 2. The first microbenchmark, *tar*, chooses five directories randomly on the ext2 file system and creates an archive file using the *tar* command. We run the *tar* command five times. Each time before the *tar* command is run, files in the directories are randomly selected and randomly changed. Similarly, we run *zip*, *latex*, and basic file operations *cp/rm/mv* on five directories randomly chosen for five times with random file changes and operations on the directories. The actions in these commands and the file changes generate block-level write requests. Two compiler applications, *gcc* and *VC++6.0*, compile the Postgres source code and our *TRAP* implementation codes, respectively. *Linux Install*, *XP Install*, and *App Install* are actual software installations on the VMWare Workstation that allows multiple OSs to run simultaneously on a single PC. The installations include *Redhat 8.0*, *Windows XP*, *Office 2000*, and *Visual C++* for Windows.

## 6 NUMERICAL RESULTS AND DISCUSSIONS

### 6.1 Space Usage Evaluation

Our first experiment is to measure the amount of storage space required to store *TRAP* data while running benchmarks on three types of databases: Oracle, Postgres, and MySQL. We concentrate on block-level storages and consider three types of data protection technologies in our experiments. *Snapshot* stores only changed data blocks at the end of each run. Traditional CDP stores all versions of a data block as disk writes occur while running the benchmarks. *TRAP* keeps parity logs as described in Section 3. To make a fair space usage comparison, we have also performed data compression in the traditional CDP architecture. The compression is done on the entire log as opposed to individual blocks. The latter would consume more space because it cannot take advantage of access

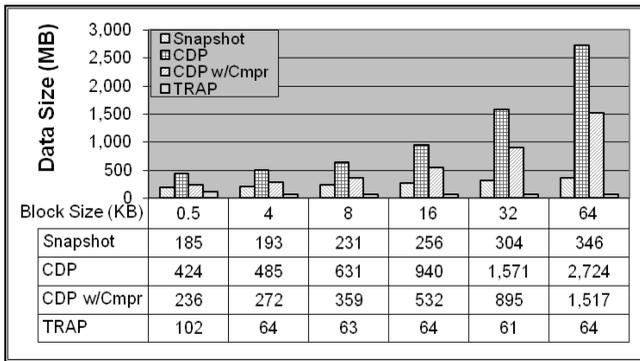


Fig. 5. Data size comparison for TPC-C on the Oracle database.

patterns among different data blocks. The compression algorithm is based on the open source library [24]. Each benchmark is run for about 1 hour on a database for a given block size. We carry out our experiments for six different parity block sizes: 512 bytes, 4 Kbytes, 8 Kbytes, 16 Kbytes, 32 Kbytes, and 64 Kbytes. Recall that the parity block size is the basic unit for parity computations. Actual data sizes of disk write requests are independent of the parity block size but are aligned with parity blocks. If a write request changes a data block that is contained in a parity block, then only one parity computation is done. If a write request changes a data block that covers more than one parity block, more parity computations have to be done. Whether or not a write data is within one parity block depends on the starting LBA and the size of the write.

Fig. 5 shows the measured results in terms of megabytes of data stored in the *TRAP* storage. There are six sets of bars corresponding to the six different block sizes. Each set contains four bars corresponding to the amount of data stored using *snapshot*, *CDP*, *CDP* with compression, and *TRAP*, respectively. It is shown in this figure that *TRAP* presents dramatic reductions in required storage space compared to other architectures. For the block size of 8 Kbytes, *TRAP* reduces the amount of data to be stored in the *TRAP* storage by an order of magnitude compared to *CDP*. For the block size of 64 Kbytes, the reduction is close to two orders of magnitude. Even with data compression being used for *CDP*, *TRAP* reduces data size by a factor of five for the block size of 8 Kbytes and a factor of 23 for the block size of 64 Kbytes, as shown in the figure.

As shown in Fig. 5, we observed in our experiments that space efficiency and performance are limited by using the block size of 512 bytes, the sector size of disks. The reason is that many write operations write large data blocks of 8 Kbytes or more. Using a 512-byte block size for parity computation, a write into an 8-Kbyte block fragments the data into at least 16 different parity groups, giving rise to more overheads and larger indexing/metadata. In the following experiments, we consider only the other five larger parity block sizes.

The results of the TPC-C benchmark on the Postgres database are shown in Fig. 6. Again, we run the TPC-C on the Postgres database for approximately 1 hour for each block size. Because Postgres was installed on a faster PC with Linux OS, the TPC-C benchmark generated more

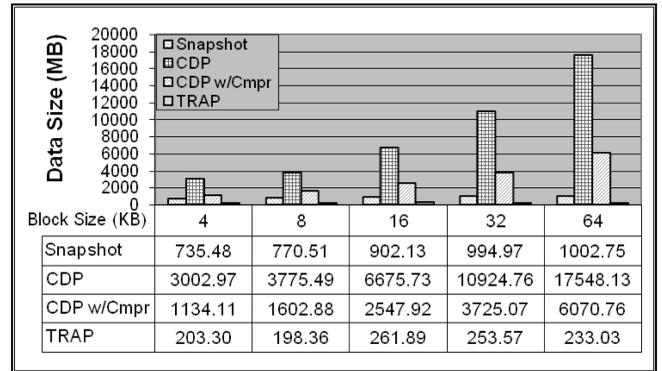


Fig. 6. Data size comparison for TPC-C on the Postgres database.

transactions on the Postgres database than on the Oracle database for the same 1-hour period. As a result, a much larger data set was written, as shown in Figs. 5 and 6. For the block size of 8 Kbytes, *CDP* needs about 3.7-Gbyte storage space to store different versions of changed data blocks in the 1-hour period. Our *TRAP*, on the other hand, needs only 0.198 Gbyte, an order of magnitude savings in storage space. If data compression is used in *CDP*, 1.6 Gbytes of data is stored, eight times more than *TRAP*. The savings are even greater for larger data block sizes. For example, for the block size of 64 Kbytes, the *TRAP* storage needs 0.23-Gbyte storage, while *CDP* requires 17.5-Gbyte storage, close to a two-order-of-magnitude improvement. Even with data compression, *TRAP* is 26 times more efficient than *CDP*. Notice that larger block sizes reduce index and metadata sizes for the same amount of data, implying another important advantage of *TRAP* since the space required by *TRAP* is not very sensitive to block sizes, as shown in the figure.

