Software RAID Technology for cluster environments

Yoshitake Shinkai, Tetsutaro Maruyama, Naomi Yoshizawa
Fujitsu Laboratories LTD.
Kawasaki, Japan
{shinkai, tetutaro, yszw}@flab.fujitsu.co.jp
tel +81-44-754-2850
fax +81-44-754-2851

Abstract
In this paper, we present a new software RAID technology called file RAID targeted at cluster environments where nodes share files through SAN (Storage Area Network). In it the file system manages separate disks and places user data over them redundantly by itself instead of by underlying device drivers while replicating the metadata on multiple nodes. This schema results in high performance by adopting the following techniques only applicable to the file system layer such as aggressive caching, reduced recovery logging and dynamic RAID type selection based on file size. It also eliminates conventional spare disks. We implemented the file RAID feature in our cluster file system HAMFS. The measurement result demonstrates a good scalability as well as good performance even for random writes for a RAID5 file.

1 Introduction

As the acceptance of SAN in the market expands, sharing files placed on disks directly through high-speed SAN becomes vital. To this end, numerous researches including our previous works [4,9] for developing cluster file systems have been conducted lately [8,10]. While cluster file systems offer improved performance over traditional network file systems by eliminating extra network overhead, large installations long for more performance utilizing multiple disks shared among nodes. RAID [11] is an efficient technology for aggregating disks as it not only improves I/O performance by placing data in a striped fashion but also improves reliability. Traditionally, the software RAID has been implemented by device drivers exporting ordinary single disk functionality. This schema has several advantages as it offers a RAID capability to applications using raw devices as well as file systems. But, providing a software RAID capability using a device driver poses several issues especially in a cluster file system environment.

[**Double Address Translation**] The physical block address associated with a user file is translated through two address translation table: logical file block # to logical device block # provided by a file system and translation from logical device block # to physical device block # provided by a device driver. This complexity leads to a vulnerability to disk failures and weak performance.

[**Double Locking**] The device driver must implement a distributed locking for protecting parity calculation. The lack of serialization causes collapsing data, when multiple nodes try to write different blocks in the same stripe line at the same time even if block-level serialization is conducted by the upper layer of cluster file system. This results in an extra communication overhead, degrading overall system performance.

[**Lack of caching**] As device driver must return after stabilizing a requested write data, additional log I/Os to protect data from node crashes and for parity calculations including a notorious parity recalculation, two reads and two writes, must be done in a synchronous way. These synchronous I/O operations slow response time considerably harming the overall system performance.

[**Lack of knowledge about target blocks**] Since the device driver does not see connections between requested blocks, it must process every write request in the same way including parity recalculation. For instance, new blocks allocated to a file but not recorded in the associated metadata need not be protected during write operation. In the event of a node failure, these blocks would be discarded silently.

To address above problems, in file RAID the file system manages multiple disks by itself and provides RAID capability by placing data on separate disks redundantly.

[**Single Translation**] The disk blocks storing user data and associated parity are allocated from separate disks selected dynamically per file base. An inode representing a file has a btree data structure containing physical block addresses indicating where the file data is placed (extent information). This eliminates the need of additional address translation, reducing complexity and improving performance.

[**Single locking**] A data token associated with a range of file blocks of a particular file is maintained to ensure consistency. For a RAID file a unit of token is extended to stripe granularity. This assures correct parity calculation without extra locking mechanism.

[**Aggressive caching**] As write requests typically complete instantly after caching data leaving actual writing to disks in background. Parity calculation and additional logging for crash recovery also may be performed asynchronously in background without affecting the response time. Moreover, in the case old image of disk blocks necessary for parity recalculation is present in the cache, the file RAID may bypass additional read operations to the disk.

[**Optimization using the knowledge concerning target blocks**] The knowledge about the contents of each physical disk blocks allows the file system to reduce the overhead inherent in conventional RAID processing.

First, the file RAID has a capability of choosing RAID type per file, RAID1 or RAID5 dynamically. For metadata having random access characteristic, the file RAID applies RAID1 to improve performance. For short files having a single file system block, the file RAID selects RAID1, storing data on two blocks on distinct disks. Otherwise it stores data and parity on multiple disks according to RAID5 schema. Disk selection is performed dynamically to balance loads of each disk.

