Optimal Implementation of Continuous Data Protection (CDP) in Linux Kernel

Xu Li and Changsheng Xie
Data Storage Division, Wuhan National Laboratory for Optoelectronics
Huazhong University of Science and Technology
Wuhan, Hubei, P. R. China
lixu.hust@gmail.com

Qing Yang
Dept. of Electrical and Computer Engineering
University of Rhode Island
Kingston, RI 02881
Email: qyang@ele.uri.edu

Abstract

To protect data and recover data in case of failures, Linux operating system has built-in MD device that implements RAID architectures. Such device can recover data in case of single hardware failure among multiple disks. But it cannot recover data that were damaged by human errors, virus attack, and disastrous failures. In this paper, we present an implementation of a device driver that is capable of recovering data to any point-in-time in case of various failures. A simple mathematical model is used to guide the optimization of our implementation in terms of space usage and recovery time. Extensive experiments have been carried out to show that the implementation is fairly robust and numerical results demonstrate that the implementation is optimal.

Keywords: *Data storage, data protection, CDP, data recovery, disk I/O architecture.*

1. Introduction

With the rapid advances in networked information services, data protection and recover have become top priority of many organizations [1,2,3,4]. In Linux operating system, there are variety of data protection and recovery programs in the open source community. Such programs can generally be classified into three categories: block level device drivers that provide data protection functionalities, file versioning, and backup and replications.

Typical block level device driver that provides data protection and recovery is MD (multiple devices) driver for software disk arrays [5]. MD implements most of RAID architectures including RAID1 through RAID5 in software, often referred to as software RAID. Such device can tolerate one disk failure and can rebuild data on the failed disk using other functional disks. However, such a device is not able to go back to a past point-in-time to recover data damaged by human errors, virus attacks, and disasters.

File versioning is another type of data protection technique that records a history of changes to files. Versioning was implemented by some early file systems such as Cedar File System [6], 3DFS [7], and CVS [8] 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 [9] and VMS [10] that have automatic versions for some file operations. Elephant [11] transparently creates a new version of a file on the first write to an open file. CVFS [12] makes versions for each individual write or small meta-data using highly efficient data structures. OceanStore [13] uses versioning not only for data recovery but also for simplifying many issues with caching and replications. The LBFS [14] 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 [15]. 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 [16] 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. But they work 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.

At block level, data protection is traditionally done using snapshots and backups. 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 [3].

In this paper, we present a new implementation of block level CDP driver in Linux operating system. Our implementation is based on the concept of TRAP architecture [17] that is capable of recovering data to any point-in-time in case of various failures. Two important design issues have been studied in depth: additional storage space usage and recovery time. We use a simple mathematical model to guide our design to optimize space usage and recovery time. Furthermore, in order to minimize possible failures caused by broken chains of parities, we provide an optimal way of organizing the parity chain with periodical snapshots inserted in the chains.

Based on our implementation, we have carried out extensive experiments to test the robustness of our program and to evaluate the performance of our implementation. Standard benchmarks are used in our experiments such as TPC-C, IOMeter, and PostMark. Our measurement results show that our implementation is space optimal and recovery time optimal.

The paper is organized as follows. Next section gives a brief overview of TRAP architecture for the purpose of completeness. Section 3 presents the detailed design of our implementation associated with the mathematical model used to guide our design. In Section 4, a detailed implementation as a Linux device driver is presented followed by our experimental settings in Section 5. Section 6 gives numerical results and discussions. We conclude our paper in Section 7.

2. Brief Overview of TRAP Architecture

As presented in [17], TRAP keeps a log of parities as a result of each write on a block. Figure 1 shows the basic design of *TRAP*. 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 $(A_1, A_2 \ldots A_i, \ldots A_n)$. The RAID controller performs the following operation to update its parity disk:

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

where $P_{T(k)}$ is the new parity for the corresponding stripe, $A_i(k)$ is the new data for data block A_i, $A_i(k-1)$ is the old data of data block A_i, and $P_{T(k-1)}$ is the old parity of the stripe. Leveraging this computation, *TRAP* appends the first part of the above equation, i.e. $P'_{T(k)} = A_i(k) \oplus A_i(k-1)$, to the parity log stored in the *TRAP* disk after a simple encoding box, as shown in Figure 1.

