In Proceedings of ISCA'06: The 33rd Annual International Symposium on Computer Architecture, Boston, USA

# TRAP-Array: A Disk Array Architecture Providing Timely Recovery to Any Point-in-time

Qing Yang, Weijun Xiao, and Jin Ren
*Dept. of Electrical and Computer Engineering*
*University of Rhode Island*
*Kingston, RI 02881*
*Email: {qyang,wjxiao,rjin}@ele.uri.edu*

## Abstract

*RAID architectures have been used for more than two decades to recover data upon disk failures. Disk failure is just one of the many causes of damaged data. Data can be damaged by virus attacks, user errors, defective software/firmware, hardware faults, and site failures. The risk of these types of data damage is far greater than disk failure with today's mature disk technology and networked information services. It has therefore become increasingly important for today's disk array to be able to recover data to any point in time when such a failure occurs. This paper presents a new disk array architecture that provides* ***T****imely* ***R****ecovery to* ***A****ny* ***P****oint-in-time, referred to as TRAP-Array. TRAP-Array stores not only the data stripe upon a write to the array, but also the time-stamped Exclusive-ORs of successive writes to each data block. By leveraging the Exclusive-OR 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 Exclusive-ORs resulting from writes. What is interesting is that TRAP architecture is amazingly space-efficient. We have implemented a prototype TRAP architecture using software at block device level and carried out extensive performance measurements using TPC-C benchmark running on Oracle and Postgress databases, TPC-W running on MySQL database, and file system benchmarks running on Linux and Windows systems. Our experiments demonstrated that TRAP is not only able to recover data to any point-in-time very quickly upon a failure but it also uses less storage space than traditional daily differential backup/snapshot. Compared to the state-of-the-art continuous data protection technologies, TRAP saves disk storage space by one to two orders of magnitude with a simple and a fast encoding algorithm. From an architecture point of view, TRAP-Array opens up another dimension for storage arrays. It is orthogonal and complementary to RAID in the sense that RAID protects data in the dimension along an array of physical disks while TRAP protects data in the dimension along the time sequence.*

## 1. Introduction

RAID architecture [1] has been the most prominent architecture advance in disk I/O systems for the past two decades. RAID1 provides *2N* 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 failures [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% [6] to 80% [7] of data losses. 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 [8,9,10]. These techniques usually take a long time to recover data [11]. In addition, data between backups are vulnerable to data loss. Recent research [12,13] has shown that data loss or data unavailability can cost up to millions of dollars per hour in many businesses. Solely depending on the traditional time-consuming backups is no longer adequate for today's information age.

This paper advocates for a disk array architecture that opens a new dimension, ***time***, to enable **T**imely **R**ecovery to **A**ny **P**oint-in-time (***TRAP***) of user data. We propose a new *TRAP* architecture that has the optimal space and performance characteristics. The idea of the new *TRAP* architecture is very simple. Instead of providing full redundancy of data in time dimension, i.e. keeping a log of all previous versions of changed data blocks in time sequence [11,14,15], we compute XORs among changed data blocks along the time dimension to improve performance and space efficiency. The resulting architecture shift is similar to the shift from RAID1 to RAID4 that improves space efficiency and performance upon RAID1. With a simple and fast encoding scheme, the new *TRAP* architecture presents dramatic space savings because of content locality that exists in real world applications. Furthermore, it provides faster data recovery to any-point-in-time than traditional techniques because of the drastically smaller amount of storage space used.

We have implemented a prototype of the new *TRAP* architecture at block device level using 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. Industry standard 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 up to 2 orders of magnitude improvements of the new *TRAP* architecture over existing technologies in terms of storage space efficiency. Such orders of magnitude improvements are practically important given the exponential growth of data [16]. We have also carried out data recovery experiments by selecting any point-in-time in the past and recovering data to the time point. Experiments have shown that all recovery attempts are successful. Recovery time of the new *TRAP* architecture is compared with existing reliable storage architectures to show that the new *TRAP* architecture can recover data to any point-in-time very quickly.

We classify different storage architectures capable of recovering data to a previous time point into 4 different categories and discuss these techniques in detail in the next section as background and related research work. The detailed design and implementation of the new *TRAP* is presented in Section 3. Section 4 presents the experimental settings and the workload characteristics. Numerical results and discussions are presented in Section 5. We conclude our paper in Section 6.

## 2. Background and Related Work

Recovery of data in the real world is measured by two key parameters: recovery point objective (RPO) and recovery time objective (RTO) [11,12]. 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 the 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 backup would represent 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 is 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 $t_2 - t_1$. Depending on the different values of RPO and RTO, there exist different storage architectures capable of recovering data upon an outage. We classify these architectures into 4 different categories as discussed below.

**TRAP-1:** Data protection and recovery have traditionally been done using periodical backups [8,10] and snapshots [9]. 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 [8, 17]. 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 backup storage space [10,18]. A good survey of various backup techniques can be found in [10]. Snapshot is a functionality that resides in most modern disk arrays [19, 20, 21], file systems [17,22,23,24,25,26,27,28], volume managers [29,30], NAS filers (network attached storages) [31,32,33] and backup software. A snapshot is a point-in-time image of a collection of data allowing on-line backup. A full-copy snapshot creates a copy of the entire data as a read only snapshot storage clone. To save space, copy-on-write 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 [30]. A snapshot can also redirect all writes to the snapshot storage [9,31] after the snapshot was created. Typically, snapshots can be created up to a half dozen a day [29] without significantly impacting application performance.

Despite the rapid advances in computer technology witnessed in the past two decades, data backup is a notable exception that is fundamentally the same as it was 20 years ago. It was well-known that backup remains a costly and highly intrusive batch operation that is prone to error and consumes an exorbitant amount of time and resources [10, 34]. As a result, RTO of backups is generally very long. Furthermore, data are vulnerable between two subsequent backups giving rise to high RPO. We categorize this type of recoverable data storages as **TRAP-1** for Time-consuming Recovery to Assigned Point-in-time.

