INCREASING THE FUNCTIONALITY AND AVAILABILITY OF REED-SOLOMON FEC CODES: A PERFORMANCE STUDY

D. Rubenstein

CMPSCI Technical Report 98-31

August 1998

Increasing the Functionality and Availability of Reed-Solomon FEC Codes: a Performance Study*

Dan Rubenstein

Technical Report 98-31
Department of Computer Science
August, 1998

Abstract

This document describes the implementation details of modifications performed to an existing C-language Vandermonde-Matrix based Forward Error Correcting (FEC) coding package [2] and to an existing C-language Cauchy-Matrix based FEC coding package [3] to increase their interface flexibility, and allow for their use in a wider variety of coding applications. We discuss why these extensions add no time complexity to encoding and decoding rates. We also describe the architecture of a coding package that we developed in Java, which is based on the C-language Vandermonde-Matrix based coding package. We compare the coding rates of our modified C-language Vandermonde-Matrix coder to those of our modified C-language Cauchy-based coder, and find that the Vandermonde based coder codes at a faster rate for a significant majority of platforms and coding scenarios considered. We then compare the coding speeds of the Vandermonde-based coder between the C and Java version, and find that the Java version running on a virtual machine performs at least two orders of magnitude slower than its C language counterpart running as compiled machine code. Finally, we examine performance improvements when the Java coder utilizes just-in-time compilation (JIT).

See http://www-net.gaia.cs.umass.edu/ drubenst/software.html for the availability of the software packages discussed in this report.

1 Introduction

Forward Error Correction (FEC) is a technique whereby additional information is sent with data to recover from transmission errors in a manner which is often more concise than by straightforward retransmission. Forward error correcting techniques can support a wide variety of loss scenarios. Initially, the techniques were applied to repairing bit errors within data packets. More recently, the techniques were applied at the packet level to recover from packet losses. This is commonly referred to as *packet level FEC*, and most often employs *Reed-Solomon codes*. Understanding and implementing these codes requires a significant understanding of coding theory, which can be found in [1]. However, it is easy to provide a simple interface to these coding techniques such that the sophisticated mathematical operations are performed within a black-box, and need not be understood by the protocol designer. We now give an example of a typical, simple, interface for packet-level coding.

Information is coded at the sender, and is sent in separate packets to receivers. Receivers use these special repair packets in conjunction with received data packets to decode the data in packets that were lost. We assume that the receiving entity that must perform the decoding either receives a packet uncorrupted, or fails to receive (loses) a packet, and is able to detect lost packets. Such a scenario is common in the transport layer of the Internet, where lower layers guarantee that received packets are not corrupted. When the network becomes overloaded, it reduces its load by dropping packets, making the packet the atomic unit

*This material was supported in part by the National Science Foundation under Grant No. NCR-9508274, and NCR-9527163, and by DARPA under Grant No. N66001-97-C-8513. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

of loss within the network. Packets which are to be transported reliably are typically given sequence numbers at the sender (origin of transmission), so that the receiver(s) can detect packet losses by noting gaps in the sequence numbers of arriving packets.

When using a coding technique such as packet level FEC, the sender groups data packets into blocks of size k (henceforth k is referred to as the *block size*), and feeds the k data packets into an encoder to produce repair packets. The number of repair packets that can be produced per block is finite. However, this number is sufficiently large that, for most purposes, it is safe to assume that the sender can produce an unlimited supply. Repair packets generated from a block of k data packets are said to belong to the block from which they were generated. Any entity, such as a receiver, with a decoder can use the decoder to retrieve the k data packets for a block once it receives any combination of k data and repair packets belonging to that block.

Several software packages have been made publically available which provide packet-level FEC encoding and decoding functionality with interfaces as described above. In particular, two C language packages which we examine in this report are a Reed-Solomon based coder that uses Vandermonde-based matrices [2], and a separate Reed-Solomon coder that uses Cauchy-based matrices [3]. Recently, bandwidth-efficient protocols have been proposed which require coding support at points within the network (i.e., not just the end-hosts). In [4], we introduced novel protocols of this type that used coding techniques to maintain the low bandwidth requirement while also reducing buffer and coding processing requirements at these non-end-host points. However, these protocols required an interface that differs from what was provided by the coding packages previously mentioned.

In this report, we give an overview of the alternative interface, and how the encoders were modified to support this interface. To support implementation of our protocols in an active networking framework, we ported the modified, Vandermonde-based coder from C to Java. Our decision to port the Vandermonde-based coder was based on a comparison of coding and decoding times of our modified C versions of the two coders, which we also present in this report. We find that almost always, the Vandermonde coder codes at a faster rate than the Cauchy-based coder, regardless of the block size, packet length, or hardware platform.

We describe the Java architecture, and compare the speed of the Java coder to that of the C coder on several platforms. We find that the Java encoder runs more than several hundreds of times slower than its C counterpart. For encoding, this slowdown is exacerbated as the block size (number of data packets used within an encoding) is decreased. We do not see any such correlation in block size for the speedup of C decoding over Java decoding. Instead, we find that the speedup by using the C version of the package mimics the speedup of the encoding package for larger block sizes. A Cauchy-based coder can be developed in Java using an architecture similar to what we present here. We expect the performance of the coder's performance in Java to suffer in performance in a manner similar to what occurs for the Vandermonde-based coder. We also find that the Java coder that performs just-in-time compilation runs between 3 and 4 times faster than the Java coder that runs over the virtual machine.

The rest of this report proceeds as follows. Section 2 gives a basic mathematical understanding of how Reed-Solomon coding is performed, and gives intuition as to why our modifications do not increase the time complexity of the coders. Section 3 discusses the modifications we made to the C versions of the FEC coding packages in order to increase their interface flexibility. A performance comparison between our modified Vandermonde-based coder and the Cauchy-based coder is given in Section 4. We describe the Java object-oriented architecture we used for our coding package in Section 5, and compare its performance with its C counterpart in Section 6. We conclude in Section 7.

2 Encoding via the Generator Matrix: A simple explanation

We now present an explanation of the mathematics used to perform encoding and decoding operations. A packet is simply a collection of bits. These bits can be thought of as a number represented in base 2. Thus, if we are given k packets, we can construct variables $x_1, \cdots, x_k$, where the value of variable x_i equals the numerical value of packet i. It follows from elementary linear algebra that given any set of k linearly independent equations involving the variables $\{x_i\}$, the values of the variables are unique and can be obtained by solving the equations. We now explain how Reed-Solomon coders utilizes this elementary linear algebra.

An identical $n + k \times k$ matrix is constructed at both the encoder and decoder. This matrix is referred to

as the *generator matrix*, G. Each row in G gives coefficients of an equation in terms of the k packet values. The ith column of the jth row, $G_{j,i}$, in the matrix contains the coefficient for the variable x_j in the ith equation. The top k rows of the matrix form a $k \times k$ identity matrix (i.e., the ith row for $i \leq k$ represents an equation that contains only the value of x_i). The process used to construct G has the following useful property that we do not prove here. The interested reader is instead referred to [1, 2, 3].

Lemma 1 *Given that G is a generator matrix (Cauchy or Vandermonde), then any $k \times k$ matrix, M, constructed from any k rows of G, has rank k, i.e., any k equations contained in M are linearly independent.*

In other words, given the vector X representing the various packet values, where

$$X = \begin{pmatrix} x_1 \\ \vdots \\ x_k \end{pmatrix},$$

then MX produces a vector Y, which gives the solution to a set of k linearly independent equations. It follows that $X = M^{-1}Y$. Thus, a decoder can retrieve the original packets given Y, G, the rows that were used to construct M, and the ability to invert a matrix.

