Published in proceedings of The 45th IEEE Asilomar Conference on Signals, Systems, and Computers (SSC 2011), November 6-9, 2011, Pacific Grove, California, USA

Finding the most fault-tolerant flat XOR-based erasure codes for storage systems

Jay J. Wylie
HP Labs
jay.wylie@hp.com

***Abstract*—We describe the techniques we developed to efficiently find the most fault-tolerant flat XOR-based erasure codes for storage systems. These techniques substantially reduce the search space for finding fault-tolerant codes (e.g., by a factor of over 52 trillion in one case). This reduction in the search space has allowed us to find the most fault-tolerant codes for larger codes than was previously thought feasible. The result of our effort to find the most fault-tolerant flat XOR-based erasure codes for storage systems has yielded a corpus of 49,215 erasure codes that we are making public.**

I. Introduction

In this paper, we describe the techniques we used to find the most fault-tolerant flat XOR-based erasure codes for storage systems. By "find", we mean exhaustive computational search of the code space. By "most fault-tolerant", we literally mean the most fault-tolerant codes. I.e., the codes with the best Hamming Distance, and in the case of two codes with the same Hamming Distance, the one with fewer possible failure sets at the Hamming Distance. By "flat", we mean an erasure code in which exactly one element (data or parity) is stored on each device (e.g., hard disk drive, SSD, or flash memory). By "XOR-based" we mean erasure codes in which each parity element is the XOR-sum of some subset of data elements. By "for storage systems", we mean systematic erasure codes for small values of k (the number of data elements) and m (the number of parity elements). The target audience for this paper are other engineers, scientists, and theoreticians interested in erasure codes for storage systems.

We provide some background on erasure codes for storage systems and our prior work on evaluating flat XOR-based erasure codes (section II). We then describe our core contribution, a computational approach to finding the most fault-tolerant flat XOR-based erasure codes for storage systems (section III). We describe, at a high level, the techniques we developed to (i) efficiently explore all codes in a code space (i.e., for some given number of data and parity elements), and (ii) efficiently determine the fault-tolerance of each code in a code space. Our effort to find the most fault-tolerant flat XOR-based erasure codes for storage systems has yielded a corpus of 49,215 erasure codes that we are making public [1]. We hope that other storage systems and coding theory researchers can benefit from the codes we found. We summarize and discuss some interesting properties of the codes in our corpus (section IV). Finally, we discuss potential future work (section V).

II. Background

An erasure code consists of n symbols, k of which are *data symbols*, and m of which are *redundant symbols*. For XOR-codes, we refer to redundant symbols as *parity symbols*. In storage systems, symbols are packed into device blocks that fail as a whole. A storage system converts any failure of such a block into an erasure and so we use failure synonymously with erasure. Instead of symbols, we refer to *elements*, which correspond to the basic I/O unit of an underlying storage device (e.g., a block).

We only consider *systematic* erasure codes: codes that store the data elements and the parity elements. Systematic erasure codes provide a common case read path that requires no decoding computation — data elements are simply concatenated to recover the stored value. Such a read path also permits an individual data element, or portion of a data element, to be read and returned; this is important for some file system and database workloads.

The fault-tolerance of an erasure code is defined by d its *Hamming distance*. An erasure code of Hamming distance d tolerates all failures of fewer than d elements. We use the following terminology to talk more exactly about the fault-tolerance of a specific erasure code. An *erasure pattern* is a set of erasures (i.e., list of failed elements) that result in at least one data element being irrecoverable. The *erasures list* for an erasure code is the list of all its erasure patterns. The erasures vector is a vector of length m in which the ith element is the total number of erasure patterns of size i in the erasures list. The dth entry of the erasures vector is the first non-zero entry. A *minimal erasure* ME is an erasure pattern in which every erasure is necessary for it to be an erasure pattern; if any erasure is removed from ME, then it is no longer an erasure pattern. The *minimal erasures list* (MEL) for an erasure code is the list of all its minimal erasures. The *minimal erasures vector* (MEV) is a vector of length m in which the ith element is the total number of minimal erasures of size i in the MEL. We believe that the MEV is the the most concise representation of the exact fault-tolerance of an erasure code and so use it to determine the most fault-tolerant erasure codes.

