Construction of Efficient OR-based Deletion–tolerant Coding Schemes

Peter Sobe and Kathrin Peter
University of Luebeck
Institute of Computer Engineering
{sobe|peterka}@iti.uni-luebeck.de

Abstract

Fault–tolerant data layouts for storage systems are based on the principle to add redundancy to groups of data blocks and store them in different fault regions. Commonly, XOR-based codes are used with an optimal redundancy overhead but with the disadvantage of relatively high calculation costs. We present a scheme that encodes input data in a highly redundant code and exploits that redundancy for a fault tolerance scheme. It allows to recalculate missed bits in fewer steps than needed for XOR-based schemes. This simple and efficient en- and decoding requires an appropriate hardware architecture or a highly parallel microprocessor architecture. Particularly, disjunctions over many input bits must be calculated, e.g. by wide OR-gates or busses that are driven by multiple logic input lines. The high redundant encoding is combined with data compression for separated data streams, each stream dedicated to a storage device. The compression not only eliminates the introduced redundancy of the used code, it also eliminates redundancy in the input data.

1. Introduction

Classical fault–tolerant data layout schemes employ redundancy schemes that are independent of the input data itself. The schemes add redundancy and use it for error detection, error correction or deletion correction. Fault–tolerant storage based on multiple disks or distributed systems mostly employ deletion correction codes. These codes are based on the knowledge about which storage is faulty and exploit that for efficient coding.

Redundancy is the key point for fault tolerant coding. Redundancy must (i) be structured, i.e. built by functional dependencies from different code word regions and (ii) must be stored in fault regions that are different from the data storage. This is the rea-

son why redundancy of data cannot be used directly. The typical case is to eliminate data redundancy at first, i.e. saving data in a compressed format (e.g. pdf, jpg, mpg, mp3) and then to add new structured redundancy. Alternatives are limited to a few cases when data redundancy can be exploited for efficient encoding. For instance, added redundancy can get restricted to significant parts of data, whereby less significant parts may be lost in case of faults. Examples were given with MPEG–aware encoding[3] or a fault-tolerant multi-server VOD architecture [4]. Roughly, one may classify common redundancy schemes in:

- **Data Replication**: Redundancy consists in complete copies of the data words. The copies must be stored each in different fault regions, and their number must be $f + 1$ or larger, when f is the number of fault regions. A 2-times replication is used for the RAID level 1 and indirectly for levels 0+1 and 1+0.

- **Simple parity-based:** Structured redundancy is added by using functional dependencies based on the bitwise exclusive OR operation (XOR). This principle is used for RAID levels 3,4 and 5, mainly to tolerate single faults. Multiple one- and multi-dimensional schemes (e.g. [5]) can be applied to tolerate situations with many failed storages. For all schemes of this class, a single input bit from each fault region is used in a calculation. When more than one parity is calculated, bits may be included in more than one calculation.

- **Sophisticated parity-based:** Horizontal-diagonal schemes use another approach. Such schemes are based on XOR and use different bits on storages for different parity calculations that build check information on two or more disks. An efficient code of this class is the Even-Odd-Scheme [1, 2].

- **Reed-Solomon-redundancy**: (described in [7])

This code is used to tolerate m faults among n data- and m redundancy storages, with $m > 0$, and $m < n$ to be efficient. It requires more complex arithmetics than bitwise XOR. Recently, research is focusing on XOR-based implementations of the operations within these arithmetics, e.g. [6].

- **Hamming-Codes:** These codes are known from channel coding and can also be used to correct deletions. Nevertheless, due to the fault characteristic of deletion instead of corruption, this kind of code is not widely used.