The sender includes the row number used to generate the encoded portion of the packet in the non-encoded header of the packet (note that the ith data packet within a block would include the row number i). After receiving k packets, the decoder constructs M from the rows of G that were used to generate the k received packets (obtained from the non-encoded header portion of each received packet). The matrix is inverted, and multiplied by the vector of received packets (with the headers stripped off), yielding a vector containing the original data packets.

We have omitted two details that are used in practice. First, the multiplication and addition operations used for matrix multiplication and inversion are performed over finite fields. Details on the respective finite fields used can be found in [2] for the Vandermonde-based operations, and in [3] for the Cauchy-based operations. Also, each data packet d_i is separated by the encoder into a fixed number of c components (e.g., one byte or one word per component), $d_i^1, \cdots, d_i^c$. For repair packet r_j, the encoding is performed over the k data packets one component at a time, producing a repair component $r_j^m, 1 \leq m \leq c$. This r_j^m can be used in conjunction with any data packets from or repairs coded from the set of $\{d_i^m\}$ to decode the mth component of a data packet from the block. A repair packet r_j is formed by concatenating the components $r_i^1, \cdots, r_i^c$. Upon receiving a repair packet, the decoder splits the repair packet into individual components, applies the matrix operations to each component to produce data packet components, and then concatenates the resulting components together, reproducing the data packets.

2.1 Supporting multiple block sizes

In [3], it is stated that a nice property of the Cauchy-based coder is that a generator matrix constructed to support a block size of k_{max} can also be used to support all block sizes $k \leq k_{max}$. This also holds true for the Vandermonde-based coder. The $n + k \times k$ generator matrix needed to support coding for a block size of k, G', can be constructed from the original generator matrix, G, for block size k_{max} using the following mapping:

$$G'_{j,i} = \left\{ \begin{array}{ll} G_{j,i} & 0 \leq j < k, 0 \leq i < k \\ G_{j+k_{max}-k,i} & k \leq j < n+k, i < k \end{array} \right\} \quad (1)$$

In other words G is converted to G' by removing rows $k+1$ through k_{max}, and from the remaining rows, removing the last $k_{max} - k$ columns. In practice, a separate matrix does not need to be constructed: the coder must simply perform the appropriate mapping.

It remains to be shown that any set of k rows chosen to form a $k \times k$ matrix, M, produces a matrix with rank k. By showing this, we know that the k rows are linearly independent, and thus any k repair packets can be used to retrieve the k original data packets. This is accomplished in the following theorem:

Theorem 1 *Let the generator matrix, G be a $n + k_{max} \times k_{max}$ matrix, and let G' be a $n + k \times k$ matrix constructed from G as described in equation 1. If any k rows are selected from G', then the $k \times k$ matrix formed has rank k.*

Proof: Assume the statement is false. Then there exists a non-0 column vector $v' = (c_1, \cdots, c_k)^T$, such that $G'v'$ equals the k-column 0-vector (we use 0 to represent the additive identity in whichever field, $\mathcal{F}$, is being used). Let $i_1, \cdots, i_k$ be the indices of the rows that were selected in G to form G'. Construct the $k_{max} \times k_{max}$ matrix G'', where the first k rows are equivalent to rows $i_1, \cdots, i_k$ in G, and the last $k_{max} - k$ rows are equivalent to rows $k + 1, \cdots, k_{max}$ in G. These last k rows are the last $k_{max} - k$ rows of the $k_{max} \times k_{max}$ identity matrix. Recall that in a field, $\mathcal{F}, \forall a \in \mathcal{F}$, the additive identity, 0, has the following properties:

- $a0 = 0a = 0$
- $a + 0 = 0 + a = a$

Consider the k_{max}-column vector $y = (c_1, \cdots, c_k, 0, \cdots, 0)^T$ (i.e., entries y_{k+1} through $y_{k_{max}}$ equal 0), and consider the product $x = (x_1, \cdots, x_{k_{max}})^T = G''y$. We have that $x_i = \sum_{j=1}^{k_{max}} G''_{i,j} y_j$. Since $y_j = 0$ for $j > k$, we have that $x_i = \sum_{j=1}^{k} G''_{i,j} y_j$. For $i \leq k$, x_i is equal to the value produced from the product of $G'_i v'$, which is 0. For $k < i \leq k_{max}$, x_i also equals 0, because G''_i is the $k + i$th row in the $k_{max} \times k_{max}$ identity matrix, such that $G''_{i,j} = 0$ for $1 \leq j \leq k$. It follows that x is the k_{max}-column 0 vector, resulting from a product between G'' and a non-0 k_{max}-column vector, y. Thus, G'' cannot be a matrix with rank k_{max}, contradicting Lemma 1. ∎

Corollary 1 *A Vandermonde-based coder can support all block sizes up to some k_{max} using a single $n + k_{max} \times k_{max}$ generator matrix. The matrix can be used to generate n repair packets. Furthermore, the computational complexity of encoding and decoding is a function of the block size, k, and not of k_{max}.*

Proof: By Lemma 1, the encoder and decoder must each maintain only the $n + k_{max} \times k_{max}$ generator matrix, and use the mapping algorithm for encoding or decoding purposes. The mapping yields a $n + k \times k$ matrix, G'. Thus, n repairs can be generated, and the computational complexity of encoding and decoding is given by the size of G' (assuming a complexity of $O(1)$ for the mapping operations). ∎

3 Modifications to C versions

Our protocols presented in [4] require an interface to the FEC encoder that is not provided in either package. One requirement is that repair packets be generated *on-the-fly*. To perform on-the-fly encoding, the encoder must provide space for a fixed number of repair packets. Each repair packet is initially cleared (set to all 0's). Each data packet is fed into the encoder one at a time. The encoder applies the ith data packet, d_i to the jth repair being built by adding $d_i G_{j,i}$ to the buffer. After this addition, packet d_i can be discarded by the encoder. Once all k packets have been applied, the repair packet can be used by a decoder.

Modifications for on-the-fly encoding were performed in both packages. Initially, both packages assumed that all data packets were provided to the encoder at the same time, so that $\sum_i d_i G_{j,i}$ could be performed within a single function call. Instead, we modified the interface so that the ith call to the encode function added $d_i G_{j,i}$ to the buffer containing the ith packet.

The second modification was necessary only in the Cauchy-based encoder. It assumed that a fixed number of repairs would be generated at the same time. We simply removed the loop that iterated over the various repairs, and passed in the appropriate repair number as a parameter. The row of G that pertained to this repair was located and used to encode the packet as described above.

The final modification took place only in the Vandermonde-based coding package and involved allowing a single generator matrix to be constructed for a block size of k_{max} that could be used to support multiple block sizes up to k_{max}. This was done by incorporating a mapping function as described in Section 2.1.

We compared the encoding and decoding rates of our modified packages to the rates from the original packages. We found no noticeable difference in the rate at which they could perform repeated encoding and decoding operations.

3.1 Interface modifications

The modifications discussed above resulted in modifications to the user interfaces of the packages. To support multiple block sizes in the Vandermonde-based encoder, the block size given to initialization function creates the generator matrix, G, for the largest block size, k_{max}, that can be supported. It was initially assumed that this block size would be used to encode and decode. Instead, the user must now pass an additional parameter to the encoder and decoder, indicating the current size of the block for which encoding / decoding is being performed.