Second, the file RAID allows installations to specify RAID type explicitly per file basis, non- RAID, RAID0, RAID1 or RAID5 according to their performance and reliability

requirements without compromising a single file system capability. If RAID5 is specified, dynamic type selection mentioned above is performed.

Third, the file RAID may bypass a RAID logging if blocks being written to disk have not been registered in the associated inode yet. Moreover, for most sequential writes expensive parity recalculation processing is eliminated even if interleaved by requests to other files. This holds true typically for creating a new file and reduces software RAID overhead significantly in a typical UNIX environments where sequential write is dominant [3].

Fourth, the file RAID significantly reduces impedance mismatch affecting on the performance [14] like alignment problems in terms of block size as it scatters data across multiple physical disks and sees their performance characteristics.

Fifth, the dynamic disk selection feature of the file RAID allocating a set of consecutive disk blocks on separate disks for improved performance and reducing control space overhead has the similar advantage as the RAID0+1 type has in terms of scattering loads across multiple disks for improved performance.

Finally, the file RAID reduces the rebuild time in the event of disk failure, since only allocated portion of space are recovered using parity. In contrast almost all blocks must be rebuilt in the case of conventional software RAID, as device drivers cannot see which blocks are currently active.

[**Spare-less**] As the file RAID controls the utilization of underlying disks by itself, recovery from disk failure may be executed by allocating alternate extents on normal disks dynamically. This mechanism eliminates the need of spare disks necessary for conventional RAID.

The rest of paper is organized as follows. After surveying related works in the following section, the basic software structure is presented in section 3. The section 4 describes recovery processing. Then in section 5 we outline our preliminary measurement result, and conclude in section 6.

2 Related Work

Several distributed file systems support RAID capability by itself as well as sharing data among nodes. [1,12] They are variants of log structured file system [13], in which striping was done in device level disallowing dynamic RAID type selection like the file RAID. This type of structure needs more complicated logic especially for load balancing among nodes and recovery from node crash compared with our design. Furthermore it is targeted at network wide striping and differs substantially from our research targeting at SAN environment.

Several hardware RAID systems support dynamic RAID type selection [6,7,15]. But they initially store data using RAID1 schema and adapt RAID type later according to access

statistics, while file RAID chooses automatically RAID type from the outset according to the characteristic of data and the size.

[2] had been performed simultaneously with our work and presents similar concept. While it says some common conclusion with us, it is a simulation work ignoring implementation details such as crash recovery. We have implemented an actual cluster file system and benchmarked it.

3 Overall Design

The basic structure of HAMFS is illustrated in figure 1. The metadata is managed centrally by a single HAMFS (called Name Server) and replicated onto another node (secondary Name Server). HAMFS running on each node gets file information including extent information from the Name Server and access user data on data volumes directly. For reducing communication overhead and the number of disk I/O, HAMFS caches user data and file information in its local memory. Consistency among cached data is guaranteed using tokens maintained on the Name Server. Space allocation is performed locally and in batch typically at file close time from reserved-pool pre-allocated in advance using space-reserving function offered by the Name Server and notifies the Name Server of committed results. In this paper we focus on the file RAID processing. For more general information of HAMFS refer to [4].

