Jerasure 2.0

Parth Deshmukh
University of Tennessee - Knoxville, pdesmu3@utk.edu

Sean Maginnis
University of Tennessee - Knoxville, smaginn1@utk.edu

Josh Chandler
University of Tennessee - Knoxville, jchand13@utk.edu

Follow this and additional works at: https://trace.tennessee.edu/utk_chanhonoproj

Part of the Databases and Information Systems Commons

Recommended Citation
Deshmukh, Parth; Maginnis, Sean; and Chandler, Josh, "Jerasure 2.0" (2011). University of Tennessee Honors Thesis Projects. https://trace.tennessee.edu/utk_chanhonoproj/1362
Jerasure 2.0 Design Document
A multi-threaded, backward-compatible, and enriched update
of the Jerasure Erasure Coding Library

Senior Design Project
Fall 2010

Faculty Mentor
Dr. James Plank

Joshua Chandler (jchand13@utk.edu)
Parth Deshmukh (pdesmu3@utk.edu)
Sean Maginnis (smaginn1@utk.edu)
The authors would like to acknowledge Dr. James Plank of The University of Tennessee - thank you for your patience!
Project Description

The Jerasure library, originally released in 2007, has proven an invaluable tool in the study of erasure coding. Over the three years since its release, however, the original authors and we have realized the need to overhaul the code in order to comport with current trends in erasure coding, provide a broader range of portability, and better utilize technologies that have come to prominence since its release, specifically multicore machines.

Such an undertaking is no small task in any of these regards: novel types of codes have transpired since this library was released (e.g.: Regenerating Codes by [DGW+07]) which need be implemented herein; multicore has clearly become the dominant paradigm of processor architecture, yet Jerasure utilizes none of the speedup inherent to thread-level parallelism. The realization of more efficient memory usage in the library with respect to bit-matrices has become imperative. Finally, the addition of utility functions to enhance the library’s ease-of-use would certainly augment the appeal of the library in erasure coding applications.

Therefore, for our Capstone Project, we seek to redesign Jerasure so as to address these issues. This design document will begin with analysis of the current speed and memory footprints of Jerasure, followed by a design plan that will capture the benefits of threading, structure-packing, and general tweaking for light-weightedness, and usability benefits resulting from added utility functions, and will conclude with the realization of this design within the Jerasure libraries.
Contents

1 Introduction: Erasure Coding at a Glance 6
   I Why erasure coding? .................................. 6
   II Terminology ........................................... 6
   III Canonical example with \( k + m \) disks ............ 7
   IV High-level introduction to Galois fields (\( GF(2^n) \)) 8

2 A Broad Overview of the existant Jerasure library utilities 8
   I Types of codes in the previous rendition .............. 8
   II Newly-implemented codes ................................ 9
      i High-level description of Regenerating codes .... 9
      ii Unsuitable for Jerasure .............................. 10

3 Analysis and design overview of the updated library 10
   I A foreword regarding backward-compatability with prior revisions .................................. 10
   II Threading of performance-critical sections .......... 11
      i Introduction ......................................... 11
      ii Threading Possibilities ............................ 11
      iii Design Considerations and Concerns ............ 12
   III Structure-packing for memory conservation and generality of use ..................................... 13
      i Introduction ......................................... 13
      ii Bitmatrix Structure and Composition ............ 13
      iii The Consequent Issue ............................. 14
      iv Design Considerations and Concerns ............ 14
      v Implementational Details ......................... 15
   IV Utility functions added to the library .............. 16
      i Introduction ......................................... 16
      ii Motivation ......................................... 16
      iii Design Considerations and Concerns ............ 16
      iv Implementational Details ......................... 17

4 Analysis and consideration of updates 18
   I Prior to updates concerning time and memory usage .... 19
      i Analysis of the Multicore Benefit .................. 19
      ii Analysis of Structure-packing .................... 19
      iii Analysis of Utility Functions .................... 20
5 Future Directions
I Expansion of Galois field size beyond 32 bit .......... 20
II Implementation of SSE ........................................ 21
III Object-oriented Jerasure? ............................... 21

6 Conclusion ......................................................... 21

7 Bibliography ..................................................... 23
1 Introduction: Erasure Coding at a Glance

We herein provide a high-level introduction to the uses, terminology, and methodology of erasure coding at large.