It is important to note that while the encoder and decoder can vary the block size, it is imperative that they agree on the value of k_{max}. This is because the $k \times k$ matrix obtained from a $n + k_{max} \times k_{max}$ generator matrix differs from one obtained from a $n + k \times k$ generator matrix when $k \neq k_{max}$.[1]

Functions to support on-the-fly encoding were provided via a separate interface, such that it could be excluded at compile time by an application that did not require on-the-fly encoding. The on-the-fly encoding interface consists of two functions: a repair initialization function, and an iterative repair modification function. The initialization function (e.g., `fec_encode_init()` in the Vandermonde package) informs the encoder that a repair is going to be constructed on-the-fly. A structure of type `on_fly_info` is created by the function that keeps track of a small bit of information pertaining to the repair. This structure is returned to the user.

The iterative repair modification function (e.g., `build_on_fly()`) requires the `on_fly_info` structure as well as a data packet. It performs the part of the matrix multiplication that requires the use of the particular data packet. After the function is called, the encoder has no more need for the data packet, and it can be discarded by the application as seen fit.

Some additional modifications were made to the Cauchy-based package interface that allowed run-time variation of the block size and packet size (bytes in the packet). These modifications produced an interface similar to that of the Vandermonde-based coder, and moved some pre-declared constants to variables that were fed in as parameters to the coding interface.

4 Performance Comparison between C coders

We compared the run-time performance of the encoder on four platforms:

- a 300 MHz Linux box running version 2.0.31
- a 4/266 MHz DEC Alpha running OSF/1
- a 110 MHz Sun Sparcstation-5 running SunOS 5.5 (Solaris 2.5.0)
- a 12/250 MHz SGI IRIX64

Both versions of the code were compiled using `gcc` with the same level of optimization (`-O9`) on all platforms. It is important to point out that for these comparisons, we did not attempt to further optimize the code beyond what was included by the authors initially for any given platform. Predefined constants (e.g., constant values that were used to determine field size and component size) were unaltered. The reader should be aware that varying these appropriately could possibly change performance of the coder. However, we believe that the typical user of these coder packages will make use of the packages as-is.

On each platform, we would fix a block size and a packet length and perform encodings for a deterministic set of up to $\binom{2k}{k}$ distinct combinations of packets from a set of k data packets and k repairs (where k is the block size used for that particular experiment). For larger block sizes, the number of codings were sometimes limited to reduce the time of the experiment ($\binom{2k}{k}$ can become quite large for large k). As a

[1] This holds only for the Vandermonde based encoder: the Cauchy based encoder does not require a parameter for k_{max} during initialization. This also means that there is no (theoretical) bound on the maximum block size.

result, most samples usually contained a higher concentration of data packets, which has a potential impact on per-packet decoding time (but should have no impact on per-packet encoding time). However, this impact can be considered more fair, since low loss rates result in a higher concentration of data packets than repair packets being received.

We would obtain the time spent by the machine (i.e., the user time) on this encoding (decoding) job via the system command `time`. Dividing the total number of packets that were encoded by the time gives the average encoding (decoding) time per packet. The multiplicative inverse gives the number of packets that can be encoded (decoded) per second. The same set of packets were encoded (decoded) with the C version of the Vandermonde-based encoder (decoder) and the C version of the Cauchy-based encoder (decoder).

4.1 Encoder comparison

All encoding comparisons use the standard encoding interface (i.e., not the on-the-fly interface). We point out that some preliminary comparisons showed that the on-the-fly interface had no noticeable effect on the encoding speed.

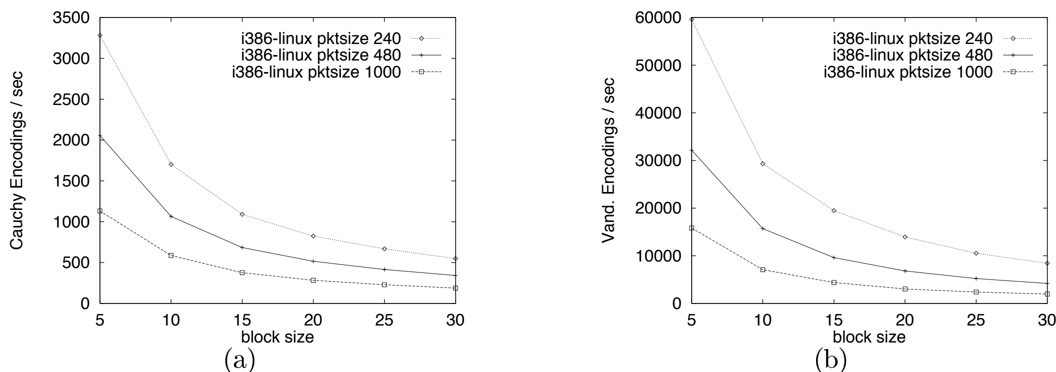


Figure 1: Encoding performance on Linux: (a) Pkts / sec in Cauchy, (b) Pkts / sec in Vandermonde. The x-axis is block size, and the various curves are for various packet sizes (given in bytes / pkt).

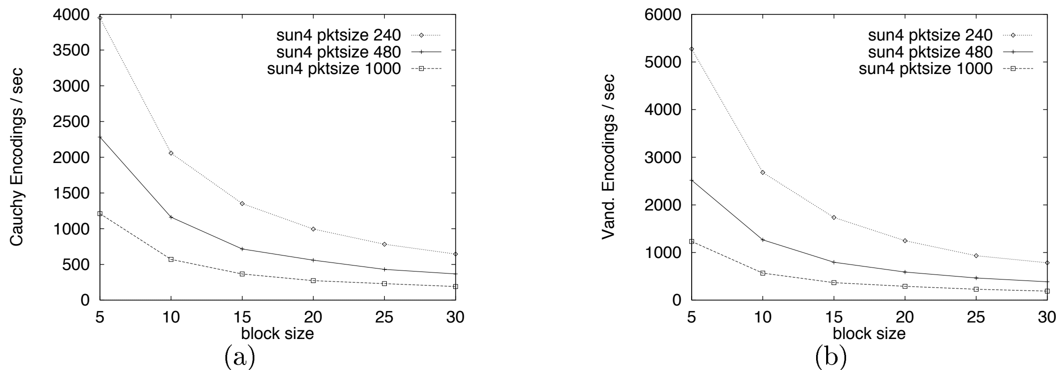


Figure 2: Encoding performance on Solaris.

Figures 1, 2, 3, and 4 give the number of packets that can be encoded per second using (a) the C version of the Cauchy-based encoder, and (b) the C version of the Vandermonde-based encoder on the Linux, Solaris, Alpha OSF/1, and IRIX platforms, respectively. The block size (k) is varied along the x-axis, and the y axis gives the encoding rate. Each curve represents a fixed packet size (in bytes / pkt).

Our results verify the results from [2] and [3] that encoding times are roughly linear in the block size and packet length. While this is not explicitly shown here, it can be seen that the encoding rate (which is the

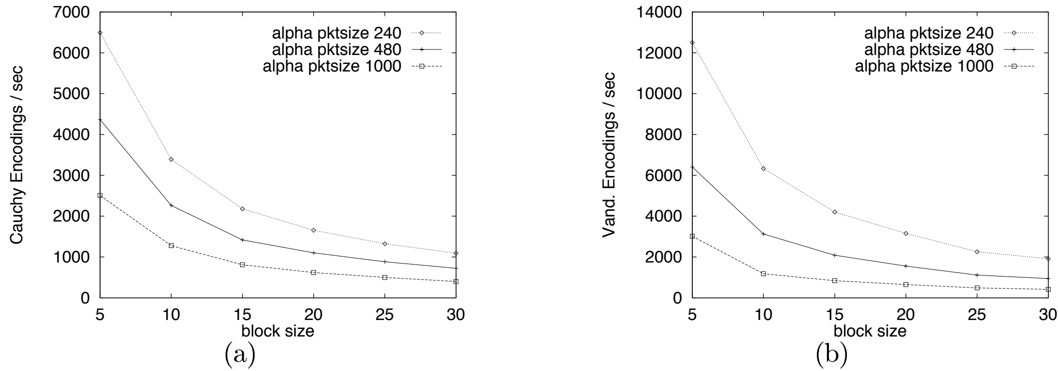


Figure 3: Encoding performance on the Alpha.

