Evaluation of a High Performance Erasure Code Implementation

Frank Uyeda, Huaxia Xia, Andrew A. Chien
Computer Science and Engineering Department
University of California, San Diego
{fuyeda, hxia, achien}@ucsd.edu
September 9, 2004

Abstract
We examine the properties and performance of LT Codes for use in a high performance distributed file system. Different variations to the algorithm were created to achieve key characteristics, such as speed and data reliability. Finally, we evaluate parameter choices in order to balance the system and network loads.

I. Introduction

As network bandwidth becomes more plentiful, high performance systems are able to rapidly transmit very large datasets between each other with high quality of service. Large-scale scientific applications can take advantage of this new capability to share and interact with massive amounts of information. This is part of the motivating force behind the Optiputer project. In order to meet the data transfer demands of high performance computing, steps must be taken to circumvent latency costs and take advantage of the maximum available bandwidth.

As part of the Optiputer project [5], the RobuSTore distributed file system addresses these obstacles by producing redundant encoded data in such a way that clients need only to retrieve a certain amount of the encoding from any combination of storage sites in order to reconstruct the original file. This allows an application to request data in parallel from every server that stores part of the encoding and then terminate the transfer after it has received a sufficient aggregate amount of information. By retrieving the data in parallel, the maximum available bandwidth can be utilized, and since only a certain percentage of the encoding is needed, the client can avoid the overhead of waiting for a sluggish host.

The current encoding implementation is a type of rateless erasure code known as a Luby Transform (LT) Code. Using the LT code as a base, certain aspects of the algorithm were modified to specialize the encoding for the needs of RobuSTore. Running on dual-processor Pentium-4 Xeon compute nodes, decoding speeds in excess of 200 MBps were obtained using reasonable encoding parameters.

This paper explores different obstacles and solutions in the use of LT codes with RobuSTore. We first look at the original LT algorithm in section II, and then define the essential criteria for RobuSTore in section III. Next we will look at several modifications to the coding algorithm used to achieve RobuSTore's goals. Our testing procedure is described in section V. The results from the speed-optimized implementations are then presented in section VI, followed by the results of the implementations with improved reliability in section VII. Section VIII provides insight into the overall performance of the system when both computation speed and network bandwidth are considered. We compare and contrast the work set forth in this paper to other implementations and uses of LT codes

in section IX. Finally, section X gives a summary of our accomplishments and highlights future work in this area.

II. Background on LT Codes

The LT Coding algorithm produces a virtually unlimited number of encoded blocks from some k original data blocks via logical XOR operations. The k original data blocks are obtained by partitioning the original data into k uniform segments and the creation of each encoded block, or "symbol", will require $O(\ln(k/\delta))$ logical operations on the original blocks. To decode the original data with a $1-\delta$ chance of success, any $k + O(\sqrt{k}\ln^2(k/\delta))$ encoded blocks should be sufficient [1].

The encoding process is relatively straight forward.

1. Choose some degree *d* for the next encoded block according to the Robust Soliton Distribution [1].
2. Randomly choose *d* different original data blocks and XOR them together to produce the encoded block.
3. Repeat steps 1 and 2 until the desired number of encoded blocks have been produced.

It should be noted that as each encoded symbol is produced, the identities of its sources must be stored as meta-data for the decoding process. In our implementation, this information is represented as a bipartite graph with edges connecting nodes corresponding to original data blocks to nodes representing encoded blocks.

The process of decoding the data is as follows:

1. When an encoded block is received, XOR it with all of its neighbors in the bipartite graph which have been recovered, and remove the edges that join the XORed nodes.
2. If the encoded block has only one remaining neighbor, then part of the original data has been recovered. Copy its data to its sole neighbor and place that data node in a queue of original nodes to process.
3. While the queue is not empty, choose a data node from the queue. XOR each received neighbor's data with the data in the original node and disconnect the nodes. For each neighbor that is XORed, perform step 2.
4. Continue receiving and processing encoded blocks until the original data has been completely recovered.

A novel feature of the LT coding algorithm is the use of the Robust Soliton distribution. The basis for this distribution comes from the probability that an encoded symbol of some degree will be able to recover a data block from a set of data blocks that have yet to be recovered. In order to balance minimal redundancy with the production of enough edges to keep the decoding successful within an established probability, Luby proposes an Ideal Soliton [1] distribution for *k* original blocks as:

$$\rho(1) = 1/k$$
$$\text{For all } i = 2,...,k,\ \rho(i) = 1/i(i-1)$$

However, in practice, the ideal distribution performs very poorly. This ideal model does not handle variance well, and therefore results in high decoding failure [1,2]. To address this problem, the Robust Soliton Distribution was proposed. For a given number of blocks, the ideal distribution was designed to keep the number of data elements waiting in the queue to be processed close to one. In order to account for variance, the Robust Soliton Distribution aims to keep the size of the queue, R, around $R = c \cdot \ln(k/\delta)\sqrt{k}$ for some $c > 0$ [1]. The following distribution was proposed to augment the Ideal Soliton to achieve this.

$$\tau(i) = \begin{cases} R/ik & \text{for } i = 1,...,R\text{-}1 \\ R\ln(R/\delta)/k & \text{for } i = k/R \\ 0 & \text{for } i = k/R+1,...,k \end{cases}$$

The Robust Soliton Distribution, $\mu(i)$, is then the normalized sum of these two distributions.

$$\mu(i) = \frac{(\rho(i) + \tau(i))}{\sum_{j=1}^{k} \rho(j) + \tau(j)}$$

III. LT Codes and RobuSTore

RobuSTore has several requirements that must be met by any encoding scheme that it uses. These requirements ensure that RobuSTore can function properly as a distributed file system while providing fault-tolerant, high performance service. Below are several criteria that our LT Code implementation must satisfy.

1. Encoded blocks must be stored and distributed ahead of time.

To achieve high parallel throughput and avoid latency, RobuSTore distributes encoded blocks among several storage devices. In order to do this, a fixed number of encoded blocks must be produced when the original file is saved. While LT Codes usually continue to produce encoded blocks until the decoding is complete, we only capture and store a fixed number of blocks which should be enough to decode with high probability.

2. The decoder must run at high speeds.

One of the key motivations for using erasure codes in the RobuSTore file system is to increase the performance by avoiding long network latencies and slow hosts. In order to take advantage of these benefits, the decoder must have a throughput high enough to saturate the available network bandwidth.

3. The algorithm must provide data reliability.

Since we are developing a file system, any data lose is unacceptable. The LT Coding algorithm is non-deterministic and, as such, there is no absolute guarantee that the original

data can be recovered from a fixed number of blocks. Our implementation must ensure 100% recoverability.