Fig. 7 shows the measured results for the TPC-W benchmark running on the MySQL database using Tomcat as the application server. We observed similar data reduction by *TRAP* as compared to *CDP*. For example, for the block size of 8 Kbytes, *TRAP* stores about 6.5 Mbytes of data in the *TRAP* storage during the benchmark run, whereas traditional *CDP* keeps 54 Mbytes of data for the same time period. If the block size is increased to 64 Kbytes, the amounts of data are about 6 Mbytes and 179 Mbytes for *TRAP* and traditional *CDP*, respectively.

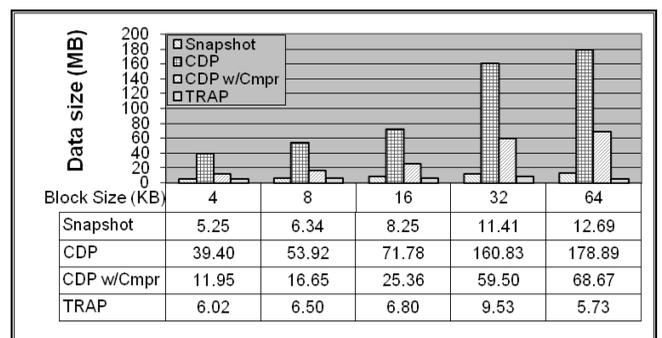


Fig. 7. Data size comparison for TPC-W on the MySQL database.

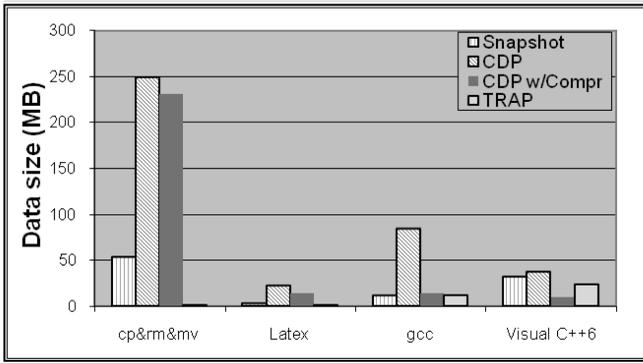


Fig. 8. Data size comparison for microbenchmarks blocksize = 8 Kbytes.

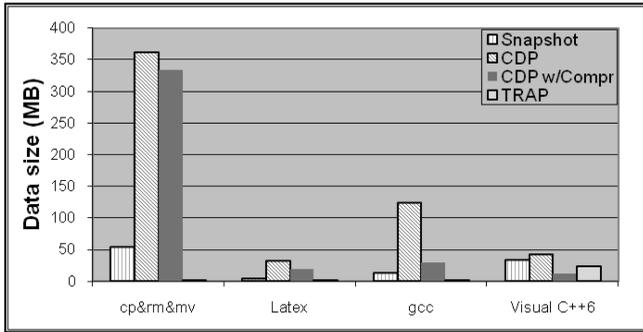


Fig. 9. Data size comparison for microbenchmarks blocksize = 16 Kbytes.

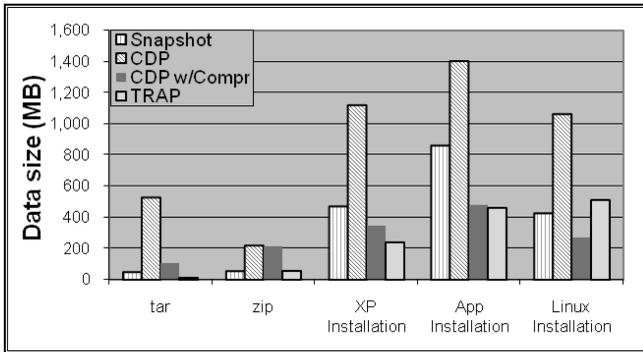


Fig. 10. Data size comparison for microbenchmarks blocksize = 8 Kbytes.

Results for file system benchmarks are shown in Figs. 8, 9, 10, and 11. Nine microbenchmarks are run for two different block sizes: 8 Kbytes and 16 Kbytes. The space savings of *TRAP* over other architectures vary from one application to another with two exceptions of Visual C++6 and Linux Installation. We observed the largest gain for *cp/rm/mv* commands and the smallest for Visual C++6. The largest gain goes up to two orders of magnitude, while the smallest gain is about 60 percent. In general, Unix file system operations demonstrate better content locality. Our analysis of Microsoft file changes indicates that some file changes result in a bitwise shift at the block level. Therefore, XOR operations at the block level are not able to catch the content locality. The data reduction ratios of all microbenchmarks are shown in Fig. 12 in logarithmic scale. As shown in the figure, the ratio varies between 1.6 and 256 times.

