Assessing the Performance of Erasure Codes in the Wide-Area

Rebecca L. Collins James S. Plank

Technical Report UT-CS-04-536
Department of Computer Science
University of Tennessee
November 2004

Abstract

The problem of efficiently retrieving a file that has been broken into blocks and distributed across the wide-area pervades applications that utilize Grid, peer-to-peer, and distributed file systems. While the use of erasure codes to improve the fault-tolerance and performance of wide-area file systems has been explored, there has been little work that assesses the performance and quantifies the impact of modifying various parameters. This paper performs such an assessment. We modify our previously defined framework for studying replication in the wide-area [6] to include both Reed-Solomon and Low-Density Parity-Check (LDPC) erasure codes. We then use this framework to assess the performance of erasure coding in three wide-area, distributed settings. We conclude that as the set size, encoding rate, type of encoding and file distribution change, performance reflects a trade-off between downloading time and decoding time.

1 Introduction

The coordination of widely distributed file servers is a complex problem that challenges wide-area, peer-to-peer and Grid file systems. Systems like OceanStore [17], LoCI [3], and BitTorrent [5] aggregate wide-area collections of storage servers to store files on the wide-area. The files are broken into fixed-size blocks, which are distributed among the disparate servers. To address issues of locality and fault-tolerance, *erasure coding* is employed, albeit normally in the primitive form of block replication. When a client needs to download a file from this collection of servers to a single location, he or she is faced with a variety of decisions to make about how to perform the download. This has been termed the “Plank-Beck” problem by Allen and Wolski, who deemed it one of two fundamental problems of data movement in Grid Computing systems [1].

In previous work, we defined a a framework for evaluating algorithms for downloading replicated wide-area files [6]. In this paper, we extend this framework to include erasure coding, and perform an evaluation of algorithms for downloading erasure-encoded files in three wide-area settings. This work is significant as it is the first piece of work to evaluate the relative performance of replication, Reed-Solomon codes and LDPC codes in a realistic, wide-area environment with aggressive, multithreaded downloading algorithms. Our results show that performance trends are influenced by three main factors: encoding rate, set size, and file distribution. The two types of erasure codes present different downloading and decoding trade-offs, and in most cases, Reed-Solomon codes perform well when sets are small, while LDPC codes perform well when sets are large.

2 The “Plank-Beck” Problem

We are given a collection of storage servers on the wide-area, and a large file that we desire to store on these servers. The file is partitioned into blocks (termed “data blocks” of a fixed size, and an erasure coding scheme is used to calculate some number of additional “coding” blocks. The collection of data and coding blocks are then distributed among the storage servers. At some point, a client at a given network location desires to download the file, either in its entirety, or in a streaming fashion. Simply stated, the “Plank-Beck” problem is: How does the client download the file with the best performance.

The simplest erasure coding scheme is replication – each block of the file is stored at more than one server. In this case, algorithms for solving the “Plank-Beck” problem may be parameterized in four dimensions [6]:

1. The number of simultaneous downloads.
2. The degree of work replication
3. The failover strategy.
4. The server selection strategy.

The optimal choice of parameters depends on many factors, including the distribution of the file and the ability for the client to utilize monitoring and forecasting [20]; however, a few general conclusions were drawn. First, the number of simultaneous downloads should be large enough to saturate the network paths to the client, without violating TCP-friendliness in shared networks. Second, work replication should be present, but not over-aggressive; ideally, it combines with a failover strategy that performs work replication when the progress of the download falls below a threshold. Finally, if monitoring software is present, fast servers (with respect to the client) should be selected, although ideal algorithms consider both the forecasted speed from server and the load induced on the server by the download itself.

When we add more complex erasure coding, some additional parameters manifest themselves. We define them after explaining erasure codes in more detail.

3 Erasure Coding

Erasure codes have arisen as a viable alternative to replication for both caching and fault-tolerance in wide-area file systems [4, 16, 18, 21]. Formally defined, with erasure coding, n data blocks are used to construct m coding, (or "check") blocks, where data and check blocks have the same size. The encoding *rate* is $\frac{n}{n+m}$. Subsets of the data and coding blocks may be used to reconstruct the original set of data blocks. Ideally, these subsets are made up of any n data or check blocks, although this is not always the case. Replication is a very simple form of erasure encoding (with $n = 1$ and $m > 1$). In comparison to replication, more complex erasure coding techniques (with $n > 1$ and $m > 1$) reduce the burden of physical storage required to maintain high levels of fault tolerance. However, such erasure coding can introduce computationally intensive encoding and decoding operations. We explore the two most popular types of erasure coding in this paper: Reed-Solomon coding, and Low-Density Parity-Check (LDPC) coding. They are summarized below.

3.1 Reed-Solomon Coding

Reed-Solomon codes have been used for fault-tolerance in a variety of settings [7–9, 12, 21]. Their basic operation is as follows. The n data blocks are partitioned into *words* of a fixed size, and a collection of n words forms a vector. A *distribution matrix* is employed so that the check blocks are calculated from the vector with m dot products. Galois Field arithmetic is used so that all elements have multiplicative inverses. Then the act of decoding is straightforward. Given any n of the data and check blocks, a decoding matrix

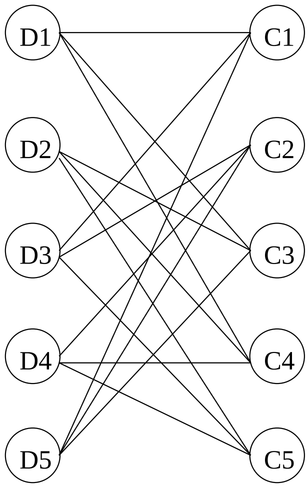

Figure 1. An example LDPC code where $n = 5$ and $m = 5$

may be derived from the distribution matrix, and the remaining data blocks may be calculated again with dot products.

A tutorial on Reed-Solomon coding is available in [13, 15]. Reed-Solomon coding is expensive for several reasons. First, the calculation of check block requires an n-way dot product. Thus, creating m check blocks of size B has a complexity of $O(Bmn)$, which grows rather quickly with n and m. Second, decoding involves an $O(n^3)$ matrix inversion. Third, decoding involves another n-way dot product for each data block that needs to be decoded. Fourth, Galois-Field multiplication is more expensive than integer multiplication. For that reason, Reed-Solomon coding is usually deemed appropriate only for limited values of n and m. See [4, 16] for some limited evaluations of Reed-Solomon coding as n and m grow.