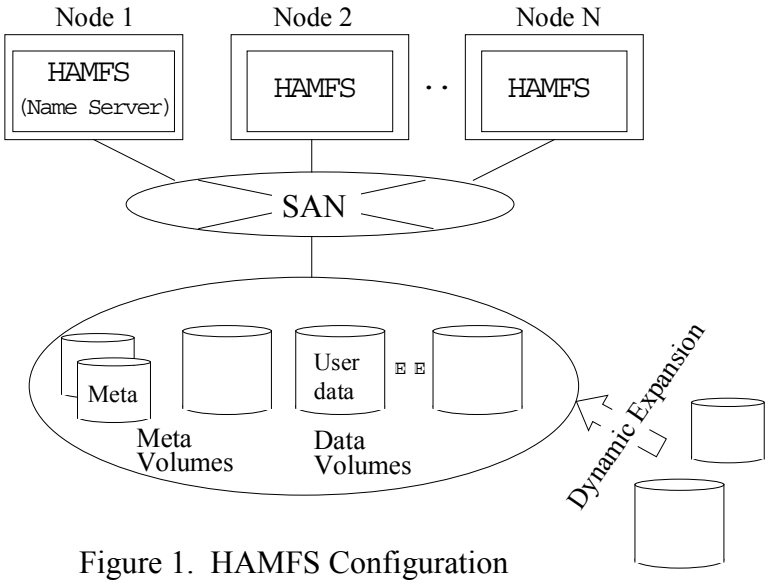

Figure 1. HAMFS Configuration

3.1 Space allocation

The disk space committed to a file is registered in a btree data structure stored in the associated inode, which has a set of quadruples representing extents; file offset, disk identifier, start disk block number, number of blocks. For a RAID file, HAMFS allocates extra extents containing parity data. Figure 2 illustrates the relation between file logical blocks and physical disk blocks assuming a RAID5 file is striped across two disks excluding parity. The file contains (2s x (n1+n4)) bytes of data, where “s” denotes stripe unit size in bytes that is equal as the file system block size and “n1”,”n4” represent the

number of blocks.. Five extents, d1-1, d1-21, d1-22, d2-1, d2-2 in respective, are assigned for user data and two extents, p1, p2 are allocated for parity data. The "dx-y" denotes an extent backing "Dx-y" part of user data. The inode associated with this file contains 7 btree entries (5 data extents and 2 parity extents).

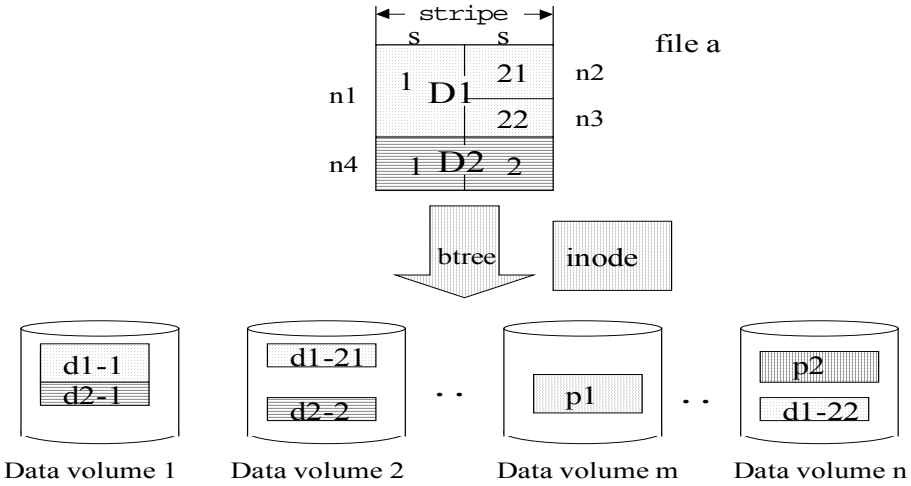

Figure 2. Data layout for a RAID file

For preserving available disk space used for creating RAID files and for recovery from disk crash, it is essential to balance the amount of available space on each disk. On the other hand, for improved performance and for reduced btree control space overhead, assigning larger extents on the same disk are preferred. To compromise these contradicting requirements, current implementation uses RAIDRAIO parameter that defines an upper threshold of maximum size of extent in terms of the amount of available space on the disk. If the size of extent allocated to a file exceeds the value calculated by this parameter and the amount of available space, separate extent on other disks is allotted instead of assigning consecutive disk space on the same disk.

3.2 Dynamic RAID type Selection

HAMFS allots extents for a file on demand on either file close, cache shortage or eviction of token. For a RAID file, it allocates parity extents in addition at the same time. When a stripe line has a single active logical block HAMFS write the cached data both on data disk block and parity disk block. This results in RAID1 without extra parity calculation. If two or more active logical blocks are present in the stripe lines it follows RAID5 fashion. This results in dynamic RAID type selection according to the file size. As small files are dominant in terms of the number of files while large files hold most of spaces in typical UNIX data installations [3], this dynamic RAID type selection leads to better performance with small space overhead. For instance assume 20% of space are used for small files and stripe width is 4. The space overhead of the file RAID under this condition is close to 40% (20% for RAID1 and 80x1/4 % for RAID5) in contrast 100% of RAID1 where all files are stored in RAID1 fashion and 25% of full RAID5.