We originally defined the minimal erasures terminology in [2] where we introduce the Minimal Erasures (ME) Algorithm. The ME Algorithm uses the structure of a flat XOR-based erasure code represented in its Tanner graph to efficiently enumerate the code's Minimal Erasures List. In this work, we

extend and improve the ME Algorithm to efficiently find the most fault-tolerant erasure codes.

Flat XOR-based storage codes for storage are quite similar to low-density parity-check (LDPC) codes [3]. Flat XOR-based codes differ from LDPC codes in that they are systematic codes and have "small" values of k and m. By small, we mean small enough to be used in a storage system (e.g., $n = k + m \leq 30$). Properties of LDPC codes based on probabilistic and/or asymptotic arguments that rely on large numbers of symbols do not apply to the flat XOR-based codes we consider.

Even though flat XOR-based codes are similar to LDPC codes, and we use the Tanner graph to determine the MEL, our assessment of fault-tolerance is not based on *stopping sets*. *Stopping sets* [4] are specific sets of failures that can prevent *iterative decoding* from successfully decoding. In a storage system, we expect to use comprehensive decode methods that require a matrix to be inverted and so ignore stopping sets.

Because of space limitations, we discuss related work on erasure codes for storage systems in a narrow manner. For broader coverage of related work on flat XOR-based erasure code constructions, and on evaluating erasure codes for storage, please see Section II of our most recent paper [5]. Traditionally, LDPC codes are constructed by randomly generating a Tanner graph based on some probability distributions of the edge counts for data and parity elements. Plank and Thomason survey and evaluate such constructions for storage systems [6]; they focus their investigation on a specific performance metric (read overhead) and cover values of k from small to moderate (up to 150). We are interested in identifying the absolute most fault-tolerant constructions for some values of k and m and so do not rely on randomized constructions. Plank et al. built upon their initial work to find codes for small values of k and m with the best read overhead [7]. They used both computational means and analytic means to produce a substantial list of optimal and near-optimal codes for use in systems research [8]. Their computational approach is similar to ours, though the metric of interest is different.

In storage systems, many erasure codes are generated via formulaic construction. I.e., an algorithm, based on some mathematical insight and parameterized by k, m, and often other values (e.g., some prime value), is used to construct an erasure code with some known properties (e.g., a specific fault-tolerance or with specific recovery properties). See our prior work for examples and discussion of such parametric formulaic constructions [5]. Such constructions tend to be based on some regular structure, which in our opinion goes against the "nature" of LDPC-like codes. We believe that parametric formulaic constructions can cover only a small portion of the possible space of constructions. Our approach to finding the most fault-tolerant flat XOR-based codes is via exhaustive computational exploration of the space of all possible such codes.

Our exhaustive computational approach to finding the most fault-tolerant flat XOR-based codes is similar to the approach Hafner used to find Weaver code constructions [9]. Hafner discovered various regular, symmetric constructions of multi-dimensional (i.e., not flat) XOR-based erasure codes for storage systems by exhaustive search for patterns of offsets and stripe widths that achieve desired level of fault-tolerance for a given rate. Our starting point differs from Hafner's in that we limit ourselves to flat codes and do not constrain ourselves to codes with rotational symmetry. The rationale for this is many-fold: we are not convinced of the benefit of rotational symmetry for erasure codes in storage systems; limiting our interest to flat codes reduces the code space we need to exhaustively explore; and, we think the corpus of flat codes may be more useful for others to build upon than multi-dimensional codes.

III. Finding the most fault-tolerant codes

To find the most fault-tolerant flat XOR-based codes, the brute force approach would consist of two steps: One, generate all possible such codes in the (k,m)-code space; Two, try all possible failure patterns to determine the exact fault-tolerance of each (k,m)-code in the code space. Such a brute force approach is prohibitively expensive: for a given (k,m), there are $2^{k\times(k+m)}$ possible XOR codes, and each such code has 2^{k+m} possible failure patterns. We followed the basic flow of the brute force approach, but made each step significantly more efficient.

A. Efficiently exploring the code space

There are many ways in which we reduced the (k,m)-code space explored while still ensuring that we found the most fault-tolerant (k,m)-code in the space. First, we note that there are only $2^{k\times m}$ possible *systematic* XOR codes which greatly reduces the code space. Second, we note that many of the $2^{k\times m}$ possible codes are in fact isomorphic to one another. Therefore, to fully explore the (k,m)-code space, we only need to enumerate all non-isomorphic systematic Tanner graphs for the space.