3.2 LDPC Coding

LDPC coding is an interesting alternative to Reed-Solomon coding. LDPC codes are based on bipartite graphs, where the data blocks are represented by the left-hand nodes, and the check blocks are represented by the right-hand nodes. An example for $n = 5$ and $m = 5$ is given in Figure 1. Each check block is calculated to be the bitwise exclusive-or of all data blocks incident to it. Thus, for example, block $C1$ in Figure 1 is equal to $D1 \oplus D3 \oplus D5$. A check block can only be used to reconstruct data blocks to which it is adjacent in the graph. For example, check block $C1$ can only be used to reconstruct data block $D1$, $D3$, or $D5$. If we assume that the blocks are downloaded randomly, sometimes more than n blocks are required to reconstruct all of the data blocks. For example, suppose blocks $D3, D4, D5, C2$, and $C5$ are retrieved. There is no way to reconstruct block $D1$ with these blocks, and a sixth block must be retrieved. Furthermore, if the next block randomly chosen is $D2$ (even though it can be decoded with blocks $D3, D4$, and $C5$), then it is still impossible to reconstruct block $D1$ and a seventh block is required. A graph G's overhead, $o(G)$, is the average number of blocks required to decode the entire set, and its *overhead factor* $f(G)$ is equal to $o(G)/n$.

Unlike Reed-Solomon codes, LDPC codes have overhead factors greater than one. However, LDPC codes have a rich theoretical history which has shown that infinitely sized codes of given rates have overhead factors that approach one [10, 19]. For small codes ($n \leq 100$), optimal codes are in general not known, but an exploration has generated codes whose overhead factors are roughly 1.15 for a rate of $\frac{1}{2}$ [16] We use selected codes from this exploration for evaluation in this paper.

4 Experimental Setup

We performed two sets of experiments to assess performance. In the first, we only study Reed-Solomon codes, and in the second, we study both Reed-Solomon and LDPC codes. The experiments were "live" experiments on a real wide-area network. The storage servers were IBP servers [14], which serve time-limited, shared writable storage in the network. The actual servers that we used were part of the Logistical Backbone [2], a collection of over 300 IBP servers in various locations worldwide.

In all experiments, we stored a 100 MB file, decomposed into 1 MB blocks plus coding blocks, and distributed them on the network. Then we tested the performance of downloading this file to a client at the University of Tennessee. The client machine ran Linux RedHat version 9, and had an Intel (R) Celeron (R) 2.2 GHz processor. Since the downloads took place over the commodity Internet, the tests were executed in a random order so that trends due to local or unusual network activity were minimized, and each data point presented is the average of ten runs.

5 Reed-Solomon Experiments

Section 5.1 summarizes the framework used to organize the experimental parameter space, while section 5.2 describes the network files used in the experiments.

5.1 The Framework for Experimentation

The problem of downloading wide-area files can be defined along four dimensions [6]:

Table 1. Reed-Solomon Experiment Space

Parameter	Range of Parameters
Simultaneous Downloads	$T \in [20, 30]$
Work Replication and Failover Strategy	$\{R, P\} \in [\{1, 1\}, \{2, (30/n)\}]$, static timeouts
Server Selection	**Fastest**$_0$, **Fastest**$_1$
Coding	$n \in [1, 2, 3, 4, 5, 6, 7, 8, (9)]$, $m \in [1, 2, 3, 4, 5, 6, 7, 8, (9)]$, such that $\frac{m}{n} <= 3$

1. **Threads (T)**: How many blocks should be retrieved in parallel?

2. **Redundancy (R)**: How many copies of the same block can be retrieved in parallel?

3. **Progress (P)**: How much progress in the download can take place before we aggressively retry a block?

4. **Server Selection Algorithm**: Which replica of a block should be retrieved when we are offered a choice of servers holding the block? We focus on two algorithms from [6], called **fastest**$_0$ and **fastest**$_1$. **Fastest**$_0$ always selects the fastest server. **Fastest**$_1$ adds a "load number" for the number of threads currently downloading from each server, and selects the server that minimizes this combination of predicted download time and load number. These two algorithms were the ones that performed best in wide-area downloads based solely on replication [6].

Based on the previous results and some preliminary testing, we experimented over the range of parameters in the four dimensions of the framework that were expected to produce the best results; they are listed in table 1, in addition to the values of n and m that were tested. Note that $\frac{m}{n}$ is the factor of coding blocks employed. For example, if $\frac{m}{n}$ is two, then the file consumes 300 MB of storage — 100 for the file, and 200 for the coding blocks. We limit $\frac{m}{n}$ to be less than or equal to three, as greater values consume what we would consider an unacceptable amount of storage for 100 MB of data.

5.2 Network Files

We employ three network files in the experiments. These differ by the way in which they are distributed on the wide-area network. The first is called the **hodgepodge distribution**. In this file, fifty regionally distinct servers are chosen and the data and check blocks are striped across these fifty servers. None of the servers is in Tennessee. The next two network files are the **regional distribution** and the **slow regional**

distribution. In these files, the data and check blocks are striped across servers in four regions. The regions chosen for the regional distribution are Alabama, California, Texas, and Wisconsin; and the regions chosen for the slow regional distribution are Southeastern Canada, Western Europe, Singapore, and Korea.

5.3 Expected Trends

There are several trends that we anticipate in our experiments with Reed-Solomon codes. Here we define a *set* of blocks to be a group of $n + m$ data/coding blocks that may be used to decode n data blocks.

- Performance should improve as m increases while n is fixed. Having more redundancy means that there are more choices of which n blocks can be used to decode, and more choices mean that the average speed of the fastest n blocks in the set is likely faster. A related trend is that performance should decline as n increases while m remains fixed.

- As the set size increases and the encoding rate remains fixed, performance will be better or worse based on the impact of two major factors:

    1. A larger set implies a larger server pool from which n blocks can be retrieved, and should improve performance for the following reason: If it is assumed that slow and fast servers are distributed randomly among the sets, then when sets are larger, it is less likely that any one set is composed entirely of slow servers.

    2. Sets with larger n have larger decoding overheads, and we expect performance to decrease when n gets too big. This is somewhat masked for smaller sets because there are many sets in a file and the decoding overhead of one set can overlap the data movement of future sets. However, as n increases, the time it takes to decode a set will eventually overtake the time it takes to download a set. Furthermore, sets near the end of the file often end up being decoded after all of the downloading has taken place — with larger n, this amounts to larger leftover computation that cannot overlap any data movement.

In the rest of this section, empirical results are presented and compared to the anticipated trends.

