Multilevel RAID Disk Arrays

Alexander Thomasian
New Jersey Institute of Technology - NJIT
athomas@cs.njit.edu

Abstract

Bricks or Storage Nodes - SNs holding a dozen or more disks are cost-effective building blocks for ultrareliable storage. We describe the Multilevel RAID - MRAID paradigm for protecting both SNs and their disks. Each SN is a k-disk-failure-tolerant kDFT array, while replication or ℓ-node failure tolerance - ℓNFTs paradigm is applied at the SN level. We provide the data layout for RAID5/5 and RAID6/5 MRAIDs and give examples of updating data and recovering lost data. The former requires storage transactions to ensure the atomicity of storage updates. In Conclusions we outline areas for further research.

1. Introduction

The volume of generated data requiring ultrareliable storage is increasing very rapidly. This is accompanied with rapid increases in disk capacities and reliability, e.g., 500 gigabyte disks and a *Mean Time to Disk Failure - MTTF* of 10^6 hours are being mentioned. *Redundant Arrays of Inexpensive/Independent Disks - RAID* paradigm [6] utilizes replication (RAID1) and Erasure Coding (RAID2-5 and RAID6) to attain higher reliability is not directly applicable to *Very Large Disk Arrays - VLDAs*. RAID1 was originally based on mirrored disks, but higher levels of redundancy may be viable, e.g., 3-way and 2×2, where two disks provide remote backup. RAID3-5 (resp. RAID6) dedicate one (resp. two) out of N disks to parity, so that the contents of one (resp. two) failed disks can be recovered. Hamming codes associated with RAID2 are not popular.

In addition to redundancy, striping was introduced in conjunction with RAID to balance disk loads by partitioning large datasets into smaller chunks of data called stripe units. Stripe units are allocated in a round-robin manner on the N disks of an array. RAID3-5 (resp. RAID6) dedicate one (resp. two) out of N stripe units to parity. Stripe units in a row are referred to as a stripe. Full-stripe writes can be carried out efficiently, since the parities are computed on the fly as successive stripe units are being written. RAID0 does not provide redundancy, but implements striping, which distinguishes it from *just a bunch of disks - JBOD*.

We envision VLDAs which are based on storage bricks or *Storage Nodes - SNs*. Each brick consists of a number of disks, an array controller, a cache, which may be partially nonvolatile - NVRAM, and interconnect capability [8]. Bricks are attached to a high bandwidth SAN (storage area network) for transmitting data, but an additional IP network is used to send/receive I/O commands and other messages.

RAID controllers at multiple levels are utilized in [2] to configure a *hierarchical RAID - HRAID* array. The higher and lower RAID levels of HRAID are specified as RAIDX(M)/Y(N), where X and Y denote the RAID level and M (resp. N) denote the number of virtual disks at the higher level, which are in fact SNs (resp. physical disks at lower level). We utilize *Multilevel RAID - MRAID* to organize numerous SNs constituting a VLDA into coherent groupings, but instead of a hierarchy of RAID controllers as in [2], we use *Data Routing Nodes - DRNs*, which may be hardened by replication, to maintain the relationship among SNs, e.g.. that two SNs are mirrors.

A recent study evaluates the reliability of arrays of storage bricks [16]. Each brick is a RAID0, RAID5, or RAID6 and a replication of one to three is considered across bricks [16]. A recursive solution method is developed to analyze the Markov chain model, which takes into account the MTTF of bricks and disks, and rebuild time which is affected by the disk transfer rate, network bandwidth, and disk capacity (and bandwidth) utilization. The performance metric of interest is "data loss events per petabyte-year". The sensitivity of *Mean Time to Data Loss - MTTDL* to MTTFs, rebuild block size, and link speed is quantified. A study dealing with reliability mechanisms in very large storage systems utilizes Markov chain models with repair to compare the MTTDL in 2- and 3-way mirroring and mirrored RAID5 arrays [24].