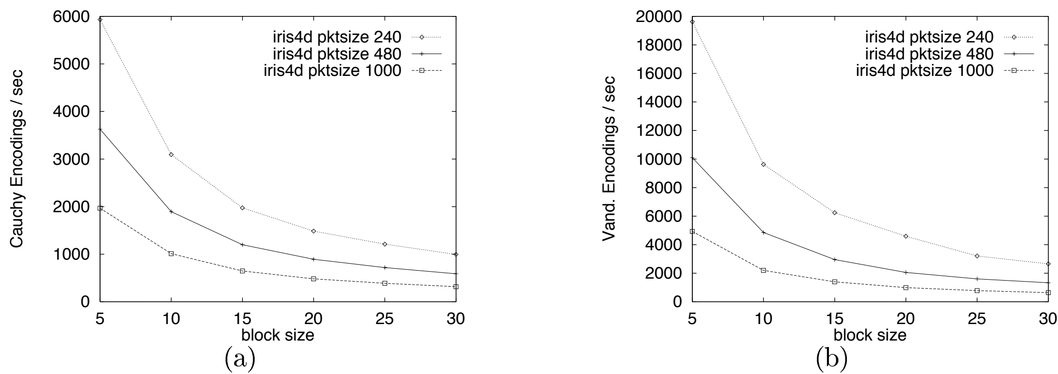


Figure 4: Encoding performance on the IRIX.

multiplicative inverse of the encoding time per packet) decays in a hyperbolic fashion.

The coding rates do not vary dramatically for the Cauchy-based encoder over the various platforms. The fastest encoding takes place on the Alpha, followed by the IRIX, the Solaris, and is slowest on the Linux box. The Alpha encoder is little more than double the speed of Linux encoder.

For the Vandermonde-based encoder, coding speed varies drastically from machine to machine, and the relative speeds of the machines differ than what is observed for the Cauchy-based encoder. The encoding speed on the Alpha is at most double that of the speed on the Linux box. For the Vandermonde-based encoder, the Linux box encodes three times faster than the IRIX box, which is 4/3 faster than the Alpha. The Alpha encodes more than twice as fast as the Solaris.

Figure 5 plots the relative performances between the C language Cauchy and Vandermonde-matrix versions on (a) Linux, (b) Solaris, (c) Alpha, and (d) Irix machines. We see that the Vandermonde-matrix encoder codes at a higher rate for almost all of our experiments. It is interesting to note that the relative difference in coding speed lessens as the packet size is increased. Also, for machines where there is a considerable difference in coding rates, this difference diminishes as the block size is increased.

4.2 Decoder comparison

Next, we turn our attention to decoding performance. For small block sizes, the Cauchy decoder is slowest on Solaris, and is slower in general on all machines than the encoder on that same machine. For large block sizes, the Linux box is slightly slower than Solaris, and on some occasions the decoder is faster than the encoder. Vandermonde-based decoding tends to be slower than Vandermonde-based encoding on a given machine.

We find again that the Vandermonde based coder can code more quickly than its Cauchy based counter-

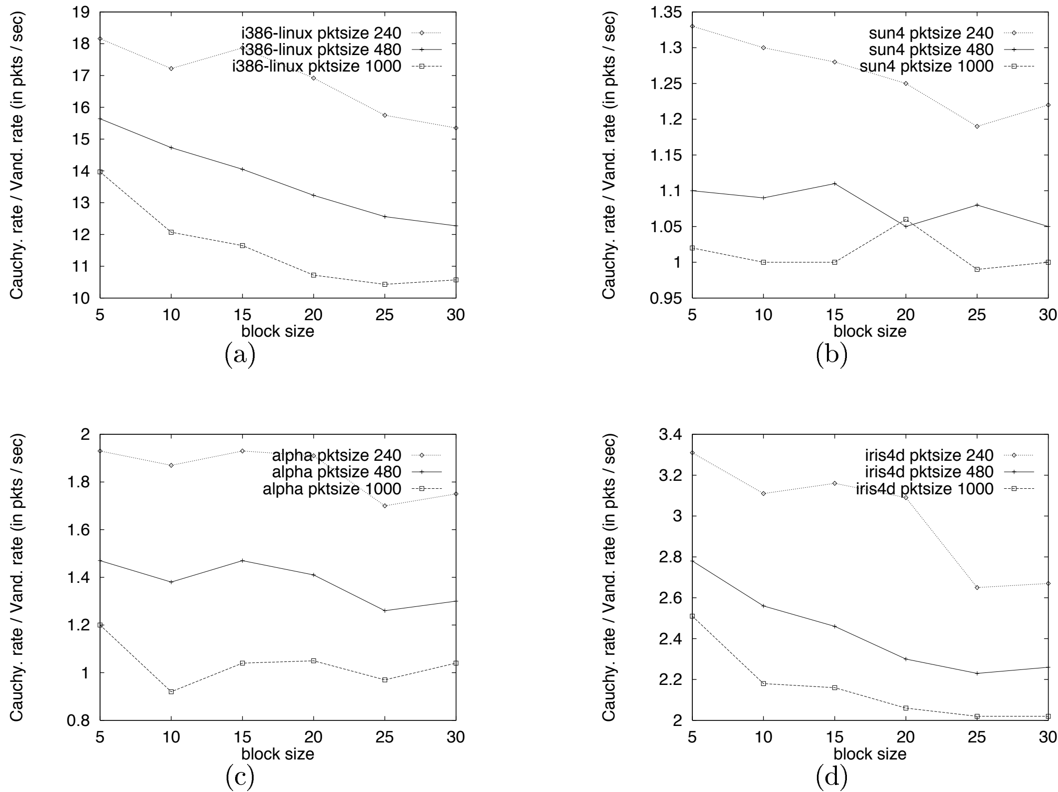


Figure 5: Encoding speedup by using Vandermonde encoding over Cauchy encoding on (a) Linux, (b) Solaris, (c) Alpha, and (d) IRIX.

part on all platforms on which we tested. Again, the hyperbolic shape of the curves assures us that decoding rate is linear in the block size. We see that for the most part, the difference in coding rates between the two decoders decreases as the packet size increases. We see no trend in their relative speed as a function of block size.