5.4 Results

Figures 2, 3, and 4 show the best performance of the three distributions when Reed-Solomon coding is used. Each block displays the best performing instances (averaged over ten runs) in the entire range of

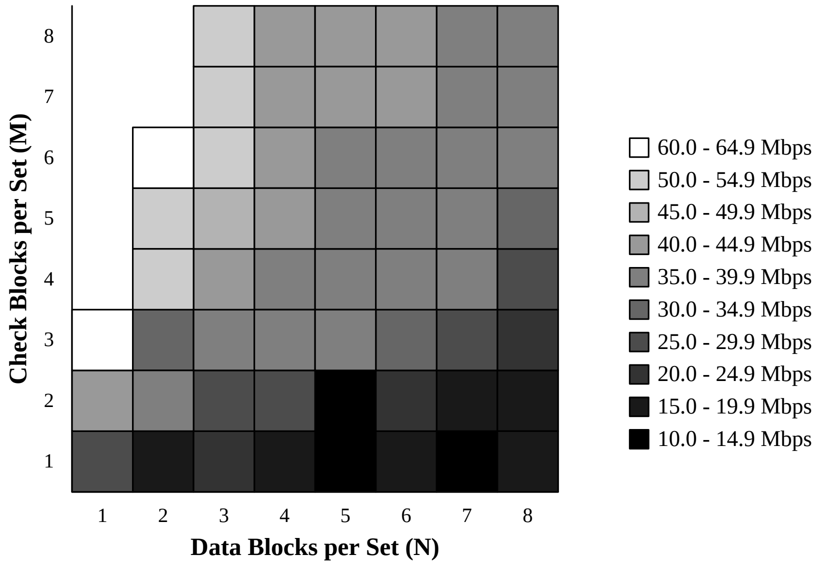

Figure 2. Best overall performance given n and m over the Slow Regional Distribution

parameters when n and m are the given values. The trends proposed in section 5.3 emerge in several ways, but the different distributions of the blocks appear to strengthen and weaken different trends.

The slow regional distribution shown in Figure 2 produces a very regular pattern with diagonal bands of performance across the grid. The two trends that dominate are that performance improves as m increases and n remains fixed, and that performance degrades as n increases and m remains fixed. Furthermore, both trends seem to have equal weight. Performance improves whenever the rate of encoding decreases, but does not show significant changes when the set size increases and the encoding rate stays the same - more specifically, increased set size does not significantly harm performance in smaller sets, nor does it improve performance due to a more advantageous block distribution.

The only difference between the regional distribution and the slow regional distribution is that all of the blocks in the regional distribution are nearby, and none of the blocks in the slow regional distribution are nearby. As such, the performance per block of the regional network file is quite good, and the only trend that emerges in Figure 3 is that performance degrades as n, and thus the decoding time per set, increases. (Note that Figures 2, 3 and 4 have different scales for the shading blocks, so that relative trends rather than absolute trends may emerge).

The hodgepodge distribution exhibits more interesting behavior in Figure 4 than either of other distributions. In general, performance improves as m increases and n remains fixed; the columns where $n = 5$

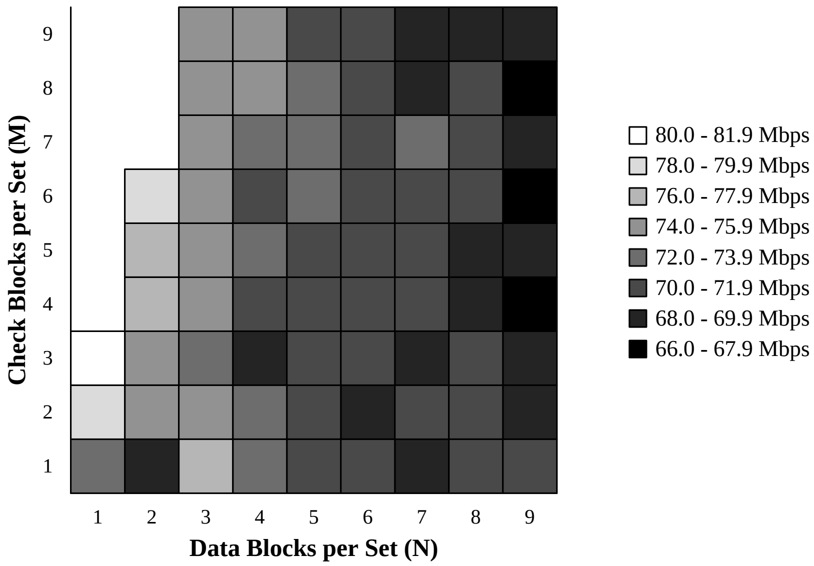

Figure 3. Best overall performance given n and m over the Regional Distribution

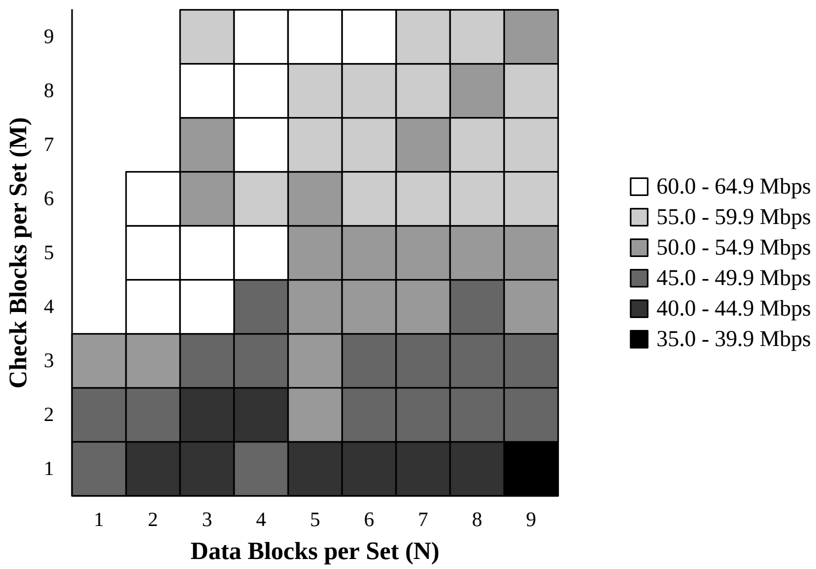

Figure 4. Best overall performance given n and m over the Hodgepodge Distribution