4. The encoding should be robust.
In order to provide fast decoding and fault tolerance, we defined the idea of "maximum degree of freedom". Simply put, there should be a maximum number of different combinations of encoded blocks that can be used to reconstitute the original data. This degree of freedom ensures that there will be no encoded blocks that the decoder always depends in order to recover the data.

IV. LT Codes in Practice

In order to utilize the benefits of LT Codes and meet the requirements of the RobuSTore file system, the algorithm was modified in several ways. The following are the variations that were implemented and tested.

Optimized Scheduling:
A key optimization to speed up the algorithm was scheduling the XOR operations. After evaluating the original implementation with gprof, the obvious bottleneck in the system was performing memory operations, namely the logical XOR. In order to reduce the number of memory accesses during decoding, we waited until all necessary blocks were present and then did the XOR operations all at once. This improvement eliminated any work that would result in a data block that had already been recovered. In addition, this method also leveraged memory and cache locality at the system level to reduce memory hits.

Optimized memory XOR:
To further drive down the time needed for XOR operations, we took advantage of the MMX instruction set and wrote a streamlined memory XOR function which used striping to maximize cache usage. Additionally, the modified function reduced the number of necessary registers. The striping provides consistent performance when the blocks are too large to fit into the cache.

Coverage Threshold:
To improve reliability and robustness, all of the original data nodes were checked after encoding to see if they met a "coverage" threshold. We defined coverage as the degree of the data node. If an original node did not have the minimum number of edges, it was included as a source for random encoded blocks until the threshold was met. This process ensured that all of the data was present in the encoding. Ensuring a certain degree of coverage improves, but does not guarantee, 100% decodability.

Original Blocks:
To ensure that the data is always recoverable, all of the original data blocks are copied into the encoded data. By adding all of the original blocks, we can always ensure a successful decoding if no blocks are lost. However, the average amount of useful redundant

information per encoded block is decreased and, on average, more blocks will be needed to perform a successful decoding.

Guaranteed Decodability:
A simple and effective solution for reliability was to run a light-weight version of the decoding algorithm on the bipartite graph during the encoding process. New graphs were generated until a decodable solution was found. Once an acceptable graph was created, the XOR operations were performed. The time cost of performing the check and possibly re-generating the coding graph is minimal compared to the cost of the XOR operations needed to produce the encoded blocks, making this is a reasonable solution.

Uniform Coverage:
To increase the degree of freedom, the idea of coverage was taken a step further. Instead of ensuring a coverage threshold, data nodes were chosen in such a way that they would receive uniform coverage. In order to accomplish this, a random permutation of the data nodes was created and place into a queue. As encoded nodes chose sources, data nodes were removed from the head of the queue and added to the encoded node. Once the queue had been depleted, a new permutation was made and placed in the queue. This process ensures that each of the original nodes is equally represented in the encoding.

Choosing Coding Parameters:
When using the LT Coding process, there are a number of parameters that can be specified and a number of criteria that are useful in evaluating the potential for a certain configuration. The most important metrics for RobuSTore are the decoding speed and the number of encoding blocks needed to recover the original data. However, high speed and a small number of blocks are opposing factors.

The three parameters that affect these two metrics are *k*, the number of data blocks; C, a parameter for the creation of the Robust Soliton Distribution; and δ, the expected rate of failure if $k + O(\sqrt{k} \ln^2(k/\delta))$ encoded blocks are evaluated. While the Ideal Soliton Distribution is a strictly decreasing function that only depends on *k*, the Robust Soliton has an added spike, whose size and location is determined by C and δ. As C increases and δ decreases the spike in the distribution occurs closer to one and increases in size. This results in additional lower degree encoded nodes. Figure 1 shows the average degree of an encoded node for different values of C and a given δ. Table 1 relates that average number of encoded blocks needed for decoding as multiples of *k*, the number of original blocks, for the original implementation.

Figure 1 – Average Node Degree vs. C for different Sigma

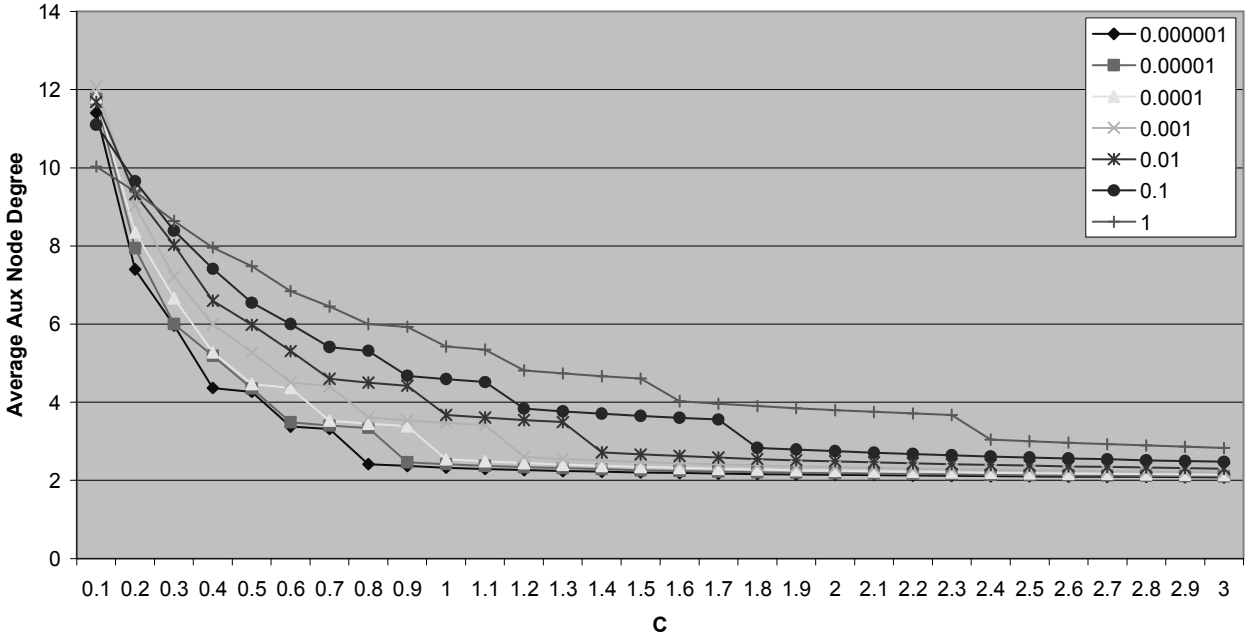


Table 1 – Avg. Blocks Needed for Reconstruction for k=1024 (multiples of k)