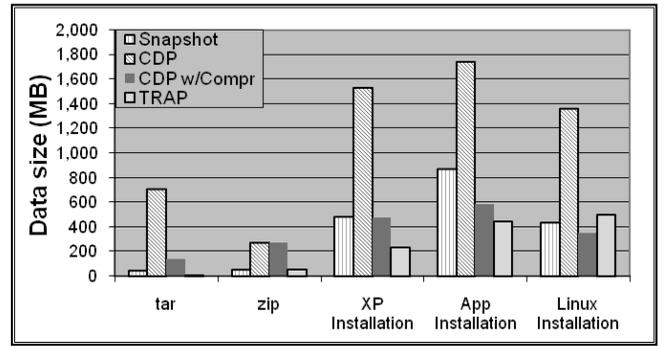


Fig. 11. Data size comparison for microbenchmarks blocksize = 16 Kbytes.

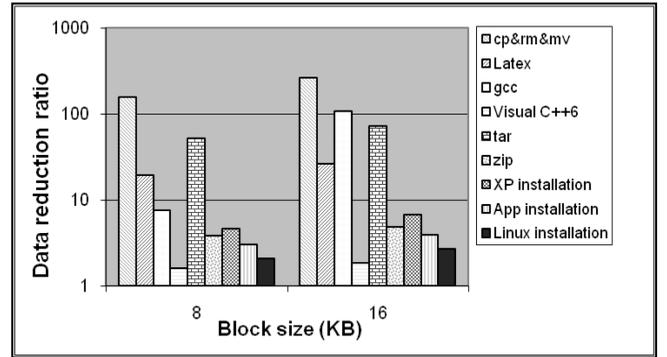


Fig. 12. Data reduction ratio of TRAP over CDP for microbenchmarks.

TABLE 3  
Measured Computation Time for XOR and Decoding Process in the TRAP Implementation on PC1

Block Size(KB)	XOR(ms)	Decode(ms)
4	0.026414	0.073972
8	0.053807	0.132586
16	0.105502	0.213022
32	0.214943	0.335425
64	0.421863	0.603595

The average gain for the 8-Kbyte block size is 28 times, and the average gain for the 16-Kbyte block size is 44 times.

## 6.2 Performance Impact

Computing and logging parities in the *TRAP* architecture may introduce additional overhead in online storages. Such overhead may negatively impact application performance. In order to quantify such impacts, we have measured the additional computation time. Table 3 lists the measured computation time of our implementation. It can be seen that the XOR computation takes tens to hundreds of microseconds using the software running on the PC. Similarly, the encoding/decoding times are in the same range. Compared to the disk I/O time, these times should be tolerable in most applications.

In order to see quantitatively how much impact such additional computation time has on real applications, we have also measured the TPC-C throughputs while running TPC-C on the Oracle and Postgres databases with two storage systems. One storage system has *TRAP* installed,

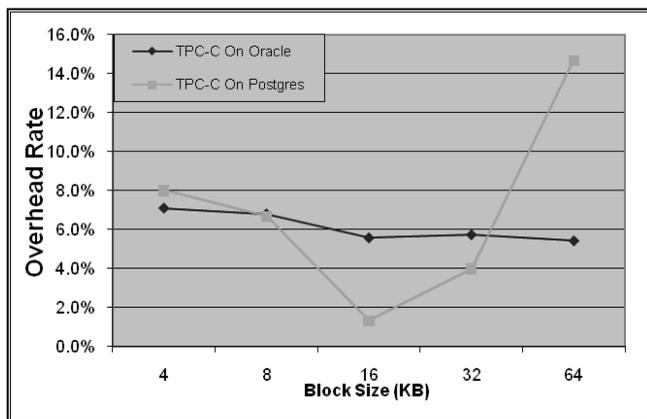


Fig. 13. Overhead of TRAP-4.

and the other has no *TRAP* installed. We then compare the two measured throughputs and calculate the overhead rate. The overhead rate is the ratio of the two measured throughputs minus one. This overhead rate is a measure of the slowdown of *TRAP*. Fig. 13 plots the overhead rates for different block sizes. Most of the overhead rates are less than 8 percent with one exception of 64 Kbytes on the Postgres database. The lowest overhead is less than 2 percent for the block size of 16 Kbytes.

It should be noted that our implementation does not assume a RAID controller. All the parity computations are done using software and considered extra overhead. As mentioned previously, *TRAP* can leverage the parity computation of RAID controllers. Therefore, if *TRAP* was implemented inside a RAID array, the overheads would be much lower.

### 6.3 Recovery Time Evaluation

The recovery of data in the real world is measured by two key parameters: recovery point objective (RPO) and recovery time objective (RTO) [13], [8]. RPO measures the maximum acceptable age of data at the time of outage. For example, if an outage occurs at time  $t_0$  and the system found such an outage at time  $t_1$ , the ideal case is to recover data as it was right before  $t_0$  or as close to  $t_0$  as possible. A daily incremental backup would represent an RPO of approximately 24 hours because the worst-case scenario would be an outage during the backup, i.e.,  $t_0$  is the time point when a backup has just started. RTO is the maximum acceptable length of time to resume normal data processing operations after an outage. RTO represents how long it takes to recover data. For the above example, if we successfully recover data at time  $t_2$  after starting the recovery process at  $t_1$ , then the RTO is  $t_2 - t_1$ .