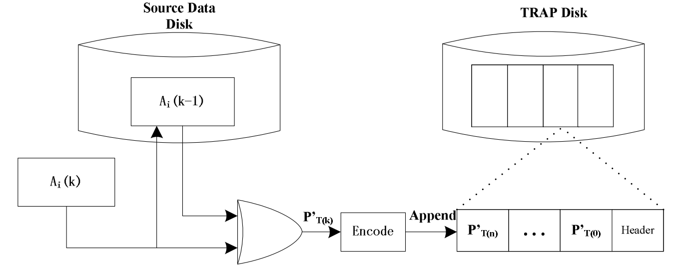


Fig. 1 Data logging method of TRAP design.

Now consider the parity log corresponding to a data block, A_i, after a series of write operations. The log contains ($P'_{T(k)}, P'_{T(k-1)} \ldots\ldots, P'_{T(2)}, P'_{T(1)}$) with time stamps $T(k), T(k-1), \ldots\ldots, T(2),$ 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 ($t_0 \leq t_1$). 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 $T(r) \leq t_0$. 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

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

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.

3. Design and Analysis of ST-CDP

The TRAP architecture discussed in the previous section provides CDP function by means of the parity chains resulting from block write operations. Since every change is kept in the chain, one can go back to any point-in-time. The traditional snapshot/backup, on the other hand, provides periodical data images of block level storage. When data recovery is necessary, these two data protection techniques work quite differently. TRAP needs to retrieve the parity chain for each data

block and perform the parity computation to recover the data block corresponding to the recovery time point. As the parity chain gets longer, so does the recovery time because of longer parity computations. Snapshot, on the other hand, just needs to restore the corresponding data blocks corresponding to the recovery time point, though the number of possible recovery points is limited by the frequency of snapshots performed. These two techniques present us with a trade-off between RPO (Recovery Point Objective) and RTO (Recovery Time Objective) [4]. Our purpose here is to design an optimal approach to data recovery by taking advantages of both techniques.

In our design and implementation, we take a hybrid approach. The idea is to break down the parity chain into sub-chains. Between any two subsequent sub-chains, we insert snapshot data image. The length of the sub-chain is a configurable parameter determined by system administrator or storage manager. We call our design ST_CDP (Snapshot in TRAP CDP). Adding snapshots between parity chains has several practical advantages. First of all, it limits the maximum recovery time. Secondly, the configurable sub-chain sizes allow a system administrator to organize the parity chains in different data structures and to optimize space usage and retrieval times. Thirdly and more importantly, this organization increases significantly the reliability and recoverability of the TRAP architecture. This is because a parity chain may become completely useless if there is any single bit error in the chain. The longer the parity chain is, the higher the probability of chain failure. Breaking up the parity chains into sub-chains and adding snapshot in between reduces the probability of such failures and increases data recoverability.

Figure 2 shows the new parity logging structure. As shown in the figure, we insert snapshots in the parity chain. As a result, sub-chains are formed that are separated by periodical snapshots. At recovery time, only one sub-chain that contains the recovery time point is needed. The recovery time for each data block is limited by the half of the sub-chain length because parity computation can be done both ways, redo and undo, as shown in [17]. To minimize chain retrieval time, one can also organize all sub-chains in an efficient data structure, which is out of scope of this paper.

From the above discussion, it is clear that the length of each sub-chain is an important parameter to determine. Let d be the length of each sub-chain in terms of the number of parity blocks in the sub-chain. We would like to determine what d value one should choose for optimal implantation of TRAP on Linux operating system. In order to provide a quantitative guidance on how to choose d, let us define the following symbols:

Symbols	Definition
d	Sub-chain (parity chain) Length: number of parity blocks in each sub-chain
IO_{rate}	IO throughput of the disk storage
S_{blk}	Data block size
S_{log}	Size of compressed parity block
C	Compression Ratio: $C = S_{blk} / S_{log}$
T_{dec}	Decoding time
T_{xor}	EX-OR operation time
T_{spn}	RPO: time span between current time and recovery time point
W_{avg}	Average number of write operations per time unit

Table 1. Definition of symbols used in analysis.