Our Tanner graph representation is a bipartite graph with data elements on one side, and parity elements on the other. A parity element is connected to each data element that is XORed in its calculation. Non-isomorphic bi-partite graph generation is well understood [10] and the `nauty` tool can efficiently generate all such graphs [11].

The `nauty` tool provides many options that we used to further reduce the portion of the (k,m)-code space we explored. First, we only needed to consider connected graphs, since a Tanner forest is necessarily less fault-tolerant than a Tanner graph for a given (k,m). Second, the `nauty` tool can shard the space of possible graphs using the `res/mod` option. This provides a path to parallelization: up to `mod` distinct machines can be used to explore the space. Such parallelization requires the results from all `mod` shards to be aggregated after the fact, but that step is trivial. Third, the weight and connectivity of each node in the bipartite graph can be restricted for `nauty`.

We use the connectivity constraints to set a minimum connectivity for data elements in the code. We set this minimum to be an estimated Hamming distance of the code, less one. Our rationale is that a data element connected to c parity elements necessarily has a minimal erasure of size $c + 1$ that consists

of itself and each parity element to which it is connected. We estimate the Hamming distance conservatively, i.e., we err on the side of too big an estimate. If our estimate is in fact too big, then no codes are found and we try again with a smaller Hamming distance estimate. If our estimate is too small, then we still find the most fault-tolerant (k,m)-code, but evaluate more of the code space than strictly necessary.

We have not systematically evaluated the reduction of the code space due to each of our optimizations. However, we did track how many codes we evaluated for each (k,m). For the (9,9)-code space, we evaluated 45,706,861,459 distinct codes. A simplistic brute force approach could have evaluated 2^{81} codes. Our approach reduced the code space, relative to a simplistic brute force approach, by a factor of over 52 trillion.

B. Efficiently determining fault-tolerance

The key tool we use to determine the fault-tolerance of a flat XOR-based erasure code is `mea`, our implementation of the ME Algorithm. The ME Algorithm, as described previously [2], uses the structure of the Tanner graph to enumerate the Minimal Erasures List (MEL) for a specific code. Briefly, and at a very high level, the ME Algorithm generates an initial set of base minimal erasures, one for each data element, and uses them to initialize the MEL. The ME Algorithm then recursively takes each minimal erasure already in the MEL and generates additional minimal erasures to add to the MEL; it does so by iteratively substituting in base erasures for parity elements in each minimal erasure. Using the structure of the code to directly enumerate the MEL requires substantially less work than the brute force approach of testing all possible failure patterns by attempting to decode.

Beyond the ME Algorithm being more efficient than brute force determination of fault-tolerance, we added many features to `mea` to more efficiently evaluate large numbers of codes. One key improvement in execution speed is to permit `mea` to terminate as soon as it discovers that the code it is evaluating cannot be the most fault-tolerant. There are two cases in which `mea` terminates early: one, it is provided with an estimate of the expected Hamming distance (as is done for improved graph generation) and it finds a minimal erasure that is smaller than the estimated Hamming distance; two, it tracks the best MEV it has found thus far and it determines that the MEV of the current code is worse than the best.

A computationally expensive corner-case for the ME Algorithm is confirming that a child erasure is in fact a minimal erasure and not the union of two disjoint minimal erasures. For the purposes of finding the most fault-tolerant code, this corner case can be ignored; such child erasures are necessarily longer than the smallest minimal erasures that distinguish the most fault-tolerant codes from the rest.

The parallelization of code space exploration has some costs. First, the optimization in which `mea` terminates early is not as effective. This is because `mea` only tracks the best codes per shard of the code space being generated by `nauty`; the various `mea` instances processing different shards to not share information during execution. Second, we made the early termination even less effective by tracking the five most fault-tolerant codes rather than the single most fault-tolerant code. I.e., for each of the five best fault-tolerances achievable in a code space, `mea` retains all the distinct codes that achieves such fault-tolerance. We did this out of curiosity and so that we could see the differences among the best few codes in a (k,m)-code space. Finally, a light-weight post-processing step is required to collect the results from all shards of the code space into a single list of most fault-tolerant (k,m)-codes.