Using our recovery program, we carry out experiments to recover data to different time points in the past. For a given block size, we first run the TPC-C benchmark on the Oracle database installed on *TRAP* for a sufficiently long time. As a result of the benchmark run, the *TRAP* storage was filled with parity logs. We then perform recoveries for each chosen time point in the past. Because of the time limit, all our parity logs and data are on disks without tape storage involved. We have made 30 recovery attempts and all of them have been able to recover correctly within the

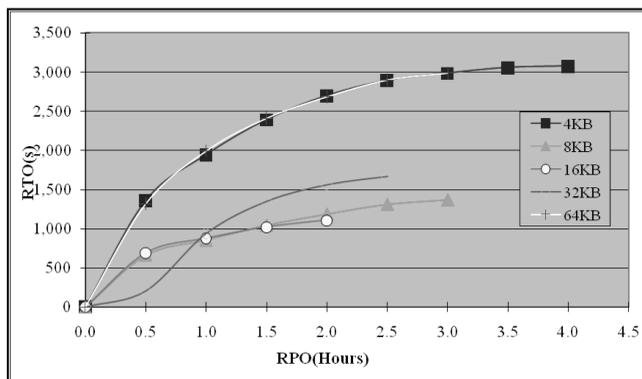


Fig. 14. The relationship between RPO and RTO.

first consistency check. Fig. 14 shows the RTO as a function of RPO for the five different block sizes. Note that our recovery process is actually an undo process using (6) as opposed to (5), which represents a redo process. An undo process starts with the newest data and traces back the parity logs, while a redo process starts with a previous data image and traces forward the parity logs. With the undo process, RTO increases as RPO increases because the farther we trace back in the parity logs, the longer time it takes to recover data. The results would be just the opposite if we were to recover data using (5). Depending on the types of outages and failure conditions, one can choose to use either process to recover data. For example, if the primary storage is damaged without newest data available, we have to recover data using a previous backup together with parity logs using (5). On the other hand, if a user accidentally performed a wrong transaction, an undo process could be used to recover data using (6).

Whether we do an undo recovery using (6) or a redo recovery using (5), RTO depends on the amount of parity data traversed during the recovery process. To illustrate this further, we plot RTO as a function of parity log sizes traversed while doing recovery, as shown in Fig. 15. The recovery time varies between a few seconds to about 1 hour for the data sizes considered. In comparison to traditional CDPs that ideally have the same RTO for different RPOs, this variable recovery time is disadvantageous. Fortunately, the smaller storage space required by *TRAP* may compensate to some extent. It should be noted that the amount of storage for the traditional CDP architecture is more than 10 Gbytes, corresponding to the parity size of 300 Mbytes. Fig. 15 can be used as a guide to users for choosing a shorter RTO recovery process, depending on the RPO, the parity log size, and the availability of newest data or a previous backup.

During our recovery experiments, we observed that block sizes of 8 Kbytes and 16 Kbytes give the shortest recovery time, as shown in Figs. 14 and 15. This result can be mainly attributed to the fact that most disk writes in our experiments fall into these block sizes. As a result, write sizes match well with parity block sizes. If the block size for parity computation were too large or too small, we would have to perform more parity computations and disk I/Os than necessary, resulting in a longer recovery time and higher overhead, as will be discussed shortly.

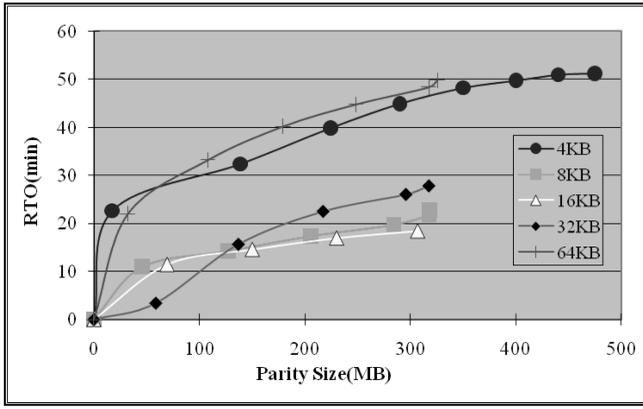


Fig. 15. RTO versus parity sizes.

As shown in (5) or (6), *TRAP* needs additional XOR computation for recovering data compared to traditional CDP. In order to quantify the impacts of the XOR computation on recovery time, we plot the proportion of time that the XOR computation takes in the total recovery time for different block sizes. As shown in Fig. 16, the XOR computation time is a small fraction of the recovery time for data recovery. It ranges from a subpercentage to the maximal of less than 3 percent. This result is expected because a large proportion of time is spent on I/O operations for retrieving parity logs.

Let us now compare the recovery time, RTO, of our *TRAP* with that of traditional CDP using the measured XOR and decoding times of *TRAP*, as shown in Table 3. Since we have only implemented the recovery program for *TRAP* but not for traditional CDP, we will carry out the following simplified analysis just to approximately compare the two. Suppose that traditional CDP reads the index node first to find out the exact location of the data block with a given time point for each changed data block. Next, the data block is read out from the CDP storage to a temporary storage. If we have a total of  $N_B$  changed data blocks, the data retrieval time for traditional CDP to recover data is approximately given by

$$(\text{inode.size}/\text{IO\_Rate} + \text{Block.size}/\text{IO\_Rate} + 2S + 2R)N_B,$$

where  $S$  and  $R$  are the average seek time and rotation latency of the hard drive, respectively. To recover data, *TRAP* needs not only to retrieve the parity log for each data

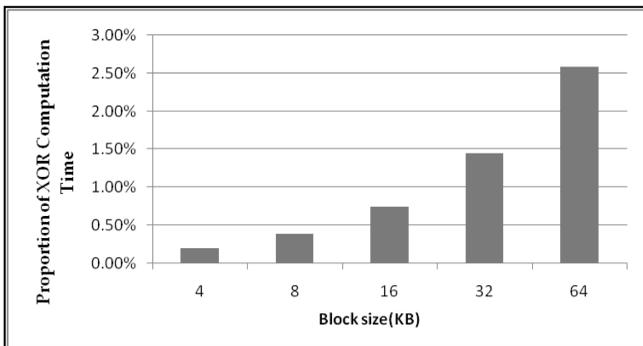
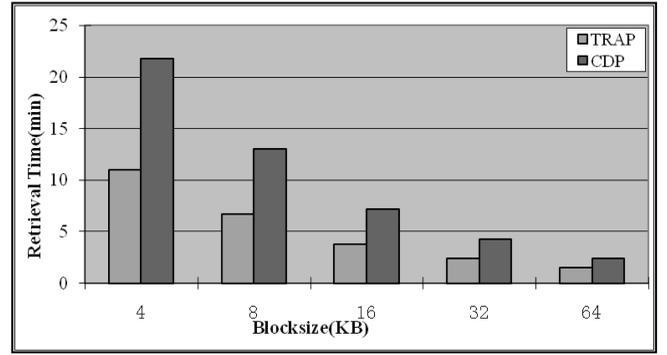


Fig. 16. Time comparison of XOR computation.