The paper is organized as follows. Sections 2 and 3 discuss erasure coding and replication to protect disks and

SNs. We specify the layout of MRAID5/5 and MRAID6/5 and their operation from the viewpoint of updating parities and recovery. We introduce storage transactions to ensure correctness in updating parities, especially when the parities are distributed across nodes. In Section 4 we summarize the discussion and outline areas for further research.

2. Coding Inside Storage Nodes

There are N nodes each with d disks, so that the total number of disks is $D = d \times N$. The number of data disks per SN is n, the number of check disks is k, and the number of spare disks is s, so that $d = n + k + s$. We postulate minimal redundancy coding, so that k check disks imply a k-disk-failure-tolerant - kDFT array.

We consider k disk failure tolerant arrays - kDFTs with $0 \leq k \leq 3$: $k = 0$ corresponds to RAID0, $k = 1$ to RAID5, $k = 2$ to RAID6, and $k = 3$ is referred to as RAID7. RAID6 and RAID7 can be implemented via Reed-Solomon codes, as well as specialized parity codes such as EVENODD [3],[4] and *rotated diagonal parity - RDP* parity layouts [7]. RM2 which tolerates two disk failures [13], but incurs a higher level of redundancy than RAID6 and exhibits a poor performance with two disk failures [19] is not considered further.

Rebuild is the systematic reconstruction of the contents of a failed disk on a dedicated spare disk or spare areas distributed among all disks. The latter is preferable to the former, since disk bandwidth is wasted otherwise [18]. The parity sparing method [5] combines two RAID5 disk arrays into one RAID5 by coalescing their parities, i.e., using the parities of one array as the spare areas of the other. Similarly, when a single disk fails in RAID6, the Q parities can be used as spare areas to convert it to a RAID5 (with P parities only) operating in normal mode. A further conversion from RAID5 to RAID0 is possible.

When a block on a failed disk is accessed, RAID5 accesses corresponding blocks on all surviving disks to reconstruct it. Since each disk also has its own load, the load on the remaining disks is roughly doubled. The load on surviving disks is tripled when two disks fail. Clustered RAID selects a parity group size g, which is smaller than the number of disks d, so that the load increase in accessing a failed disk is given by the declustering ratio: $\alpha = (g-1)/(d-1)$ [12]. The load increase in RAID5 and RAID6 disk arrays processing read and write requests is given in [21].

When a spare disk or areas are available, rather than recreating data blocks on demand, the rebuild process systematically reconstructs successive data blocks of the failed disk on a spare disk [12]. RAID5 rebuild processing may be unsuccessful due to a second disk failure, which is highly unlikely unless disk failures are correlated, but rather when *latent sector failures - LSFs* are encountered. This is one of the justification for RAID6 [3].

Updating small data blocks in kDFTs is referred to as the small write penalty, since it requires $2(k+1)$ *read-modify-writes - RMWs* to update data and parity blocks [6]. RMW can be implemented as a request to read old data and parity blocks, followed by a disk rotation to write both. Given $d_{diff} = d_{new} \oplus d_{old}$ the new parity is computed at the disk holding the parity as: $p_{new} = p_{old} \oplus d_{diff}$. "Write holes" caused by system failures can be obviated by computing the parity at the disk array controller, which does logging for recovery [15]. This requires independent requests to read and write data and parity blocks. The reconstruct write method of updating parities is preferable in some cases [20].

3. Internode Level Replication and Erasure Coding

Replication or erasure coding across the SNs provides data protection for SNs as well as their disks, when local recovery is not possible. Note that the failure rate for SNs (excluding disk failures) may be higher than disks. External disk requests arriving at communication nodes are sent to DRNs (data router nodes), which are aware of data placement in the array. A DRN may consist of a cluster of DRNs for load-sharing, scalability, and fault-tolerance. A cluster of DRNs constitutes a data-sharing system [14]. Data in VLDA is partitioned into fragments and fragments are allocated to DRNs.