We have not systematically evaluated the efficiency of each optimization we introduced to find the most fault-tolerant codes. Two specific accomplishments put the efficiency of these techniques into context though. Gaidioz et al. [12] used Monte Carlo approaches to construct small codes because exhaustive "techniques are not feasible for generating larger codes, like the ones we are interested in using," where larger referred to $k > 3$ and $m > 5$. We used the techniques described above to find a more fault-tolerant (8,6)-code than Gaidioz et al. discovered in less than an hour on a laptop. Woitaszek et al. [13] used a "test suite [that] contains exactly 962,144,153 test cases and requires about 34 CPU days to execute for a single graph" to verify the fault-tolerance of a small XOR-based code they wanted to use; we were able to determine the fault-tolerance of the (48,48)-code found by Woitaszek et al. at a greater level of detail than the authors in a few CPU hours on a laptop.

IV. Most fault-tolerant codes

The result of our effort to find the most fault-tolerant flat XOR-based erasure codes for storage systems has yielded a corpus of 49,215 erasure codes that we are making public [1]. The corpus lists the most fault-tolerant flat XOR-based erasure codes for storage systems for $(k \leq 5, m \leq 10)$, $(k \leq 9, m \leq 9)$, $(k \leq 16, m \leq 5)$, and $(k \leq 20, m \leq 4)$. Over all of the (k,m)-code spaces we evaluated, the 2,670 codes that provide the most fault-tolerance, as measured by the MEV, are listed. We also include the 46,545 codes that provide the next four best fault-tolerances, as measured by the MEV, over these code spaces. We did this computational search in approximately 10 days a few years ago on a cluster of 300 HP DL360 computers with 2.8 GHz Intel Xeon processor and 4 GB of RAM.

Since our corpus will be released in its entirety [1], we focus on presenting summary statistics here and discussing interesting features of the code corpus. First, we note that the parts of the corpus that can be easily verified by inspection are correct. For example, all replication codes, i.e., codes with $k = 1$, in the corpus are listed correctly. Similarly, all codes with $(k > 1, m = 1)$ in the corpus correctly list RAID4 (a single parity element that is the XOR-sum of all data elements) as the most fault-tolerant code. These easily verified codes are the only flat XOR-based codes that are MDS.

Table I lists the Hamming distance for the best codes (as measured by the MEV) we found for each value of k and m. Note that all the most fault-tolerant codes we found for $k > 4$ and $m > 5$ stand out in so far as others thought it to be prohibitive to process such codes via "brute force" [13].

Published in proceedings of The 45th IEEE Asilomar Conference on Signals, Systems, and Computers (SSC 2011), November 6-9, 2011, Pacific Grove, California, USA

k	m =1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10	11
2	2	2	3	4	4	5	6	6	7	8
3	2	2	3	4	4	4	5	6	6	7
4	2	2	3	4	4	4	5	6	6	7
5	2	2	2	3	4	4	4	5	6	7
6	2	2	2	3	4	4	4	5	6	-
7	2	2	2	3	4	4	4	5	6	-
8	2	2	2	3	4	4	4	5	6	-
9	2	2	2	3	4	4	4	5	6	-
10	2	2	2	3	4	-	-	-	-	-
11	2	2	2	3	4	-	-	-	-	-
12	2	2	2	2	3	-	-	-	-	-
13	2	2	2	2	3	-	-	-	-	-
14	2	2	2	2	3	-	-	-	-	-
15	2	2	2	2	3	-	-	-	-	-
16	2	2	2	2	3	-	-	-	-	-
17	2	2	2	2	-	-	-	-	-	-
18	2	2	2	2	-	-	-	-	-	-
19	2	2	2	2	-	-	-	-	-	-
20	2	2	2	2	-	-	-	-	-	-

TABLE I
HAMMING DISTANCE OF MOST FAULT-TOLERANT CODES.

k	m =1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
2	1	2	2	1	2	2	1	2	2	1
3	1	2	2	1	3	6	7	3	6	3
4	1	1	1	1	3	6	10	3	7	3
5	1	2	3	3	2	12	8	6	3	2
6	1	2	6	6	7	5	57	47	13	-
7	1	1	7	10	8	57	20	162	401	-
8	1	2	3	3	6	59	162	859	102	-
9	1	2	6	7	3	13	17	102	19	-
10	1	1	3	3	2	-	-	-	-	-
11	1	2	1	1	1	-	-	-	-	-
12	1	2	3	4	3	-	-	-	-	-
13	1	1	6	13	10	-	-	-	-	-
14	1	2	7	33	24	-	-	-	-	-
15	1	2	3	51	42	-	-	-	-	-
16	1	1	6	14	19	-	-	-	-	-
17	1	2	3	33	-	-	-	-	-	-
18	1	2	1	13	-	-	-	-	-	-
19	1	1	3	4	-	-	-	-	-	-
20	1	2	6	13	-	-	-	-	-	-

TABLE II
NUMBER OF DISTINCT CODES THAT SHARE MOST FAULT-TOLERANT MEV.