An example for a system that employs full data replication is CEFT-PVFS [10], an extension of PVFS-1 that mirrors each block in the distributed system. A system that implements block-wise data distribution combined with configurable XOR and R/S redundancy is NetRAID[8, 9]. Figure 1 depicts the commonly applied way for data storage. Often, original data is compressed in a first step. Then structured redundancy is added, e.g. adding XOR-parity to each block. To form independent fault regions, blocks of data and redundancy are distributed across several independent storage devices. Principally, codes with low storage overhead are used.
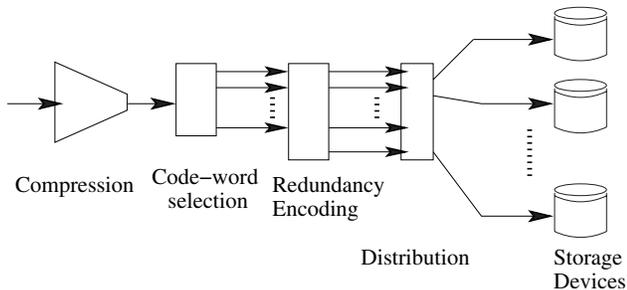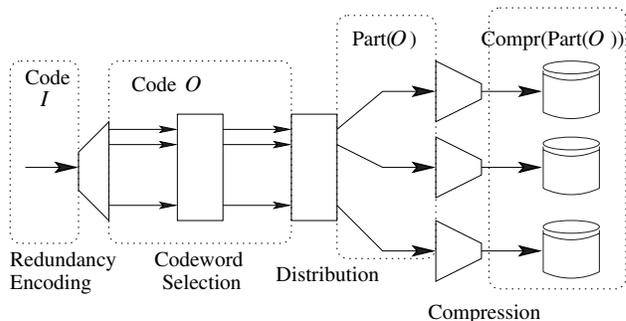

Figure 1. Common data path to the storage medium

Another way for storing data in a space–efficient and fault–tolerant way is to encode input code I in another code O that already contains structured redundancy. This code is directly employed for distribution of data and has already to represent the independent fault regions of the storage system. From that point, data can be distributed onto several independent storages providing fault tolerance automatically. Recovery is based on the error correction or deletion tolerance capabilities of the chosen code O. The code may employ a higher degree of redundancy than an optimal code, in order to allow a simple and efficient encoding. A main fraction

of redundancy is removed by compressing the data assigned to a storage device. This principle is illustrated in figure 2.

Compression in the scope of a storage device is capable to remove redundancy of the original input data (the redundancy in I) as well as the redundancy caused by the data symbol distribution in O. So, the proposed scheme promises to reach nearly the same space-efficiency as the common methods. With a more efficient en- and decoding it is worth to be analyzed and tried to get used in distributed storage systems.

Figure 2. Alternative data path to the storage medium

2 Redundant Coding

2.1 Deletion–Tolerant Codes

The hamming distance is a lower bound for the number of tolerable bit errors in a code. When each two code words differ in at least d_{min} bits, one can determine the number of faults that are safely detected and corrected. A decoder that exclusively detects errors, is able to detect $d_{min} - 1$ erroneous bits. When the decoder corrects errors, $(d_{min} - 1)/2$ bit errors can be corrected.

These numbers are lower bounds, i.e. for particular position patterns there exist codes that tolerate a higher number of errors[1].

When positions of errors are known, i.e. which disks are failed, codes offer better properties and $d_{min} - 1$ errors can be corrected. The hamming distance of a

---

[1] This ability is caused by a particular structure of the redundancy. For instance a code with 8 data bits $d_0, \ldots, d_7$ and two parity bits $p_0 = xor(d_0, \ldots d_3)$ and $p_1 = xor(d_4, \ldots, d_7)$ also has a hamming distance of two and thus a single error is detectable. But also when two errors in d_x and d_y with $x \in \{0, \ldots, 3\}$ and $y \in \{4, \ldots, 7\}$ occur, this situation can be tolerated because of the error position patterns and the redundancy structure.

simple binary parity code is $d_{min} = 2$. Another code with the same hamming distance of $d_{min} = 2$ is a (1 out-of N) code. Thus, in both codes a single error can be corrected, when the failed storage (i.e. the deleted bit) is known for decoding.