Table 2. Summary of Reed-Solomon and LDPC Coding

	Reed-Solomon	**LDPC**
Primary Operation	Galois-Field Dot Products	XOR
Quality of Encoding	Optimal	Suboptimal
Decoding Complexity	O(n^3)	O(n ln($1/\epsilon$)), where $\epsilon \in \mathbb{R}$ [11]
Overhead Factor	1	Roughly 1.15

and 6 are good examples of this behavior. Performance also diminishes as n increases; observe for example, the rows where $m = 5$ and 8. These are the same trends that turned up in the slow regional distribution; however, the upper right quadrant of the grid has better performance relative to the rest of the grid than the upper right quadrant of the slow regional grid. There are two possible reasons for this: first, the performance is improving due to distribution advantages that arise in larger sets, and second, the trend that performance improves as m increases is stronger than the trend that performance declines as n increases. (these observations are entirely focused with "small" sets; as n increases beyond a point, decoding will take much longer than downloading, regardless of set size or distribution) The hodgepodge distribution spreads the blocks across 50 servers that are not related to each other in any way, while the regional and slow regional distributions spread the blocks across only 4 regions that typically contain less that 50 servers. Thus, it is likely that the loads of servers in the same region interfere with each other, and the hodgepodge distribution may have performance advantages because of this that are not visible in the regional and slow regional network files.

6 Reed-Solomon vs. LDPC

Table 2 summarizes some of the key differences between Reed-Solomon and LDPC coding. When comparing the two types of coding, the properties of key importance are the encoding and decoding times, and the average number of blocks that are necessary to reconstruct a set. LDPC codes have a great advantage in terms of encoding and decoding time over Reed-Solomon codes; in addition, Reed-Solomon decoding requires n blocks from a set before decoding can begin, while LDPC decoding can take place on-the-fly. However, for small n, the extra blocks that LDPC codes can require for decoding can cause substantial performance degradation. Moreover, for systems where the network connection is slow, Reed-Solomon codes can sometimes outperform LDPC codes despite the increased decoding penalty [16].

Table 3 shows the parameter space explored in the next set of experiments, which compare Reed-Solomon coding to LDPC coding. Since LDPC coding may fare poorly in very small sets, sets up to

Table 3. LDPC vs. Reed-Solomon Experiment Space

Parameter	Range of Parameters
Simultaneous Downloads	$T \in [20, 30]$
Work Replication and Failover Strategy	$\{R, P\} \in [\{1, 1\}, \{2, (30/n)\}]$, static timeouts
Server Selection	**Fastest**$_0$, **Fastest**$_1$
Block Selection	**Db-first**, **Dont-care**
Coding	$\{n, m\} \in [\{5, 5\}, \{10, 10\}, \{20, 20\}, \{50, 50\}, \{100, 100\}]$

size 200 were tested. The experiments test the same values for T, R, P, and server selection algorithm that were used in the Reed-Solomon experiments. In addition, a new block selection criteria based on the type of block is introduced, and will be detailed in section 6.1. The actual codes used are listed in the Appendix. They were derived as a part of the exploration in [16], and therefore are not provably optimal.

6.1 Subtleties of LDPC Implementation

The implementation of LDPC coding involves several subtleties that are not present in that of Reed-Solomon coding. First, the fact that LDPC coding sometimes requires more than n blocks affects not only *how many* blocks must be retrieved, but also limits *which* blocks the client application can choose. A wide range of block scheduling strategies may be applied to an LDPC coding set. At one extreme, blocks are downloaded randomly, until enough blocks have been retrieved to decode the set. It is likely that some of the coding blocks will become useless by the time they are retrieved, and that some data blocks may be decoded before they are retrieved. Both of these possibilities increase the number of blocks that are needlessly downloaded. At the other extreme, a download may be simulated in order to determine an "optimal" set of blocks that can be used to decode the set. Note that it is possible every time to choose a set of exactly n blocks that can be used for decoding. The difficulty here is that it must be determined up front which blocks are coming from the fastest servers. In our previous experiments we tested a number of different server selection algorithms that judged servers based on speed and on load. The speed of servers remains fixed in most of the server scheduling algorithms, but the load is always dynamic, and any optimal schedule would have to approximate the load of servers not only throughout the download of a given set, but between the downloads of different sets in the file, since they often overlap. Moreover, such a scheduling algorithm is somewhat complicated to implement and may not offer significant performance enhancements once its own computation time is factored into performance. The following experiments use a compromise between the two extreme scheduling options: when selecting a block to download, the downloading algorithm will skip

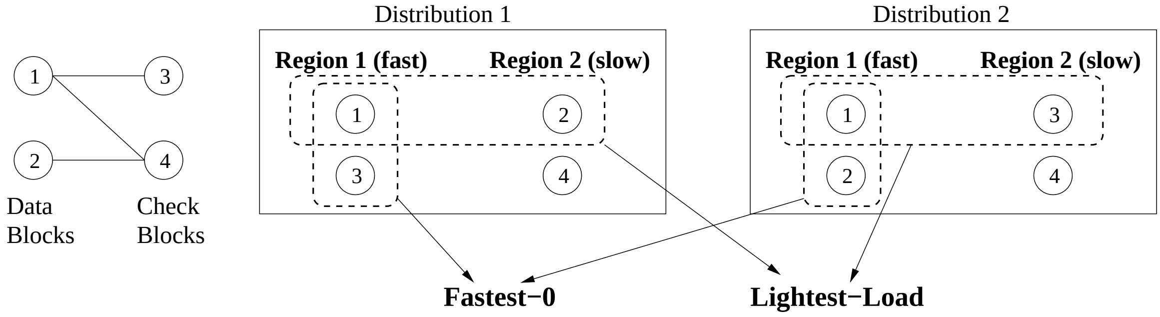

Figure 5. Distribution Woes. The performance of different server scheduling algorithms can vary greatly depending on the distribution of a file with LDPC coding. In distribution 1, the **Lightest-load** algorithm performs best, while in distribution 2, the **Fastest**$_0$ algorithm performs best.

over check blocks that can no longer contribute to decoding and data blocks that are already decoded. The algorithm also allows one of two block preferences to be specified:

- Data blocks first (**db-first**): data blocks are always preferred over check blocks.
- Don't care (**dont-care**): the type of blocks is ignored, and blocks are chosen solely based on the speed and load of the servers on which they reside.

In the previous experiments over only Reed-Solomon codes, only the **dont-care** algorithm was used.