SN replication is preferable to erasure coding across SNs for high access rates. M-way replication increases access bandwidth by this factor. Read requests can be processed at any one of the M SNs, but updates should be written to all SNs, i.e., read-one, write-all paradigm [15]. With two-way replication when a single SN fails then the read load at the other SN is doubled. This problem can be dealt with by using a more sophisticated data allocation methods for mirrored data such as interleaved declustering, chained declustering, and group rotate declustering, whose reliability and performance is compared against basic mirroring in [22].

According to the linearizability correctness paradigm reads must access the latest version of written data [9], which is simple to implement when all requests to access replicated data are channeled through one DRN. Appropriate concurrency control methods for replicated data are required otherwise [15]. Since storage level replication and redundancies, i.e., parities, are not exposed to the application level, *storage transactions* in addition to application level transactions, are required to ensure correctness.

Erasure coding introduces less redundancy than replication and is applied to a subset of SNs, rather than all

SNs at once. We postulate c SN clusters, so that there are $M = N/c$ SNs per cluster with ℓ out of M check SNs. In addition to striping across the disks of an SNS, striping may be carried over across the disks in a cluster. Striping at this level results in a balanced load across the disks. The case $M = d$ (same number of clusters as disks) lends itself to a straightforward layout for parities, so we explore it first. We first set $k = \ell = 1$, which implies nested RAID arrays: RAID5(M)/5(d). Parities at the two levels are referred to as P and Q parities, respectively.

A dedicated SN which holds all Q parities has the disadvantages associated with RAID4 disk arrays: (i) In a write intensive environment to small blocks it may become a bottleneck. (ii) When all disk accesses are reads the check SN remains unutilized, unless there are SN failures. If all the blocks on the check SN are assigned to Q parities, then the Q parities will be unprotected against disk failures at that SN. If one of the disk at the check SN fails then the corresponding disks at the other SNs need to be read to reconstruct its contents (assuming spare space is available). An alternative solution is not to protect P parities and use the unutilized space at the check SN to local P parity blocks, so that they can protect Q parities. In other words, the Q parities at the check SN are treated as data blocks, which are protected by P parities.

Given the disadvantages of dedicating the full capacity of an SN to parity, we distribute the Q parities across M SNs, while the capacity equivalent of one disk per SN is already dedicated to P parities. The starting point is $M-1$ "data" disks and a single check SN with Q parities. The even distribution of Q parities over the M SNs is straightforward for $M = d$, in which case each SN allocates $1/d$ of its capacity to P parities and $1/d$ to Q parities. The two parities can be placed in parallel diagonals for load balancing purposes. P parities protect data blocks and also the Q parities in the same stripe, while Q parities apply across the SNs. A distributed sparing scheme is also applicable at the SN level.

Example 1. A disk array with four SNs and four disks: $M = d = 4$ is used as an example in this paper. Each block (or stripe unit) is specified as $x^m_{i,j}$, where $1 \le i \le d$ is the stripe or row number, $1 \le j \le d$ is the disk number, and $1 \le m \le M$ is the SN number. Only four rows are shown, since the pattern repeats itself. The Q parities are rotated from disk to disk and SN to SN to balance the load for updating disks.

To update $d^2_{4,1}$ we first compute $d^{2diff}_{4,1} = d^{2new}_{4,1} \oplus d^{2old}_{4,1}$ and next update its parities in parallel: $p^{2new}_{4,3} = p^{2old}_{4,3} \oplus d^{2diff}_{4,1}$, $q^{1new}_{4,1} = q^{1old}_{4,1} \oplus d^{2diff}_{4,1}$. We next compute $q^{1diff}_{4,1} = q^{1new}_{4,1} \oplus q^{1old}_{4,1}$ to update its P parity: $p^{1new}_{4,4} = p^{1old}_{4,4} \oplus q^{1diff}_{4,1}$.