2.2 XOR-based codes

Each storage medium or each node in a distributed storage system can be seen as a fault region, i.e. these regions fail independently of each other. The bits of a code word must be interleaved across these fault regions. Assuming N different fault regions that are used to store the data word without parity, the number of computation steps for a simple XOR calculation is given in the following.

- O(N) when XOR is computed sequentially. Usually, the word parallelism of a microprocessor allows to encode w code words of N bits each in O(N) time steps. Common microprocessors provide bit-wise XOR instructions for data words of width w.

- O(ldN) when XOR is calculated along a reduction tree with parallel logic operations. The word parallelism of a microprocessor can be exploited to calculate w' XOR trees in parallel. In O($log_2 N$) steps w' codewords of N data bits each can be calculated.

For hardware acceleration the minimal cost is in the order O(ldN) with a word parallelism of w'. This means that a single parity bit, as well as a group of w' parity can be calculated in a time proportional in a logarithmic way to the number of fault regions. The word parallelism w' is not bounded in theory, but limited by the width of data paths.

2.3 (1 out-of N) Codes

For the redundant code O we chose a (1 out-of N) code. Here, each code symbol of O contains exactly one bit that is 1 and the rest of bits are 0. Such a code with N bits forms N different code words. Compared to XOR-based codes, a (1 out-of N) code allows encoding and bit recovery in O(1), i.e. a constant number of time steps. The code redundancy is rather high. To encode N symbols with equal likelihood, $ld(N)$ bits would be necessary in the redundancy-free case. When N bits are used, $r = N - ld(N)$ bits per symbol are redundancy. The entropy[2] of symbols coded by O is

---

[2] $H(x) = \sum_i -p(x_i) ld(p(x_i))$ with x_i as the i-th code symbol and $p(x_i)$ the likelihood of the occurrence of x_i.

slightly increased compared to the symbols in the original code I. It can be calculated from the probabilities of 1-bit and 0-bit occurrences. In the following, the entropy per bit is calculated for O-code symbols that originate from t-bit words in I:

$$E_{bit} = -\frac{1}{2^t}ld(\frac{1}{2^t}) - \frac{2^t - 1}{2^t}ld(\frac{2^t - 1}{2^t})$$

$$= \frac{t}{2^t} - \frac{2^t - 1}{2^t}(ld(2^t - 1) - t) \qquad (1)$$

The entropy in O is derived by

$$E_{symbol} = E_{bit} \cdot 2^t. \qquad (2)$$

With a growing t the entropy E_{symbol} approaches $t + 1.44$. This means that data in code O could be compressed very close to the size of its representation in I with additional 1.44 bits in average when an ideal compressor is used. The approach of compression is used later for compressing data after distribution.

For distribution across the fault regions, data is kept uncompressed so far. The high redundancy is traded for a very efficient en- and decoding that can be implemented in both hardware and software, e.g. using a lookup table that is addressed by the input symbols. In hardware, a channel-wise encoder array of size N can be build. Each channel encoder e_i compares the input symbol in I with the value i and the input word. In case of equality, 1 is taken into the channel, otherwise 0. An example for encoding, splitting and channel compression is shown in figure 3.

Decoding is implemented by reading the single bits from the storage channels. Single bits from all N channels form the code word in O. The channel that owns the 1 initiates the output of a fixed codeword in I, all other channels return a word containing all zero. In a second step, the outputs of all channel decoders are bitwise OR-ed. Two steps are needed to obtain the data word, coded in I.

A software–based decoder implementation has to find the 1-bit by comparing words by zero and selecting the 1-bit position of the word containing the 1-bit. This position and the word-index can form a offset to a table where the code words of I are located. The deletion tolerance of the decoder is based on the property that exactly a single 1-bit exists in each code word. So, if the available part of the codeword contains this 1-bit, the missed bit must be 0. Vice versa, when the available part does not include a 1-bit, then the missed bit must be 1. The decision about that is implemented by a NOR-operation.