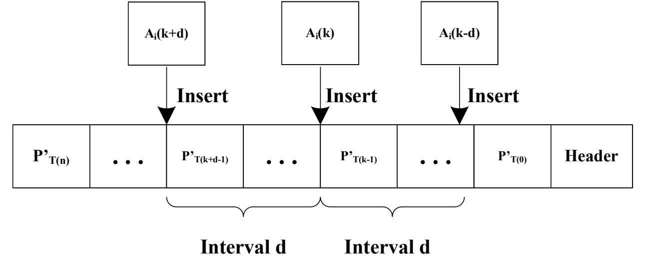


Fig. 2 Data logging method of ST-CDP design.

If we do not break up parity chains, the recovery time of each data block is given by

$$(T_{dec}+T_{xor}+ S_{log} /IO_{rate})* W_{avg}* T_{span}. \quad (3)$$

Now consider our ST-CDP design with sub-chains of d parity blocks. As mentioned previously, the recovery time for each data block is limited by the half of the sub-chain length, d, because parity computation can be done both ways, redo and undo. If we assume that the recovery time point is uniformly distributed among d points within a sub-chain if the RPO falls into that chain, the expected parity blocks needed to do EX_OR computations for the data recovery is given by

$$E(d) = \frac{1}{2}\sum_{k=0}^{\lceil \frac{d}{2} \rceil} k * P(x = k) + \frac{1}{2}\sum_{k=1+\lceil \frac{d}{2} \rceil}^{d} (d-k) * P(x = k)$$

$$\approx \sum_{k=0}^{\lceil \frac{d}{2} \rceil} \frac{2k}{d} \approx \left\lceil \frac{d}{4} \right\rceil. \quad (4)$$

The recovery time of each data block in case of ST-CDP is given by

$$T(d) = (T_{dec}+T_{xor}+ S_{log} /IO_{rate})* E(d)+ S_{blk} /IO_{rate} \quad (5)$$

The first half of the above equation gives the parity computation time and the second half gives the data copy time. It is interesting to note that this recovery time is independent of RPO but dependent on d value. The recovery time increases as d increases.

Let us now consider the cost of the ST-CDP program. The major cost is the additional storage space needed to store the parity logs and snapshot data while running ST-CDP. We would like to examine the average storage increase per time unit while running the ST-CDP, which is given by

$$S(d) = S_{log}*W_{avg} + S_{blk}*W_{avg}/d,$$

where the first term gives the space for parity log and the second term gives the snapshot space. Since the number of snapshots inserted is inversely proportional to ***d,*** the space usage of snapshots is also inversely proportional to ***d***.

The other important cost is the time it takes to recover data. Ideally, we would like to use as little storage space as possible and recover data as quickly as possible. We will use these two factors to determine how good a data protection technology is. We therefore use the product of these two cost factors as the compound cost of ST-CDP. Let

$$\begin{aligned} F(d) &= T(d)*S(d) \\ &= [(T_{dec}+T_{xor}+S_{log}/IO_{rate})*E(d)+S_{blk}/IO_{rate}]*(S_{log}*W_{avg}+S_{blk}*W_{avg}/d) \\ &= (c_1*E(d)+c_2)*(c_3+c_4/d) \\ &= (c_1*d/4+c_1/4+c2)(c_3+c_4/d) \end{aligned} \qquad (6)$$

where $c_1=(T_{dec}+T_{xor}+S_{log}/IO_{rate})$, $c_2=S_{blk}/IO_{rate}$, $c_3= S_{log}*W_{avg}$, and $c_4= S_{blk}*W_{avg}$. These are constants independent of ***d***.

