A Remote Mirroring Architecture with Adaptively Cooperative Pipelining[⋆]

Yongzhi Song, Zhenhai Zhao, Bing Liu, Tingting Qin,
Gang Wang, and Xiaoguang Liu

Nankai-Baidu Joint Lab, College of Information Technical Science, Nankai University
94 Weijin Road, Tianjin 300071, China
syzcch@sina.com, zhaozhenhai1985@gmail.com, liubing87@126.com,
paula_qin_1987@126.com, wgzwp@163.com, liuxg74@yahoo.com.cn

Abstract. In recent years, the remote mirroring technology has attracted increasing attention. In this paper, we present a novel adaptively cooperative pipelining model for remote mirroring systems. Unlike the traditional pipelining model, this new model takes the decentralization of processors into account and adopts an adaptive batching strategy to alleviate imbalanced pipeline stages caused by this property. To release the heavy load on CPU exerted by compression, encryption, TCP/IP protocol stack and so on, we design fine-grained pipelining, multi-threaded pipelining and hybrid pipelining. We implement a remote mirroring prototype based on Linux LVM2. The experimental results show that, the adaptively cooperative pipelining model balances the primary and the backup sites - the two stages of the pipeline effectively, and fine-grained pipelining, multi-threaded pipelining and hybrid pipelining improve the performance remarkably.

Keywords: remote mirroring, cooperative pipelining, adaptive batching, fine-grained, multi-threaded.

1 Introduction

Consistently, data protection is a hot topic in IT academia and industry. Especially in recent years, after several great disasters, some enterprises with perfect data protection resumed quickly, while many others went bankrupt because of data loss. So data protection technologies have attracted increasing attention. Remote mirroring is a popular data protection technology that tolerates local natural and human-made disasters by keeping a real-time mirror of the primary site in a geographically remote place. There are two typical remote mirroring strategies: synchronous and asynchronous [1]. The latter is preferable due to the former's heavy sensitivity to the Round Trip Time (RTT) [2].

[⋆] Supported partly by the National High Technology Research and Development Program of China (2008AA01Z401), NSFC of China (60903028), SRFDP of China (20070055054), and Science and Technology Development Plan of Tianjin (08JCYBJC13000).

In this paper, we present a new cooperative pipelining model to depict remote mirroring systems. By "cooperative" we mean that the pipeline is across the primary site, the network and the remote backup site. Since each subtask naturally has an "owner", so we cannot distribute them arbitrarily to balance the pipeline stages. For this, we present an adaptive batching algorithm. Write requests are propagated to the backup site in batches and the batch size (interval) is adjusted dynamically according to the processing speed of the primary and the backup sites. We implement data compression and encryption in our prototype to reduce network traffic and enhance security respectively. These operations put a lot of pressure on CPU. For this, we design three accelerating methods: fine-grained pipelining, multi-threaded pipelining and hybrid pipelining.

The rest of this paper is organized as follows: In Section 2, we focus on the related work in the recent years. In Section 3, we illustrate the basic architecture of our system and present the adaptive cooperative pipeline. In Section 4, we introduce the implementation and evaluate it from results of a quantity of experiments. Finally, the conclusions and future work is given in Section 5.

2 Related Work

EMC Symmetrix Remote Data Facility (SRDF) [3] is a synchronous block-level remote replication technique that will switch to semi-synchronous mode if the performance is below the threshold. Veritas Volume Replicator (VVR) [4] is a logical volume level remote replication technique. It supports multiple remote copies and performs asynchronous replication using a log and transaction mechanism. Dot Hill's batch remote replication service [5] schedules point-in-time snapshots of the local volume, then transfers the snapshot data changes to one or more remote systems.

Network Appliance's SnapMirror [1] uses snapshot to keep the backup volume up to date. The WAFL file system is used to keep track of the blocks that have been updated. Seneca [6] delays sending a batch of updates to the remote site, in the hope that write coalescing will occur. Writes is coalesced only within a batch, and batches must be committed atomically at the remote site to avoid inconsistency.

Our prototype is also implemented in the logical volume level like VVR and Dot Hill's remote replication service. Like Seneca, the updates are sent to the backup site in batches for performance reasons. Our prototype also adopts an adaptive mechanism. However, it is based on asynchronous mode and for pipeline stage balancing rather than network conditions adapting.