**TRAP-2:** Besides periodical data backups, data can also be protected at file system level using file versioning that records a history of changes to files. Versioning was implemented by some early file systems such as Cedar File System [24], 3DFS [35], and CVS [36] to list a few. Typically, users need to create versions manually in these systems. There are also copy-on-write versioning systems exemplified by Tops-20 [37] and VMS [38] that have automatic versions for some file operations. Elephant [28] transparently creates a new version of a file on the first write to an open file. CVFS [39] versions each individual write or small meta-data using highly efficient data structures. OceanStore [40] uses versioning not only for data recovery but also for simplifying many issues with caching and replications. The LBFS [41] 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 [23]. 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 [42] 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. File versioning is categorized as *TRAP*-2. *TRAP*-2 differs from *TRAP*-1 in the sense that *TRAP*-2 works mainly at file system level not at block device level. Block level storages usually provide high performance and efficiency especially for applications such as databases that access raw devices.

**TRAP-3:** To provide timely recovery to any point-in-time at block device level, one can keep a log of changed data for each data block in a time sequence [11,14,34]. In the storage industry, this type of storage is usually referred to as CDP (Continuous Data Protection) storage. In this type of systems, a write operation will replace the old data in the same logic block address (LBA) to another disk storage instead of overwriting it. As a result, successive writes to the same LBA will generate a sequence of different versions of the block with associated timestamps indicating the time of the corresponding write operations. These replaced data blocks are stored in a log structure, maintaining a history of the data blocks that have been modified. Since every change on a block is kept, it is possible to view a storage volume as it existed at any point in time, dramatically reducing RPO. The RTO depends on the size of the storage for the logs, indexing structure, and consistency checks. The data image at the time of an outage is considered to be "crash consistent" at block level because the orders of all write operations are strictly preserved. Modern file systems and databases have tools to perform consistency checks and recover data that are file system/application consistent [14,43,44].

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. Typically, about 20% of active storage volumes change per day, with an average of 5 to 10 overwrites to a block. If we have one terabyte data storage, a CDP storage will require one to two terabytes of space to store the logs reflecting data changes in one day. A week of such operations will require 5 to 10 terabytes of storage space.

There have been research efforts attempting to reduce storage space requirement for *TRAP-3*. Morrey III and Grunwald [14] 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, Li, and Patterson [18] proposed an efficient storage architecture that identifies previously stored data segments to conserve storage space. 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 smart search algorithm and intelligent cache designs can help in speeding up the process [14,18]. 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.

## 3. TRAP-4 Architecture

The idea of *TRAP-4* architecture 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 [45] as a result of each write on the block. Figure 1 shows the basic design of *TRAP-4*. Suppose that at time ***T(k),*** the host writes into a data block with logic block address $A_i$ that belongs to a data stripe *($A_1$, $A_2$ … $A_i$, … $A_n$)*. The RAID controller performs the following operation to update its parity disk:

$$P_{T(k)} = A_i(k) \oplus A_i(k\text{-}1) \oplus P_{T(k\text{-}1)} \qquad (1)$$

where $P_{T(k)}$ is the new parity for the corresponding stripe, $A_i(k)$ is the new data for data block $A_i$, $A_i(k\text{-}1)$ is the old data of data block $A_i$, and $P_{T(k\text{-}1)}$ is the old parity of the stripe. Leveraging this computation, *TRAP-4* appends the first part of the above equation, i.e. $P'_{T(k)} = A_i(k) \oplus A_i(k\text{-}1)$, to the parity log stored in the *TRAP* disk after a simple encoding box, as shown in Figure 1. Our extensive experiments have demonstrated a very strong *content locality* that exists in real world applications. For the workloads that we have studied, only 5% to 20% of bits inside a data block actually change on a write operation. The parity, $P'_{T(k)}$, reflects the exact changes at 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 bit stream 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.

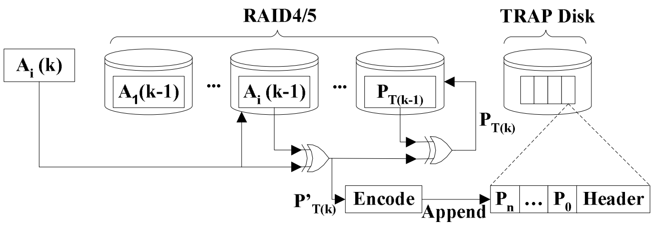


**Figure 1. Block diagram of TRAP-4 design**

Now consider the parity log corresponding to a data block, $A_i$, after a series of write operations. The log contains ($P'_{T(k)}$, $P'_{T(k-1)}$ ……, $P'_{T(2)}$, $P'_{T(1)}$) with time stamps $T(k)$, $T(k\text{-}1)$, ……, $T(2)$, and $T(1)$ associated with the parities. Suppose that an outage occurred at time $t_1$, and we would like to recover data to the image as it was at time $t_0$ ($t_0 \le t_1$). To do such a recovery, for each data block $A_i$, we first find the largest $T(r)$ in the corresponding parity log such that $T(r) \le t_0$. We then perform the following computation:

$$A_i(r) = P'_{T(r)} \oplus P'_{T(r-1)} \oplus \ldots \oplus P'_{T(1)} \oplus A_i(0), \qquad (2)$$

where $A_i(r)$ denotes the data image of $A_i$ at time $T(r)$ and $A_i(0)$ denotes the data image of $A_i$ at time $T(0)$. Note that

$$P'_{T(l)} \oplus A_i(l\text{-}1) = A_i(l) \oplus A_i(l\text{-}1) \oplus A_i(l\text{-}1) = A_i(l),$$

for all $l=1,2, \ldots r$. Therefore, Equation (2) gives $A_i(r)$ correctly assuming that the original data image, $A_i(0)$, exists.