Now, let us consider the derivative of ***F(d)*** and set it to 0. We have

$$F'(d)=0 \rightarrow d_0 = \sqrt{\frac{(c_1+4c_2)c_4}{c_1c_3}} \qquad (7)$$

Since the second derivative,

$$F''(d) = (c_1+4c_2)C_4*d^{-3}/2,\ F''(d = d_0) = c_1c_3/2d_0>0,$$

the minimum value of ***F(d)*** exists when $d=d_0$. We will choose ***d*** to be the integer closest to d_0 as our optimal sub-chain size.

4. Driver Implementation

Based on the design and analysis presented in the previous sections, we have implemented our ST-CDP in Linux Kernel. Our implementation is developed as an added kernel module on top of MD RAID5. The ST-CDP was developed as a standalone block device driver independent of higher level file systems. As a result, it can support variety of applications including different file systems and database applications.

The ST-CDP has two major functional modules, CDP logging module and recovery module. The CDP logging module works at run time to keep journaling of parities and snapshots. It bypasses all I/O read operations and intercepts all I/O write operations. There are two parallel threads, one handling normal write operations and the other performing CDP functions. There are two major parts in the CDP functional module. The first part does the parity computation and logging. It also keeps track of metadata for the parity logs. The second part carries out snapshot operations when triggered. The snapshot operations starts whenever the number of parities collected for a block reaches value ***d*** defined in the previous section. The underlying storage is partitioned into two volumes: source volume and CDP volume. The source volume stores the production data while the CDP volume stores the parity logs and snapshot data.

The recovery module of the ST-CDP is a program that runs offline. When data recovery needs to be done, the recovery module starts by retrieving parity logs and snapshot data. Based on the designated RPO, it searches the parity chains for each data block to find the sub-chain that contains the desired RPO. Once such sub-chain is found, the recovery program searches for a parity block that has the timestamp matches the closest to the RPO. OX-OR operations are then performed to recover the right data block. After all changed data blocks are recovered, the data will be written to the source volume and recovery process is done. It is also possible that the RPO matches one of the snapshots in the CDP volume. In this case, no parity computation is necessary. The recovery program just copies the snapshot data to the source volume.

5. Experimental Settings

For the purpose of testing of our ST-CDP implementation and performance evaluation, we have carried out measurement experiments. Figure 3 shows the high level block diagram of our experimental settings. To allow multiple clients and multiple storage servers in a networked environment, we implemented the lower level storage device using iSCSI protocol as shown in Figure 3. Our ST-CDP module runs on the storage server at block device level of Linux operating system. The client machine has file system, database, and application benchmarks installed. The details of the

hardware and software environment in our experiments are shown in Table 2 below.

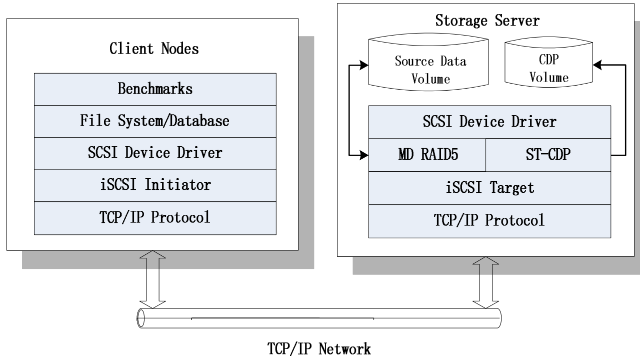


Fig. 3 System architecture of ST-CDP implementation.

Right workloads are important for performance studies [18]. In order to have an accurate evaluation, we use real world I/O workloads and 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 [19]. TPC-C simulates the execution of a set of distributed and on-line transactions (OLTP) for a period of two to 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. From data storage point of view, these transactions will generate reads and writes that will change data blocks on disks. For Postgres Database, we use the implementation from TPCC-UVA [20]. 8 warehouses with 25 users are built on Postgres database. Details regarding TPC-C workloads specification can be found in [19].