The trend is as expected: as m increases, Hamming distance increases.

Table II lists the number of distinct codes (i.e., non-isomorphic Tanner graphs) that share the best MEV. Results like those for $(k, m) = (11, 3)$, $(11, 4)$ and $(11, 5)$ standout because only one distinct code achieves the most fault-tolerance in these code spaces. Table III lists the number of distinct codes that achieve the best Hamming distance for the code space. Because we only tracked the five best MEV for each code space, we list a plus sign for any code spaces in which the five best MEV all have the same Hamming distance. Some specific results stand out in this table. For $(k, m) = (11, 4)$ and $(11, 5)$, only one distinct code achieves the best Hamming distance. For $(k = 9, m = 9)$, a huge code space, only 19 distinct codes achieve the best Hamming distance.

As mentioned earlier, Plank et al. identified flat XOR-based codes with the best read overhead [7], [8]. The (10,4)-code they identified as having optimal overhead (of 10.6771) has an MEV of $(0, 1, 27, 72)$; compare this to the MEV of the most fault-tolerant such code: $(0, 0, 28, 77)$. In fact, the read overhead optimal code identified by Plank et al., when compared with our corpus of best codes, has the second best fault-tolerance.

V. FUTURE WORK

We believe that some engineering effort and re-running this analysis on more modern computers could find the most fault-tolerant flat XOR-codes for even more code spaces. The `nauty` tool is written in C and so likely runs efficiently. However, our `mea` tool set is written in python; a rewrite in C and more careful design of some internal data structures could yield a substantial execution speedup. Even if implementation re-engineering provided a factor of ten speedup, and modern compute hardware yielded another factor of ten speedup, the exponential nature of the size of code spaces is still a barrier to, for example, finding the most fault-tolerant (20,20)-code. To reach such a goal, we believe further advances in our understanding of how to prune a (k,m)-code space and how to efficiently determine the exact fault-tolerance of a (k,m)-code is necessary.

A. Efficiently exploring the code space

We used prior results to provide estimates of the expected Hamming distance to guide the code space exploration. A better Hamming distance estimator could improve code space exploration.

We have experimented with estimating the Hamming weight (i.e., number of edges in the Tanner graph) of the most fault-tolerant code in a code space. We started with an estimate of the exact Hamming weight of the most fault-tolerant code. We then expand the Hamming weight range, i.e., from a single initial number to a range from one less to one greater than the initial number, and so on. We continue expanding the Hamming weight range until we stop finding more fault-tolerant codes. Unfortunately, our effort so far has not resulted in any substantial reduction in the computation required to explore a code space.

The `nauty` tool provides the `res/mod` option that allows us to parallelize code space exploration. We are not convinced that this feature was designed to efficiently parallelize graph generation. I.e., there may be overheads to this parallelization that we do not understand. If so, then reducing the degree to which we parallelize code space exploration, or improving the means which `nauty` uses to parallelize such work, could make parallel code space exploration more efficient.

k	m =1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
2	1	2	2	1	4	2	1	4	2	1
3	1	4+	2	1	8	31+	11	3	30+	3
4	1	6+	1	1	14	66+	10	3	79+	3
5	1	9+	32+	35+	14	121+	152+	417+	223+	2
6	1	12+	29+	35	12	163+	381+	1633+	565+	-
7	1	14+	49+	28	10	243+	5386+	4611+	557	-
8	1	12+	48+	16	6	313+	4271+	859	102	-
9	1	12+	70+	7	3	370+	366+	102	19	-
10	1	17+	57+	3	2	-	-	-	-	-
11	1	12+	53+	1	1	-	-	-	-	-
12	1	12+	67+	352+	595+	-	-	-	-	-
13	1	17+	45+	222+	334+	-	-	-	-	-
14	1	12+	70+	417+	630+	-	-	-	-	-
15	1	12+	58+	267+	647+	-	-	-	-	-
16	1	17+	83+	213+	497+	-	-	-	-	-
17	1	12+	61+	290+	-	-	-	-	-	-
18	1	12+	53+	361+	-	-	-	-	-	-
19	1	17+	67+	352+	-	-	-	-	-	-
20	1	12+	45+	290+	-	-	-	-	-	-

TABLE III
NUMBER OF DISTINCT CODES THAT ACHIEVE THE BEST HAMMING DISTANCE.