Fig. 17. Retrieval time comparison for recovery between CDP and *TRAP*.

block but also to decode the parity and compute XORs. Let  $T_{\text{DEC}}$  and  $T_{\text{XOR}}$  denote the decoding time and XOR time. The data retrieval time for *TRAP* to recover data is approximately given by

$$(T_{\text{DEC}} + T_{\text{XOR}} + \text{Avg.log.size}/\text{IO.Rate} + S + R)N_B,$$

where *Avg\_log\_size* is the average parity log size for each data block. Our experiments show that the average log size is 38 Kbytes. Therefore, an entire log is read from the *TRAP* disk every time when we try to recover one block of data. It is important to note that the data log sizes of traditional CDP are generally too large to be read in one disk operation. That is why it needs two disk operations, one for reading the I-node (header) of the corresponding log and the other for the data block pointed by the I-node. Using the above two formulas, we plot the data retrieval time of the *TRAP* and traditional CDP architectures, as shown in Fig. 17, assuming the average seek time to be 9 ms, the average rotation latency to be 4.15 ms, and the *IO\_Rate* to be 45 Mbytes/s. Note that the time it takes to do a consistency check and a write in place should be the same for both systems. As shown in the figure, *TRAP* generally takes a shorter time to retrieve data from the *TRAP* storage even though additional computations are necessary for decoding and XOR. However, the actual recovery time depends on the real implementation of each recovery algorithm and many other factors such as caching effect and indexing structure.

## 7 RELATED WORK

Depending on the different values of RPO and RTO, there exist different storage architectures capable of recovering data upon an outage. We summarize existing related works in three different categories based on different RPOs.

**Snapshot or incremental backup.** Data protection and recovery have traditionally been done using periodical backups [10], [12] and snapshots [11]. Typically, backups are done nightly when data storage is not being used since the process is time consuming and degrades application performance. During the backup process, user data are transferred to a tape, a virtual tape, or a disk for disk-to-disk backup [10], [34]. To save backup storage, most organizations perform full backups weekly or monthly with daily incremental backups in between. Data compression is often used to reduce the backup storage space [12], [35]. A good survey of various backup techniques can be found in [12].

Snapshot is a functionality that resides in most modern disk arrays [36], [37], [38], file systems [34], [39], [40], [41], [42], [43], [44], [45], volume managers [46], [47], NAS filers (network attached storages) [48], [49], [50], and backup software. A snapshot is a point-in-time image of a collection of data allowing online backup. A full-copy snapshot creates a copy of the entire data as a read-only snapshot storage clone. To save space, the COW snapshot copies a data block from the primary storage to the snapshot storage upon the first write to the block after the snapshot was created [47]. A snapshot can also redirect all writes to the snapshot storage [11], [49] after the snapshot was created. Typically, snapshots can be created up to a half dozen a day [46] without significantly impacting application performance.

**File versioning.** Besides periodical data backups, data can also be protected at the file system level using file versioning, which records a history of changes to files. Versioning was implemented by some early file systems such as the Cedar File System [41], 3DFS [51], and CVS [52] to list a few. Typically, users need to create versions manually in these systems. There are also COW versioning systems exemplified by Topsy-20 [53] and VMS [54] that have automatic versions for some file operations. Elephant [45] transparently creates a new version of a file on the first write to an open file. CVFS [55] versions each individual write or small metadata using highly efficient data structures. OceanStore [56] uses versioning not only for data recovery but also for simplifying many issues with caching and replications. The LBFS [57] file system exploits similarities between files and versions of the same files to save network bandwidth for a file system on low-bandwidth networks. Peterson and Burns have recently implemented the ext3cow file system that brings snapshot and file versioning to the open source community [40]. Other programs such as *rsync*, *rdiff*, and *diff* also provide versioning of files. To improve efficiency, flexibility, and portability of file versioning, Muniswamy-Reddy et al. [58] presented a lightweight user-oriented versioning file system called *Versionfs* that supports various storage policies configured by users.

File versioning provides a time-shifting file system that allows a system to recover to a previous version of files. These versioning file systems have controllable RTO and RPO. But they are generally file system dependent and may not be directly applicable to enterprise data centers that use different file systems and databases.