There are many other remote replication products, such as IBM's Extended Remote Copy (XRC) [7], HP Continuous Access Storage Appliance (CASA) [8] and so on. In recent academic studies, [9] presents a prototype in which the synchronous and asynchronous mode are specified by the upper level applications or the system. [2] uses Forward Error Correction (FEC) and "callback" mechanism for high reliability.

3 Adaptively Cooperative Pipelining

Fig. 1 shows the architecture of our prototype. It is an asynchronous remote mirroring system implemented in Linux LVM2 [10]. NBD (the Network Block Device) [11] is used for data transmission between the primary and the backup sites. When a write request arrives, it is duplicated, and then the original enters the local queue and the replica enters the remote queue. The requests in the remote queue are sent to the backup site in batches at intervals. Since NBD transfers messages using TCP, the consistency is guaranteed as long as each batch is committed atomically in the backup site. In order to reduce the network traffic, requests are compressed before they are being sent. They are also encrypted to provide good security. It is easy to see that the order of compressing before encrypting is superior to the reverse order in computational complexity. Therefore, in the backup site, the requests are decrypted and then decompressed before being committed.

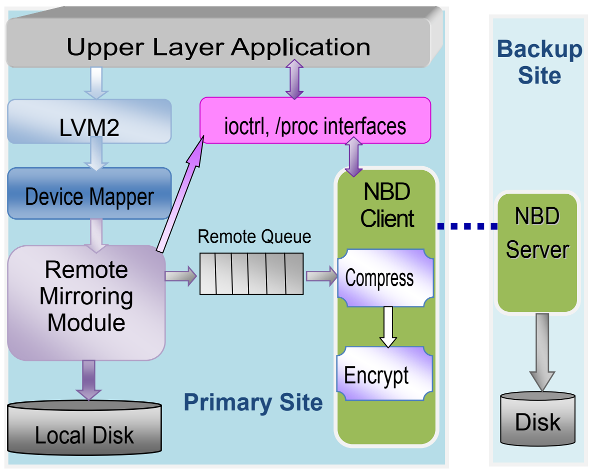

Fig. 1. Prototype architecture

3.1 Cooperative Pipelining

In order to maximize the throughput, an obvious way is to overlap the operations in the primary site and the operations in the backup site. That is, after a batch is sent, instead of waiting the reply, the primary site immediately throws itself into processing the next batch while the backup site processes the previous one. So we can describe the processing of the requests by a two-stage pipelining model. We call the two stages *the primary stage* and *the backup stage* respectively. This pipeline is "*cooperative*", that is, each batch is processed by the primary site and the backup site cooperatively. We know that, for a given task, the more the stages and the nearer the size of the stages, the higher the performance of the pipeline is. However, the cooperative pipelining model has some unique properties against this.

For traditional pipelining, to increase the speed of a single pipeline, one would break down the tasks into smaller and smaller units. For example, pipelines in modern CPUs typically have more than 20 stages. However, a cooperative pipeline is naturally composed of eight subtasks including batch assembling, compression, encryption, batch transmission, decryption, decompression, disk writing and reply. Since it is a software pipeline, further task decomposition will induce significant interaction overhead. Moreover, we can not accurately predict the execution time of some stages (primarily the disk operations and the network transmission whose performance depend on the current state of the system heavily). This is a serious obstacle to high efficient pipelining.

Typical distributed pipelining models, such as the pipelined gaussian elimination algorithm [12], break down the tasks into subtasks with the same size and assign them to proper processors. However, each subtask in a cooperative pipeline has a specific "owner" so that it can not be assigned to other processors. For example, the backup site can not perform data compression which must be done at the primary site. This inherently immutable task mapping contributes to the difficulty in load balancing. We must equal the speed of the primary stage and the backup stage for perfect load balance. Otherwise, processor will still be idle even if we break down the task into smaller subtasks with the same size. Moreover, if we want to improve performance by deepening the pipeline, we must further divide the primary stage and the backup stage identically.

3.2 Adaptive Batching

As mentioned above, the primary site can process the next batch immediately after the previous batch is sent. In a single-user system, this "best-effort" strategy guarantees the optimal performance. However, it has some drawbacks.

If the speed of the two stages are different, for instance, the primary stage is faster than the backup stage, the two sites are out of step under best-effort strategy. If the user application keeps up the pressure on the storage subsystem, the gap between the primary site and the backup site becomes wider and wider until the primary site exhausts system resource. Then the primary site will slow down to wait for the replies from the backup site to release enough resource. This brings unsteady user experience (response time of the primary site). Moreover, exhausting system resource by one process is not good for a multi-user system. This will impact the stability of the system and the performance of other processes seriously.