The high ratio of redundancy added by the (1 out-of N)-code later is eliminated by channel compression. To estimate the compression degree within each channel,
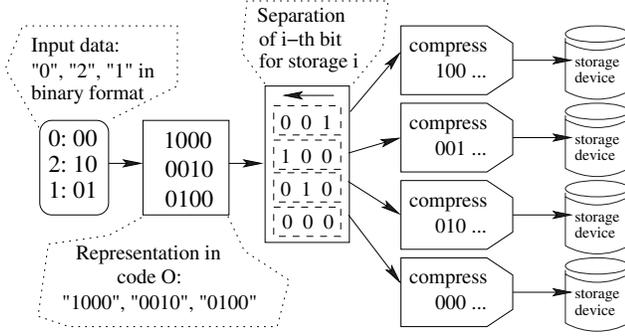
Figure 3. Data path for the (1 out-of N)-code

we calculated the size of data for each channel assuming an entropy encoder. We vary the symbol length of the input code in table 1. For input code I a symbol length t is used. Then the symbol length of O is 2^t. This length directly corresponds to the number of fault regions, i.e. the number of used disks that can fail independently. For each storage channel we obtain an entropy $H(x)$ for a bit symbol in the cannel. Using an ideal compressor, in the compressed code $compr(O)$ the average size of a word is $E_{symbol} = 2^t H(x)$. This result can be used as well for the sequence of bits within a storage channel. This storage size is used to calculate the overhead R related to a word with t bits in code I. The overhead R is compared with the overhead induced by a XOR parity R_{XOR} that represents the minimum of needed redundancy.

Symbol length		Storage	Redundancy ratio	
I	O	size	R	R_{XOR}
t	2^t	$2^t H(x)$	$\frac{2^t H(x)}{t}$	$\frac{t+1}{t}$
1	2	2	2	2
2	4	3.24	1.62	1.5
3	8	4.34	1.45	1.33
4	16	5.39	1.35	1.25
⋮	⋮	⋮	⋮	⋮
8	256	9.44	1.18	1.12
⋮	⋮	⋮	⋮	⋮
16	65536	17.44	1.089	1.062

Table 1. Properties of the (1 out-of N)-Encoding with channel–wise compression

2.4 Concatenated (1-out-of M) codes

For concatenated codes, the original code word is split into k parts. These parts are encoded separately in a (1 out-of M_k) code, with $\sum_k ld(M_k) = ld(N)$. This variant reflects the change of granularity in code word selection and parallelism in encoding. Figure 4 shows the direct (1 out-of 256) encoding of an 8-bit word and the concatenated encoding of two sub words, each with a (1 out-of 16) code.

The concatenated encoding of smaller sub words reduces the storage overhead of O. When a (1 out-of N) encoding requires $N = 2^t$ bits for encoding of a t–bit word, the s–concatenated encoding requires $s \cdot 2^{(t/s)}$ bits, where s sub words of equal length are formed. In contrast, the overhead for compressed storage in O is slightly enlarged and the parallelism in the en- and decoder is shifted to a simultaneous processing of multiple code words.
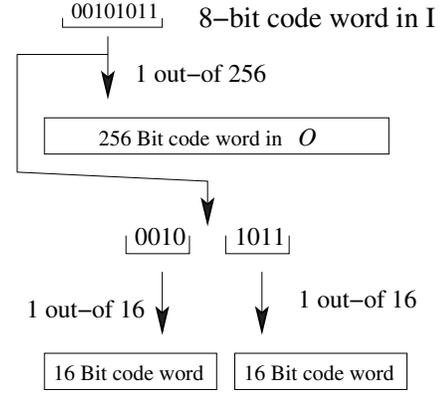
Figure 4. Concatenated Encoding