3.3 Caching

HAMFS caches user data as long as possible to reduce the number of I/O operations. Consistencies among caches present on multiple nodes are preserved using data tokens. Data token is maintained per file block basis for normal files. In case of RAID files HAMFS extends a range of token to stripe granularity. This ensures correctness of parity processing. In addition, the escalation feature that the Name Server replies a full rage of token covering the whole file in case no contention exists minimizes locking overhead considerably. HAMFS tries to allocate a consecutive block on each disk to reduce the number of I/O operations using clustering. For a typical large sequential write the file RAID calculates parity without parity recalculation, as all data in a stripe are present in cache. For reducing memory copy operations in the RAID processing, existing HAMFS caching structure is extended so that a single cache entry can correspond to multiple disk blocks and a single file block may have multiple associated cache entries.

3.4 RAID logging

Before writing back the cached data associated with a RAID file onto disks, HAMFS creates a RAID log record identifying the stripe lines, which are I/O in progress and might remain inconsistent upon node crash. This log record contains the following information: write identifier, inode number, file offset, size and completed identifier. The write identifier is a unique sequence number, which identifies this I/O operation. The file RAID maintains write identifiers associated with write I/O operation in progress. Upon completion of I/O it updates the “completed identifier”, which represents all write I/Os associated with numbers smaller than this value have completed. The “completed identifier” is used to discard stale log records for reducing recovery time and for reuse of log area. When cached data are coalesced to reduce the number of I/O operations only one log entry is created. In addition the RAID logging is skipped if the stripe lines have no data registered in the associated inode. Node crash during writing this type of extents onto disks leaves no inconsistency because all corresponding disk blocks will be returned to free state at the crash recovery. The above schema reduces RAID logging overhead significantly against conventional software RAID as described before.

2.5 Early commit

RAID log is maintained per node basis and stored in a local RAID log file. But in a cluster environment where multiple nodes are alive, RAID log is transferred to another node’s memory instead of writing onto the RAID log file for shorting RAID logging I/O time. To preserve log data maintained in local memory upon power failure the file RAID uses UPS and saves the contents of local memory onto disk. This feature allows the file RAID to provide RAID capability without special hardware such as NVRAM.

4 Recovery processing

4.1 Crash recovery

When a node crashes an alive-node that holds the associated RAID log restores consistency of potential inconsistent stripe lines. The restoration is performed by reading user data on the stripe lines identified by RAID log, calculating parity data from them and then restores new parity data.

4.2 Disk Failure recovery

When a read access to a disk fails the file RAID determines the failure type by reading diagnostic blocks located at the predefined location on the disk. If the diagnosis completes successfully it proceeds to block recovery. Otherwise the file RAID consults the Name Server whether the failure is up to the I/O path from the current node to the disk or not. Upon this request the Name Server inquires other nodes if the disk is accessible from the node. In the case of successful return from this inquiry the Name Server acknowledges the original node with a path failure, otherwise reply with a device failure.

[Block failure] In the event of a block failure, the file RAID allocates an alternate block typically on the same disk and restores original data on the failed block using parity. Then it notifies the Name Server of switching the block. The Name Server updates the associated inode reflecting new block, leaving the failed block in "permanent error" state to prevent further allocation

[Path Failure] In the event of a path failure, the file RAID acknowledges the user program with an I/O error.

[Device Failure] If device failure is replied from the Name Server, the file RAID initiates a device recovery. The device recovery first scans inodes and finds extents on the failed disk unit. When a extent on the failed disk is present, it allocates alternate extents on unique disk that does not match with disks used for the affected stripe lines and recover data on them. Then, it notifies the Name Server of alternate extents for replacing extent information in the inode. Inode maps, which exists per data volume basis and identifies sets of file using the data volume are used to shorten above search.