In a word, best-effort strategy does not coordinate the primary site and the backup site well. Or, it "coordinates" the two sites by exhausting system resource. So we introduce an *adaptive batching* algorithm for coordination. We set a *batch interval* when system initializing. This interval defines the request accumulating and the batch processing periods. That is, the requests accumulated in the previous period are processed in batches in the next period. Every time a batch finishes, we adjust the batch interval to approach the execution time of the primary stage and the execution time of the backup stage. Therefore, if the batch interval converges to a stable condition eventually, the primary stage and the backup stage will have the same execution time (both equal to the batch interval). Note that the execution time of disk operations and network transmission is not proportion to the batch size. So, although interval adjusts lengthens (or shortens) of both stages, the increment of the faster one (generally the primary stage) may be longer than that of the slower one (generally the backup site which contains disk operations and network transmission), therefore the batch interval converges.

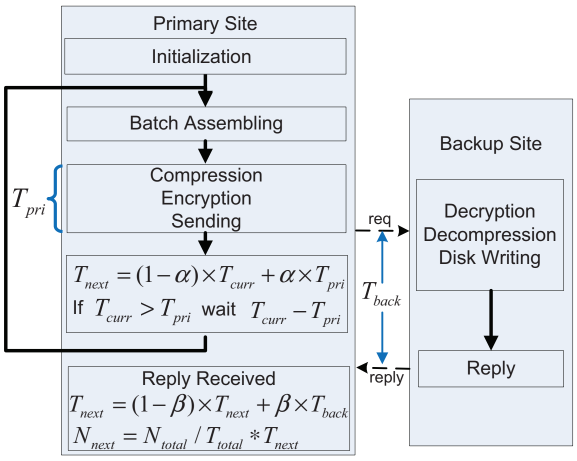
Primary Site
Initialization
Batch Assembling
Compression
Encryption
Sending
T_{pri}
$T_{next} = (1-\alpha)\times T_{curr} + \alpha\times T_{pri}$
If $T_{curr} > T_{pri}$ wait $T_{curr} - T_{pri}$
Reply Received
$T_{next} = (1-\beta)\times T_{next} + \beta\times T_{back}$
$N_{next} = N_{total} / T_{total} * T_{next}$
req
T_{back}
reply
Backup Site
Decryption
Decompression
Disk Writing
Reply

Fig. 2. Adaptive batching algorithm

Unlike best-effort strategy, adaptive batching algorithm lets the faster site sleep for a while instead of processing the next batch immediately when the two sites are out of step, that is, slows down the faster site to force the two sites in step. The advantage of this strategy is obvious: the faster site will not exhaust system resource, therefore it does not impact other processes in the system. Another advantage of adaptive batching is the good adaptability to change of network conditions. If network fluctuates, by interval adjusting, the primary site and the backup site adapt to the speed of network automatically and timely. The following *adaptive formulas* are used to adjust the batch interval.

$$T_{next} = (1-\alpha) \times T_{curr} + \alpha \times T_{pri} \tag{1}$$

$$T_{next} = (1-\beta) \times T_{next} + \beta \times T_{back} \tag{2}$$

where T_{curr} and T_{next} denote the current and the next intervals respectively, and T_{pri} and T_{back} denote the execution time of the primary stage and the backup stage respectively. In our implementation, T_{pri} includes time spent in batch assembling, compression, encrypting and batch sending, and T_{back} includes time spent in batch receiving, decryption, decompression, disk write and reply. α and

β are *adjusting factors* which are real numbers between 0 and 1. They control how fast the batch interval approaches to real processing time and how fast the pipeline adapts to network change. To counter continuous heavy load, we use a *batch size threshold* to control the maximum number of requests in a batch:

$$N_{next} = N_{total}/T_{total} \times T_{next} \tag{3}$$

where N_{total} denotes the total number of requests that have been processed, and T_{total} denotes the total processing time. That is, the processing capacity is estimated by statistics and is used to set the next threshold N_{next}. Fig. 2 illustrates the adaptive batching algorithm. When the primary stage finishes, Formula 1 is used to adjust the batch interval and the primary site will sleep a while if the primary stage is shorter than the current interval. Formula 2 is used when the reply is received.

The adaptive batching algorithm has several variants for different purposes. For example, we can set lower and higher thresholds for batch interval. This implies the range of acceptable RPO (recovery point objective [13]). Moreover, the batch size threshold can be used to provide QoS. We can fix the ratio of threshold to interval which implies the fixed processing speed, therefore we fix the resource occupation.