Besides benchmarks running on databases, we have also run two file system benchmarks IoMeter and PostMark. IoMeter is a flexible and configurable benchmark tool that is also widely used in industries and the research community [21]. It can be used to measure the performance of a mounted file system or a block device. We run the IoMeter on NTFS with 4K-block size for two types of workloads: 100% random writes, and 30% writes and 70% reads. PostMark is another widely used file system benchmark tool written by Network Appliance, Inc [22]. It measures performance in terms of transaction rates in an ephemeral small-file environment by creating a large pool of continually changing files. Once the pool has been created, a specified number of transactions occur. Each transaction consists of a pair of smaller transactions, i.e. Create file/Delete file and Read file/Append file. Each transaction's type and files it affected are chosen randomly. The read and write block size can be tuned. In our experiments, we set PostMark workload to include 10,000 files and to perform 20,000 transactions. Read and Write buffer sizes are set to 4KB.

	2 Client Nodes	Storage Server
CPU	Intel Xeon 2.8GHZ	Intel Core 2 E2140, 1.6GHz
RAM	DDR2 533 , 2GB	DDR2 333, 1GB
Disk	SATA 300GB	SATA 300GB
OS	Red Hat Linux 9.0 (Kernel2.6.9)	Gentoo Linux (Kernel 2.6.20)
Switch	Cisco 3750-E Gb	
NIC	2*PCI 1GB/s	
Benchmarks	TPC-C on Postgres database IoMeter PostMark	

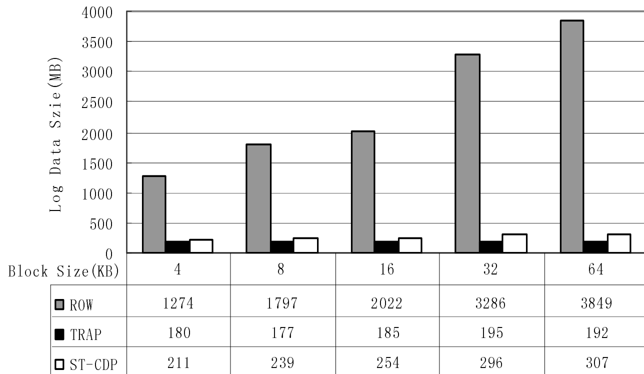
Table 2. List of testing environments.

6. Numerical Results and Discussions

In this section, we present our measurement results in terms of space usage, recovery time, and run time performance impact of ST-CDP. We compare the performance results of three data protection techniques: namely native TRAP with no sub-chains, ST-CDP, and pure periodical snapshots. The snapshot we evaluate here is redirect-on-write snapshot, ROW for short, as opposed to copy-on-write snapshots [23]. In our experiments, we set the values of ***d*** to 71、79、85、91、and 94 corresponding to block sizes of 4KB，8KB，16KB，32KB，and 64KB, respectively. These values are selected based on our analysis presented in the previous section.

Our first experiment is to measure the additional space usage of the three data protection technologies. Figure 4 shows the measured results. We plotted the space usage of the three data protection technologies for different block sizes ranging from 4KB through 64KB. It can be seen from this figure that snapshot takes most space because it keeps the original data blocks of all changed data. Native TRAP takes the least amount of space because of locality property of write operations as evidenced in [17]. The space usage of ST-CDP is somewhere in between the other two because it stores both parity logs and small amount of snapshots between sub-chains. Because we choose the optimal value of ***d*** for each block size, the space overhead of ST-CDP is closer to that of TRAP than that of ROW snapshot. Our

observation is that ST-CDP provides continuous data protection with substantial less storage overhead than continuous real-time snapshots.



Block Size(KB)	4	8	16	32	64
ROW	1274	1797	2022	3286	3849
TRAP	180	177	185	195	192
ST-CDP	211	239	254	296	307

Fig. 4 Storage space comparison for TPC-C on Postgres database.

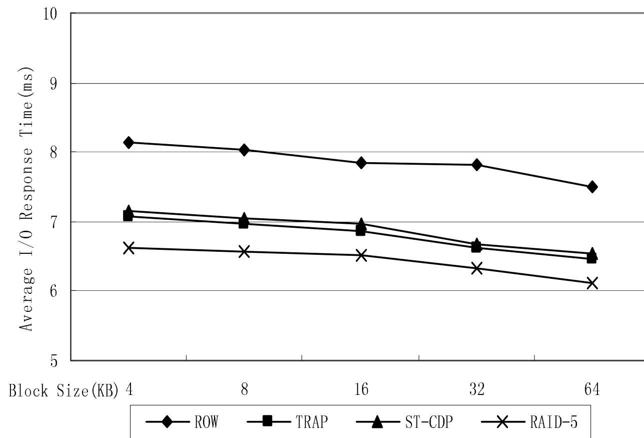


Fig. 5 Average I/O response time comparison for 70% reads and 30% writes of IoMeter benchmark.