The second major subtlety in the implementation of LDPC coding is that the distribution of the file can have a great impact on the performance of different server scheduling algorithms. Consider the example depicted in Figure 5, where $n = 2$, and $m = 2$, and there are two server regions. Figure 5 shows two possible distributions of the blocks, and which blocks would be chosen from each distribution by the **Fastest**$_0$ and **Lightest-load** server scheduling algorithms [6], where the **Lightest-load** algorithm always chooses the server with the lightest load. Depending on the distribution, one of the algorithms results in two blocks that can be used to reconstruct the entire set, and the other does not. The unfortunate consequence of this characteristic is that the performance can vary greatly between different server scheduling algorithms not because of the algorithms themselves, but because of the file's distribution. The following experiments use the **Fastest**$_0$, and **Fastest**$_1$ server scheduling algorithms, and the same distributions described in section 5, which do not address the distribution subtleties of LDPC coding – it is a subject of further work to address the impact of these subtleties on download performance.

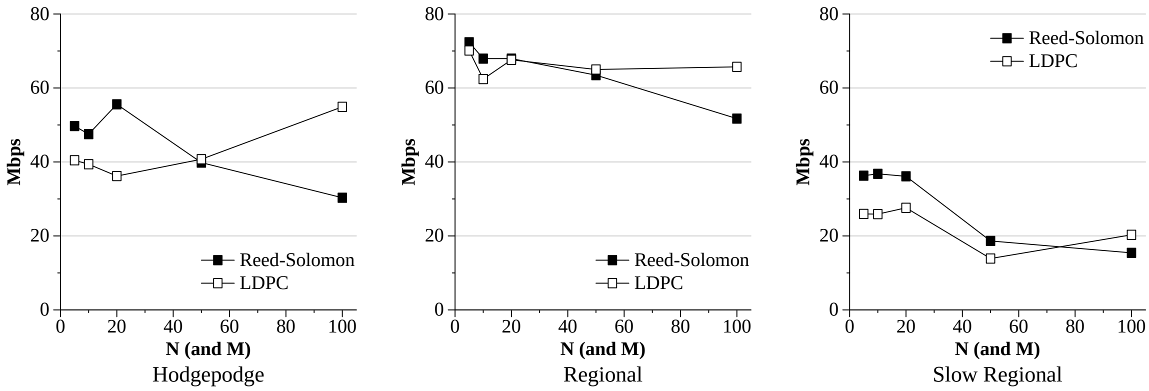

Figure 6. Reed-Solomon vs. LDPC coding (performance including both download and decoding times)

6.2 The Trade-off between Download and Decode

The best performing instances of Reed-Solomon and LDPC coding are shown in Figure 6. LDPC coding performs no better than, and in some cases much worse than Reed-Solomon coding when n is less than 50; however, when n is greater than 50, LDPC coding vastly outperforms Reed-Solomon coding. As n increases past 20, the performance of Reed-Solomon coding steadily declines, while the performance of LDPC coding tends to improve, as in the hodgepodge distribution, or level off, as in the regional distribution. The fact that the hodgepodge distribution is able to improve as the size of the set increases while the regional distribution is not reinforces the observation from section 5.4 that the lack of topographical relationship between the blocks in the hodgepodge distribution allows this distribution to take advantage of the fast servers without being penalized for regional interference between servers. Concerning the best overall performance, the best data point over all set sizes in the hodgepodge distribution occurs at $n = m = 100$ for LDPC coding, while in both the regional and the slow regional distributions, Reed-Solomon achieves the best data point over all at $n = m = 5$ and $n = m = 10$, respectively.

When judging the merits of either type of coding scheme in this particular application, it is important to remember that any size set can scale to arbitrarily large files without incurring additional overhead per set. In general, given an application and a choice between Reed-Solomon or LDPC coding it is probably best to choose the scheme and set n,m that:

- Is able to scale in the future along with the application.

- Does not exceed physical storage limitations.

Table 4. Block Preferences

Distribution n, m	LDPC preference	RS preference
Slow Regional 5,5	**dont-care**	**dont-care**
Slow Regional 10,10	**dont-care**	**dont-care**
Slow Regional 20,20	**dont-care**	**dont-care**
Slow Regional 50,50	**db-first**	**dont-care**
Slow Regional 100,100	**dont-care**	**dont-care**
Hodgepodge 5,5	**dont-care**	**dont-care**
Hodgepodge 10,10	**dont-care**	**dont-care**
Hodgepodge 20,20	**dont-care**	**dont-care**
Hodgepodge 50,50	**db-first**	**db-first**
Hodgepodge 100,100	**dont-care**	**db-first**
Regional 5,5	**db-first**	**db-first**
Regional 10,10	**db-first**	**db-first**
Regional 20,20	**db-first**	**dont-care**
Regional 50,50	**db-first**	**db-first**
Regional 100,100	**db-first**	**db-first**

- Meets desired levels of fault tolerance; note that Reed-Solomon coding has stronger guarantees than LDPC coding in this respect.
- Satisfies each of the three previous criteria with the best performance where performance consists of both download time and decoding time.

6.3 Block Preferences

In these experiments, blocks can be downloaded selectively based on their type. Data blocks are generally preferable to check blocks since fewer data blocks require less decoding, but when check blocks are much closer than data blocks, sometimes the advantage of getting close blocks is worth the decoding penalty. Table 4 shows which block preference algorithm is used in the best performing instances of the codes over the three distributions. A trade-off between download time and decoding time is apparent: the **dont-care** algorithm is favored in the Slow Regional distribution and in the Hodgepodge distribution, since the penalty of retrieving slow blocks is greater than the computational cost of decoding. In the Regional distribution, the **db-first** algorithm is favored — since the majority of blocks are closer to the server, the penalty for selecting blocks that require no decoding is small. In addition, when $n \geq 50$ (i.e., the decoding time is greater), the **db-first** algorithm is slightly favored. To illustrate this trade-off further, Figures 7 and 8 show the total time

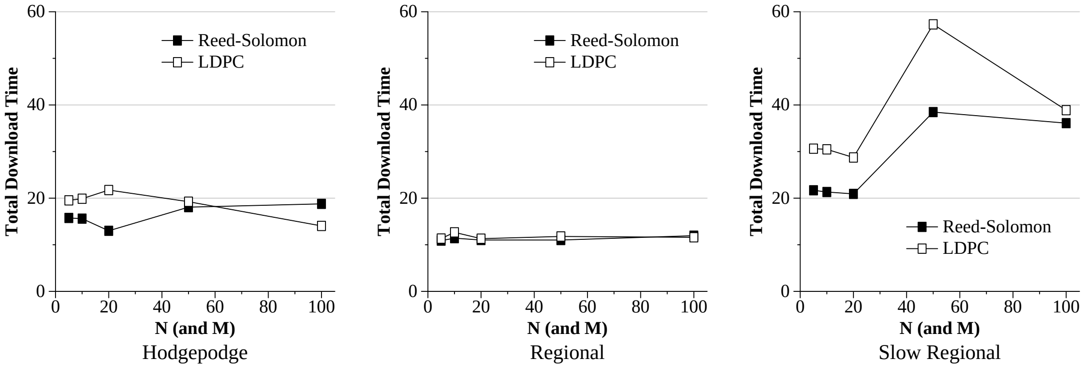

Figure 7. Total Download Time (beginning of first IBP load to ending of last) in Best Performances