In either type of recovery alternate extents are allocated on alive-disks on demand. So, no spare disks used for conventional RAID is required.

5 Measurement Results

To evaluate our prototype we created two synthetic workloads. The first is a sequential I/O and the latter is a random I/O for a large file.

5.1 Measurement Environment

We used two nodes connected through fast Ethernet, which runs the Name Server and the normal HAMFS respectively. Four disks are connected to the second node, on which benchmark jobs run, through two Ultra160 SCSI buses two disks per a SCSI bus. RAID log is sent to the first node from the second node. The detail of the measurement configuration is shown in Table 1.

5.2 Sequential I/O

Figure 3 illustrates the result of sequential write access as a function of request size. For this measurement we made a simple benchmark program creating a 100MB of file. Right figure shows a base performance for this measurement. Non RAID denotes an ordinary file and Raw refer to raw I/O. Raw-n represents the aggregated performance of n Raw I/O

jobs executed concurrently. As seen in the figure, while the sustained performance of measured disk is 29MB/s, aggregate performance is limited to around 65MB/s.

	type	OS	Memory	I/O	CPU	LAN
Node1	Name Server	SunOS R5.6	32MB	-	PentiumII 200MH	Fast Ether
Node2	Normal	FreeBSD 4.0	64MB	Adaptec-39160, Quantum ATRUS V9.1 (29MB/s, 6.3 ms) x 4units	PentiumIII 500MHz	

Table 1. Measurement Configuration

The left figure depicts the performance of the file RAID. RAID0-n denotes a RAID0 file with stripe width n. RAID5-n refers to a RAID5 file with stripe width n excluding parity portion. This figure shows the write performance scales according to the stripe width, RAID1 < RAD5-2 < RAID5-3. The reason why RAID1/5 performance is not good as the corresponding RAID0 performance is the upper limit of the aggregate performance, not parity processing overhead. Some ratio of bandwidth is used for parity data. For instance, RAID1 uses 57.4MB/s aggregate bandwidth (2 x 28.7MB/s). Meantime RAID5-2 uses 58.2MB/s (3/2 x 38.8 MB/s) aggregate bandwidth, respectively. This aggregate bandwidth matches well the corresponding RAID0 performance, RAID0-2 and RAID0-3, respectively.

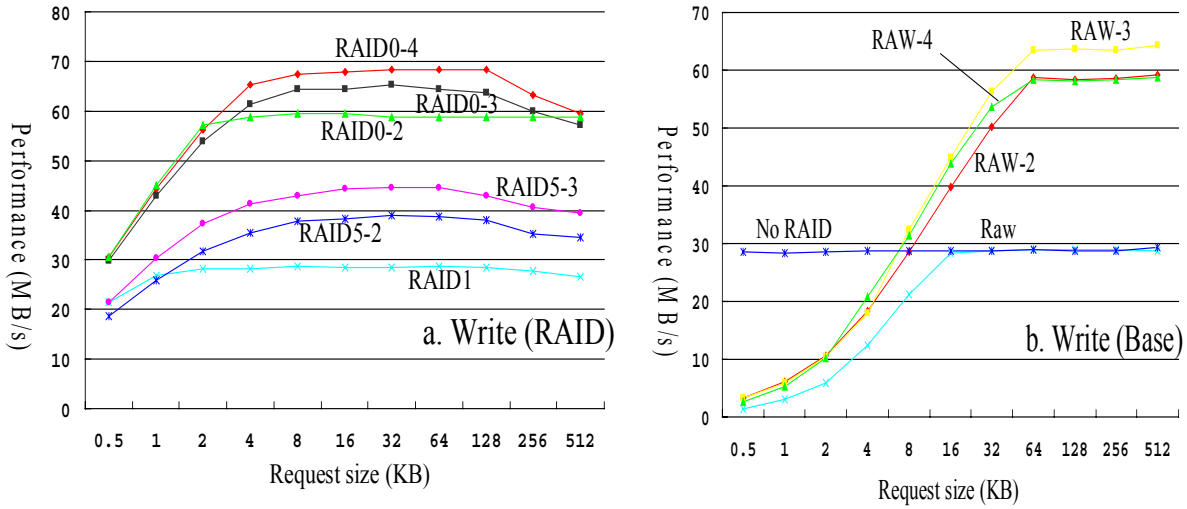

Figure 3. Sequential write access performance

Figure 4 shows the result of the sequential read access measurement as a function of request size. Performance scales well the number of disks, RAID1 < RAID5-2 < RAID5-3□Because of the same reason as write performance, scalability is limited by the aggregate performance. While RAID5 performance should be in line with RAID0