2.5 Hierarchic Code

A further reduction of the temporary storage overhead in O is reached by encoding the 1-position in O in a hierarchic way. A hierarchy level is introduced by dividing the O code word into two halves. The information which half contains the 1-bit is encoded by a two bit sequence, according to a (1 out-of 2) code. That (1 out-of 2) word is used as the leading sequence, followed by the half of the word that contained the 1-bit. With such a hierarchy level, the data word size in O is reduced to $2 + m/2$, compared to the original amount of m. Hierarchy levels can be applied recursively. Figure 5 illustrates the encoding with two hierarchy levels. Adding hierarchies can be seen as a compression that leads to a (1 out-of 2) binary code, when applied

completely through the code word in a recursive way. In such a complete hierarchic code, each bit from a binary code is represented by two bits where the sequence "10" stands for 1 and "01" stands for 0. An example is $(010)_I$ that is represented in O by $(00000100)_O$. Using $t - 1 = 2$ levels it gets hierarchically encoded to $(01)(10)(01)$. The (1 out-of 2) sequences also can be encoded in a single step each, simply by OR-ing the bits of each half.
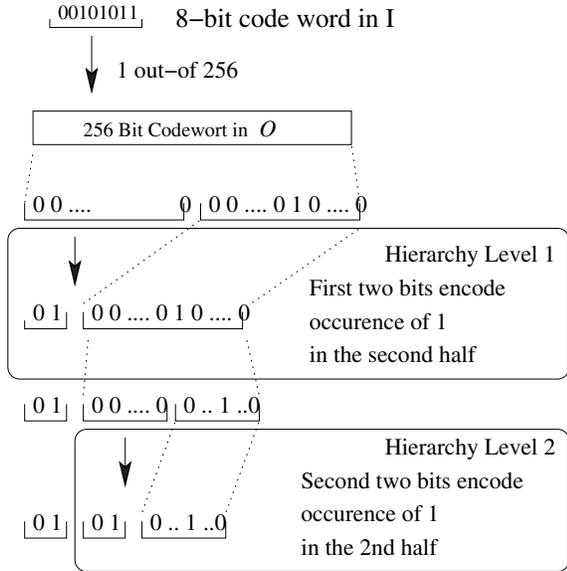
Figure 5. Hierarchic Encoding

The number of hierarchy levels must be traded with a corresponding number of logic stages for en- and decoding. These stages are introduced by filtering out the corresponding half using multiplexors. The OR operations can be (but do not have to be) done in parallel. Figure 6 illustrates the example of an 8 bit word coded in O that is encoded in two hierarchic levels.
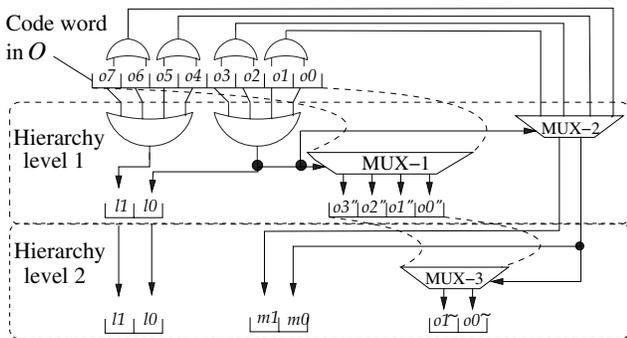
Figure 6. Encoding in Levels

The storage overhead for hierarchic encoding and a single hierarchy level is listed in table 2. We use the symbol O'' for the codeword half that contains the 1-bit. The storage size is obtained by $S = 2H(x') + (2^t/2)H(x'')$, with x' denoting the bits of the leading sequence L and x'' the bits in O''.

Symbol length			Storage size	Redundancy ratio	
I	O	L and O''		R	R_{XOR}
t	2^t	$2 + 2^t/2$	S	$\frac{S}{t}$	$\frac{t+1}{t}$
2	4	$2 + 2$	$2 + 2$	2	1.5
3	8	$2 + 4$	$2 + 3.24$	1.75	1.33
4	16	$2 + 8$	$2 + 4.34$	1.58	1.25
⋮	⋮	⋮	⋮	⋮	⋮
8	256	$2 + 128$	$2 + 8.44$	1.30	1.11
⋮	⋮	⋮	⋮	⋮	⋮
16	65636	$2 + 32768$	$2 + 16.44$	1.15	1.06

Table 2. Properties of a (1 out-of N) code with a single hierarchy level

For such a scheme with a single hierarchy level, the redundancy ratio approaches $(2 + (t - 1) + 1.44)/t$ asymptotically. With an increasing number of hierarchy levels the redundancy ratio approaches $(2 \cdot t)/t = 2$ asymptotically.