Sigma\C	0.1	0.5	1	1.1	1.2	1.3	1.4	1.5	1.7	2	3	5
0.001	1.48	1.98	2.18	2.18	2.90	3.00	2.94	3.03	3.11	3.27	3.42	3.61
0.002	1.45	1.90	2.11	2.17	2.20	2.89	2.97	3.02	3.10	3.15	3.43	3.55
0.005	1.39	1.95	2.08	2.13	2.13	2.20	2.91	2.91	2.98	3.12	3.34	3.56
0.007	1.37	1.90	2.10	2.14	2.17	2.16	2.83	2.84	2.92	3.10	3.29	3.54
0.01	1.36	1.87	2.06	2.07	2.06	2.20	2.80	2.86	2.95	3.05	3.32	3.53
0.02	1.32	1.75	2.05	2.06	2.09	2.06	2.14	2.78	2.85	2.94	3.23	3.53
0.05	1.27	1.72	1.71	2.00	2.06	2.06	2.05	2.18	2.78	2.80	3.13	3.39
0.07	1.26	1.67	1.71	1.73	2.00	2.05	2.10	2.09	2.68	2.78	3.15	3.46
0.1	1.24	1.61	1.65	1.70	1.99	2.04	2.01	2.07	2.10	2.78	3.07	3.32
0.2	1.21	1.57	1.63	1.69	1.65	1.97	2.01	2.01	2.10	2.61	3.05	3.27
0.5	1.18	1.48	1.53	1.60	1.64	1.66	1.93	1.96	2.02	2.10	2.73	3.13
0.7	1.17	1.43	1.50	1.58	1.63	1.64	1.63	1.89	1.95	2.01	2.74	3.11
1	1.16	1.41	1.47	1.48	1.60	1.61	1.63	1.68	1.94	2.01	2.67	3.08

Table 2 gives the product of the average node degree and the average number of blocks needed. This is an important metric when evaluating speed as it is closely related to the number of XOR operations performed in the original algorithm. It is important to notice that the implementations which include scheduled XOR operations have different behaviors which eliminate many of the XOR operations.

Table 2 – Product of Avg. Node Degree and Avg. Blocks Needed for Reconstruction

Sigma\C	0.1	0.5	1	1.1	1.2	1.3	1.4	1.5	1.7	2	3	5
0.001	17.68	10.49	7.56	7.50	7.53	7.63	7.39	7.45	7.47	7.60	7.51	7.57
0.002	17.40	10.09	7.47	7.56	7.50	7.48	7.58	7.54	7.64	7.49	7.64	7.51
0.005	17.01	11.73	7.53	7.53	7.41	7.52	7.60	7.48	7.49	7.62	7.58	7.61
0.007	16.34	11.33	7.62	7.62	7.63	7.46	7.56	7.47	7.42	7.57	7.50	7.54
0.01	15.88	11.18	7.59	7.47	7.28	7.63	7.57	7.61	7.60	7.61	7.67	7.60
0.02	15.39	10.50	7.67	7.56	7.56	7.36	7.54	7.67	7.57	7.48	7.61	7.65
0.05	14.55	11.25	7.65	7.67	7.67	7.52	7.48	7.75	7.65	7.50	7.52	7.48
0.07	14.16	10.96	7.74	7.69	7.61	7.60	7.70	7.57	7.57	7.50	7.66	7.73
0.1	13.93	10.56	7.58	7.64	7.59	7.64	7.52	7.58	7.50	7.54	7.57	7.42
0.2	13.23	10.95	7.67	7.77	7.50	7.64	7.65	7.58	7.69	7.52	7.77	7.50
0.5	12.23	10.75	8.15	7.63	7.66	7.65	7.71	7.75	7.71	7.69	7.43	7.41
0.7	11.97	10.28	8.06	7.59	7.73	7.65	7.54	7.59	7.56	7.49	7.55	7.50
1	11.68	10.61	8.02	7.92	7.71	7.64	7.56	7.70	7.70	7.63	7.55	7.54

From Table 1, we see that smaller values of C and larger values of δ produce encodings that require fewer encoded blocks. This is expected since the average degree of the encoded blocks is highest for these values. Table 2 shows that there is not much variation in the products for values of C greater than 1 and that these values give the lowest values, which translate into the least number of symbol operations. From these observations, we can safely choose values of C greater than, but close to 1, and larger values for δ. While these observations hold for the original implementation and serve as a good basis for testing, the modified algorithms exhibit different behavior to given choices of C and δ.

The third parameter, k, affects several aspects of the algorithm. Most importantly it relates to both the number of necessary blocks, $k + O(\sqrt{k}\ln^2(k/\delta))$, and also the number of symbol operations, $O(k \cdot \ln(k/\delta))$ [1]. As k increases, the number of blocks and the number of symbol operations grow in a non-linear fashion. Therefore, smaller values of k should result in relatively fewer necessary blocks and faster decoding times.

V. Experiment Setup

The following tests fixed the number of data blocks at k=1024 and the file size at 128MB. Based on the average number of encoded blocks needed to decode, we produced $5k$ encoded blocks to ensure that the tests would successfully complete with high probability. Encoded blocks were pushed to the decoder in random order. Ten tests were run for each combination of parameters using dual Xeon 2.4GHz machines with 1GB of memory.

VI. Results from Speed Optimizations

The intension of these modifications was to improve the algorithm's efficiency in order to achieve faster decoding times when using a reasonable number of blocks. The optimizations performed on the LT Code algorithm took two forms, reducing the number of operations and performing the operations more efficiently.

1. The original implementation – Greedy Scheduled & Original XOR

Table 3 - Avg. MBps

Sigma\C	1	1.1	1.2	1.3	1.4	1.5	1.7
0.1	67.88516	65.1774	72.42279	70.40005	71.03017	69.37574	75.36792
0.2	69.56745	67.87941	69.37398	72.94715	73.02349	77.65614	73.15825
0.5	67.5528	70.12628	69.07971	68.97836	73.83843	71.67508	73.76106
0.7	64.70222	71.94008	68.43411	70.80492	69.89602	70.71551	72.33103
1	67.92839	65.34122	69.00225	71.69747	67.93825	71.09644	71.51722

2. Optimized Scheduling & Original XOR

Table 4 - Avg. MBps

Sigma\C	1	1.1	1.2	1.3	1.4	1.5	1.7
0.1	187.2019	188.4609	196.6718	197.7784	202.3057	211.1028	203.1425
0.2	185.3084	185.7814	182.276	187.6694	201.9992	200.8429	199.5137
0.5	169.1851	182.5684	180.6636	186.5384	192.8676	187.4026	204.4487
0.7	164.5086	185.0814	178.6015	191.4239	182.9313	196.9798	185.7542
1	174.2982	172.3834	177.4054	190.4721	182.1788	179.9855	192.6644

The dramatic speed up from the optimized scheduling algorithm can be explained by the reduction of XOR operations. With this scheduling scheme over half of the original XOR operations can be eliminated. Below are tables of the average number of 2-input XOR operations that were performed during the decoding process before and after the optimized scheduling was introduced. While calls to the memory XOR function consume the majority of the processing time, the actual bottleneck for the system is the memory throughput required by the function. For each XOR operation in the original scheduling, two reads and one write must be performed. However, for the optimized scheduling, many of the memory reads can be avoided since the intermediate results remain in the cache. More precisely, for an encoding with k original data blocks, there will only be k memory write operations and $k + (\text{avg. XORs})$ memory reads during the entire decoding process.