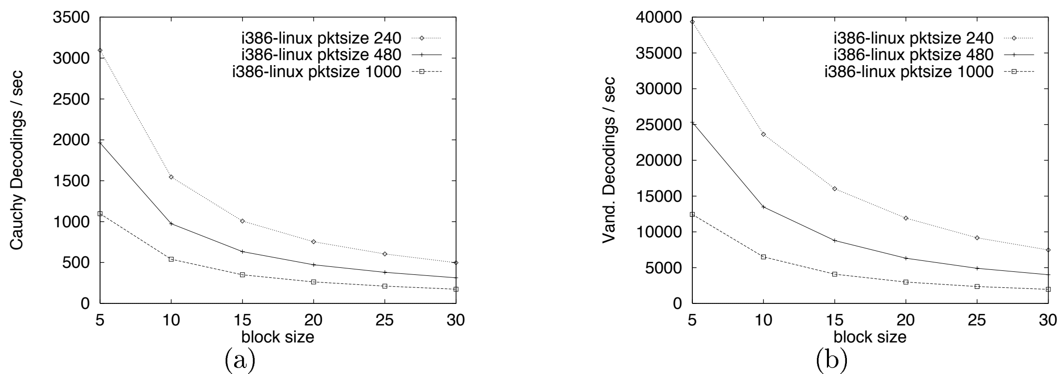


Figure 6: Decoding performance on Linux: (a) Pkts / sec in Cauchy, (b) Pkts / sec in Vandermonde. The x-axis is block size, and the various curves are for various packet sizes.

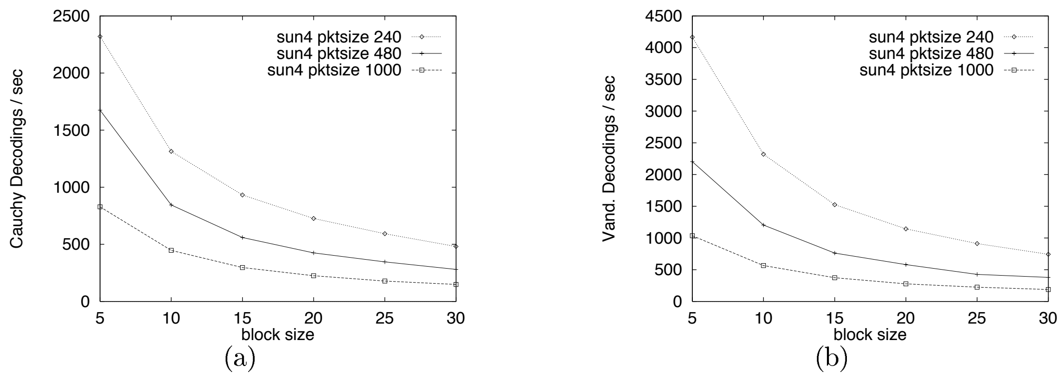


Figure 7: Decoding performance on Solaris.

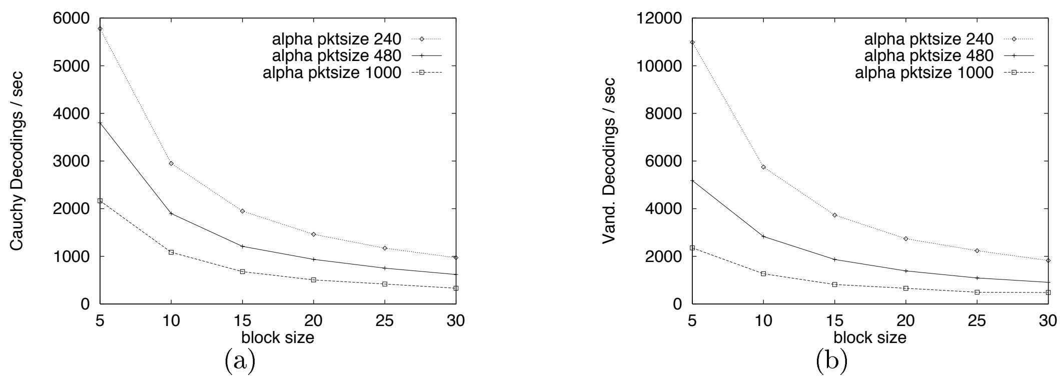


Figure 8: Decoding performance on the Alpha.

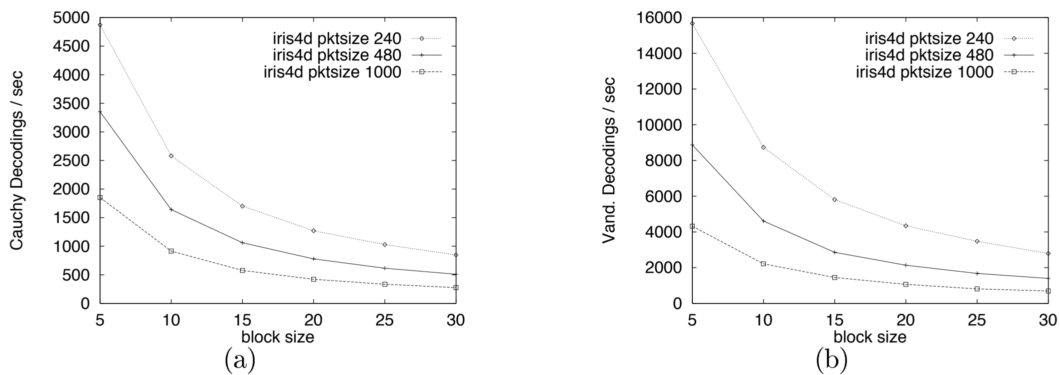
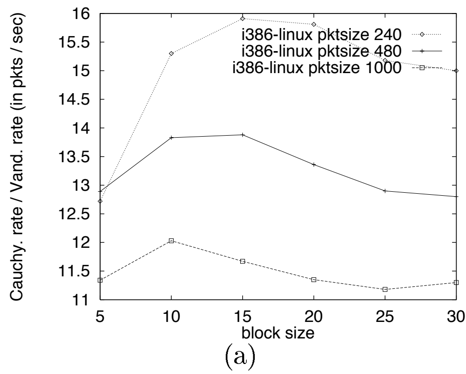
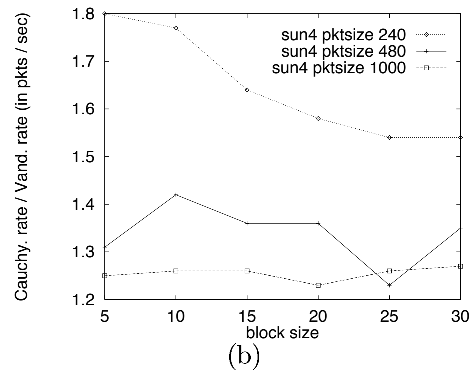


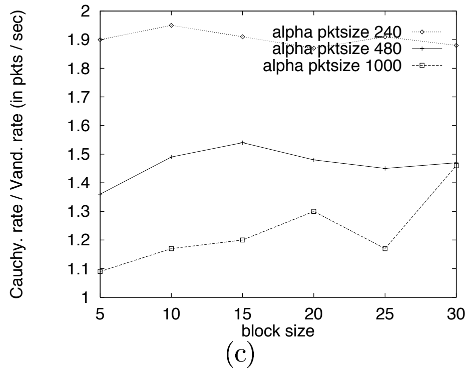
Figure 9: Decoding performance on the IRIX.



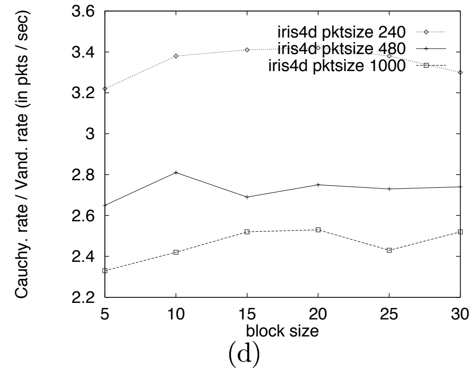
(a)



(b)



(c)



(d)

Figure 10: Decoding speedup by using Vandermonde over Cauchy on (a) Linux, (b) Solaris, (c) Alpha, and (d) IRIX.