performance, as read access for RAID5 does not need parity processing, there is an apparent gap between RAID5 and associated RAID0. RAID5-3 shows better performance than RAID0-3, while RAID5-2 is worse than RAID0-2. We suspect the difference of read-ahead count might cause the imparity.

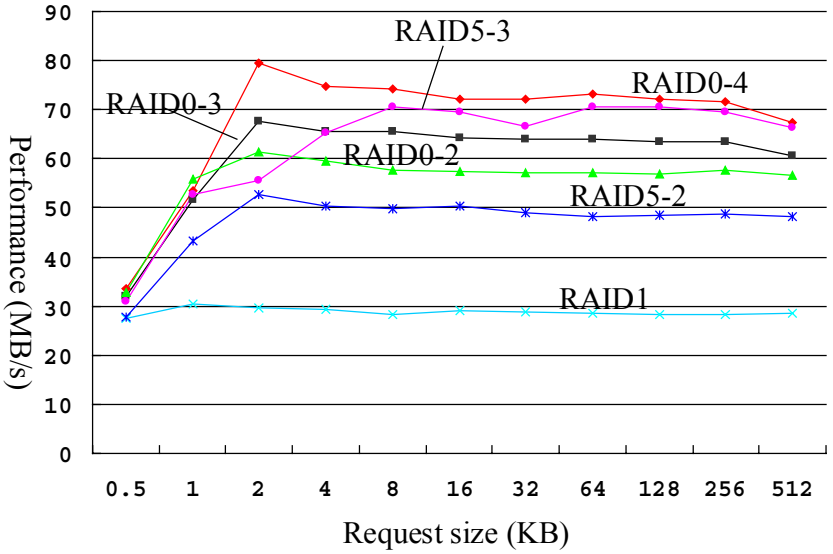

Figure 4. Sequential read access performance

5.3 Random Access

Figure 5 shows the result of a random I/O workload. In this measurement the benchmark job writes 100MB of blocks at random on an existing 100MB file. As parity recalculation is performed in background, the penalty of RAID5 is less apparent. RAID5-2 and RAID5-3 achieves 82% and 85% of associated RAID0 performance, compared with RAID0-2 and RAID0-3 in respective.

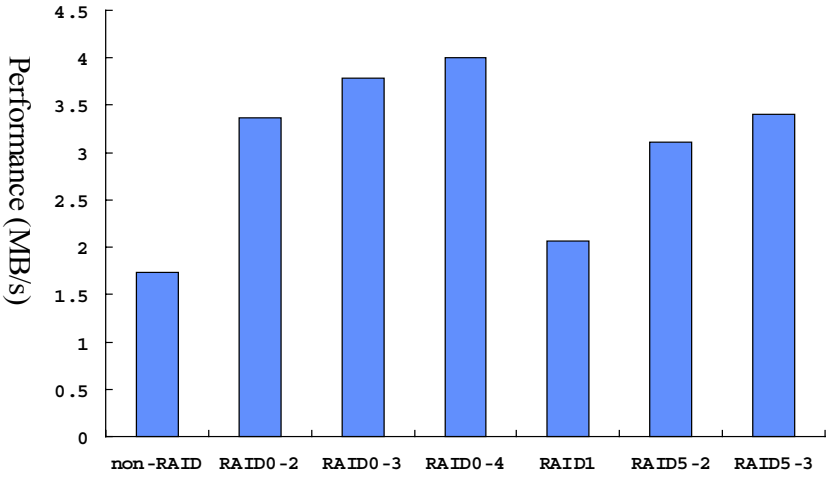

Figure 5. Random access performance

6 Conclusion

In this paper we demonstrated a new software RAID technique named file RAID, which solves most of problems inherent in conventional software RAID technology implemented as a device driver. The file RAID has the following unique characteristics. It is simple, suffers from fewer overheads, and facilitates system expansion, as it is integrated with a cluster file system offering multiple disks support capability. It works with only commodity hardware resulting in a reduced system cost. While hardware RAID evolves the software RAID would remain vital for increasing the performance by aggregating multiple hardware RAID or creating low cost data installations. This is especially important for storage appliance such as Network Attached Storage.