The above process represents a typical recovery process upon an outage that results in data loss or data damage while earlier data is available in a full backup or a mirror storage. An undo process is also possible with the parity log if the newest data is available by doing the following computation instead of Equation (2):

$$A_i(r) = A_i(k) \oplus P'_{T(k)} \oplus P'_{T(k-1)} \oplus \ldots \oplus P'_{T(r+1)}, \qquad (3)$$

where $A_i(k)$ represents the latest data of block $A_i$.

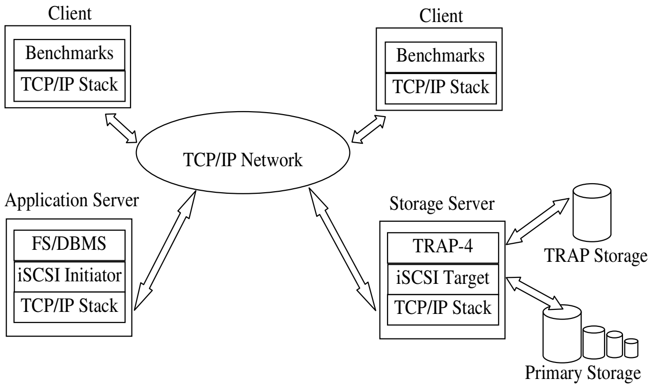


**Figure 2. System architecture of TRAP-4 implementation**

We have designed and implemented a software prototype of *TRAP-4*. 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*-4. The prototype driver takes write requests from a file system or database system at block level. Upon receiving a write request, *TRAP*-4 performs 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 Figure 2. In the iSCSI protocol, there are two communication parties, referred to as iSCSI initiator and iSCSI target [46]. An iSCSI initiator runs under the file system or database applications as a device driver. As I/O operations come from applications, the initiator generates I/O requests using SCSI commands wrapped inside TCP/IP packets that are sent to the iSCSI target. Our *TRAP*-4 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 $P'_{T(k)}$ as discussed above. Our implementation works on a configurable and fixed block size, referred to as *parity block size*. 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 [47] 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.

As shown in Figure 2, our implementation is on top of the standard TCP/IP protocol. As a result, our *TRAP-4* 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-4* can protect important data from site failures or disaster events.

We have also implemented a recovery program for our *TRAP*-4. For a given recovery time point (RPO), $\boldsymbol{t_r}$, the recovery program retrieves the parity log to find the timestamp, $\boldsymbol{T(r)}$, such that $\boldsymbol{T(r) \leq t_r}$, for every data block that have been changed. We then decode the parity blocks and compute XOR using either Equation (2) or Equation (3) to obtain the data block as it was at time $\boldsymbol{t_r}$ 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 do 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 the recovered data is in a "crash consistency" state. We are currently working on possible techniques to assist applications to quickly recover to the most recent consistent point at the application level. It should also be noted that a bit error in the parity log could potentially break the entire log chain, which would not be the case for *TRAP-3* that keeps all data blocks. There are two possible solutions to this: adding an error correcting code to each parity block or mirror the entire parity log. Fortunately, *TRAP-4* uses orders of magnitude less storage, as will be evidenced in Section 5. Doubling parity log is still more efficient than *TRAP-3*. More research is needed to study the trade-offs regarding tolerating bits errors in parity logs.

## 4. Evaluation Methodology

This section presents the evaluation methodology that we use to quantitatively study the performance of *TRAP*-4 as compared to other *TRAP* levels. Our objective here is to evaluate three main parameters: storage space efficiency, RTO and RPO, and performance impacts on applications.

### 4.1 Experimental Setup

Using our implementation described in the last section, we install our *TRAP-4* on a PC serving as a storage server, shown in Figure 2. There are four PCs that are interconnected using the Intel's NetStructure 10/100/1000Mbps 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.

| | |
|---|---|
| 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 2 (Linux Kernel 2.4.20) |
| 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) |

**Table 1. Hardware and software environments**

In order to test our *TRAP-4* under different applications and different software environments, we set up both Linux and Windows operating systems in our experiments. The software environments on these PCs are listed in Table 1. We install Fedora 2 (Linux Kernel 2.4.20) on one of the PCs and Microsoft Windows XP Professional on other PCs. On the Linux machine, the UNH iSCSI implementation [48] is installed. On the Windows machines the Microsoft iSCSI initiator [49] 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-4* module on the storage server PC inside the iSCSI targets.

On top of the *TRAP-4* module and the operating systems, 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 2. 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 Tomcat 4.1 application server for processing web application requests issued by benchmarks.

## 4.2 Workload Characteristics

Right workloads are important for performance studies [50]. In order to have an accurate evaluation of different *TRAP* levels, we use standard 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 [ 51 ]. TPC-C simulates the execution of a set of distributed and on-line transactions (OLTP) for a period of between two and eight 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 complexity 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 Oracle Database, we use one of the TPC-C implementations written by Hammerora Project [52]. We build data tables for 5 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 3GB storage. For Postgres Database, we use the implementation from TPCC-UVA [53]. 10 warehouses with 50 users are built on Postgres database taking 2GB storage space. Details regarding TPC-C workloads specification can be found in [51].

Our second benchmark, TPC-W, is a transactional web benchmark developed by Transaction Processing Performance Council that models an on-line bookstore [54]. The benchmark comprises a set of operations on a web server and a backend database system. It simulates a typical on-line/E-commerce application environment. Typical operations include web browsing, shopping, and order processing. We use the Java TPC-W implementation of University of Wisconsin-Madison [55] and build an experimental environment. This implementation uses Tomcat 4.1 as an application server and MySQL 5.0 as a backend 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 micro-benchmarks as listed in Table 2. The first micro-benchmark, *tar*, chooses five directories randomly on ext2 file system and creates an archive file using *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 5 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 V*C++6.0*, compile Postgress source code and our *TRAP* implementation codes, respectively. *Linux Install, XP Install,* and *App Install* are actual software installations on 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.