Table 5 - Avg. XOR Operations – Greedy Scheduling (Original)

Sigma\C	1	1.1	1.2	1.3	1.4	1.5	1.7
0.7	8216.54	7541.24	7637.93	7625.51	7604.23	7457.19	7465.07
0.2	7726.09	7579.38	7571.47	7723.49	7593.58	7572.73	7426.41
0.1	7713.39	7799.26	7769.66	7785.86	7817.45	7638.43	7541.22
0.5	7976.1	7640.36	7577.77	7634.76	7661.36	7752.36	7447.94
1	8134.86	8048.14	7813.82	7822.09	7599.22	7705.47	7831.12

Table 6 -Avg. XOR Operations – Optimized Scheduling

Sigma\C	1	1.1	1.2	1.3	1.4	1.5	1.7
0.1	3782.16	3743.89	3386.86	3403.97	3351.59	3137.05	3032.59
0.2	3919.03	3878.25	3716.28	3436.42	3417.73	3216.78	3127.58
0.5	4451.07	4006.23	3849.98	3952.8	3693.01	3557.17	3485.74
0.7	4366.7	3976.05	4003.21	3911.45	3829.87	3565.12	3432.12
1	4436.99	4357.59	4094.96	4025.7	3840	3845.2	3660.5

3. Optimized Scheduling & Optimized XOR

The memory XOR function was modified in two ways. First, the 2-input 32-bit operation was replaced with the Intel MMX 64-bit version. This reduced the number of processor cycles devoted to computing the XOR and also made use of the processor's special MMX registers, leaving the general register open for other values. Second, data striping was introduced so that the operands and any intermediate results could stay in the cache until they were no longer needed. Below are the results of the improved memory XOR operation.

Table 7 - Avg. MBps

Sigma\C	1	1.1	1.2	1.3	1.4	1.5	1.7
0.1	182.8556	181.7817	197.934	206.3672	210.8067	202.2333	208.7896
0.2	173.2348	193.8107	192.4676	199.893	197.0093	200.3505	205.4907
0.5	172.3564	179.4596	181.2496	189.5616	192.2246	192.8085	202.6708
0.7	167.2648	176.4274	174.8152	172.2848	172.2233	193.2714	197.0925
1	174.6569	174.2886	181.3088	183.7134	185.6997	179.0326	196.1095

The optimized XOR function does not show any notable performance benefit for these testing parameters since it has the greatest effect when used with larger data blocks (smaller values of k). With the current parameters, the block size is 128KB, which is already a good match for the caches on the test machines. For larger blocks, the optimized version can maintain higher throughput compared to the performance of the original XOR implementation, which deteriorates due to additional cache misses. Ultimately, the performance is bounded by the memory throughput, thus utilizing the caches is critical.

VII. Results from Reliability and Robustness Modifications

After the speed optimizations had been integrated into the algorithm, several modifications were made to improve the reliability and robustness of the encodings. The following results were obtained with the optimized scheduling and optimized XOR options enabled.

1. Including the Original Blocks

The encoder was set to include all of the original data blocks as encoded blocks. Since all of the original blocks were already included in the encoding, we did not allow the encoder to produce any other encoded nodes with degree equal to one. By including all of the original blocks, we can ensure that all encodings will be decodable.

Table 8 - Avg. MBps

Sigma\C	1	1.1	1.2	1.3	1.4	1.5	1.7
0.1	194.8817	199.9829	207.171	221.7728	215.1289	223.6352	226.2422
0.2	195.7527	194.4314	188.7477	211.542	212.5632	208.8365	214.9657
0.5	170.8894	194.1178	198.2732	191.8115	204.8867	210.5193	205.5439
0.7	180.2153	194.303	195.2366	190.9267	189.0547	192.2865	213.0392
1	173.7224	181.5339	178.0226	181.3043	193.8084	187.1918	197.3282

Table 9 - Avg. Blocks as multiples of k

Sigma\C	1	1.1	1.2	1.3	1.4	1.5	1.7
0.1	1.76123	1.669238	2.15791	2.114063	2.172363	2.229102	2.405957
0.2	1.793359	1.695898	1.691504	1.932031	2.243457	2.273828	2.130078
0.5	1.534863	1.842383	1.863672	1.875488	2.109473	2.038672	1.931738
0.7	1.51582	1.739551	1.728418	1.972168	1.658496	1.991504	2.20459
1	1.433496	1.521973	1.59502	1.699902	1.739258	1.808398	1.971484

It is clear to see that more encoded blocks are needed on average in order to reconstruct the original data. In addition, the decoding speeds are noticeably faster. There results are expected since the inclusion of the original data blocks in the encoded data decreases the average node degree. In this scenario, more blocks are needed to account for all of the original blocks, but few XOR operations are required.

2. Specifying a Coverage Threshold

During the encoding process, the algorithm makes sure that every data node is part of at least some number of encoded nodes. For this test, we set the minimum coverage at three. This check ensures that none of the original blocks are missed during the random assignment of edges. Furthermore, it makes sure that the entire decoding process is not dependent on a small number of encoded blocks that contain information about a poorly covered data node. While this implementation reduces the possibility that the data can not be recovered, it does not fully eliminate it.

Table 10 - Avg. MBps

Sigma\C	1	1.1	1.2	1.3	1.4	1.5	1.7
0.1	183.8205	183.3571	201.5032	199.1531	198.8691	202.7484	207.0833
0.2	179.0851	179.8119	182.4829	189.5924	201.2879	194.2353	204.6021
0.5	169.1483	177.1901	179.7195	184.3208	199.8105	185.4573	195.9053
0.7	165.878	179.7948	172.9102	180.4744	186.5881	189.2392	185.057
1	167.9597	166.8491	171.8411	175.4305	171.0451	174.0317	191.5861

Table 11 - Avg. Blocks as multiples of k

Sigma\C	1	1.1	1.2	1.3	1.4	1.5	1.7
0.1	1.69458	1.678613	2.041309	1.995264	1.943066	2.084814	2.109033
0.2	1.640234	1.58457	1.719531	1.917725	2.007959	1.964551	2.068701
0.5	1.553076	1.694434	1.638379	1.65	2.001807	2.016553	1.926074
0.7	1.514111	1.500977	1.565967	1.666162	1.779102	1.851563	1.973096
1	1.454688	1.510889	1.590674	1.522607	1.662842	1.706348	1.925732

3. Including the Original Blocks and Specifying a Coverage Threshold

While the results from the Coverage Threshold tests were very good, the possibility of losing data due to an unrecoverable encoding is unacceptable. To address this problem, the encoder included all of the original blocks in the encoded data and made sure that every data node was part of at least three encoded nodes. Unlike the previous implementation which included the original blocks only once, we allowed the encoder to produce

additional nodes of degree one. The original LT Coding algorithm does not check for identical encoded blocks, and in an effort to mimic that model, we did likewise. Unfortunately, this produces any abundance of lower degree nodes, which forces the decoder to use even more blocks during decoding. One benefit to the numerous low degree blocks is fewer XOR operations, and hence a faster decoding throughput.