5 Java Coder Architecture

We developed a Java version of the Vandermonde-based FEC coder. Our motivation is that several research projects have been developed on platforms that do not interface with C code. An example is the ANTS active networking toolkit from MIT [6]. Our Java version was derived from the C-language version of the Vandermonde-based coder. Here, we briefly discuss at a high level the object oriented architecture of the Java coder.

5.1 Basic architecture

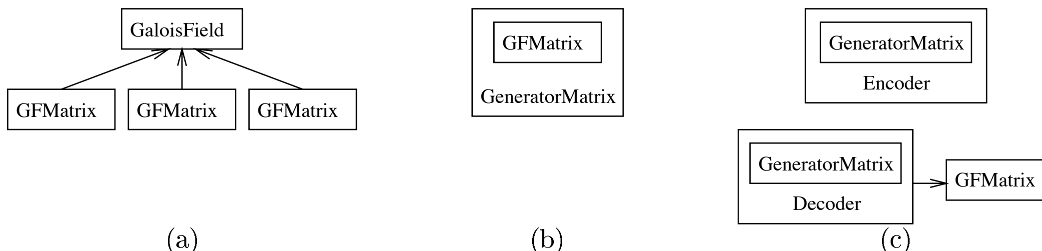


Figure 11: Arch

Figure 11 gives the basic architecture of the coding package. A `GaloisField` object provides operations for a field. Currently, all aspects of this field are fixed (e.g., the number of elements within the field). The object's methods are called whenever an addition or multiplication needs to be performed between elements of the field, or when an element's inverse (multiplicative or additive) needs to be obtained.

A `GFMatrix` object holds a matrix. Each instance of a `GFMatrix` points to a static instance of a `GaloisField`, as shown in Figure 11(a). Entries in the matrix are integers, but matrix addition and multiplication operations performed on these entries are done as specified by the methods of the static `GaloisField` object. Thus, only one `GaloisField` object exists per executing package. The `GFMatrix` object contains methods to support some of the standard operations over this matrix in the specified Galois Field, including matrix multiplication, multiplication by a single row vector, and matrix inversion.

The `GeneratorMatrix` object (Figure 11(b)) is inherited from the `GFMatrix` object, and initializes itself as the generator matrix. The `Encoder` and `Decoder` objects (Figure 11(c)) are each inherited from the `GeneratorMatrix` object, and thus contain generator matrices. The `Decoder` object also constructs an additional matrix each time a decode is requested. It holds the inverted matrix that gets multiplied by the vector of repairs to return the original data packets.

5.2 Advanced components

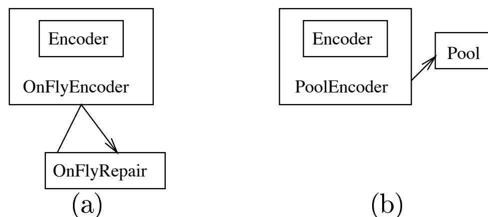


Figure 12: Arch

`OnFlyEncoder` (Figure 12(a)) is an encoder that can perform both regular encoding as well as on the fly encoding, and is derived from the `Encoder` object. Similar to the C version, when performing on-the-fly encoding, the coder requires the user to maintain an `OnFlyRepair` object that holds state pertaining to the

repair while it is in the process of being built. After calling the `EncodeInit()` method to initialize a repair, the user iteratively calls the `ApplySourcePkt()` method with a data packet as a parameter until all data packets within a block have been applied to the repair.[2]

`PoolEncoder` (Figure 12(b)) is another version of an encoder that uses a buffer pool during encoding. The goal was to reduce memory allocations by reusing previously allocated space in the hope that this would increase the rate at which the encoder could encode. It also contains a `Pool` object which maintains previously used arrays that can be reused. A `PoolDecoder` object was constructed under the same premise. It should be noted that multi-threaded applications should contain utilize a different `Pool` object per thread. Trials comparing the speed of the non-pool coders to the pool coders revealed no noticeable difference in speedup. Our comparisons make use of the non-pool coders.

6 Performance Comparison

Our comparison between the Java-language Vandermonde coder and the C-language Vandermonde coder uses the same set of experiments that we used to compare the C-language Vandermonde coder with the C-language Cauchy coder.[3] As before, the C-language coder was compiled with a high level of optimization (`-O9`) on all platforms. The Java coder was also compiled with optimizations turned on (`-O`).

6.1 Encoder comparison

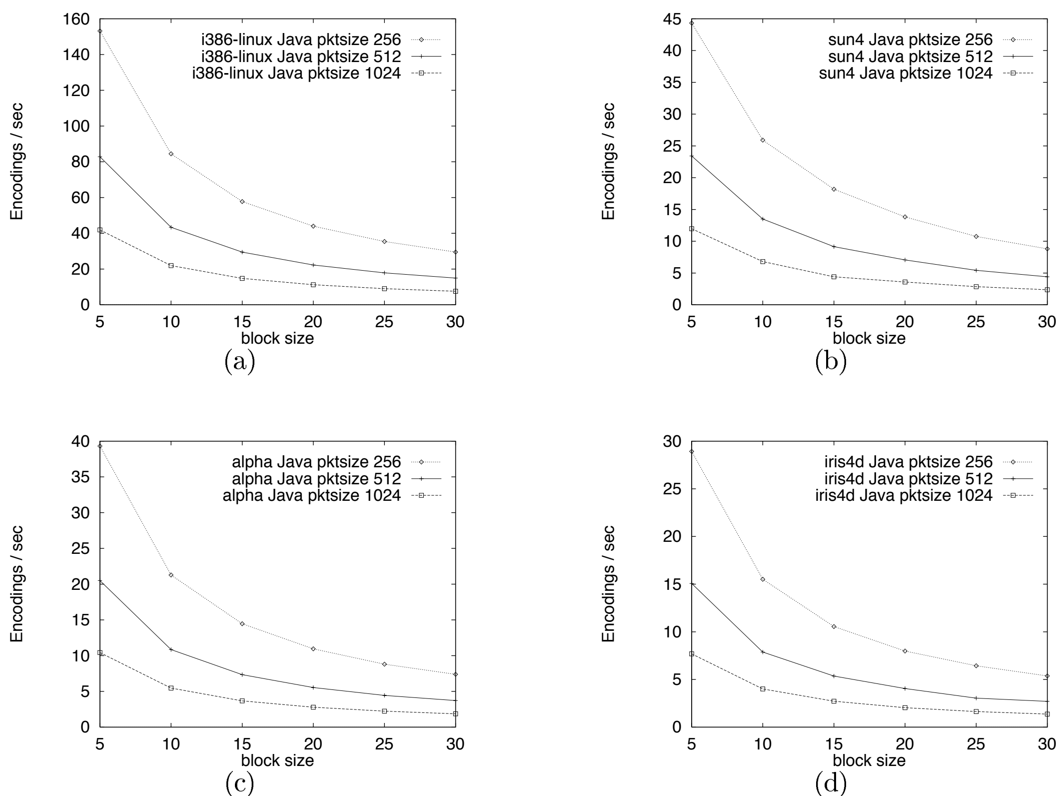


Figure 13: Encoding performance on (a) Linux, (b) Solaris, (c) Alpha, and (d) IRIX of the Java encoder package.

[2]We have discovered that the block size used in generating an on-the-fly repair can be determined dynamically. This is clearly possible due to Corollary 1.