**Traditional CDP.** To provide timely recovery to any point in time at the block level, one can keep a log of changed data for each data block in a time sequence [13], [16], [59]. In the storage industry, this type of storage is usually referred to as the CDP storage. Laden et al. proposed four alternative architectures for CDP in a storage controller and compared them analytically with respect to both write performance and space usage overhead [60]. Zhu and Chiueh proposed a user-level CDP architecture that is both efficient and portable [61]. They implemented four variants of this CDP architecture for NFS servers and compared their performance characteristics. Lu et al. presented an iSCSI storage system named Mariner to provide comprehensive CDP on commodity ATA disk and Gigabit Ethernet technologies [62]. The main drawback of the CDP storage is the huge amount of storage space required,

which has thus far prevented it from being widely adopted. There have been research efforts attempting to reduce the storage space requirement for traditional CDP. Flouris and Bilas [63] proposed a storage architecture named Clotho providing transparent data versioning at the block level. Clotho coalesces the updates that take place within a period of time by creating new versions for them. This versioning happens at discrete time points, not necessarily continuous as done in CDP. What is interesting in their work is that they observed storage space savings by binary differential compression to store only the delta data that has been modified since the last version. Morrey III and Grunwald [16] observed that for some workloads, a large fraction of disk sectors to be written contain identical content to previously written sectors within or across volumes. By maintaining information (128-bit content summary hash) about the contents of individual sectors, duplicate writes are avoided. Zhu et al. [35] proposed an efficient storage architecture that identifies previously stored data segments to conserve storage space. A recent study [64] described three techniques to avoid the disk bottleneck in the Data Domain deduplication file system. These data reduction techniques generally require a search in the storage for an identical data block before a write is performed. Such a search operation is generally time consuming, although a smart search algorithm and intelligent cache designs can help in speeding up the process [16], [35]. These data reduction techniques are more appropriate for periodic backups or replications where timing is not as much a critical concern as the timing of online storage operations.

It should be noted that keeping a log of changed data has been studied and used in other contexts other than data recovery. For example, the log-structured file system has been researched extensively for improving disk write performance [65], [66]. There are variations of such log-structured file system, such as DCD [28] proposed by Hu and Yang and Vagabond [67] proposed by Norvag and Bratbergsengen for optimizing disk write performance. Norvag and Bratbergsengen also noticed the space savings of storing a delta object in the buffer space when an object is changed, which suggests that data locality exists in write operations.

## 8 CONCLUSIONS

We have presented a novel disk array architecture capable of providing Timely Recovery to Any Point in time for CDP, referred to as the *TRAP* architecture. A prototype of the new *TRAP* architecture has been implemented as a block-level device driver. File systems such as ext2 and NTFS and databases such as Oracle, Postgres, and MySQL have been installed on the prototype implementation. Real-world benchmarks including TPC-C, TPC-W, and file system benchmarks are used to test the performance of the new storage architecture. Extensive experiments have demonstrated improvements of up to two orders of magnitude in terms of storage efficiency. In addition, we have given theoretical proofs for the one-way recovery of traditional snapshots and incremental backups compared to the two-way recovery of the *TRAP* architecture. Recovery experiments have also been carried out several dozen times

to show the quick recovery time of the new architecture. Measurements have also shown that the new architecture has little negative performance impact on application performance while providing CDP capability.

## ACKNOWLEDGMENTS

This research is sponsored in part by the US National Science Foundation (NSF) under Grant CCR-0312613, SGER Grants 0610538, and CCF-0811333. This work is also supported in part by the National Natural Science Foundation of China under Grant NSFC-60736013. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF. The authors would like to thank the anonymous reviewers for the valuable comments that improved the quality of this paper.