Table 12 - Avg. MBps

Sigma\C	1	1.1	1.2	1.3	1.4	1.5	1.7
0.1	205.4863	205.0873	219.0133	224.481	228.8149	228.0048	237.1714
0.2	191.0704	202.9727	194.6265	215.5549	208.6296	229.4761	226.5123
0.5	192.7276	191.023	205.4291	202.4576	213.9331	213.5806	222.1768
0.7	185.8853	199.9064	199.729	197.7476	186.2845	206.7715	209.5227
1	178.2516	176.2841	200.591	189.8512	185.436	199.2595	212.8395

Table 13 - Avg. Blocks as multiples of k

Sigma\C	1	1.1	1.2	1.3	1.4	1.5	1.7
0.1	1.846387	1.958105	2.31709	2.278418	2.344922	2.38457	2.495996
0.2	1.839551	1.885938	1.935352	2.302246	2.291504	2.435156	2.572461
0.5	1.86709	2.025098	1.938379	2.021875	2.006738	2.423242	2.224707
0.7	1.660059	1.95	1.946582	1.980273	2.041602	2.073047	2.130957
1	1.768457	1.614746	1.892188	1.929102	1.863281	1.941211	2.081738

4. Specifying a Coverage Threshold and Guaranteed Decodability

In order to ensure decidability without the reception overhead of including all of the original blocks, a light-weight version of the decoding algorithm was run on the bipartite meta-data graph during the encoding process. This check and the possible graph regeneration were very quick in comparison to the XOR operations. This modification guarantees decodability but also maintains the decoding speed and reception overhead of the optimized original implementation.

Table 14 - Avg. MBps

Sigma\C	1	1.1	1.2	1.3	1.4	1.5	1.7
0.1	185.525	187.6934	198.7276	209.3207	204.422	201.9158	213.1997
0.2	184.1342	184.7919	180.1663	196.4269	204.5893	207.2135	207.9365
0.5	167.23	180.3972	180.0027	186.9943	192.6963	192.3808	200.8509
0.7	169.2232	176.2942	181.1987	185.9171	192.1051	195.8659	202.2354
1	167.2817	170.4065	182.4955	184.1416	184.6647	183.4999	195.0134

Table 15 - Avg. Blocks as multiples of k

Sigma\C	1	1.1	1.2	1.3	1.4	1.5	1.7
0.1	1.658691	1.615869	1.949658	2.098877	2.150781	2.033252	2.056641
0.2	1.660889	1.657959	1.636084	1.896045	2.005957	2.022559	2.0979
0.5	1.559961	1.535205	1.660205	1.649219	1.948193	2.059326	2.035254
0.7	1.554541	1.581055	1.568213	1.539502	1.632813	2.075049	2.021338
1	1.512207	1.457715	1.499121	1.54292	1.568066	1.778906	1.911768

5. Uniform Coverage and Guaranteed Decodability

To increase the degree of freedom, the idea of coverage was taken a step further. Instead of setting a lower threshold on a data node's degree, we attempt to give all data nodes the same degree. This ensures that all pieces of the original data are equally represented in the encoding.

Table 16 - Avg. MBps

Sigma\C	1	1.1	1.2	1.3	1.4	1.5	1.7
0.1	177.1758	185.8256	204.1372	200.8858	209.4234	217.0385	212.7888
0.2	176.6163	175.1468	187.8626	201.1126	202.57	211.1644	205.471
0.5	166.339	171.1617	184.316	176.5173	198.6438	191.7638	208.7998
0.7	160.569	174.143	179.1221	182.7572	184.8161	197.1052	202.9509
1	127.8182	168.0613	176.2462	182.5931	182.0857	174.389	195.6421

Table 17 - Avg. Blocks as multiples of k

Sigma\C	1	1.1	1.2	1.3	1.4	1.5	1.7
0.1	1.52041	1.574219	1.545605	1.60625	1.725879	1.753906	1.791895
0.2	1.396973	1.513184	1.544434	1.596191	1.727832	1.662695	1.839551
0.5	1.534473	1.413672	1.451074	1.420313	1.686328	1.539551	1.627441
0.7	1.453711	1.402539	1.516602	1.448145	1.4375	1.642578	1.689844
1	1.431445	1.48584	1.370215	1.458203	1.368066	1.430078	1.547852

It is interesting to note that while the computation speeds are very similar to the implementation with a coverage threshold, the average number of blocks needed for decoding is noticeably lower. This suggests that the uniform coverage has provided some level of improvement in the degree of freedom for the encoding.

VIII. Evaluation of Network and Computation Speeds

In addition to the processing time needed to decode a set of encoded blocks, it is also important to consider the total time necessary to both receive and process the blocks. As encoded blocks are received, some of the XOR operations can be performed while waiting for the rest of the necessary blocks. Our goal is to balance the receiving and processing rates to optimize our efficiency and speed. In order to apply our results to a wide variety of hardware configurations, our simulations used the ratio between the network bandwidth and the computation bandwidth as a parameter. Several simulations were run using the guaranteed decodability and uniform coverage implementation and a wide range of values for C and δ to investigate the overall performance when this parallelism is considered.