Note that updating the Q parity associated with $p^2_{4,3}$ would result in cascading updates starting with: $p^{2diff}_{4,3} = p^{2new}_{4,3} \oplus p^{2old}_{4,3}$, $q^{3new}_{4,3} = q^{3old}_{4,3} \oplus p^{2diff}_{4,3}$. *In this implementation P parities protect Q parities, but this is not true vice-versa.* Note that this coincides with the second option of not protecting parities for a dedicated check SN.

Disk failures can be dealt with firstly inside an SN, but otherwise using the erasure coding scheme across SN. As an example of an SN failure we assume that SN_1 has failed and reconstruct its first stripe.

$d^1_{1,1} = d^2_{1,1} \oplus q^4_{1,1}$. $d^1_{1,2} = q^3_{1,2} \oplus d^4_{1,2}$. $q^1_{1,4} = d^2_{1,4} \oplus d^3_{1,4}$. $p^1_{1,3} = d^1_{1,1} \oplus d^1_{1,2} \oplus q^1_{1,4}$.

Just in case disk 1 or D_1 at SN_2 has also failed, it should be reconstructed before SN_1 can be reconstructed: $d^2_{1,1} = p^2_{1,2} \oplus q^2_{1,3} \oplus d^2_{1,4}$. In fact for higher efficiency it is best to interleave rebuilding at D_1 at SN_2 with rebuilding SN_1. **End of Example 1.**

Storage Transactions A race condition arises in updating $p^1_{1,3}$ when $d^1_{1,1}$ and $d^1_{1,2}$ are updated at the same time, since $p^1_{1,3}$ might be set to either $p^{1new}_{1,3} = d^{1old}_{1,1} \oplus d^{1new}_{1,1} \oplus p^{1old}_{1,3}$ or $p^{1new}_{1,3} = d^{1old}_{1,2} \oplus d^{1new}_{1,2} \oplus p^{1old}_{1,3}$, while $p^1_{1,3}$ should reflect the outcome of both updates: $p^{1new}_{1,3} = d^{1new}_{1,1} \oplus d^{1new}_{1,2}$. A similar situation arises when $d^1_{1,1}$ and $d^2_{1,1}$ are updated as far as the updating of $q^4_{1,1}$ is concerned.

The well-known solution to the lost update problem is to associate a transaction with each update. We use the strict *two-phase locking - 2PL* concurrency control method, where each transaction holds all of the locks to the end [15]. R and W stand for reading and writing of the data block in parentheses. Reads (resp. writes) are preceded by the acquisition of a shared (resp. exclusive). The NVRAM at each node can be used for logging for recovery [15]. A *two-phase commit - 2PC* protocol is required to ensure the atomicity of distributed transactions [15]. which result from updating Q parities, as shown below.

Transaction (update $d^{1new}_{1,1}$) = { $1 - R(d^{1old}_{1,1})$, $2 - W(d^{1new}_{1,1})$, $3 - d^{1diff}_{1,1} = d^{1new}_{1,1} \oplus d^{1old}_{1,1}$, $4 - R(p^{1old}_{1,3}), R(q^{4old}_{1,1})$, $5 - p^{1new}_{1,3} = d^{1diff}_{1,1} \oplus p^{1old}_{1,3}, q^{4new}_{1,1} = d^{1diff}_{1,1} \oplus q^{4old}_{1,1}, W(p^{1new}_{1,3})$, $6 - q^{4diff}_{1,1} = q^{4new}_{1,1} \oplus q^{4old}_{1,1}, R(P^{4old}_{1,4})$, $7 - p^{4new}_{1,4} = p^{4old}_{1,4} \oplus q^{4diff}_{1,1}$, $8 - W(p^{4new}_{1,4})$}.

Assuming that disk D_1 at SN_1 has failed then $d^{1old}_{1,1}$ is not accessible. A transaction attempting to read an inaccessible data block is aborted according to [1]. The system then issues a new transaction which includes a fork-join request to reconstruct $d^1_{1,1}$, i.e., $d^1_{1,1} = d^1_{1,2} \oplus p^1_{1,3} \oplus q^1_{1,4}$. A more sophisticated transaction processing mode is postulated in this study, so that SNs which upon receiving requests to read or write data blocks issue appropriate substransactions depending on the detailed state of the SN, which is only known locally. The transaction is aborted only when the requested block cannot be recovered.