## REFERENCES

- [1] D.A. Patterson, G. Gibson, and R.H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. ACM SIGMOD '88*, pp. 109-116, 1988.
- [2] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures," *Proc. 21st Ann. Int'l Symp. Computer Architecture (ISCA)*, 1994.
- [3] G.A. Alvarez, W.A. Burkhard, and F. Christian, "Tolerating Multiple Failures in RAID Architectures with Optimal Storage and Uniform Declustering," *Proc. 24th Ann. Int'l Symp. Computer Architecture (ISCA)*, 1997.
- [4] C.I. Park, "Efficient Placement of Parity and Data to Tolerate Two Disk Failures in Disk Arrays Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, pp. 1177-1184, Nov. 1995.
- [5] P. Corbett, B. English, A. Goel, T. Gracanac, S. Kleiman, J. Leong, and S. Sankar, "Row-Diagonal Parity for Double Disk Failure Correction," *Proc. Third Usenix Conf. File and Storage Technologies (FAST '04)*, Mar. 2004.
- [6] D.M. Smith, "The Cost of Lost Data," *J. Contemporary Business Practice*, vol. 6, no. 3, 2003.
- [7] D. Patterson et al., "Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies," Computer Science Technical Report UCB/CSD-0201175, Univ. of California, Berkeley, Mar. 2002.
- [8] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes, "Designing for Disasters," *Proc. Third Usenix Conf. File and Storage Technologies (FAST)*, 2004.
- [9] D. Patterson, "A New Focus for a New Century: Availability and Maintainability >> Performance," *FAST Keynote*, <http://www.cs.berkeley.edu/~patterson/talks/keynote.html>, Jan. 2002.
- [10] M. Rock and P. Poresky, "Shorten Your Backup Window," *Storage*, special issue on managing the information that drives the enterprise, pp. 28-34, Sept. 2005.
- [11] G. Duzy, "Match Snaps to Apps," *Storage*, special issue on managing the information that drives the enterprise, pp. 46-52, Sept. 2005.
- [12] A.L. Chervenak, V. Vellanki, and Z. Kurmas, "Protecting File Systems: A Survey of Backup Techniques," *Proc. Joint NASA and IEEE Mass Storage Conf.*, Mar. 1998.
- [13] J. Damoulakis, "Continuous Protection," *Storage*, vol. 3, no. 4, pp. 33-39, June 2004.
- [14] The 451 Group, *Total Recall: Challenges and Opportunities for the Data Protection Industry*, [http://www.the451group.com/reports/executive\\_summary.php?id=218](http://www.the451group.com/reports/executive_summary.php?id=218), May 2006.
- [15] Q. Yang, W. Xiao, and J. Ren, "TRAP-Array: A Disk Array Architecture Providing Timely Recovery to Any Point-in-Time," *Proc. 33rd Int'l Symp. Computer Architecture (ISCA '06)*, June 2006.
- [16] C.B. Morrey III and D. Grunwald, "Peabody: The Time Traveling Disk," *Proc. 11th NASA Goddard/20th IEEE Conf. Mass Storage Systems and Technologies (MSST '03)*, Apr. 2003.
- [17] B. O'Neill, "Any-Point-in-Time Backups," *Storage*, special issue on managing the information that drives the enterprise, Sept. 2005.
- [18] J. Gray, *Turing Lectures*, <http://research.microsoft.com/~gray>, 2008.
- [19] H. Simitci, *Storage Network Performance Analysis*. Wiley, 2003.
- [20] J.P. Tremblay and R. Manohar, *Discrete Mathematical Structures with Applications to Computer Science*. McGraw-Hill, 1975.
- [21] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson, "RAID: High-Performance, Reliable Secondary Storage," *ACM Computing Surveys*, June 1994.
- [22] HP Corporation, *Miscellaneous RAID-5 Operations*, <http://www.docs.hp.com/en/B7961-90018/ch08s12.html>, 2001.
- [23] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner, *iSCSI Draft Standard*, <http://www.ietf.org/internet-drafts/draftietf-ips-iscsi-20.txt>, Jan. 2003.
- [24] G. Roelofs and J.L. Gailly, *zlib Library*, <http://www.zlib.net>, 2005.
- [25] B. Furht, J. Greenberg, and R. Westwater, *Motion Estimation Algorithms for Video Compression*. Springer, 1996.
- [26] UNH, *iSCSI Reference Implementation*, <http://unh-iscsi.sourceforge.net>, 2005.
- [27] Microsoft Corporation, *Microsoft iSCSI Software Initiator Version 2.0*, <http://www.microsoft.com/windowsserver/system/storage/default.mspix>, 2005.
- [28] Y. Hu and Q. Yang, "DCD-Disk Caching Disk: A New Approach for Boosting I/O Performance," *Proc. 23rd Ann. Int'l Symp. Computer Architecture (ISCA '96)*, May 1996.
- [29] Transaction Processing Performance Council, *TPC BenchmarkTM C Standard Specification*, <http://tpc.org/tpcc>, 2005.
- [30] S. Shaw, *Hammerora: Load Testing Oracle Databases with Open Source Tools*, <http://hammerora.sourceforge.net>, 2004.
- [31] J. Piernas, T. Cortes, and J.M. García, *TPCC-UVA: A Free, Open-Source Implementation of the TPC-C Benchmark*, <http://www.infor.uva.es/~diego/tpcc-uva.html>, 2005.
- [32] H.W. Cain, R. Rajwar, M. Marden, and M.H. Lipasti, "An Architectural Evaluation of Java TPC-W," *Proc. Seventh Int'l Symp. High-Performance Computer Architecture (HPCA '01)*, Jan. 2001.
- [33] M.H. Lipasti, *Java TPC-W Implementation Distribution*, <http://www.ece.wisc.edu/~pharm/tpcw.shtml>, 2003.
- [34] L.P. Cox, C.D. Murray, and B.D. Noble, "Pastiche: Making Backup Cheap and Easy," *Proc. Fifth Usenix Symp. Operating System Design and Implementation (OSDI '02)*, Dec. 2002.
- [35] M.B. Zhu, K. Li, and R.H. Patterson, *Efficient Data Storage System*, US patent 6928526.
- [36] E.K. Lee and C.A. Thekkath, "Petal: Distributed Virtual Disks," *Proc. ACM Seventh Int'l Conf. Architecture Support for Programming Languages an Operating Systems (ASPLOS-7)*, 1996.
- [37] EMC Corporation, *EMC TimeFinder Product Description Guide*, [http://www.emc.com/products/product\\_pdfs/timefinder\\_pdg.pdf](http://www.emc.com/products/product_pdfs/timefinder_pdg.pdf), 1998.
- [38] Hitachi Ltd., *Hitachi ShadowImage Implementation Service*, [http://www.hds.com/pdf\\_143\\_implem\\_shadowimage.pdf](http://www.hds.com/pdf_143_implem_shadowimage.pdf), June 2001.
- [39] J.J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," *Proc. 13th ACM Symp. Operating System Principles (SOSP '91)*, Oct. 1991.
- [40] Z. Peterson and R.C. Burns, "Ext3cow: A Time-Shifting File System for Regulatory Compliance," *ACM Trans. Storage*, vol. 1, no. 2, pp. 190-212, 2005.
- [41] D.K. Gifford, R.M. Needham, and M.D. Schroeder, "Cedar File System," *Comm. ACM*, vol. 31, no. 3, pp. 288-298, Mar. 1988.
- [42] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West, "Scale and Performance in a Distributed File System," *ACM Trans. Computer Systems*, vol. 6, no. 1, pp. 51-81, Feb. 1988.
- [43] N.C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley, "Logical versus Physical File System Backup," *Proc. Third Usenix Symp. Operating System Design and Implementation (OSDI '99)*, pp. 239-250, Feb. 1999.
- [44] S. Quinlan and S. Dorward, "Venti: A New Approach to Archival Storage," *Proc. Usenix Conf. File and Storage Technologies (FAST '02)*, pp. 89-101, Jan. 2002.
- [45] D.S. Santry, M.J. Feeley, N.C. Hutchinson, A.C. Veitch, R.W. Carton, and J. Ofir, "Deciding When to Forget in the Elephant File System," *Proc. 17th ACM Symp. Operating System Principles (SOSP '99)*, pp. 110-123, Dec. 1999.
- [46] A. Sankaran, K. Guinn, and D. Nguyen, *Volume Shadow Copy Service*, <http://www.microsoft.com>, Mar. 2004.