3   Architectures

The discussed codes show clearly advantages in recovery complexity against common parity based codes. For a (1 out-of N) code, recovery of a missed bit simply consists of an inverted conjunction of all other bits. The (1 out-of N) code can be generated efficiently by a broadcast bus and a set of parallel comparators. The output of this comparator array leads directly to the codeword in O.

The storage and transfer overhead before compression could be a serious argument against the usage of (1 out-of N) codes. We argue that storage and transfer overhead can be compensated by a proper architecture, i.e. can be hidden in the computing nodes or in the network. The efficient implementation of the OR operation is a key point for a hardware acceleration to reach a better encoding and decoding throughput in comparison to parity-based codes.

3.1 OR-based Hardware Architecture

Figure 7 illustrates a possible hardware architecture for a (1 out-of N) code. For encoding, the code word in I is broadcasted over a bus to a set of N parallel comparators. Each comparator compares with a particular code word in I and generates a 1-bit if the result of comparison is equality, otherwise it generates a 0-bit. The data streams generated by the comparators are compressed and then stored on a storage medium. Comparator, compressor and the storage medium form a single fault region, i.e. a single fault region may fail in a fail-stop manner.

Figure 7. Encoding in Hardware

Reading data from storage invokes a decoding process in the presented hardware architecture. Data from the storage mediums get decompressed into a representation in code O'. Faulty storages are considered in the code O' that is equal to O except for the possibility of a single deleted bit. The deleted bit is coded by an arbitrary bit value, but its position is signaled. Decoding from O to I base on the principle of parallel code generators and a bus that transports the generated codewords to a data sink. In the fault free case, for a single clock period solely a single bit in O' is equal to the value 1. So a single code generator sends the corresponding code word in I. The bus can be seen as a word-wide set of parallel OR gates over N input lines. In case of a faulty storage, the bit recovery is getting activated. A fault region covers the storage medium (disk, RAM–based memory) and the assigned decompressor. The fault must be signaled by FD=1, then an arbitrary data output of a single faulty storage can be tolerated. For all non-faulty storages, FD is signaling 0.

Figure 8 shows the used hardware architecture for

Figure 8. Decoding in Hardware

decoding. The bit recovery is activated for the faulty storage by closing the AND-gate for direct data output and opening the AND-gate for the inverted conjunction result (marked as * in figure 8). Due to the usage of master-slave flipflops for decoupling input and output, the inverted conjunction result may only be used a clock period later. Thus, the logic for bit recovery needs two clock cycles, a first one for propagation of the fault free bits into the flipflops, and a second one for calculating the missed bit and store them in the flipflop array as well. For the first clock period, the * value must be 0, in the second period it has to represent the OR-result of all inputs from O'. This can easily be reached by clock division and appropriate gating. For decoding, a fault region only covers the storage and the decompressor. The set of gates, flipflops, the OR-logic and the code generators are assumed to be failure resilient for the presented architecture. By the deletion correction, the code O' is transformed into O, the fault-free representation of the data words in a (1 out-of N) code.

3.2 Software-based Architecture for Multiprocessors

The basis principle of generating the bits of a (1 out-of N) code by parallel encoders can be adopted in software as well. It requires a parallel hardware architecture, e.g. a multiprocessor system that supports a couple of parallel threads that execute encoder and decoder functions. The data processed by these threads must be assigned to different storages. Encoding is done by parallel comparator threads that check for particular codewords in an input buffer, generate the code words in O bit-wise and do the compression. Decoding

also relies on parallelism in a multiprocessor system. Multiple threads read compressed data from different storages, decompress and mix the obtained 1-bits into a storage area that contains codewords of O'.