Distributed dynamic locking with blocking (or wait on conflict) policy may lead to distributed deadlocks, whose

Node 1				Node 2				Node 3				Node 4			
$d^1_{1,1}$	$d^1_{1,2}$	$p^1_{1,3}$	$q^1_{1,4}$	$d^2_{1,1}$	$p^2_{1,2}$	$q^2_{1,3}$	$d^2_{1,4}$	$p^3_{1,1}$	$q^3_{1,2}$	$d^3_{1,3}$	$d^3_{1,4}$	$q^4_{1,1}$	$d^4_{1,2}$	$d^4_{1,3}$	$p^4_{1,4}$
$d^1_{2,1}$	$p^1_{2,2}$	$q^1_{2,3}$	$d^1_{2,4}$	$p^2_{2,1}$	$q^2_{2,2}$	$d^2_{2,3}$	$d^2_{2,4}$	$q^3_{2,1}$	$d^3_{2,2}$	$d^3_{2,3}$	$p^3_{2,4}$	$d^4_{2,1}$	$d^4_{2,2}$	$p^4_{2,3}$	$q^4_{2,4}$
$p^1_{3,1}$	$q^1_{3,2}$	$d^1_{3,3}$	$d^1_{3,4}$	$q^2_{3,1}$	$d^2_{3,2}$	$d^2_{3,3}$	$p^2_{3,4}$	$d^3_{3,1}$	$d^3_{3,2}$	$p^3_{3,3}$	$q^3_{3,4}$	$d^4_{3,1}$	$p^4_{3,2}$	$q^4_{3,3}$	$q^4_{3,4}$
$q^1_{4,1}$	$d^1_{4,2}$	$d^1_{4,3}$	$p^1_{4,4}$	$d^2_{4,1}$	$d^2_{4,2}$	$p^2_{4,3}$	$q^2_{4,4}$	$d^3_{4,1}$	$p^3_{4,2}$	$q^3_{4,3}$	$d^3_{4,4}$	$p^4_{4,1}$	$q^4_{4,2}$	$d^4_{4,3}$	$d^4_{4,4}$

detection and resolution incurs extra messages. Distributed concurrency control methods based on local resolution of lock conflicts (at the SNs) via aborts are appropriate in this case. Transaction aborts are affordable, since the probability of data conflict is very small and aborted transactions are restarted by the system. The wound-wait, wait-die [17] belong to this category. The time-stamp ordering method is the preferred concurrency control method according to [1].

The system then issues a new transaction which includes a fork-join request to reconstruct $d^1_{1,1}$, i.e., $d^1_{1,1} = d^1_{1,2} \oplus p^1_{1,3} \oplus q^1_{1,4}$. A more sophisticated transaction processing mode is postulated in this study, so that SNs which upon receiving requests to read or write data blocks issue appropriate substransactions depending on the detailed state of the SN, which is only known locally. The transaction is aborted only when the requested block cannot be recovered.

Distributed dynamic locking with blocking (or wait on conflict) policy may lead to distributed deadlocks, whose detection and resolution incurs extra messages. Distributed concurrency control methods based on local resolution of lock conflicts (at the SNs) via aborts are appropriate in this case. Transaction aborts are affordable, since the probability of data conflict is very small and aborted transactions are restarted by the system. The wound-wait and wait-die methods [17] belong to this category. The time-stamp ordering method is the preferred concurrency control method according to [1].

Example 2. There are $M = 5$ SNs and $d = 5$ disks per SN. Each SN is a RAID5: $k = 1$ and the parity is denoted by P. Across the SNs we have RAID6 with $\ell = 2$ and parities denoted by Q and S. Only the first row is shown for brevity.