[3]A small difference is the packet size. Due to internal configurations of the Cauchy-based coder set by the original authors, the packet size had to be some multiple of 40 bytes. Here, we set packet sizes to be fractional sizes of a kilobyte.

Figure 13 gives number of packets that can be encoded per second using the Java version of the encoder on the (a) Linux, (b) Solaris, (c) Alpha OSF/1, and (d) IRIX platforms, respectively. The block size (k) is varied along the x-axis, and the y axis gives the encoding rate. Each curve represents a fixed packet size (in bytes per packet).

The Linux box executes Java encoding roughly four times faster than the Solaris and Alpha machines, and is about five times faster than the IRIX machine.

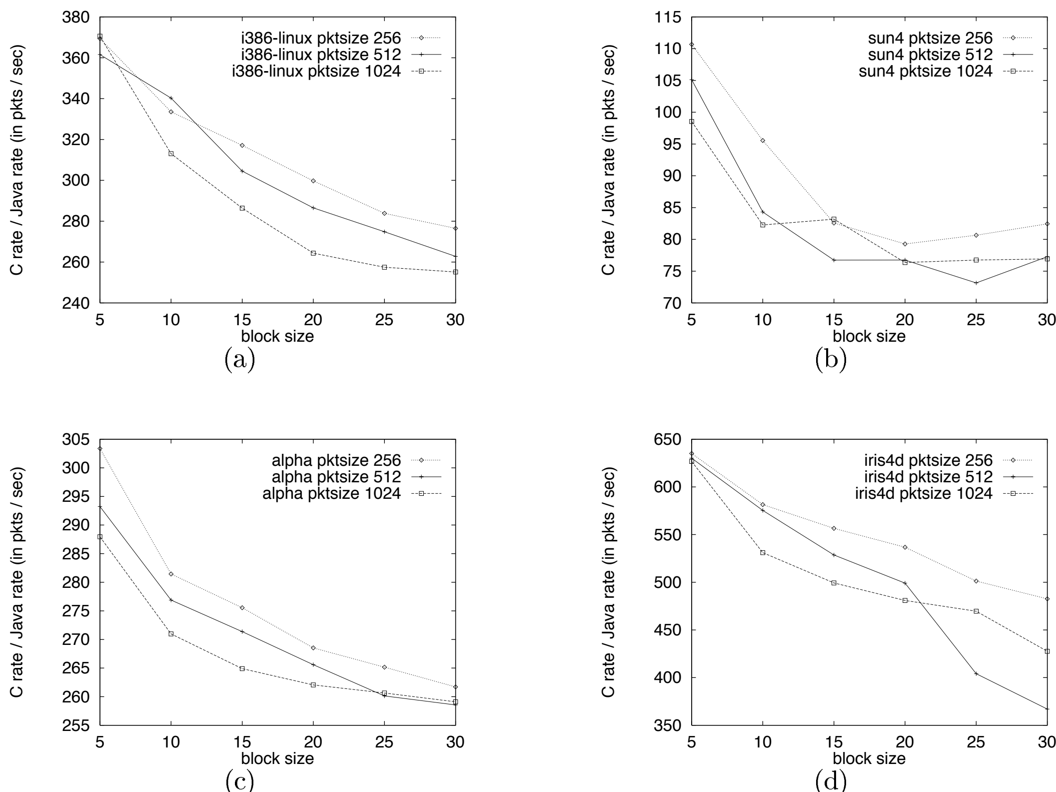


Figure 14: Encoding speedup by using C over Java on (a) Linux, (b) Solaris, (c) Alpha, and (d) IRIX.

Figure 14 plots the relative performances between C and Java Versions on (a) Linux, (b) Solaris, (c) Alpha, and (d) Irix machines. We see that encoding in Java is two orders of magnitude slower than encoding in C. This is most dramatic on the IRIX machine, and least dramatic on Solaris. Also, the speed difference tends to decrease with an increasing block size.

6.2 Decoder comparison

Next, we turn our attention to decoding performance. We find that the decoding performance of the Java implementation takes roughly the same time as that of the encoding performance. A comparison of decoding speeds between C and Java packages reveals no trend based on block size or packet length. The variance between rates on each machine is less than what was observed for the encoding rates as block size is varied (except for the ratio on the IRIX machine for a block size of 10, packet size of 1024). The difference between Java and C speeds for a decoder on a given machine is roughly identical to the difference between Java and C speeds for the encoder on that same machine for a large block size.

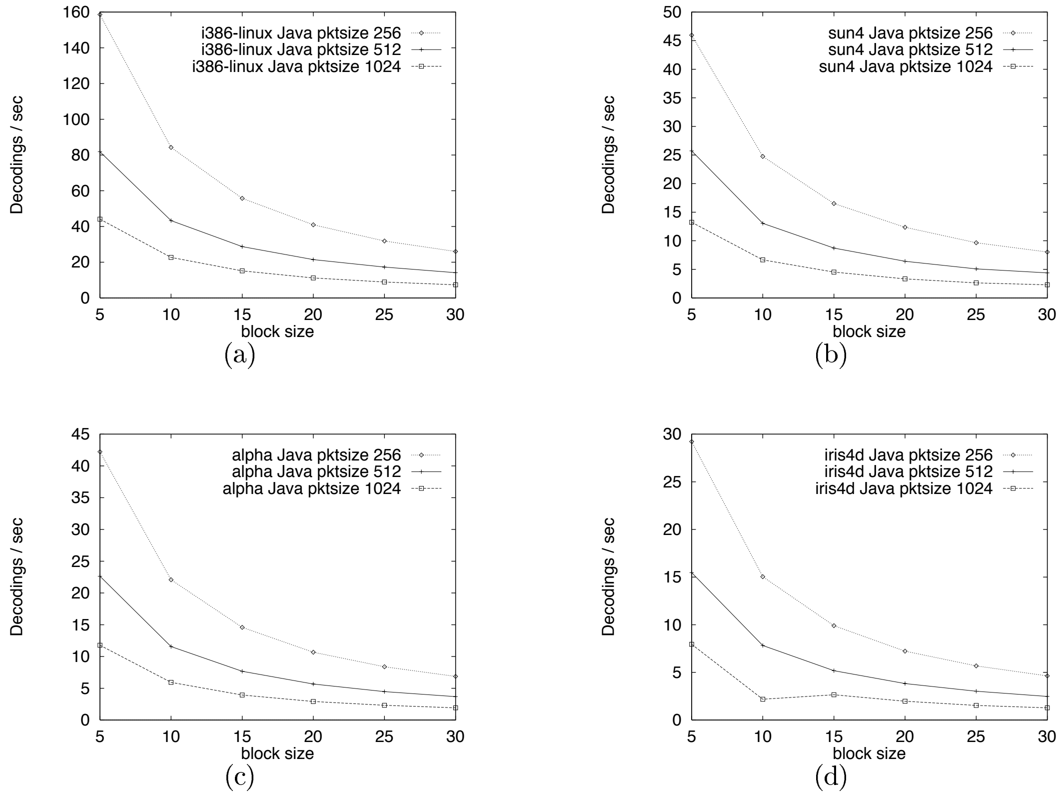
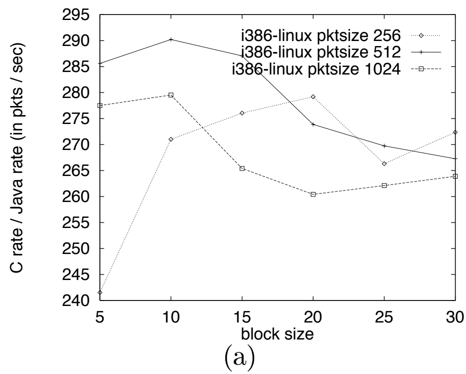
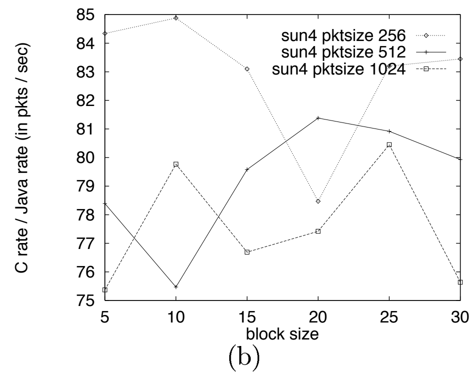