3.3 Accelerating Techniques

Data compression/decompression and encryption/decryption put a lot of pressure on CPU. Considering the popularity of multi-core systems, accelerating adaptively cooperative pipelines using parallel techniques is a natural idea. So we design two accelerating approaches: *fine-grained pipelining* and *multi-threaded pipelining*. A combination of these two approaches called *hybrid pipelining* is also considered.

Fine-Grained Pipelining. A common way to accelerate a single pipeline is to deepen the pipeline, that is, breaking down the tasks into smaller units, thus lengthening the pipeline and increasing overlap in execution. However, as mentioned above, for a cooperative pipeline, we can not re-decompose the task arbitrarily. Instead, we must decompose the primary stage and the backup stage into the same number of smaller stages with the same size. It is difficult to decompose the two existing stages identically. We adopt two strategies for this:

- We decompose the two existing stages manually by experience. In our implementation, the primary stage is decomposed into two sub-stages: the *compression stage* containing batch assembling and data compression, and the *encryption stage* containing data encryption and batch sending. The backup stage certainly is also decomposed into two sub-stages: the *computation stage* containing batch receiving, data decryption and decompression, and the *I/O stage* containing disk write and reply. Both primary and backup sites invoke two threads. Each thread is responsible for a sub-stage.

- Obviously, experience only guarantees that the four stages are approximately equal. In order to make them nearer and counter their dynamic change, adaptive batching algorithm is used again. After each stage finishes, the corresponding adaptive formula is applied to adjust the batch interval.

Multi-threaded Pipelining. Another intuitive accelerating approach is to decompose the each existing stage into subtasks in parallel instead of smaller stages. Since each batch contains dozens to hundreds of requests, a simple and effective decomposition technique is data decomposition. In our implementation, each batch is decomposed into two sub-batches with the same size. Both primary and backup sites invoke two threads. Each thread is responsible for a sub-batch. They process the sub-batches in parallel, and then send them in serial for consistency reasons.

Like fine-grained pipelining, multi-threaded pipelining also faces the difficulty in load balancing. Fortunately, the problem is much easier in this method. Note that the computation time of a request is proportional to the number of data bytes in it. So, if the workload contains requests all of the same size (for example, the workload in our experiments generated by Iometer), load balance is guaranteed simply by partitioning sub-batches according to the number of requests. Otherwise, we can partition sub-batches according to the total number of bytes.

Serial network transmission seems to be a drawback of multi-threaded pipelining. However, time spent in this operation is only a small part of the total execution time, thus serial network transmission does not impact the overall performance much. Our experimental results verified this point.

In addition, multi-threaded pipelining is more flexible than fine-grained pipelining. It can even be used to deal with unequal cooperative stages. For example, if the backup stage is twice as long as the primary stage, the backup site can use twice the threads than primary site to equal the execution time of the two stages.

Hybrid Pipelining. Our experimental results showed that neither four-stage pipelining nor double-threaded pipelining fully occupies CPU. Theoretically, deepening the pipeline further or using more threads will make full use of CPU power. However, as mentioned above, very deep pipeline will introduce significant interaction overhead. For the latter strategy, decomposing a batch into too many sub-batches may induce load imbalance and too many threads may increase interaction overhead. So we combine these two techniques. Both primary and backup stages are decomposed into two sub-stages, and each sub-stage is accelerated further by multi-thread technique. We call this method *hybrid pipelining.*

In fact, fine-grained pipelining is an *inter-batch parallelization*, that is, each batch is processed by only one processor at a time, and several batches are processed by multiple processors simultaneously. Multi-threaded pipelining is an *inner-batch parallelization*, that is, batches are processed one-by-one, and each batch is processed by multiple processors simultaneously. Hybrid pipelining is a two-dimensional parallelization.

4 Experimental Evaluation

4.1 Prototype Implementation

We implemented adaptive batching algorithm as a "remote copy" module in Linux LVM2. Like snapshot module in LVM2, this module treats the remote mirror as an attached volume of the original volume. Three accelerating techniques were also implemented. The underlying OS was RedHat AS server 5 (kernel version 2.6.18-128.el5). LVM2 2.02.39, device mapper 1.02.28 and NBD 2.8.8 were used. LZW algorithm [14] was chosen as the compression/decompression algorithm, and AES algorithm [15] was chosen as the encryption/decryption algorithm. A log mechanism was implemented in backup site for batch automatic committing.