$(d^1_{1,1}, d^1_{1,2}, p^1_{1,3}, q^1_{1,4}, s^1_{1,5})$, $(d^2_{1,1}, p^2_{1,2}, q^2_{1,3}, s^2_{1,4}, d^2_{1,5})$,
$(p^3_{1,1}, q^3_{1,2}, s^3_{1,3}, d^3_{1,4}, d^3_{1,5})$, $(q^4_{1,1}, s^4_{1,2}, d^4_{1,3}, d^4_{1,4}, p^5_{1,5})$,
$(s^5_{1,1}, d^5_{1,2}, d^5_{1,3}, p^5_{1,4}, q^5_{1,5})$.

If SN_1 and SN_2 fail we can reconstruct $d^1_{1,1}$ and $d^2_{1,1}$ using $q^4_{1,1}$ and $s^5_{1,1}$. $d^1_{1,2}$ can be reconstructed by noting that $d^1_{1,2} \oplus d^4_{1,2} = q^3_{1,2}$, although $s^4_{1,1}$ could also be used for this purpose. We similarly note that $d^2_{1,5} \oplus d^3_{1,5} = q^5_{1,5}$. Greater than k disk failure at an SN are special cases of full node failures and can be protected using Q and S parities. **End of Example 2.**

4. Conclusions

We have described the *Multilevel RAID - MRAID* paradigm to organize disks into reliable groupings in VLDAs. Replication at the higher level and a kDFT at the lower level is a simple organization to protect SNs at the higher level and disks at the lower level. Erasure coding at both levels results in significant savings in redundancy. We have specified two MRAIDs with RAID5 and RAID6 at the higher level and RAID5 at the lower level. A design decision made in this study is to protect parities across SNs with parities protecting the disk at each SN, but not vice-versa. Examples of updating parities and recovery from disk and SN failures are given. Combinations of replication and erasure coding can be used to provide fault-tolerance at three levels, inside SNs, across SNs in a cluster, and across clusters (replicated clusters).

We introduce the need for storage transactions to ensure correct interaction of read and write requests Distributed transactions are required across SNs and the problem is more challenging since we allow disk and SN failures. Given that the probability of conflicting updates is small, the preferred concurrency control method will minimize the number of internode messages, rather than just data conflicts.

To gain insight into MRAID reliability we have developed approximate models to compare the reliability of representative MRAID arrays combing mirroring with RAID5 and RAID6 [23]. The applicability of Markov chain models for estimating MRAID MTTDLs is also explored, but we have also resorted to simulation.

We are developing cost models for MRAIDs to evaluate the performance of the system, as was done in [19] for 2DFTS. Timing diagrams in the form of task systems will be used to determine the latency associated with more complex operations.

Operation in degraded and rebuild mode is also of interest. In both cases replication seems to preferable to erasure coding, because of the very high bandwidth requirements in the latter case for SN repair. Replication is a good substitute for the backup/restore paradigm to magnetic tapes, since the backup of very large disks takes excessive amounts of time.

References

[1] K. Amiri, G. A. Gibson, and R. Golding. “Highly concurrent shared storage”, *Proc. 20th Int’l Conf. Distributed Computing Systems - ICDCS*, Taipei, Taiwan, April 2000, pp. 298–307.

[2] S. H. Baek, B. W. Kim, E. Jeung, and C. W. Park. “Reliability and performance of hierarchical RAID with multiple controllers”, *Proc. 20th Annual ACM Symp. on Principles of Distributed Computing - PODC*, Newport, RI, August 2001, pp. 246–254.

[3] M. Blaum, J. Brady, J. Bruck, and J. Menon. “EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures”, *IEEE Trans. Computers TC-44*(2): 192–202 (February 1995).

[4] M. Blaum, J. Brady, J. Bruck, J. Menon, and A. Vardy. “The EVENODD code and its generalizations”, in *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, H. Jin. T. Cortes, and R. Buyya (editors), Wiley 2002, pp. 187-205.