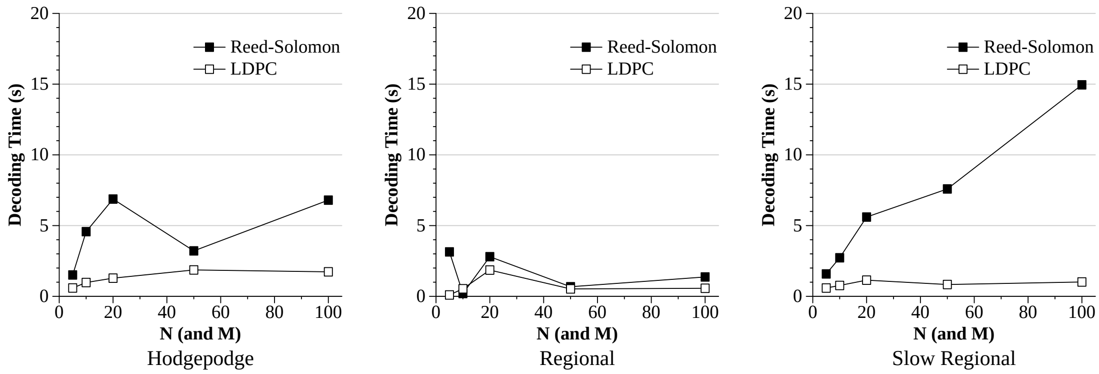

Figure 8. Total Decoding Time in Best Performances

spent downloading and decoding in the best performing instances of each distribution. Note that although one might think that the computational overhead would be similar for equal values of n and m, in the Regional distribution, the computational overhead is lower than the rest because the fastest instances prefer data blocks to coding blocks. In general, these Figures clearly show that Reed-Solomon codes require more time to decode than LDPC codes. However, since the decoding of sets can overlap with the downloading of subsequent sets, the cost of decoding will negatively impact performance only when the decoding time surpasses the downloading time, and when the last sets of the file retrieved are being decoded.

7 Conclusions

When downloading algorithms are applied to a wide-area file system based on erasure codes, additional considerations must be taken into account. First, performance depends greatly on the interactions of encoding rate, set size, and file distribution. The predominant trend in our experiments is that performance improves as the rate of encoding decreases, and that performance ultimately diminishes as n increases, but to a somewhat lesser extent, it appears that larger sets do have an advantage in terms of download time depending on the distribution of the file. Performance amounts to a balance between the time it takes to download the necessary number blocks and the time it takes to decode the blocks that have been retrieved, and thus distribution, which is intimately related to download time, strengthens and weakens the trends that arise from encoding rate and set size.

Though less thoroughly explored in this work, the type of erasure codes being used also has interactions with encoding rate, set size, and distribution, that shift performance. LDPC codes outperform Reed-Solomon codes in large sets because LDPC decoding is very inexpensive, but LDPC codes also require more blocks than Reed-Solomon codes and mildly limit which blocks can be used.

In the end, given a specific application or wide-area file system based on erasure codes, decisions about set size, encoding rate, and coding scheme should be based on the following issues: first, what kind of distribution is being used, and can it be changed; second, what level of fault tolerance is necessary, and what set size and coding scheme can achieve this level given the available physical storage; last, given the set size and coding scheme pairs that meet storage constraints and fault tolerance requirements, which has the best performance and the best ability to scale in ways that the file system is likely to scale in the future.

8 Acknowledgments

This material is based upon work supported by the National Science Foundation under grants CNS-0437508, ACI-0204007, ANI-0222945, and EIA-9972889. The authors thank Micah Beck and Scott Atchley for helpful discussions, and Larry Peterson for PlanetLab access.

References

[1] M. S. Allen and R. Wolski. The Livny and Plank-Beck Problems: Studies in data movement on the computational grid. In *SC2003*, Phoenix, November 2003.

[2] A. Bassi, M. Beck, T. Moore, and J. S. Plank. The logistical backbone: Scalable infrastructure for global data grids. In *Asian Computing Science Conference 2002*, Hanoi, Vietnam, December 2002.

[3] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable network storage. In *ACM SIGCOMM '02*, Pittsburgh, August 2002.

[4] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *ACM SIGCOMM '98*, pages 56–67, Vancouver, August 1998.

[5] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, Berkely, CA, June 2003.

[6] R. L. Collins and J. S. Plank. Downloading replicated, wide-area files – a framework and empirical evaluation. In *3rd IEEE International Symposium on Network Computing and Applications (NCA-2004)*, Cambridge, MA, August 2004.

[7] Paul J.M. Havinga. Energy efficiency of error correction on wireless systems, 1999.

[8] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[9] Witold Litwin and Thomas Schwarz. LH* RS : A high-availability scalable distributed data structure using reed solomon codes. In *SIGMOD Conference*, pages 237–248, 2000.

[10] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann. Practical loss-resilient codes. In *29th Annual ACM Symposium on Theory of Computing,*, pages 150–159, El Paso, TX, 1997. ACM.

[11] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman. Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47(2):569–584, February 2001.

[12] J. S. Plank. Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques. In *15th Symposium on Reliable Distributed Systems*, pages 76–85, October 1996.

[13] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[14] J. S. Plank, A. Bassi, M. Beck, T. Moore, D. M. Swany, and R. Wolski. Managing data storage in the network. *IEEE Internet Computing*, 5(5):50–58, September/October 2001.

[15] J. S. Plank and Y. Ding. Note: Correction to the 1997 tutorial on reed-solomon coding. Technical Report CS-03-504, University of Tennessee, April 2003.

[16] 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: The International Conference on Dependable Systems and Networks*. IEEE, June 2004.

[17] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49, 2001.

[18] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *First International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[19] S. B. Wicker and S. Kim. *Fundamentals of Codes, Graphs, and Iterative Decoding*. Kluwer Academic Publishers, Norwell, MA, 2003.

[20] R. Wolski, N Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5):757–768, October 1999.

[21] Z. Zhang and Q. Lian. Reperasure: Replication protocol using erasure-code in peer-to-peer storage network. In *21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, pages 330–339, October 2002.