When a faulty storage is present, a correction thread is capable to recalculate the missed bit simply by OR-ing all bits in the codewords of O' and setting the missed bit according to the inverted OR-result. Finally, the data in code I is obtained by a set of threads that compare for 1-bits in particular positions and write the related code word in the output sequence. For that, N parallel comparator threads are necessary.

For a multiprocessor system, usually the entire computer is a single fault region. Thus, solely the different storages can be seen as the fault regions. Other software architectures and machine models (e.g. distributed systems) may show completely different structured fault regions.

4 Summary

An alternative approach to parity-based fault-tolerant data layouts is the OR-based redundancy. Parity–based schemes need a number of $O(log_2 N)$ steps for parity calculation and deletion correction, where N is the number of fault regions. In contrast, the OR operation can be done in a constant number of time steps. Thus, the cost of encoding and decoding under faults is independent from the number of fault regions (e.g. number of disks). This advantageous approach comes with two difficulties. First, a (1 out-of N) code must be used that is very space-consuming. This can be compensated by compression of data within the fault regions. Secondly, an efficient implementation of the OR operation must be either present in hardware or must get implemented by software that exploits the parallelism of the used machine. In this paper, it has been shown that nearly the same space-efficiency as parity-based codes can be reached and that the OR-based codes are feasible, when an appropriate hardware architecture is used. The applied compression can also be used for reducing data redundancy, so the proposed storage architecture comes with the byproduct of doing data compression without any extra cost. First results show that for input data that allow a high compression ratio, the compression effect is also reached in the (1 out–of N) representation.

References

[1] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures . *IEEE Transactions on Computers*, 44(2), February 1995.

[2] V. Bohossian, C.C. Fan, P.S.LeMahieu, M.D. Riedel, L. Xu, and J. Bruck. Computing in the RAIN - A Reliable Array of Independent Nodes. Technical report, California Institute of Technology, September 1999.

[3] E. N. Elnozahy. Storage Strategies for Fault-Tolerant Video Servers. Technical Report CMU-CS-96-144, Carnegie Mellon University, 1996.

[4] R. Friedman, L. Baram, and S. Abaranal. Fault Tolerant Multi-Server Video on Demand Service. In *IPDPS Proceedings*. IEEE Computer Society Press, 2003.

[5] Lisa Hellerstein, Garth A. Gibson, Richard M. Karp, Randy H. Katz, and David A. Patterson. Coding Techniques for Handling Failures in Large Disk Arrays. *Algorithmica*, 12(2/3):182–208, 1994.

[6] Ping-Hsun Hsieh, Ing-Yi Chen, Yu-Ting Lin, and Sy-Yen Kuo. An XOR Based Reed-Solomon Algorithm for Advanced RAID Systems. In *DFT '04: Proceedings of the Defect and Fault Tolerance in VLSI Systems, 19th IEEE International Symposium on (DFT'04)*, pages 165–172, Washington, DC, USA, 2004. IEEE Computer Society.

[7] J. S. Plank. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. *SOFTWARE - PRACTICE AND EXPERIENCE*, 27(9):995–1012, September 1997.

[8] P. Sobe. Data Consistent Up- and Downstreaming in a Distributed Storage System. In *Proceedings of. Int. . Workshop on Storage Network Architecture and Parallel I/Os*, pages 19–26. IEEE Computer Society, 2003.

[9] P. Sobe. Stable Checkpointing in Distributed Systems without Shared Disks. In *IPDPS 2003 Proceedings, Workshop on Fault-Tolerant Parallel and Distributed Systems*. IEEE Computer Society, 2003.

[10] Y. Zhu. Design, Implementation, and Performance Evaluation of a Cost-Effective Fault-Tolerant Parallel Virtual File System. In *Proceedings of. Int. . Workshop on Storage Network Architecture and Parallel I/Os*, pages 53–64. IEEE Computer Society, 2003.