[5] J. Chandy and A. L. N. Reddy. “Failure evaluation of disk array organizations”, *Proc. 13th Int’l Conf. on Distributed Computing Systems - ICDCS*, Pittsburgh, PA, May 1993, pp. 319–326.

[6] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. “RAID: High-performance, reliable secondary storage”, *ACM Computing Surveys 26*(2): 145–185 (June 1994).

[7] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. “Row-diagonal parity for double disk failure correction”, *Proc. 3rd Conf. File and Storage Technologies - FAST’04*, San Francisco, CA, March/April 2004.

[8] J. Gray. “Storage bricks - Keynote speech”, *First USENIX Conf. on File and Storage Technologies - FAST’02*, Monterey, CA, January 2002.

[9] M. P. Herlihy and J. M Wing. “Linearizability: A correctness criterion for concurrent objects”, *ACM Trans. Programming Languages and Systems - ASPLOS-12*(3): 463-492.

[10] C. R. Lumb, R. Golding, and G. R. Ganger. “D-SPTF: Decentralized request distribution in brick-based storage systems”, *Proc 11th Int’l Conf. Architectural Support for Programming Languages and Operating Systems - ASPLOS*, Cambridge, MA, October 2004, pp. 37–47.

[11] A. Merchant and P. S. Yu. “Analytic modeling of clustered RAID with mapping based on nearly random permutation”, *IEEE Trans. Computers TC-45*(3): 367–373 (March 1996).

[12] R. R. Muntz and J. C.-S. Lui. “Performance analysis of disk arrays under failure”, *Proc. 16th Int’l Conf. Very Large Data Bases - VLDB*, Brisbane, Queensland, Australia, August 1990, pp. 162–173.

[13] C.-I. Park. “Efficient placement of parity and data to tolerate two disk failures in disk array systems”, *IEEE Trans. Parallel and Distributed Systems TPDS-6*(11): 1177–1184 (November 1995).

[14] E. Rahm. “Empirical performance evaluation of concurrency and coherency control protocols for database sharing systems”, *ACM Trans. Database Systems TODS-18*(2): 333 - 377 (June 1993).

[15] R. Ramakrishnan and J. Gehrke. *Database Management Systems, Third Edition*, McGraw-Hill, 2003.

[16] K. K. Rao, J. L. Hafner, and R. A. Golding. “Reliability for networked storage nodes”, *IBM Research Report RJ10358*, Almaden Research Center, San Jose, CA, September 2005.

[17] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II. “System level concurrency control in distributed database systems”, *ACM Trans. Database Systems TODS-3*(2): 178–198 (June 1978).

[18] A. Thomasian and J. Menon. “RAID5 performance with distributed sparing”, *IEEE Trans. Parallel and Distributed Systems TPDS-8*(6): 640-657 (June 1997).

[19] A. Thomasian, C. Han, G. Fu, and C. Liu. “A performance tool for RAID disk arrays”, *Proc. Quantitative Evaluation of Systems - QEST’04*, Enschede, The Netherlands, September 2004, pp. 8–17.

[20] A. Thomasian. “Read-modify-writes versus reconstruct writes in RAID”, *Information Processing Letters - IPL 93*(4): 163-168 (February 2005).

[21] A. Thomasian. “Access costs in clustered RAID disk arrays”, *The Computer Journal 48*(11): 702-713 (November 2005).

[22] A. Thomasian and J. Xu. “Reliability and performance of mirrored disk organizations”, revised and resubmitted November 2005.

[23] A. Thomasian. “Shortcut method for reliability comparisons in RAID”, *The Journal of Systems and Software*, to appear 2006.

[24] Q. Xin, E. L. Miller, T. J. E. Schwarz, D. D. E. Long, S. A. Brandt, and W. Litwin. “Reliability mechanisms for very large storage systems”, *Proc. 20th IEEE/11th NASA Goddard Conf. on Mass Storage Systems - MSS’03*, San Diego, CA. April 2003, pp. 146–156.