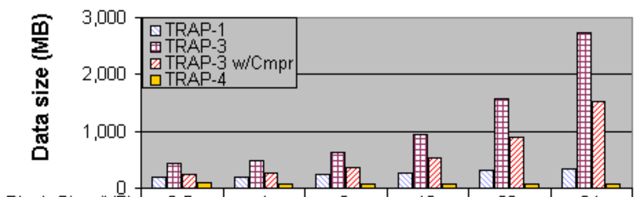
| Benchmark | Brief description |
|---|---|
| tar | Run 5 times randomly on ext2 |
| gcc | Compile Postgres 7.1.2 source code on ext2 |
| zip | Compress an image directory on ext2 |
| Latex | Make DVI and PDF files with latex source files on ext2 |
| cp/rm/mv | Execute basic file operations (cp, rm and 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 *TRAP* implementation codes |

**Table 2. File system micro benchmarks**.

## 5. Numerical Results and Discussions

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 *TRAP* architectures in our experiments. *TRAP-1* stores only changed data blocks at the end of each run. This represents a typical copy-on-write snapshot or an incremental backup that only backs up changed data blocks. *TRAP-3* stores all versions of a data block as disk writes occur while running the benchmarks. *TRAP-4* keeps parity logs as described in Section 3. To make a fair space usage comparison, we have also performed data compression in the *TRAP*-3 architecture. The compression algorithm is based on the open source library [47]. Each benchmark is run for about 1 hour on a database for a given block size. We carry out our experiments for 6 different parity block sizes: 512B, 4KB, 8KB, 16KB, 32KB, and 64KB. Recall that this 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.



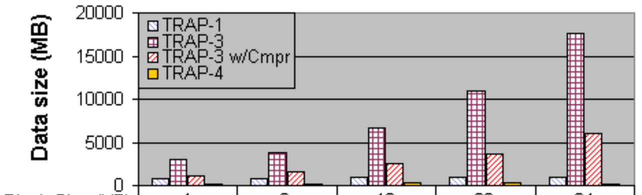
| Block Size (KB) | 0.5 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|
| TRAP-1 | 185 | 193 | 231 | 256 | 304 | 346 |
| TRAP-3 | 424 | 485 | 631 | 940 | 1,571 | 2,724 |
| TRAP-3 w/Cmpr | 236 | 272 | 359 | 532 | 895 | 1,517 |
| TRAP-4 | 102 | 64 | 63 | 64 | 61 | 64 |

**Figure 3. Data size comparison for TPC-C on Oracle database**

Figure 3 shows the measured results in terms of Mbytes 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 *TRAP-1, TRAP-3, TRAP-3* with compression, and *TRAP-4*, respectively. It is shown in this figure that *TRAP-4* presents dramatic reductions in required storage space compared to other *TRAP* architectures. For the block size of 8KB, *TRAP-4* reduces the amount of data to be stored in the *TRAP* storage by an order of magnitude compared to *TRAP-3*. For the block size of 64KB, the reduction is close to 2 orders of magnitude. Even with data compression being used for *TRAP-3*, *TRAP-4* reduces data size by a factor of 5 for the block size of 8KB and a factor of 23 for the block size of 64KB, as shown in the figure.

It is interesting to observe in Figure 3 that our *TRAP-4* uses even smaller storage space than *TRAP-1* that represents periodical backups. In this experiment, 25 users continuously generate transactions to 5 warehouses following the TPC-C specification with no thinking period. The amount of I/O requests generated with this workload in an hour is probably similar to one day's I/Os of medium size organizations. In this case, the amount of data in *TRAP-1* would be the amount of data for a daily backup. If this is the case, our *TRAP-4* uses smaller storage space than daily backup while being able to recover data to any point-in-time. That is, with less storage space than today's daily backup *TRAP-4* achieves near 0 RPO as opposed to 24 hours RPO.

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

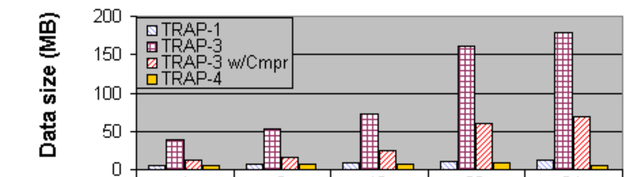


| Block Size (KB) | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|
| TRAP-1 | 735.48 | 770.51 | 902.13 | 994.97 | 1002.75 |
| TRAP-3 | 3002.97 | 3775.49 | 6675.73 | 10924.76 | 17548.13 |
| TRAP-3 w/Cmpr | 1134.11 | 1602.88 | 2547.92 | 3725.07 | 6070.76 |
| TRAP-4 | 203.30 | 198.36 | 261.89 | 253.57 | 233.03 |

**Figure 4. Data size comparison for TPC-C on Postgres database**

Results of the TPC-C benchmark on Postgres database are shown in Figure 4. Again, we run the TPC-C on 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 transactions on Postgres database than on Oracle database for the same one-hour period. As a result, much larger data set was written as shown in Figure 4 and Figure 3. For the block size of 8KB, *TRAP*-3 needs about 3.7GB storage space to store different versions of changed data blocks in the one-hour period. Our *TRAP*-4, on the other hand, needs only 0.198GB, an order of magnitude savings in storage space. If data compression is used in *TRAP*-3, 1.6GB of data is stored in the *TRAP* storage, 8 times more than *TRAP*-4. The savings are even greater for larger data block sizes. For example, for the block size of 64KB, *TRAP*-4 storage needs 0.23GB storage while *TRAP*-3 requires 17.5GB storage, close to 2 orders of magnitude improvement. Even with data compression, *TRAP*-4 is 26 times more efficient than *TRAP*-3. Notice that larger block sizes reduces index and meta data sizes for the same amount of data, implying another important advantage of *TRAP*-4 since space required by *TRAP*-4 is not very sensitive to block sizes as shown in the figure.