References

[1] John H. Hartman and John K. Ousterhout, “The Zebra striped network file system”, ACM Transactions on Computer Systems, 13(3) 274-310, Aug. 1995.

[2] J.-H. Kim and S.-W. Eom and S. Noh and Y.-H. Won, “Striping and Buffer Caching for Software RAID File Systems in Workstation Clusters”,19th IEEE International Conference on Distributed Computing Systems (ICDCS '99),Austin, Texas, May 1999

[3] M. Baker, J. H. Hartman, Michael, D. Kupfer, K. Shirriff and J. K. Ousterhout, "Measurements of a Distributed File System, Proceedings. of the Thirteenth ACM Symposium on Operating System Principles,198-212, Pacific Grove, California, October, 1991

[4] Y. Shinkai, Y. Tsuchiya, T. Murakami and J. Williams, HAMFS File System, Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems, Lausanne, Switzerland, Oct., 1999

[5] C. Chou, L. Golubchik and J. Lui, "Striping Doesn't Scale: How to Achieve Scalability for Continuous Media Servers with Replication", The 20th International Conference on Distributed Computing Systems (ICDCS 2000), Taipei, Taiwan, April, 2000

[6] N. Muppalaneni and K. Gopinath, A Multi-tier RAID Storage System with RAID1 and RAID5, Proceedings of 14th International Parallel and Distributed Processing Symposium (IPDPS'00), Cancun, Mexico, May, 2000

[7] J. Wilkes, R. Golding, C. Staelin and T. Sullivan, The HP AutoRAID hierarchical storage system, Proceedings of the fifteenth ACM symposium on Operating systems principle, 96-108, Cooper Mountain Resort, Colorado, December, 1995

[8] K. W. Preslan, A. Barry, J. Brassow et al, , Implementing Journaling in a Linux Shared Disk File System, Proceedings of the Seventeenth IEEE Symposium on Mass Storage Systems, 351-378, College Park, Maryland, March, 2000

[9] Y. Shinkai, Y. Tsuchiya, T. Murakami and J. Williams, Alternatives of implementing a Cluster File Systems, Proceedings of the Seventeenth IEEE Symposium on Mass Storage Systems, 161-177, College Park, Maryland, March, 2000

[10] R. L. Haskin and F. B. Schmuck, The Tiger Shark File System, Proceedings. of forty-first IEEE Computer Society International Conference(COMPCON), Santa Clara, California, February, 1996

[11] David A. Patterson, Garth Gibson and Randy H. Katz, A case for redundant arrays of inexpensive disks (RAID), Proc. of SIGMOD'88, 109-116, Chicago, Illinois, 1988
[12] T. E. Anderson, M. Dahlin and J. M. Neefe and David A. Patterson, Serverless Network File Systems, ACM Transactions on Computer Systems,14(1) 41-79, February,1996
[13] Mendel Rosenblum and John K. Ousterhout, The Design and Implementation of a Log-Structured File System, Proceedings of the Thirteenth ACM Symposium on Operating System Principles, 1-15, Pacific Grove, California, October, 1991
[14] T. M. Ruwart, Disk Subsystem Performance Evaluation: From Disk Drives to Storage Area Networks, Proceedings of the Seventeenth IEEE Symposium on Mass Storage Systems, College Park, Maryland, March, 2000
[15] K. Mogi and M. Kitsuregawa, Hot mirroring: a method of hiding parity update penalty and degradation rebuilds for RAID5, Proceedings of ACM SIGMOD'96, 183-193, Montreal, Quebec, June, 1996