Since ST-CDP carries out parity computation and snapshot operations at run time, an immediate question is how it impact application performance. Our next experiment is to evaluate the performance impact of ST-CDP on applications. For this purpose, we run IOMeter to measure the IO performance while enabling the ST-CDP module. Figure 5 shows our measured results in terms of average I/O response time as functions of block sizes. We plotted 4 performance curves corresponding to snapshots, TRAP, ST-CDP, and RAID5 alone with no data protection program running. Performance of RAID5 is used as a reference for us to observe the negative impacts of the three data protection technologies. We noticed that snapshot has the most performance impact and TRAP has the least. ST-CDP is in between but close to that of TRAP. For block size of 4KB, ST-CDP's performance is about 8.3% lower than that of RAID5. For block size of 32KB, such performance difference is about 5.4%. The maximum performance drop of TRAP and snapshot compared to RAID5 are 7.1%（4KB） and 23.2%（32KB）, respectively, whereas the maximum performance drop of ST-CDP is about 8.3%.

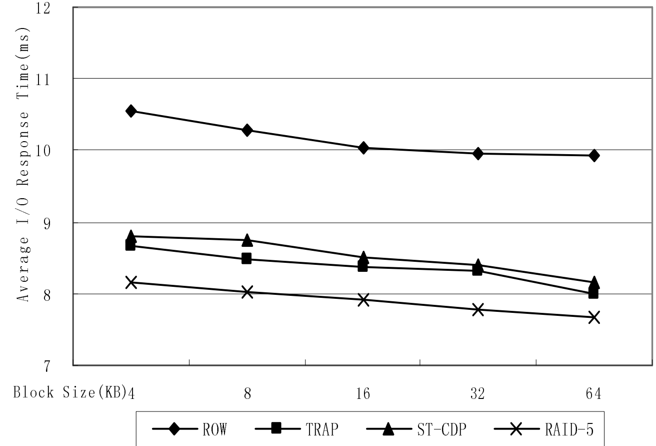


Fig. 6 Average I/O response time comparison for 100% random writes of IoMeter benchmark.

Figure 6 shows the results of IOMeter with 100% random writes. Results similar to that of Figure 5 are observed. The maximum performance drops of the three data protection techniques compared to that of RAID5 are 9.2%(8KB)， 6.3%（4KB），and 29.6%（64KB） for ST-CDP, TRAP, and snapshots respectively.

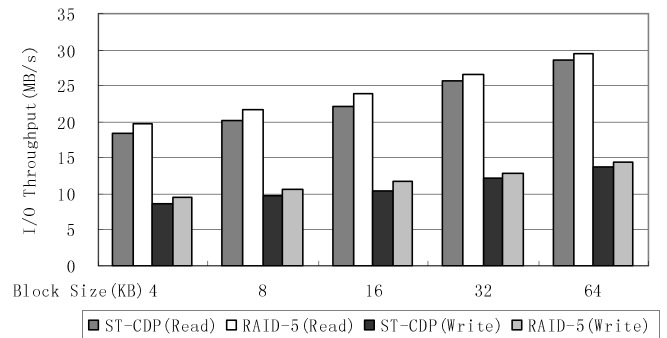


Fig. 7 I/O throughput comparison for PostMark benchmark.

PostMark results are shown in Figure 7. In this figure, we compared the performance of ST-CDP with the performance RAID5 to see how much performance degradation caused by the ST-CDP overhead. We noticed that the performance of both RAID5 and ST-CDP increases as the block size increases. The performance differences between RAID5 and ST-CDP

are very small and these performance differences do not change with block size. This observed result can be attributed to the fact that the parity computation of ST-CDP is a part of RAID5 parity computation. Therefore, the overhead of ST-CDP is manageable.