Figure 5 shows the measured results for TPC-W benchmark running on MySQL database using Tomcat as the application server. We observed similar data reduction by *TRAP*-4 as compared to *TRAP*-3. For example, for block size of 8KB, *TRAP*-4 stores about 6.5MB of data in the *TRAP* storage during the benchmark run whereas traditional CDP (*TRAP*-3) keeps 54MB of data in the *TRAP* storage for the same time period. If block size is increased to 64KB, the amounts of data are about 6MB and 179MB for *TRAP*-4 and traditional CDP (*TRAP*-3), respectively.



| Block Size (KB) | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|
| TRAP-1 | 5.25 | 6.34 | 8.25 | 11.41 | 12.69 |
| TRAP-3 | 39.40 | 53.92 | 71.78 | 160.83 | 178.89 |
| TRAP-3 w/Cmpr | 11.95 | 16.65 | 25.36 | 59.50 | 68.67 |
| TRAP-4 | 6.02 | 6.50 | 6.80 | 9.53 | 5.73 |

**Figure 5. Data size comparison for TPC-W on MySQL database**

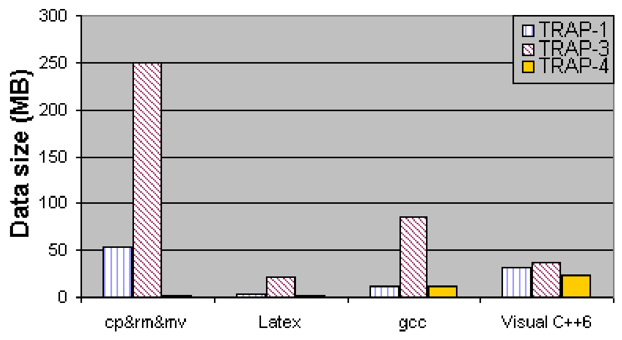


**Figure 6. Data size comparison for micro benchmarks (blocksize=8KB)**

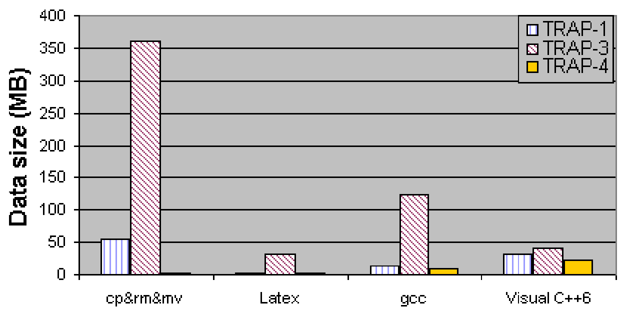


**Figure 7. Data size comparison for micro benchmarks (blocksize=16KB)**

Results for file system benchmarks are shown in Figures 6-9. Nine micro benchmarks are run for two different block sizes, 8KB and 16KB. Space savings of *TRAP-4* over other *TRAP* levels vary from one application to another. We observed largest gain for *cp/rm/mv* commands and smallest for Visual C++6. The largest gain goes up to 2 orders of magnitude while the smallest gain is about 60%. In general, Unix file system operations demonstrate better content locality. Our analysis of Microsoft file changes indicates that some file changes result in bit-wise shift at block level. Therefore, XOR operations at block level are not able to catch the content locality. The data reduction ratios of all micro benchmarks are shown in Figure 10 in logarithmic scale. As shown in the figure, the ratio varies between 1.6 and 256 times. The average gain for 8KB block size is 28 times and the average gain for 16KB block size is 44 times.

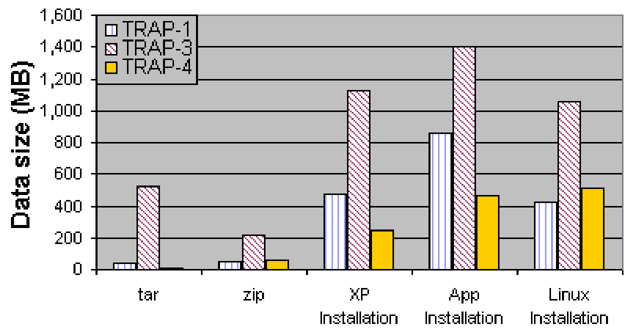


**Figure 8. Data size comparison for micro benchmarks (blocksize=8KB)**

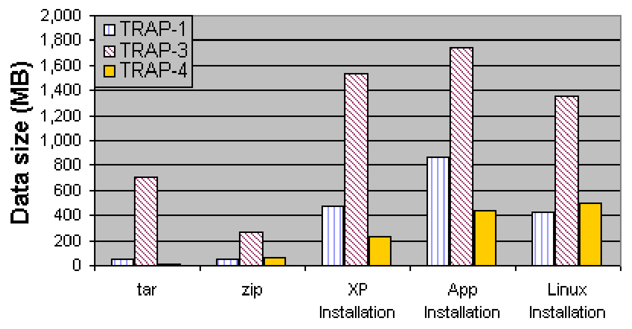


**Figure 9. Data size comparison for micro benchmarks (blocksize=16KB)**

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 Oracle database installed on *TRAP*-4 for sufficiently long time. As a result of the benchmark run, *TRAP*-4 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 with no tape storage involved. We have made 30 recovery attempts and all of them have been able to recover correctly within first consistency check. Figure 11 shows the RTO as functions of RPO for the 5 different block sizes. Note that our recovery process is actually an undo process using Equation (3) as opposed to Equation (2) that represents a redo process. An undo process starts with the newest data and traces back the parity logs while redo process starts with a previous data and traces forward the parity logs. With the undo process, the 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 Equation (2). 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 Equation (2). On the other hand, if a user accidentally performed a wrong transaction, an undo process could be used to recover data using Equation (3).