B. Efficiently determining fault-tolerance

In our tool chain, `mea` consumes the output of a single instance of `nauty` when code exploration is parallelized. The optimizations to terminate early do not work as well in this context. Some form of communication among instances of `mea` that share the currently best known MEV could improve this optimization when code exploration is done in parallel.

We have experimented with having the ME Algorithm determine a short prefix of the MEV of the code (i.e., the first two non-zero entries in the MEV). The only cut off we could determine to be correct though was the number of symbol elements in the minimal erasure. I.e., for an estimated Hamming distance of 3, `mea` had to evaluate all child erasures up to the point that only minimal erasures with four or more data elements in them remained. This effort has not yet resulted in substantial reduction in code evaluation time.

We believe the main way to improve the efficiency of determining fault-tolerance though is to improve the ME Algorithm. Any techniques that can more efficiently find minimal erasures at the Hamming distance could significantly improve our ability to analyze larger code spaces.

ACKNOWLEDGMENTS

Thanks to my key collaborators on erasure codes research over the last five years (Kevin Greenan, Jim Plank, Ram Swaminathan) for feedback, ideas and discussions that lead to these results. Thanks to Alex Dimakis for organizing the Distributed Storage Systems session at the 45th Asilomar Conference on Signals, Systems and Computers. Finally, thanks to the other attendees that provided me with feedback and asked me interesting questions.

REFERENCES

[1] J. J. Wylie, "List of most fault-tolerant flat XOR-based erasure codes for storage systems," HP Labs, Tech. Rep. HPL-2011-217, November 2011.

[2] J. J. Wylie and R. Swaminathan, "Determining fault tolerance of XOR-based erasure codes efficiently," in *DSN-2007*. IEEE, June 2007, pp. 206–215.

[3] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. A. Spielman, and V. Stemann, "Practical loss-resilient codes," in *STOC-1997*. ACM Press, 1997, pp. 150–159.

[4] M. Schwartz and A. Vardy, "On the stopping distance and the stopping redundancy of codes," *IEEE Trans. on Inf. Theory*, vol. 52, no. 3, pp. 922–932, 2006.

[5] K. M. Greenan, X. Li, and J. J. Wylie, "Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs." in *26th IEEE Symposium on Massive Storage Systems and Technologies*. IEEE, May 2010.

[6] J. S. Plank and M. G. Thomason, "A practical analysis of low-density parity-check erasure codes for wide-area storage applications," in *DSN-2004*. IEEE, June 2004, pp. 115–124.

[7] J. S. Plank, A. L. Buchsbaum, R. L. Collins, and M. G. Thomason, "Small parity-check erasure codes - exploration and observations," in *DSN-2005*. IEEE, July 2005.

[8] J. S. Plank, "Enumeration of small, optimal and near-optimal parity-check erasure codes," Department of Computer Science, University of Tennessee, Tech. Rep. UT-CS-04-535, November 2004.

[9] J. L. Hafner, "WEAVER Codes: Highly fault tolerant erasure codes for storage systems," in *FAST-2005*. USENIX Association, December 2005, pp. 212–224.

[10] B. McKay, "Practical graph isomorphism," *Congressus Numerantium*, vol. 30, pp. 45–87, 1981.

[11] ——, "`nauty` version 2.2 (including `gtools`)," http://cs.anu.edu.au/~bdm/nauty/.

[12] B. Gaidioz, B. Koblitz, and N. Santos, "Exploring high performance distributed file storage using LDPC codes," *Parallel Computing*, vol. 33, pp. 264–274, May 2007.

[13] M. Woitaszek and H. M. Tufo, "Fault tolerance of Tornado codes for archival storage," in *15th IEEE International Symposium on High Performance Distributed Computing*, 2006.