Our results indicate that there is a relatively small window where network bandwidth and computation speed can be balanced using C and δ. Experimentation showed that for ratios of network to computation speed less than 0.06, parameters giving very high average node degrees, and hence the fewest number of necessary blocks for decoding, provided the best performance. This indicates that the system bottleneck is the network speed. For network to computation ratios greater than 0.8 extremely low average node degrees gave the best performance. For these configurations, a bottleneck in computation speed creates

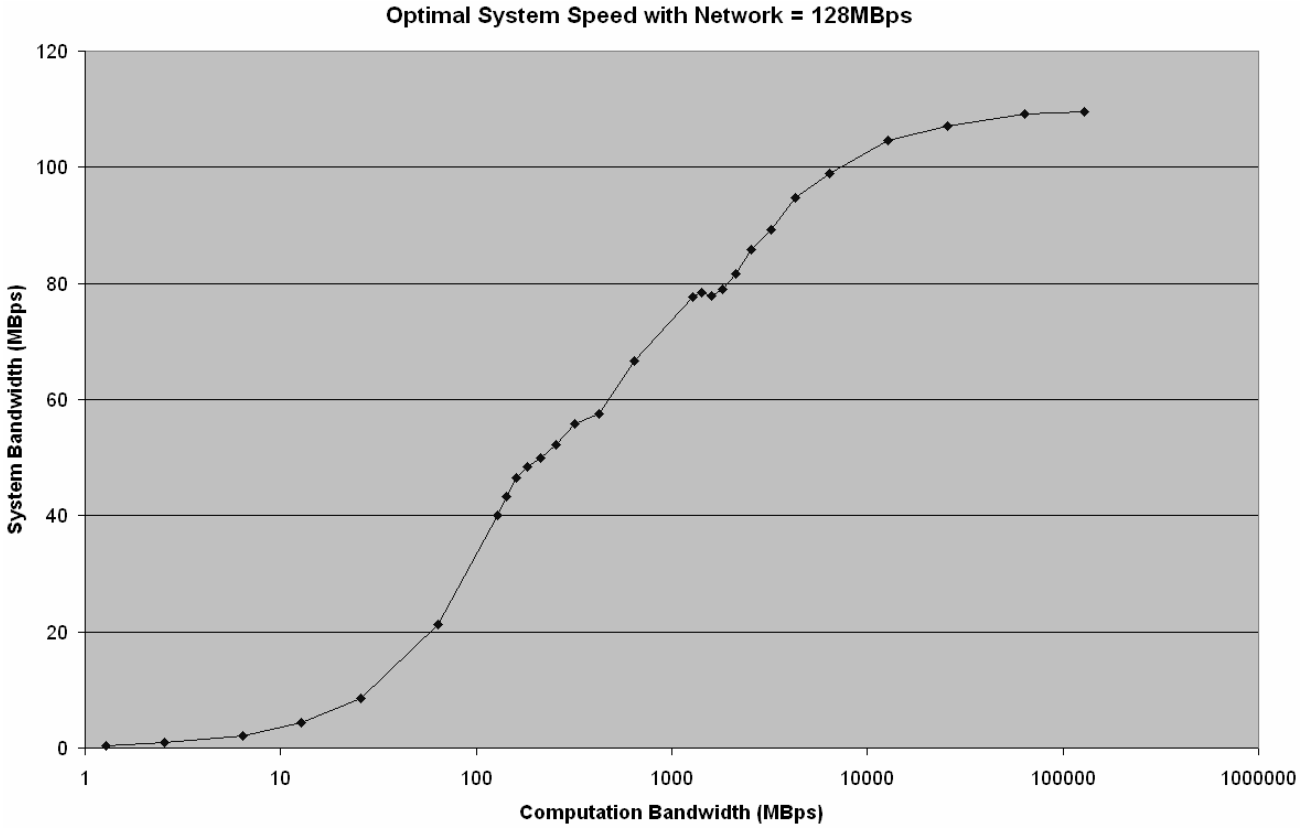
a heavy reliance on the network to transport a large quantity of low degree nodes which can be rapidly evaluated. Table 18 summarizes the best coding parameters along with the average node degrees for given network to computation bandwidth ratios.

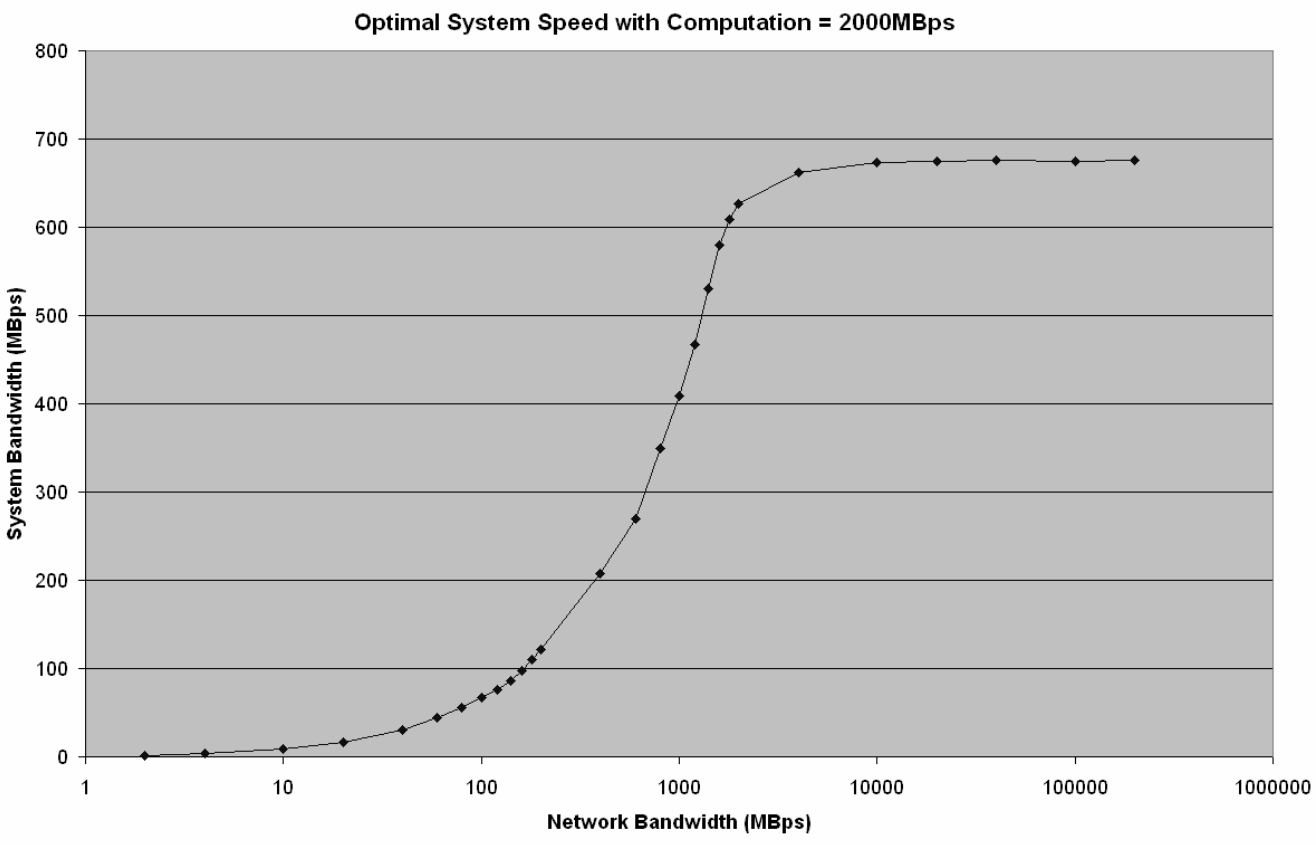
In order to see the benefits of scaling the network and computational bandwidth of a system, we evaluated the theoretical performance of our current test machines. For our systems, the measured memory bandwidth for the memory XOR operation is 2.1 GBps and the network interface is specified at 1 Gbps (~128MBps). The figures below show the total system throughput using the optimal coding parameters when scaling either network bandwidth or computation bandwidth.