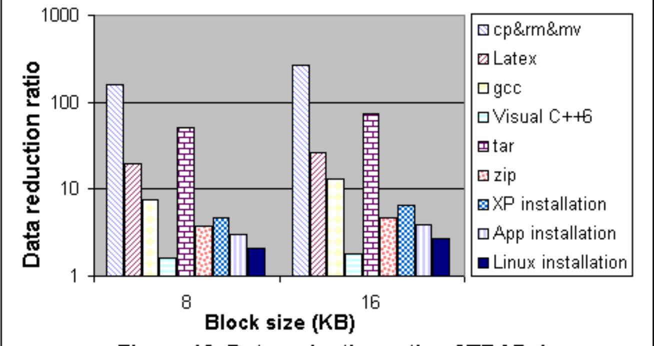


**Figure 10. Data reduction ratio of TRAP-4 over TRAP-3 for micro benchmarks**

Whether we do an undo recovery using Equation (3) or a redo recovery using Equation (2), the RTO depends on the amount of parity data traversed during the recovery process. To illustrate this further, we plot RTO as functions of parity log sizes traversed while doing recovery as shown in Figure 12. The recovery time varies between a few seconds to about 1 hour for the data sizes considered. It should be noted that the amount of storage for *TRAP*-3 architecture is over 10GB corresponding to the parity size of 300 MB. Figure 12 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.

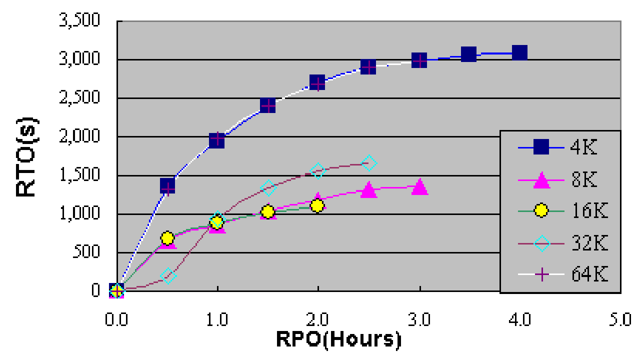


**Figure 11. The relationship between RPO and RTO**

During our recovery experiments we observed that block sizes of 8KB and 16KB give the shortest recovery time, as shown in Figures 11 and 12. 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 longer recovery time and higher overhead as will be discussed shortly.

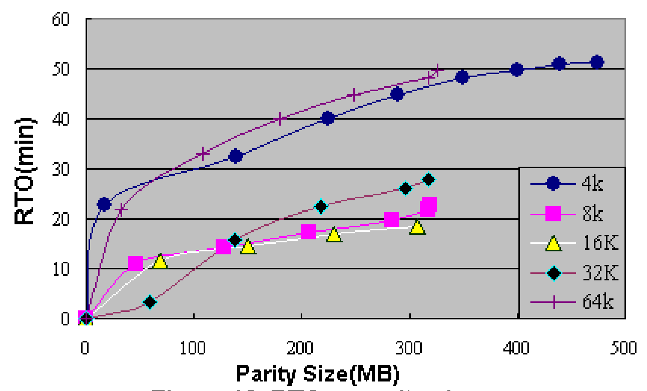


**Figure 12. RTO vs parity sizes**

| 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 |

**Table 3. Measured computation time for XOR and decoding process in TRAP-4 implementation on PC1.**

In order to compare the recovery time, RTO, of our *TRAP*-4 with that of *TRAP*-3, we measure the time it takes to do the XOR and decoding operations of *TRAP*-4 as shown in Table 3. Since we have only implemented the recovery program for *TRAP*-4 but not for *TRAP*-3, we will carry out the following simplified analysis just to approximately compare the two. Suppose that *TRAP*-3 reads the index node first to find out the exact location of the data block with a given time stamp for each changed data block. Next the data block is read out from the *TRAP* storage to a temporary storage. If we have a total of $N_B$ changed data blocks, the data retrieval time for *TRAP*-3 to recover data is approximately given by

$$(inode\_size/IO\_Rate+Block\_size/IO\_Rate+2S+2R)N_B,$$

where S and R are average seek time and rotation latency of the hard drive, respectively. To recover data, *TRAP*-4 needs not only to retrieve parity log for each data block but also to decode parity and to compute XORs. Let $T_{DEC}$ and $T_{XOR}$ denote the decoding time and XOR time. The data retrieval time for *TRAP*-4 to recover data is approximately given by

$$(T_{DEC}+T_{XOR}+Avg\_log\_size/IO\_Rate+S+R)N_B,$$

where *Avg_log_size* is the average parity log size for each data block. Our experiments show that the average log size is 38KB. Therefore, an entire parity log with the data block at the end is read from *TRAP* disk every time when we try to recover one block of data. It is important to note that the data log sizes of *TRAP*-3 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 formulae, we plot the data retrieval time of the two *TRAP* architectures as shown in Figure 13 assuming the average seek time to be 9ms, the average rotation latency to be 4.15ms, and the *IO_Rate* to be 45MB/s. Note that the time it takes to do consistency check and write in-place should be the same for both systems. As shown in the figure, *TRAP-4* generally takes 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.