- [47] A.J. Lewis, J. Thormer, and P. Caulfield, *LVM How-To*, <http://www.tldp.org/HOWTO/LVM-HOWTO.html>, 2006.
- [48] D. Hitz, J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," *Proc. Usenix Winter Technical Conf.*, pp. 235-245, 1994.
- [49] W. Xiao, Y. Liu, Q. Yang, J. Ren, and C. Xie, "Implementation and Performance Evaluation of Two Snapshot Methods on iSCSI Target Storages," *Proc. 14th NASA Goddard/23rd IEEE Conf. Mass Storage Systems and Technologies (MSST '06)*, May 2006.
- [50] G.A. Gibson and R.V. Meter, "Network Attached Storage Architecture," *Comm. ACM*, vol. 43, no. 11, pp. 37-45, Nov. 2000.
- [51] D.G. Korn and E. Krell, "The 3-D File System," *Proc. Usenix Summer Conf.*, pp. 147-156, 1989.
- [52] B. Berliner and J. Polk, *Concurrent Versions System (CVS)*, <http://www.cvshome.org>, 2001.
- [53] L. Moses, "An Introductory Guide to TOPS-20," Technical Report TM-82-22, Information Sciences Inst., Univ. of Southern California, 1982.
- [54] K. McCoy, *VMS File System Internals*. Digital Press, 1990.
- [55] C.A.N. Soules, G.R. Goodson, J.D. Strunk, and G.R. Ganger, "Metadata Efficiency in Versioning File Systems," *Proc. Second USENIX Conf. File and Storage Technologies (FAST '03)*, pp. 43-58, Mar. 2003.
- [56] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: The OceanStore Prototype," *Proc. Second Usenix Conf. File and Storage Technologies (FAST '03)*, Mar. 2003.
- [57] A. Muthitacharoen, B. Chen, and D. Mazières, "A Low-Bandwidth Network File System," *Proc. 18th ACM Symp. Operating Systems Principles (SOSP '01)*, Oct. 2001.
- [58] K. Muniswamy-Reddy, C.P. Wright, A. Himmer, and E. Zadok, "A Versatile and User-Oriented Versioning File System," *Proc. Third Usenix Conf. File and Storage Technologies (FAST)*, 2004.
- [59] J. Damoulakis, "Time to Say Goodbye to Backup," *Storage*, vol. 4, no. 9, pp. 64-66, Nov. 2006.
- [60] G. Laden, P. Ta-shma, E. Yaffe, and M. Factor, "Architectures for Controller Based CDP," *Proc. Fifth Usenix Conf. File and Storage Technologies (FAST '07)*, Feb. 2007.
- [61] N. Zhu and T. Chiueh, "Portable and Efficient Continuous Data Protection for Network File Servers," *Proc. 37th Ann. IEEE/IFIP Int'l Conf. Dependable Systems and Networks (DSN '07)*, June 2007.
- [62] M. Lu, S. Lin, and T. Chiueh, "Efficient Logging and Replication Techniques for Comprehensive Data Protection," *Proc. 24th IEEE Conf. Mass Storage Systems and Technologies (MSST)*, Sept. 2007.
- [63] M.D. Flouris and A. Bilas, "Clotho: Transparent Data Versioning at the Block I/O Level," *Proc. 12th NASA Goddard/21st IEEE Conf. Mass Storage Systems and Technologies (MSST '04)*, Apr. 2004.
- [64] B. Zhu and K. Li, "Avoiding the Disk Bottleneck in the Data Domain Deduplication File System," *Proc. Sixth Usenix Conf. File and Storage Technologies (FAST)*, 2008.
- [65] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Computer Systems*, pp. 26-52, Feb. 1992.
- [66] M. Seltzer, K. Bostic, M.K. McKusick, and C. Staelin, "An Implementation of a Log-Structured File System for UNIX," *Proc. Winter Usenix Technical Conf.*, pp. 307-326, Jan. 1993.
- [67] K. Norvag and K. Bratbergsengen, "Log-Only Temporal Object Storage," *Proc. Eighth Int'l Conf. Database and Expert Systems Applications (DEXA '97)*, Sept. 1997.



student member of the IEEE and the IEEE Computer Society.

**Weijun Xiao** received the bachelor's and master's degrees in computer science from Huazhong University of Science and Technology, China, in 1995 and 1998, respectively. He is currently a PhD candidate in the Department of Electrical, Computer, and Biomedical Engineering, University of Rhode Island, Kingston. His research interests include computer architecture, networked storage system, embedded system, and performance evaluation. He is a



**Jin Ren** received the bachelor's degree in material science and engineering and the master's degree in computer science from Huazhong University of Science and Technology, China, in 1999 and 2002, respectively. He is currently a PhD candidate in the Department of Electrical, Computer, and Biomedical Engineering, University of Rhode Island, Kingston. His research interests include computer architecture, networked storage system, and performance evaluation.



**Qing Yang** received the BSc degree in computer science from Huazhong University of Science and Technology, Wuhan, China, in 1982, the MSc degree in electrical engineering from the University of Toronto, Canada, in 1985, and the PhD degree in computer engineering from the Center for Advanced Computer Studies, University of Louisiana, Lafayette, in 1988. Presently, he is a distinguished engineering professor in the Department of Electrical, Computer, and Biomedical Engineering, University of Rhode Island, Kingston, where he has been a faculty member since 1988. His research interests include computer architectures, memory systems, disk I/O systems, data storages, parallel and distributed computing, performance evaluation, and local area networks. He is a senior member of the IEEE and the IEEE Computer Society and a member of the ACM SIGARCH.

► For more information on this or any other computing topic, please visit our Digital Library at [www.computer.org/publications/dlib](http://www.computer.org/publications/dlib).