4.2 Experimental Setup

All experiments were performed on two single-core 2.66GHz Intel Xeon nodes. One acted as the primary site, and another acted as the backup site. Each machine has 2GB of memory and a hardware RAID-5 composed of six 37GB SAS disks. The two nodes were connected by a Gigabit Ethernet. In order to test three parallel models, we turned off log mechanism in backup site to put enough pressure on CPU. Both α and β were set to 0.5. The batch interval was initialized to 30ms. Iometer [16] was used to generate workload. Since write requests trigger remote mirroring module, we only test write workload. Unless otherwise expressly stated, sequential write workload was used. In order to eliminate the impact of asynchronous mode on performance, we recorded the experimental results after the performance curve reported by Iometer becomes stable over time. Each data point is the average of three samples.

4.3 Experimental Results

We first performed a baseline test. Fig.3 shows the result. "Pri" denotes the RAID-5 devices in the primary site and "back" denotes the RAID-5 devices in the backup site. "LVM" denotes the original volume in the primary site. "Async" denotes the asynchronous remote mirroring system without adaptive batching,

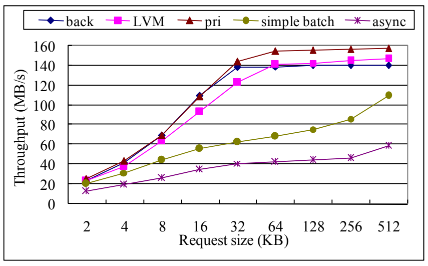

Fig. 3. Baseline test

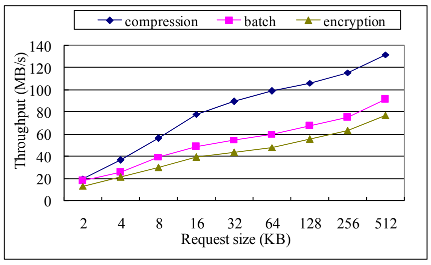

Fig. 4. Computation pressure

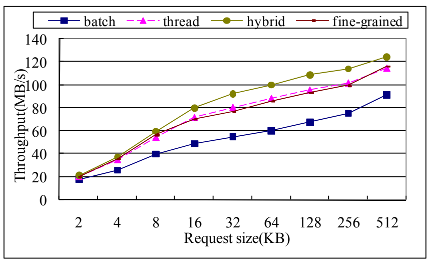

Fig. 5. Parallel models

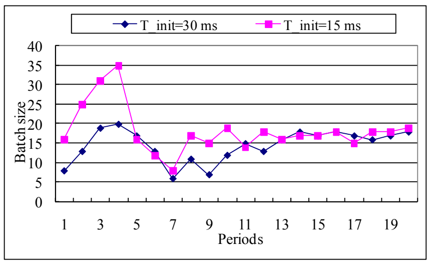

Fig. 6. Batch converging

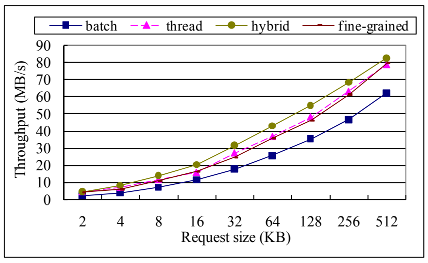

Fig. 7. 50% random write

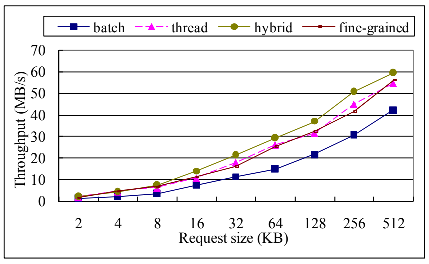

Fig. 8. 100% random write

“simple batch” denotes the one with adaptive batching but without data compression and encryption. We can see that adaptive batching algorithm improves performance greatly though it is far below raw devices yet.

Fig.4 shows the impact of data compression and encryption on performance. “Compression” means with data compression but without encryption, “encryption” means with encryption but without decompression, and “batch” means with both compression and decryption. As we expected, introducing compression improves performance significantly due to network traffic decreasing, and encryption impacts performance seriously due to high computational complexity. The complete version is between the other two versions.