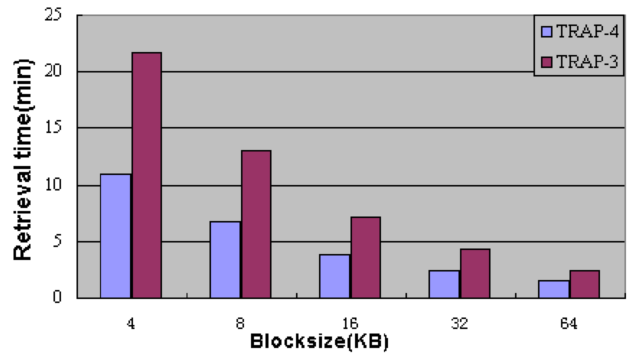


**Figure 13. Retrieval time comparsion for recovery between TRAP-3 and TRAP-4**

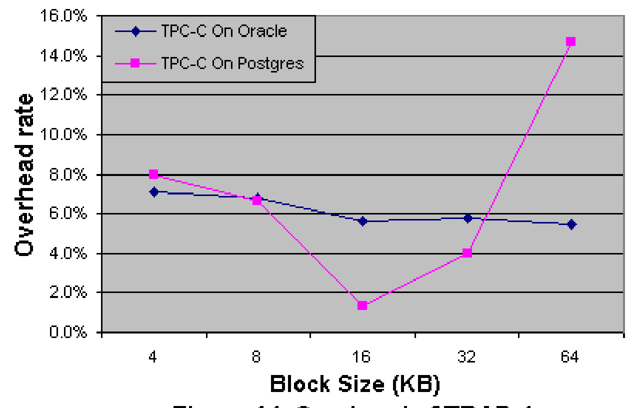


**Figure 14. Overhead of TRAP-4**

Computing and logging parities in *TRAP-4* 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 as shown in Table 3. In addition, we have also measured the TPC-C throughputs while running TPC-C on Oracle and Postgres databases with two storage systems. One storage system has *TRAP-4* installed and the other has no *TRAP-4* 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 1. This overhead rate is a measure of slow down of the *TRAP*-4. Figure 14 plots the overhead rates for different block sizes. Most of the overhead rates are less than 8% with one exception of 64KB on Postgres database. The lowest overhead is less than 2% for the block size of 16KB. It should be noted that our implementation does not assume a RAID controller. All the parity computations are considered extra overheads. As mentioned previously, *TRAP-4* can leverage the parity computation of RAID controllers. Therefore, if *TRAP*-4 were implemented inside a RAID array, the overheads would be much lower.

## 6. Conclusions

We have presented a new disk array architecture capable of providing **t**imely **r**ecovery to **a**ny **p**oint-in-time for user data stored in the array, referred to as *TRAP* array. 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. Standard 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 up to 2 orders of magnitude improvements in terms of storage efficiency. 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 continuous data protection capability.

The executable code of our TRAP implementation is available online at www.ele.uri.edu/hpcl.

## Acknowledgments

This research is sponsored in part by National Science Foundation under grants CCR-0073377, CCR-0312613, and SGER Grant 0610538. 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 National Science Foundation. The authors would like to thank John DiPippo for his supports and technical discussions, and Kerry Yang for her help in editing and proof reading of the paper. The authors are grateful to the anonymous reviewers for their valuable comments that helped in improving the paper. We also thank Slater Interactive Office of Rhode Island Economic Development Council for the generous financial support on part of this research work.

## 7. References

[1] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)", In *Proc. of the ACM SIGMOD International Conference on Management of Data*, 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," In *Proc. of the 21st Annual International Symposium on Computer Architecture*, Chicago, IL, 1994.

[3] G.A. Alvarez, W. A. Burkhard, and F. Christian, "Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering," In *Proc. of the 24th Annual International Symposium on Computer Architecture,* Denver, CO, 1997.

[4] C. I. Park, "Efficient placement of parity and data to tolerate two disk failures in disk arrays systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol.6, pp. 1177-1184, Nov. 1995.

[5] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-Diagonal parity for double disk failure correction," In *Proc. of the 3rd USENIX Conference on FAST,* San Francisco, CA, March 2004.

[6] D. M. Smith, "The cost of lost data," *Journal of Contemporary Business Practice,* Vol. 6, No. 3, 2003.

[7] D. Patterson, A. Brown and et al. "Recovery oriented computing (ROC): Motivation, Definition, Techniques, and Case Studies," *Computer Science Technical Report UCB/CSD-0201175*, U.C. Berkeley, March 15, 2002.

[8] 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.

[9] G. Duzy, "Match snaps to apps," *Storage, Special Issue on Managing the information that drives the enterprise*, pp. 46-52, Sept. 2005.

[10] A.L. Chervenak, V. Vellanki, and Z. Kurmas, "Protecting file systems: A survery of backup techniques," In *Proc. of Joint NASA and IEEE Mass Storage Conference*, College Park, MD, March 1998.

[11] J. Damoulakis, "Continuous protection,"*Storage,* Vol. 3, No. 4, pp. 33-39, June 2004.

[12] K. Keeton, C. Santos, D. Beyer, J. Chase, J. Wilkes, "Designing for disasters," In *Proc. of 3rd Conference on File and Storage Technologies*, San Francisco, CA, 2004.

[13] D. Patterson, "A New Focus for a New Century: Availability and Maintainability >> Performance," In *FAST Keynote*, January 2002, www.cs.berkeley.edu/ ~patterson/talks/keynote.html.

[14] C. B. Morrey III and D. Grunwald, "Peabody: The time traveling disk," In *Proc. of IEEE Mass Storage Conference*, San Diego, CA, April 2003.

[15] B. O'Neill, "Any-point-in-time backups," *Storage, Special Issue on Managing the Information that Drives the Enterprise*, Sept. 2005.

[16] J. Gray, "Turing Lectures," http://research. Microsoft.com/~gray.

[17] L. P. Cox, C. D. Murray, B. D. Noble, "Pastiche: making backup cheap and easy," In *Proc. of the 5th USENIX Symposium on Operating System Design and Implementation*, Boston, MA, Dec. 2002.

[18] M. B. Zhu, Kai Li, R. H. Patterson, "Efficient data storage system," US Patent No. 6,928,526.

[19] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," In *Proc. of the 7th International Conference on Architecture Support for Programming Languages an Operating Systems (ASPLOS-7)*, Cambridge, MA, 1996.