From the figures, we can see that significant increases in the memory bandwidth from 2000MBps will produce no more than a 30% improvement in overall throughput. However, doubling the network bandwidth will nearly double the system performance!

Table 18 – Optimal Coding parameters

Net2comp	C	Sigma	Avg. Degree
< 0.06	Small	1	High
0.06	0.1	1	10.03576
0.07	1.2	1	4.815068
0.08	1.4	0.5	4.000171
0.09	1.6	1	4.025548
0.1	1.4	0.5	4.000171
0.2	2	1	3.799221
0.3	2	0.2	2.864407
0.4	1.9	0.05	2.690313
0.5	5	0.7	2.412966
0.6	10	0.7	2.148624
0.7	30	0.00001	1.969518
0.8	100	0.00005	1.958965
0.9	100	0.1	1.940646
> 0.9	Large	> 0.0001	Low



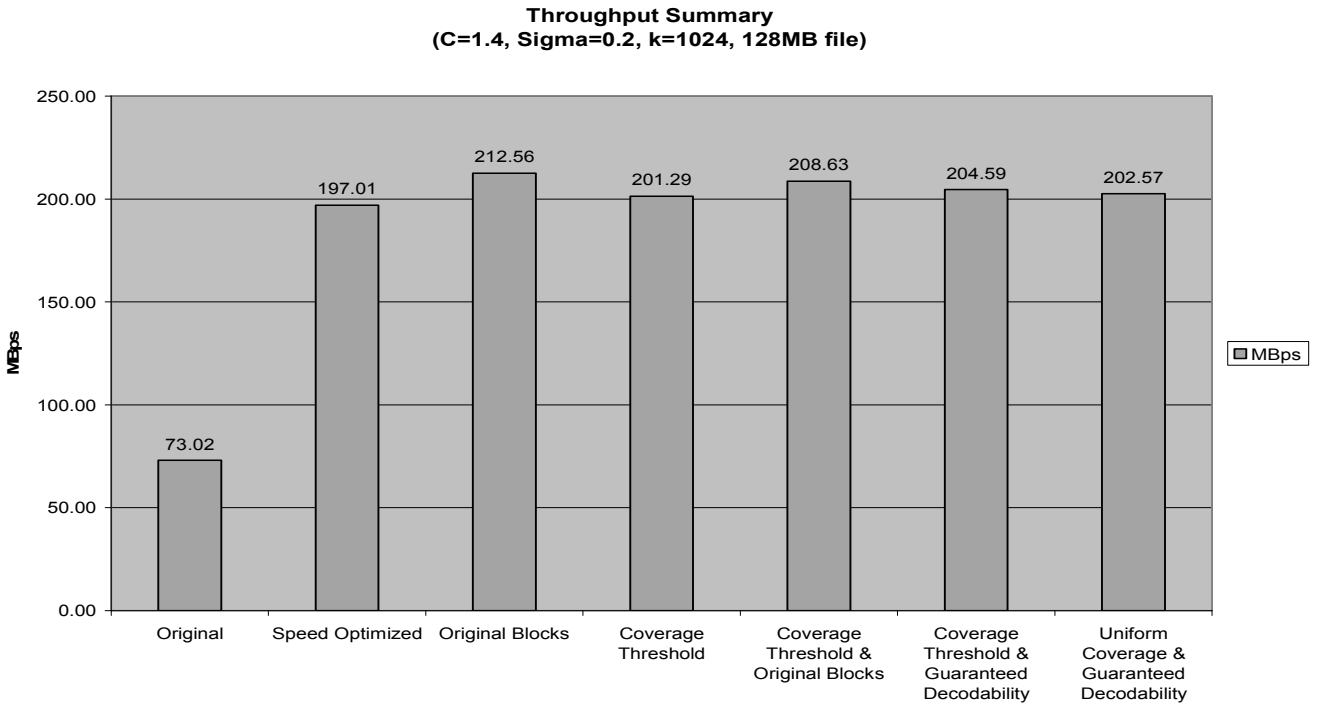


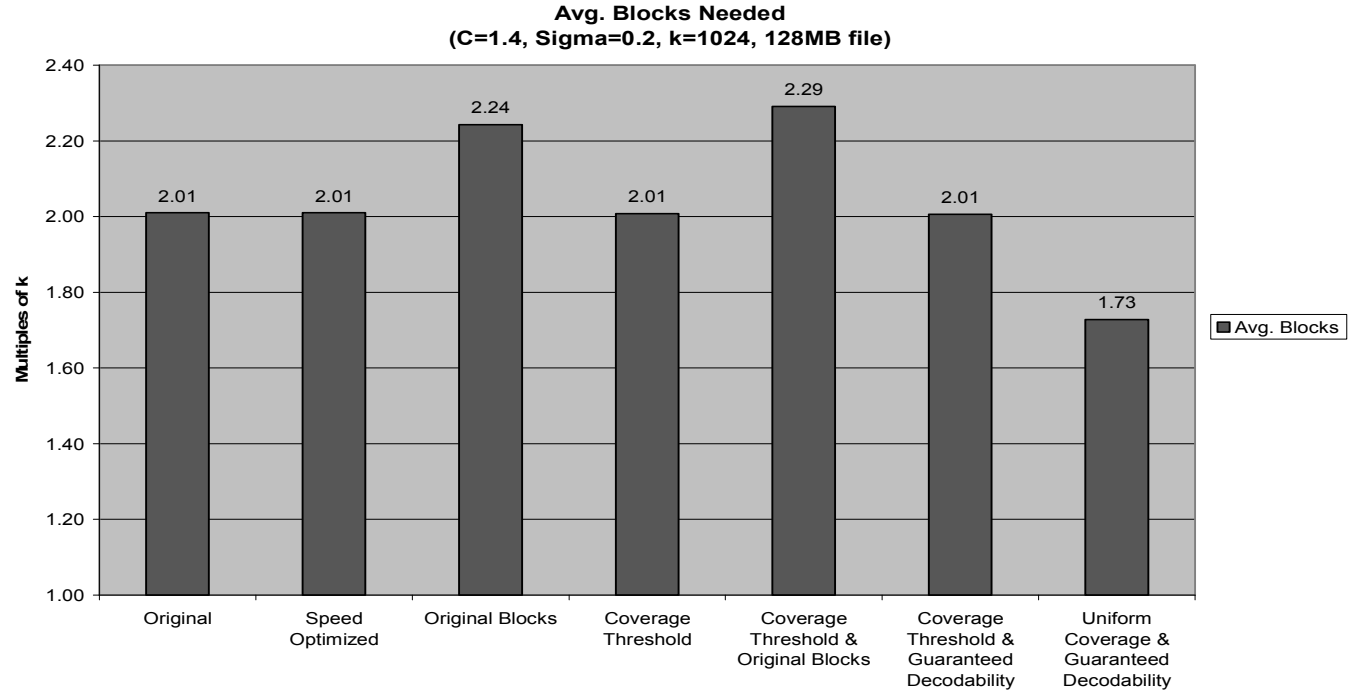
IX. Related Work

The idea of using erasure codes for reliable storage is not a completely new one. Material has been published about Lincoln Erasure Code implementations for distributed storage, and also on research done at the University of California, Berkley on the OceanStore project. The OceanStore project has implemented Reed-Solomon codes and Tornado codes to provide data reliability [3], but the computation costs of these codes constrain their actual throughput [4]. Researchers at MIT's Lincoln Labs have developed their own erasure code implementation, the Lincoln Erasure Code, which sends the original data blocks along with a specified number of parity blocks. In their 2003 report, their implementation claimed an advantage over standard LT codes of 760 Mbps to 220 Mbps for a 13MB file on a 1.7GHz Xeon machine [4].

X. Summary and Future Work

Through our study of LT codes for RobuSTore we have investigated necessary components of a practical algorithm and worked to shape a specialized version of LT codes to meets those requirements. Although Cooley, et. al., had misgivings about the speed and reliability of LT codes for distributed storage [4], our experiments show that LT codes can perform at high speeds with reasonable reception overheads. Below is a summary of our different implementations evaluated on decoding speed and necessary blocks.





In addition, it is interesting to look at the trade off between decoding speed and necessary blocks. For our later implementations, it becomes apparent that larger values of C produced faster decoding speeds, but also require more blocks. In tests with extreme values of C, consistent decoding speeds as high as 324MBps were achieved with an average of 2.5 times the number of original blocks. Conversely, low values of C and large values of sigma can produce encodings with low reception overheads (<1.2) while still

maintaining throughputs over 100MBps. Based on our analysis of the overall reception and computation throughput, the values of C and sigma can be adjusted to give optimal performance for a given hardware configuration.

Ongoing work for LT Codes to be used in RobuSTore should focus on the robustness of the encoding. Developing a means to evaluate the maximum degree of freedom, as well as, better methodologies to achieve maximum freedom would ensure better data reliability and allow RobuSTore more flexibility when receiving data from storage sites. The goal is to achieve this without significantly degrading the decoding bandwidth or increasing the average number of necessary blocks for decoding. The implementations using uniform coverage should have improved degree of freedom but an accurate way to compare their performance to that of the other implementations is still needed.