9 Appendix

Below we list the five LDPC codes used for the experiments. They are specified as the edge lists of each left-hand node.

- $n = 5$, $m = 5$: $\{$(0, 2, 3), (2, 3, 4), (0, 1, 4), (1, 3, 4), (0, 1, 2)$\}$. Overhead = 5.464. This is the graph pictured in Figure 1.

- $n = 10$, $m = 10$: $\{$(0, 1, 3, 8), (0, 2, 6), (1, 2, 4, 9), (2, 3, 4, 6), (3, 5, 7, 8), (1, 2, 7, 8), (6, 8, 9), (1, 5, 6), (0, 5, 7, 9), (0, 1, 4)$\}$. Overhead = 11.416.

- $n = 20$, $m = 20$: $\{$(3, 4, 5, 6, 8, 10, 14, 15), (0, 2, 6, 11, 17), (3, 6, 7, 9, 10, 15, 16, 17), (8, 10, 15, 17, 19), (2, 6, 9, 10, 14), (1, 2, 4, 6, 9, 10, 12, 15), (1, 8, 9, 12, 15), (4, 9, 10, 15, 19), (3, 6, 9, 10, 11, 17, 19), (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19), (0, 1, 5, 6, 8, 10, 12, 15, 16, 18), (0, 1, 10, 14, 15), (2, 6, 10, 12, 13, 15), (2, 6, 15, 16, 18), (1, 6, 10, 15, 16), (1, 6, 10, 13, 14, 15), (3, 5, 6, 10, 15, 17), (2, 7, 10, 15), (6, 10, 11, 15), (6, 7, 10, 13, 15, 17, 18)$\}$. Overhead is 23.884.

- $n = 50, m = 50$: {(1, 4, 9, 12, 14, 15, 16, 19, 23, 25, 29, 32, 46), (1, 3, 4, 5, 6, 7, 9, 12, 13, 14, 15, 16, 17, 18, 19, 23, 25, 29, 31, 35, 41, 45, 46, 47, 49), (1, 3, 4, 6, 7, 9, 11, 12, 14, 15, 16, 17, 18, 19, 23, 25, 35, 36, 40, 41, 45, 46, 47), (1, 4, 6, 9, 12, 13, 14, 15, 16, 17, 18, 19, 22, 25, 27, 28, 29, 33, 35, 36, 39, 44, 45, 46, 49), (1, 3, 4, 6, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19, 25, 26, 28, 29, 33, 35, 41, 43, 45, 46, 47), (1, 3, 4, 6, 8, 9, 11, 12, 14, 15, 16, 17, 18, 19, 23, 25, 28, 33, 34, 35, 44, 45, 46, 47, 49), (0, 1, 3, 4, 5, 6, 7, 8, 9, 11, 12, 14, 15, 16, 17, 18, 19, 21, 23, 25, 26, 35, 36, 41, 45, 46, 47, 49), (1, 2, 3, 4, 6, 7, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 22, 25, 28, 29, 30, 33, 34, 35, 46, 47, 49), (1, 4, 6, 9, 11, 12, 13, 14, 15, 16, 18, 19, 23, 25, 27, 32, 34, 40, 45, 46, 48, 49), (1, 3, 4, 9, 14, 16, 18, 25, 27, 46, 49), (1, 3, 4, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19, 23, 24, 25, 27, 29, 34, 35, 36, 38, 40, 41, 45, 46, 47, 49), (4, 6, 14, 46), (1, 4, 5, 6, 7, 9, 12, 14, 15, 16, 18, 19, 20, 21, 22, 25, 27, 28, 33, 34, 35, 41, 42, 44, 45, 46, 47, 49), (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 44, 45, 46, 47, 49), (1, 4, 9, 11, 14, 16, 18, 19, 22, 23, 25, 27, 46, 49), (4, 12, 18, 25), (1, 7, 14, 16, 23, 46, 49), (0, 1, 3, 4, 5, 6, 9, 14, 16, 17, 18, 19, 22, 25, 27, 28, 31, 35, 46), (1, 3, 4, 5, 6, 7, 9, 12, 14, 15, 16, 17, 18, 19, 22, 25, 27, 33, 35, 37, 41, 46, 49), (1, 3, 4, 6, 7, 9, 12, 14, 15, 16, 17, 18, 19, 22, 23, 25, 28, 29, 30, 33, 35, 46, 47, 48, 49), (4, 7, 9, 16, 35, 46), (1, 3, 4, 6, 9, 11, 12, 13, 14, 15, 16, 18, 19, 25, 29, 35, 40, 43, 45, 46, 49), (0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19, 22, 23, 25, 26, 27, 28, 29, 30, 32, 33, 34, 35, 36, 37, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49), (1, 3, 4, 6, 7, 9, 12, 14, 15, 16, 17, 18, 19, 23, 25, 26, 27, 28, 34, 35, 36, 39, 41, 46, 49), (1, 3, 4, 6, 7, 9, 11, 12, 13, 14, 15, 16, 18, 19, 22, 23, 27, 28, 29, 31, 34, 35, 36, 37, 38, 40, 41, 43, 44, 45, 46, 47, 49), (1, 4, 5, 6, 15, 16, 18, 19, 35, 46, 49), (1, 3, 4, 6, 7, 11, 12, 14, 15, 16, 17, 18, 19, 22, 23, 27, 28, 29, 34, 35, 46, 47, 48), (1, 3, 4, 9, 14, 15, 16, 17, 18, 19, 22, 23, 25, 29, 35, 41, 44, 45, 46, 47, 48, 49), (1, 2, 4, 6, 7, 9, 12, 13, 14, 15, 16, 18, 19, 22, 23, 24, 25, 27, 28, 34, 35, 36, 46, 47, 49), (1, 3, 4, 9, 12, 14, 15, 16, 18, 19, 23, 25, 41, 46), (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 22, 23, 25, 26, 27, 28, 29, 30, 31, 33, 34, 35, 36, 39, 40, 41, 42, 46, 47, 49), (1, 9, 16, 18, 25, 31, 46, 47), (1, 3, 4, 6, 7, 9, 12, 14, 15, 16, 17, 18, 19, 22, 23, 25, 27, 29, 35, 41, 45, 46, 49), (1, 3, 7, 16, 19, 25, 41, 45), (1, 3, 4, 14, 16, 19, 36, 46), (1, 3, 4, 9, 12, 14, 15, 16, 18, 19, 22, 23, 24, 25, 26, 28, 31, 34, 35, 46, 47, 49), (1, 4, 6, 18), (1, 4, 16, 18, 35, 36, 44, 46, 49), (1, 4, 6, 12, 14, 16, 19, 22, 46, 47), (1, 4, 6, 14, 18, 19, 22, 35, 46, 49), (1, 3, 4, 7, 8, 9, 11, 12, 14, 15, 16, 17, 18, 19, 23, 25, 27, 29, 34, 35, 39, 41, 42, 45, 46, 47, 48, 49), (1, 4, 6, 14, 15, 16, 18, 19, 28, 35, 46, 47), (1, 4, 11, 16, 18, 46), (1, 3, 4, 6, 7, 9, 11, 12, 13, 14, 15, 16, 18, 19, 20, 23, 25, 27, 28, 33, 35, 36, 40, 41, 45, 46, 47, 48, 49), (4, 9, 16, 18, 19, 47), (3, 4, 9, 14, 16, 18, 19, 34), (1, 4, 18, 19, 25, 35, 45, 46), (1, 4, 6, 9, 14, 15, 16, 18, 19, 22, 29, 35, 41), (1, 3, 4, 9, 11, 12, 13, 14, 15, 16, 18, 19, 21, 22, 23, 25, 27, 28, 29, 34, 35, 41, 45, 46, 49), (6, 16, 22, 35)$\}$. Overhead is 63.96944.