[20] EMC Corporation, "EMC TimeFinder Product Description Guide," 1998,http://www.emc.com/ products/product_pdfs/timefinder_pdg.pdf.

[21] Hitachi Ltd., "Hitachi ShadowImage implementation service," June 2001,http://www.hds.com /pdf_143_ implem_shadowimage.pdf

[22] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the Coda file system," In *Proc. of 13th ACM Symposium on Operating System Principles*, Pacific Grove, CA, Oct. 1991.

[23] Z. Peterson and R. C. Burns, "Ext3cow: A Time-Shifting File System for Regulatory Compliance", *ACM Transactions on Storage*, Vol.1, No.2, pp. 190-212, 2005.

[24] D.K. Gifford, R.M. Needham and M.D. Schroeder, "Cedar file system," *Communication of the ACM*, Vol.31, No.3, pp. 288-298, March 1988.

[25] 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 Transactions on Computer Systems*, Vol.6, No.1, pp.51-81, Feb. 1988.

[26] N.C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley, "Logical vs. Physical file system backup," In *Proc. of 3rd Symposium. on Operating system Design and Implementation*, New Orleans, LA, Feb 1999, pp. 239-250.

[27] S. Quinlan and S. Dorward, "Venti: a new approach to archival storage," In *Proc of the 2002 Conference on*

*File and Storage Technologies*, Monterey, CA, Jan. 2002, pp. 89-101.

[28] 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," In *Proc. of 17th ACM Symposium on Operating System Principles*, Charleston, SC, Dec. 1999, pp. 110-123.

[29] A. Sankaran, K. Guinn, and D. Nguyen, "Volume Shadow Copy Service," March 2004, http://www.microsoft.com.

[30] A.J.Lewis, J. Thormer, and P. Caulfield, "LVM How-To," http://www.tldp.org/HOWTO/LVM-HOWTO.html.

[31] D. Hitz, J. Lau, and M. Malcolm, "File system design for an NFS file server appliance," In *Proc. of the USENIX Winter Technical Conference*, San Francisco, CA, 1994, pp. 235-245.

[32] W. Xiao, Y. Liu, Q. Yang, J. Ren, and C. Xie, "Implementation and Performance Evaluation of Two Snapshot Methods on iSCSI Target Storages," In *Proc. of NASA/IEEE Conference on Mass Storage Systems and Technologies,* May, 2006,

[33] G.A. Gibson and R.V. Meter, "Network Attached Storage Architecture," *Communications of the ACM*, Vol. 43, No 11, pp.37-45, November 2000.

[34] J. Damoulakis, "Time to say goodbye to backup?" *Storage*, Vol. 4, No. 9, pp.64-66, Nov. 2006.

[35] D. G. Korn and E. Krell, "The 3-D file system," In *Proc. of the USENIX Summer Conference,* Baltimore, DC, Summer 1989, pp.147-156.

[36] B. Berliner and J. Polk, "Concurrent Versions System (CVS)," 2001, http://www.cvshome.org.

[37] L. Moses, "An introductory guide to TOPS-20," *Tech. Report TM-82-22*, USC/Information Sciences Institutes, 1982.

[38] K. McCoy, "*VMS File System Internals*," Digital Press, 1990.

[39] C.A.N. Soules, G. R. Goodson, J. D. Strunk, and G.R. Ganger, "Metadata efficieny in versioning file systems," In *Proc. of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, March 2003, pp. 43-58.

[40] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: The OceanStore prototype," In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST),* San Francisco, CA, March 2003.

[41] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," In *Proc. of the Eighteenth ACM symposium on Operating systems principles*, Alberta, Canada, October 2001.

[42] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok, "A versatile and user-oriented versioning file system," In *Proc. of the 3rd USENIX Conference on File and Storage Technologies,* San Francisco, CA, 2004.

[43] M. Ji, A. Veitch, and J. Wilkes, "Seneca: remote mirroring done write," In *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, TX, 2003, pp. 253-268.

[44] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha, "A Logic of File Systems," in *Proc. Of 4th USENIX Conference on Filesystems and Storage Technologies*, 2005.

[45 ] Q. Yang "Data replication method over a limited bandwidth network by mirroring parities," Patent pending, U*S Patent and Trademark office,* 62278-PCT, August, 2004.

[46] 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.

[47] G. Roelofs and J.L. Gailly, "zlib library," 2005, http://www.zlib.net.

[ 48 ] UNH, "iSCSI reference implementation," 2005, http://unh-iscsi.sourceforge.net.

[49] Microsoft Corp., "Microsoft iSCSI Software Initiator Version 2.0," 2005, http://www.microsoft.com/windowsserversystem/storage/default.mspx.

[50] Yiming Hu and Qing Yang, "DCD---Disk Caching Disk: A New Approach for Boosting I/O Performance," In *23rd Annual International Symposium on Computer Architecture (ISCA),* Philadelphia, PA, May 1996.

[51] Transaction Processing Performance Council, "TPC BenchmarkTM C Standard Specification," 2005, http://tpc.org/tpcc.

[52] S. Shaw, "Hammerora: Load Testing Oracle Databases with Open Source Tools," 2004, http://hammerora.sourceforge.net.

[53] J. Piernas, T. Cortes and J. M. García, "tpcc-uva: A free, open-source implementation of the TPC-C Benchmark," 2005, http://www.infor.uva.es/~diego/tpcc-uva.html.

[54] H.W. Cain, R. Rajwar, M. Marden and M.H. Lipasti, "An Architectural Evaluation of Java TPC-W," *HPCA 2001*, Nuevo Leone, Mexico, Jan. 2001.

[55] Mikko H. Lipasti, "Java TPC-W Implementation Distribution," 2003, http://www.ece.wisc.edu/~pharm/tpcw.shtml.