While obstacles still remain in creating a high-performance distributed file systems based on erasure coding, the work done thus far with LT Codes shows the feasibility and potential for such systems. Without any special hardware, decoding speeds exceeding 200MBps were achieved using fewer than 1.7 times the number of original blocks. These speeds will come close to saturating 4-Gigabit links while providing low latency access to distributed data. In addition, we see that even with our current compute capabilities, increasing memory bandwidth does not give much advantage while using a 1GBps network link. However, the advance of network bandwidths will allow sizable improvements in the overall system speed using our current compute hardware. With the current trends in the growth of network throughput versus processor speed, RobuSTore is in a position to offer excellent high performance, fault-tolerant service in the coming years.

Acknowledgements:
Supported in part by the National Science Foundation under awards NSF Cooperative Agreement ANI-0225642 (OptIPuter), NSF CCR-0331645 (VGrADS), NSF ACI-0305390, and NSF Research Infrastructure Grant EIA-0303622. Support from the California Institute for Telecommunications and Information Technology, UCSD Center for Networked Systems, BigBangwidth, and Fujitsu is also gratefully acknowledged.

Appendix A: Function Headers

Function: Encode()
Description: Given a segment of original data, this function will encode it and return N encoded data blocks in pDataBlocks (memory space already allocated) and a serialized version of the coding graph.

Returns: TRUE if the original data was successfully encoded, FALSE, otherwise.

int Encode(void *pOrigData, int origSize, int K, int N, EncInfo **ppInfo, int* infoSize,
DataBlock **pEncData);

void *pOrigData	Original data chunk
int origSize	The number of bytes of original data
int K	Number of original data blocks
int N	Number of total encoded data blocks
EncInfo **ppInfo	Graph and file information produced when encoding, malloc() by this function, MUST be freed with EncRelease() after usage.
int* infoSize	Address of an int to pass back the size of the EncInfo
DataBlock **pEncData	N encoded blocks to be returned, memory space will be allocated and MUST be freed later.

Function: DecInit()
Description: Initializes the decoder by inputing the coding graph and specifying where to store original data segment.

Returns: an initialized DecState object to be used with DecPush(). The returned memory MUST be deallocated with a call to DecRelease().

DecState* DecInit(EncInfo* pInfo, void *pOrigData);

EncInfo *pInfo	Graph and block size produced when encoding.
void *pOrigData	Allocated memory space for reconstructed data.

Function: DecPush()
Description: Pushs an encoded block into the decoding ripple, and tell if it's ready to reconstruct the original data segment.

Returns: True if enough blocks have been received to reconstruct the original data, and the data is available.

int DecPush(DataBlock* pDataBlock, DecState* state);

DataBlock* pDataBlock	The next data block to evaluate
DecState* state	State information from previous calls to DecPush()

References

[1] Luby, Michael, "LT Codes", *43rd Annual IEEE Symposium on Foundations of Computer Science*, 2002.

[2] Khisti, Ashish, "Tornado Codes and Luby Transform Codes"

[3] Kubiatowicz, J., et. al. "OceanStore: An Architecture for Global-Scale Persistent Storage." *Proceedings of the Ninth International Conference on Architectual Support of Programming Languages and Operationg Systems* (ASPLOS 2000), November 2000, 190-201.

[4] Cooley, Joseph A., et. al. "Software-base Erasure Codes for Scalable Distributed Storage". *Twentieth IEEE/Eleventh NASA Goddard Conference on Mass Storage Systems & Technologies*, April 2003

[5] Chien, Andrew, et al. "OptIPuter System Software Framework", UCSD Technical Report CS2004-0786