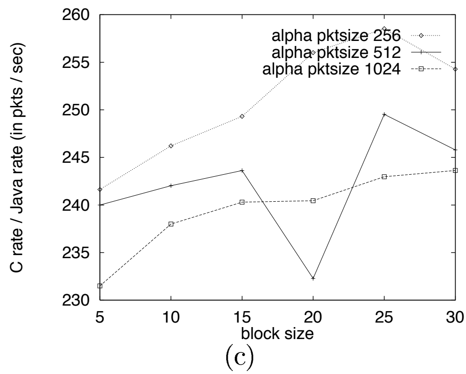
Figure 15: Decoding performance on (a) Linux, (b) Solaris, (c) Alpha, and (d) IRIX in Pkts / sec for the Java decoder. The x-axis is block size, and the various curves are for various packet sizes.



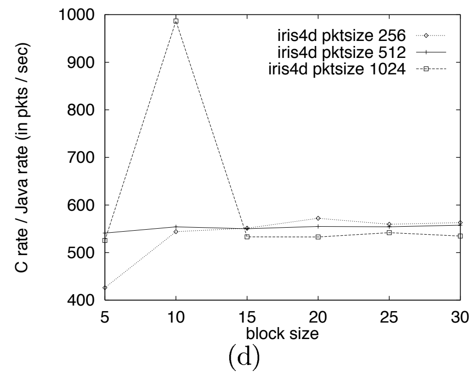
(a)



(b)



(c)



(d)

Figure 16: Decoding speedup by using C over Java on (a) Linux, (b) Solaris, (c) Alpha, and (d) IRIX.

6.3 Benefit of a Just-in-time compiler (JIT)

A Java machine that compiles to native code before execution via just-in-time compilation can speed up execution of long-running code that contains large amounts of computation. Code must run for a considerable amount of time to offset the time penalty incurred while the virtual machine byte-codes are compiled to native code.

We upgraded our Solaris machine to Solaris 2.5.1, and installed Java Development Kit 1.1.6 (jdk.1.1.6), which performs just-in-time compilation. This is the only platform on which we have access to a JIT.

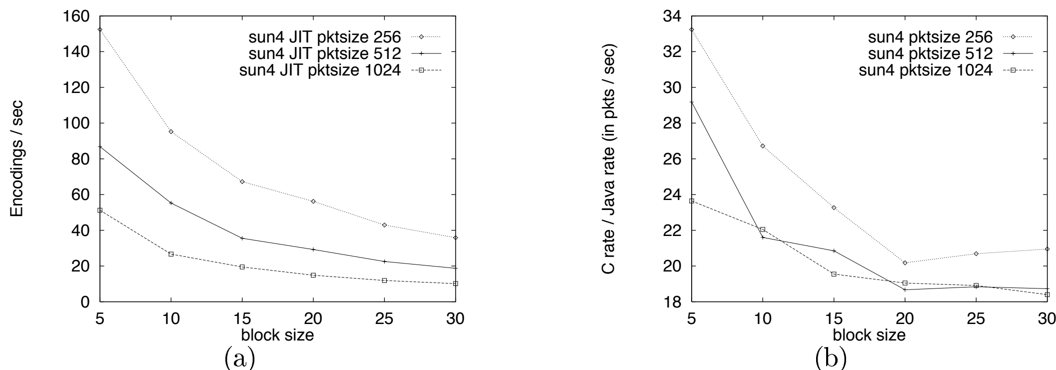


Figure 17: Encoding performance (a) on Solaris with JIT and (b) compared to C performance.

Figure 17 shows (a) the encoding rate in packets per second using the JIT and (b) the rate relative to the rate of the C encoder. We see that JIT compilation speeds up performance on Solaris roughly three or four times, making its Java performance comparable to what is observed on the faster Alpha processor.

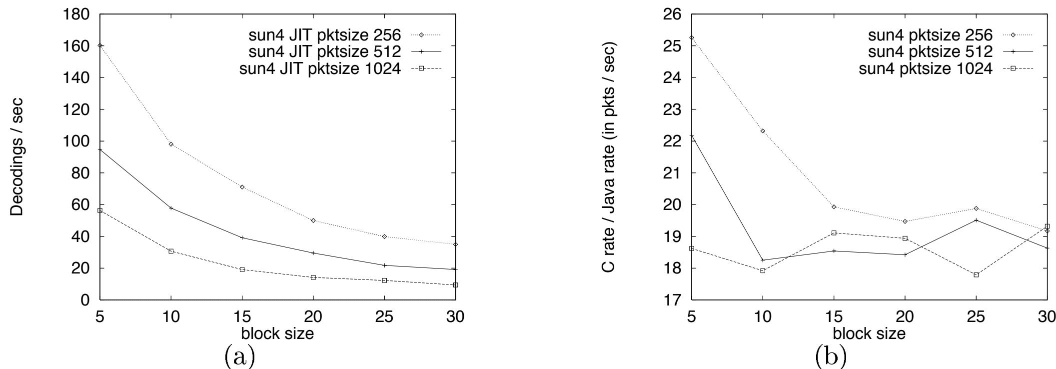


Figure 18: Decoding performance (a) on Solaris with JIT and (b) compared to C performance.

Figure 18 shows (a) the decoding rate in packets per second using the JIT and (b) the rate relative to the rate of the C decoder. Results are similar to those observed for the JIT encoder.

7 Conclusion

We have examined several aspects of increasing portability of packet level FEC encoding. First, we made modifications to two existing C-language Reed-Solomon Coding packages, and observed that these modifications had no impact on the speed at which encoding and decoding could be performed. Second, we constructed a Java coder that is based on the C-language Vandermonde-matrix based coder. We find that our Java version is hundreds slower than the original C version, but that using just-in-time (JIT) compilation can reduce this difference to only 25 to 30 times slower.

References

[1] Richard E. Blahut, *Theory and Practice of Error Control Codes*. Addison-Wesley, Reading, MA, 1983.

[2] Luigi Rizzo, *Effective Erasure Codes for Reliable Computer Communication Protocols*, Computer Communication Review, April 1997.

[3] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, *An XOR-Based Erasure-Resilient Coding Scheme*, International Computer Sciences Institute Technical Report ICSI TR-95-048, August 1995.

[4] Dan Rubenstein, Sneha Kasera, Don Towsley, and Jim Kurose, *Improving Reliable Multicast Using Active Parity Encoding Services (APES)*, UMass CMPSCI Technical Report 98-79, July 1998.

[5] Flanagan, D., *Java in a Nutshell*, O'Reilly & Associates, Inc., May, 1996.

[6] D. Wetherall, J. Guttag, and D.L. Tennenhouse, *ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols*, IEEE OPENARCH'98, San Fransisco, CA, April 1998.