- $n = 100$, $m = 100$: $\{$(6, 7, 18, 22, 31, 78), (8, 22, 45, 54, 57, 63, 73, 74), (1, 25, 28, 32, 33, 68, 72), (4, 11, 12, 13, 18, 30, 46), (4, 16, 25, 27, 34, 78), (2, 31, 39, 52, 66, 78, 98), (25, 31, 34, 42, 72, 87), (3, 10, 16, 25, 32, 41, 97), (0, 4, 7, 18, 53, 69, 78), (2, 7, 52, 55, 74, 75, 82, 88), (10, 43, 59, 64, 67, 87, 99), (23, 26, 48, 55, 62, 68, 88), (0, 7, 10, 25, 36, 51, 79), (15, 24, 48, 75, 84, 97), (4, 16, 18, 38, 61, 70, 77, 96), (39, 53, 65, 76, 78, 91, 94), (7, 18, 25, 55, 62), (4, 14, 16, 18, 25, 48, 54, 80), (34, 47, 54, 55, 61, 75), (10, 20, 22, 48, 60, 62, 78), (14, 29, 32, 34, 48, 51, 66), (14, 15, 34, 50, 51, 91), (1, 7, 39, 61, 83, 95), (7, 18, 27, 34, 63, 69, 95), (7, 11, 14, 27, 28, 55, 73), (7, 25, 48, 58, 86, 90, 95), (0, 10, 34, 35, 64, 89, 95), (5, 10, 20, 31, 68, 71, 84, 90), (28, 40, 48, 54, 55, 80, 84, 96), (3, 7, 19, 37, 40, 59, 94), (7, 15, 17, 20, 27, 83), (9, 21, 26, 31, 54, 78, 85, 97), (24, 25, 29, 48, 69, 81, 85), (21, 41, 52, 55, 62, 65), (16, 34, 65, 85, 86, 89, 99), (3, 9, 18, 24, 25, 83, 91), (3, 7, 10, 18, 48, 68, 94, 98), (31, 37, 49, 58, 72, 78), (6, 7, 10, 42, 47, 80, 92), (13, 18, 36, 50, 51, 67, 72), (14, 55, 57, 72, 75, 77, 85, 89), (31, 39, 45, 70, 72, 79, 94), (4, 6, 14, 25, 41, 79, 84), (13, 20, 34, 40, 46, 51, 60), (3, 19, 31, 34, 42, 65, 71, 73), (12, 19, 27, 44, 58, 65, 70, 72), (9, 17, 30, 44, 58, 64, 70), (1, 40, 55, 81, 83, 89, 92), (18, 23, 29, 38, 63, 65, 77), (21, 38, 66, 72, 81, 82, 92), (8, 25, 34, 81, 94, 98), (17, 34, 43, 52, 55, 76, 77), (9, 25, 32, 41, 48, 55, 56, 70), (0, 6, 13, 19, 46, 63, 92), (25, 48, 49, 50, 85, 88, 92), (24, 26, 49, 52, 55, 84, 86, 99), (4, 25, 34, 38, 48, 57, 97), (5, 10, 13, 17, 31, 34, 58), (12, 25, 31, 45, 57, 65, 82, 92), (10, 29, 44, 77, 82, 87, 96), (8, 11, 31, 34, 60, 88, 96), (10, 12, 21, 35, 37, 55, 78, 82), (3, 21, 46, 47, 48, 74, 78, 96), (36, 38, 46, 48, 59, 83), (1, 4, 31, 40, 46, 78, 81), (1, 18, 34, 39, 57, 71, 98), (4, 31, 33, 41, 53, 67, 72), (16, 17, 18, 23, 40, 50, 85), (5, 7, 35, 54, 65, 72, 75), (5, 8, 26, 30, 31, 55, 94), (4, 44, 48, 56, 60, 61, 78), (27, 31, 65, 71, 76, 78, 88, 97), (2, 23, 26, 32, 37, 49, 78), (25, 42, 72, 77, 79, 83), (4, 15, 25, 51, 76, 93, 99), (4, 18, 56, 64, 65, 72), (4, 36, 41, 48, 59, 76, 80), (15, 18, 24, 28, 33, 65, 72), (7, 28, 34, 47, 71, 74, 99), (0, 9, 10, 25, 43, 44, 60), (4, 6, 39, 72, 73, 75, 89), (7, 12, 26, 31, 33, 34, 48, 66), (18, 25, 31, 34, 65, 72, 90), (6, 10, 13, 36, 43, 49, 61), (1, 28, 33, 35, 41, 48, 78, 84), (7, 19, 66, 79, 93, 98), (2, 23, 25, 50, 58, 60, 63), (10, 42, 45, 65, 78, 87, 91), (2, 8, 10, 18, 22, 53, 72), (53, 55, 62, 65, 80), (7, 35, 47, 71, 90, 92), (0, 10, 31, 56, 65, 72), (4, 48, 57, 65, 88), (20, 24, 34, 66, 72, 78, 82), (30, 31, 55, 65, 90, 93), (4, 32, 45, 55, 68, 72, 76, 86), (18, 34, 35, 43, 64, 72, 99), (7, 18, 20, 37, 72, 78, 93), (7, 11, 22, 65, 69, 95), (4, 10, 18, 31, 67, 73)$\}$. Overhead is 114.81.