Our next experiment is to measure the recovery time that is very important performance parameter for data protection technologies. We considered the 5GB of data of normal I/O operations and try to recover data to different RPOs. We measured the recovery times of native TRAP and ST-CDP and compared their respective recovery times. Figure 8 shows the measured recovery time as function of RPO. As can be seen from this figure, TRAP's recovery time increases as RPO increases because of EX-OR computation of long parity chains. On the other hand, the recovery time of ST-CDP keeps flat while RPO changes. For example, for 4KB block, recovering data to half hour ago takes about 1,246 seconds with native TRAP. To recover data to 8 hours ago, TRAP takes about 4,910 seconds, 3.9 times longer. For block size of 64KB, recovery time of TRAP becomes smaller. It takes about 723s and 2,061s to recover data to half an hour ago and 8 hours ago, respectively. On the other hand, ST-CDP module can recover data much faster irrespective of RPO. As shown in Figure 8, the recovery time varies from 212 seconds to 253 seconds, very little change!

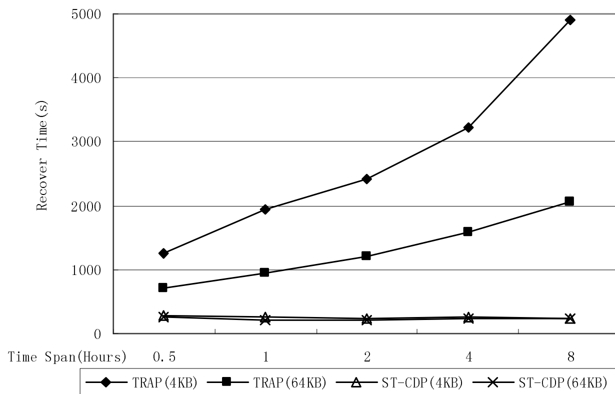


Fig. 8 Recover time comparison between ST-CDP and TRAP.

7. Conclusions

In this paper, we have presented a design, analysis, and a Linux implementation of a continuous data protection technique referred to as ST-CDP. It is based on TRAP [17] technology that keeps logs of parities of changed data blocks and interspersed with snapshot data. The implementation is done at block device level as an independent device driver that can be added to MD software RAID device. Extensive experiments have been carried out to show the implementation is fairly robust. Standard benchmarks are used to evaluate the performance and cost of the implementation. Numerical results have shown that the overhead is manageable. The major advantage of ST-CDP is low RTO that is RPO independent. Our future work includes investigating reliability and recoverability of TRAP by adding error correcting code in parity chains and implementing it in hardware RAID controllers.

Acknowledgments

The authors would like to thank Jing Yang and Yu Cheng of Data Storage Division, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, for their help in system implementation and testing. Their contribution to this work is greatly appreciated. This research is sponsored in part by National Science Foundation of China under grants number NSFC-60736013, 60603074, and 60603075 and Chinese 973 grant number 2004CB318203 and US National Science Foundation under grants CCR-0312613 and CCF0610538. 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.

References

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

[2] 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.

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

[4] 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.

[5] Linux Kernel Drivers, Available: http://sourceforge.net

[6] 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.

[7] 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.

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

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Q. Yang , W. Xiao , and J. Ren, "TRAP-Array: A Disk Array Architecture Providing Timely Recovery to Any Point-in-time", In *Proceedings of the 33rd annual international symposium on Computer Architecture,* June 2006, pp. 289-301.

[18] 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.

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

[20] 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.

[21] Intel, "IoMeter: Performance Analysis Tool," http://www.iometer.org/.

[22] J. Katcher, "PostMark: A new file system bench - mark," *Network Appliance*, Tech. Rep. 3022, 1997.

[23] W. Xiao, Y Liu, Q. Yang, J Ren, and C Xie, "Implementation and performance evaluation of two snapshot methods on iSCSI target stores," in *Proc. of IEEE/NASA Conf. on Mass Storage Systems and Technologies*, May 2006.