Fig.5 shows how greatly the three parallel models improve performance. We can see that all three models accelerate the pipeline remarkably. Fine-grained pipelining and Multi-threaded pipelining exhibit almost the same performance. Hybrid pipelining improves the performance further because CPU is not overloaded - we observed an about 87% CPU occupation during hybrid pipelining test.

We also traced the batch interval (batch size). The results are showed in Fig.6. The batch interval was initialized to 30ms and 15ms respectively. The request size is 4KB. The batch sizes of the first 20 periods were recorded. We can see that the adaptive batching algorithm indeed coordinates the primary and the backup sites well. The batch interval converged to a stable condition quickly.

We also test random write performance. As Fig.7 and Fig.8 shows, compared with the result of sequential write test, the performance gap between the serial version and the parallel versions narrows. The reason is that the proportion of I/O part in the total running time increases and three parallel models accelerate only the computation part.

5 Conclusion and Future Work

In this paper, we presented a novel cooperative pipelining model for remote mirroring systems. Unlike traditional pipelining models, this new model considers the decentralization of processors. To solve the imbalance between the primary and the backup stages, we proposed an adaptive batching algorithm. To release the heavy load on CPU exerted by compression, encryption and TCP/IP protocol stack, we designed three parallel models: fine-grained pipelining, multi-threaded pipelining and hybrid pipelining. We implemented these techniques in our prototype. The experimental results showed that, the adaptively cooperative pipelining model balances the primary and the backup stages effectively, and the three parallel models improve performance remarkably.

All the experiments were performed in a LAN environment. Testing our prototype in the (emulated) WAN environment is an important future work. FEC (Forward Error Correcting) by using efficient erasure codes is also planned.

References

1. Patterson, R.H., Manley, S., Federwisch, M., Hitz, D., Kleiman, S., Owara, S.: SnapMirror: File-System-Based Asynchronous Mirroring for Disaster Recovery. In: Proceedings of the 1st USENIX Conference on File and Storage Technologies, FAST 2002, Monterey, California, USA, January 2002, pp. 117–129 (2002)
2. Weatherspoon, H., Ganesh, L., Marian, T., Balakrishnan, M., Birman, K.: Smoke and Mirrors: Reflecting Files at a Geographically Remote Location Without Loss of Performance. In: Proceedings of the 7th USENIX Conference on File and Storage Technologies, FAST 2009, San Francisco, California, USA, February 2009, pp. 211–224 (2009)
3. EMC SRDF - Zero Data Loss Solutions for Extended Distance Replication. Technical Report P/N 300-006-714, EMC Corporation (April 2009)
4. VERITAS Volume Replicator (tm) 3.5 Administrator's Guide (Solaris). Technical Report 249505, Symantec Corporation, Mountain View, CA, USA (June 2002)
5. Secure Data Protection With Dot Hills Batch Remote Replication. White Paper, dot Hill Corporation (July 2009)
6. Ji, M., Veitch, A.C., Wilkes, J.: Seneca: Remote Mirroring Done Write. In: Proceedings of the General Track: 2003 USENIX Annual Technical Conference, San Antonio, Texas, USA, pp. 253–268 (June 2003)
7. DFSMS/MVS Version 1 Remote Copy Administrator's Guide and Reference 4th edition. Technical Report SC35-0169-03, IBM Corporation (December 1997)
8. HP OpenView continuous access storage appliance. White Paper, Hewlett-Packard Company (November 2002)
9. Liu, X., Niv, G., Shenoy, P.J., Ramakrishnan, K.K., van der Merwe, J.E.: The Case for Semantic Aware Remote Replication. In: Proceedings of the 2006 ACM Workshop on Storage Security and Survivability, StorageSS 2006, Alexandria, VA, USA, October 2006, pp. 79–84 (2006)

10. LVM, http://sources.redhat.com/lvm/
11. Breuer, P.T., Lopez, A.M., Ares, A.G.: The Network Block Device. Linux Journal 2000(73), 40 (2000)
12. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing, 2nd edn. Addison-Wesley, Essex (2003)
13. Keeton, K., Santos, C.A., Beyer, D., Chase, J.S., Wilkes, J.: Designing for Disasters. In: Proceedings of the 3rd USENIX Conference on File and Storage Technologies, FAST 2004, San Francisco, California, USA, March 2004, pp. 59–72 (2004)
14. Welch, T.A.: A Technique for High-Performance Data Compression. IEEE Computer 17(6), 8–19 (1984)
15. NIST Advanced Encryption Standard (AES). Federal Information Processing Standards Publication (2001)
16. Iometer, http://www.iometer.org/