I Why erasure coding?

Erasure coding finds its theory design primarily cultivated in the fields of distributed systems and long-term storage. When data is flushed to disk, there exists no guarantee of its permanance - over time, data on the disk may become corrupt or lost. In systems where large amounts of data are generated and used over long periods of time, it becomes imperative to ensure the parity and permanance of such.

Herein lies the crux of erasure coding: how do we preserve data against loss over time, or due to degredation of the medium upon which it is stored? The solutions to this problem have been many, fomenting an impressive corpus of research and implementation over the years.

II Terminology

Erasure coding carries with it some standard terminology in describing the means by which encoding of data, and subsequent decoding, are performed. We define the following terms:

- **matrix**: Erasure coding is, at its core, an application of linear algebra - that is, encoding and decoding are performed by a series of matrix multiplications and inversions. The core of any erasure coding algorithm lies in the matrices used as input and produced by the algorithm.

- **k**: the number of *data pointers* - this may be thought of in terms of data disks in a RAID array. The parameter refers to the count of media upon which raw data is held.

- **m**: the number of *coding pointers* - continuing the RAID analogy, these would be equivalent to the parity disks. Erasure coding is not possible without *m* extra data mediums.

- **n**: this term is used to refer to the total number of data and coding pointers - that is \( n = k + m \).

- **w**: this parameter refers to the *word size* - this is the size of the Galois field used to do coding operations (more on this below). Often, \( w \)
will be seen in the context of Galois field size notation - for instance, $GF(2^w)$.

- $s$: this is the data size. The source data must be partitioned into blocks of $s$ to fit each of $m$ and $k$. Actual byte usage here may be somewhat bigger than the source data, as the bytes per "device" must be equal - this may require padding with null bytes. $s$ does not affect computation on the number of elements in the coding matrix, but is here mentioned as we consider the size and number of files where erasure-coding could feasibly be used.

- $p$: this is the packet size. Certain erasure coding algorithms are performed by converting source data into a bitmatrix: the original data in binary. The conversion from a general coding matrix to a bitmatrix requires this parameter to generate a matrix equivalent to its progenitor. The restrictions on this parameter in the current implementation of Jerasure is that, $p \times w$ must be multiple of $s$, and that $p$ be a multiple of the size of a long integer.

### III Canonical example with $k + m$ disks

We offer an example of the encoding/decoding process:

1.) We begin with some input data of size $s$ which we would like to protect against degradation. This data is stored upon $k$ disks.

2.) The data is encoded - this encoding is stored on the remaining $m$ disks. We do this by creating a matrix consisting of $k$ rows, each row containing the contents of its encapsulating disk. This matrix is multiplied by a Vandermonde matrix, whose first $k$ rows are the identity matrix, and whose final $m$ rows are for encoding, which are particular to the algorithm being used (as per [PlDi05]; consult this document for a more detailed description of these).

3.) The result of this multiplication is a $k + m$ matrix, where the first $k$ rows are the original data, and the last $m$ rows are the encoded data.

4.) When we detect disk corruption, damage, or loss, we may decode and retrieve the original data. We do this by eliminating the rows of both the original coding matrix and the encoded data matrix corresponding with the errant disks, inverting the coding matrix, and multiplying this inversion by the reduced data matrix (called the survivor matrix). The
result of this is the original data. Of course, this portion of the process is subject to a few constraints:

- If we cannot utilize the contents of more than $m$ disks, the data is unrecoverable.
- If the Galois field size is not sufficiently large, the coding matrix will be uninvertible, and again data will be unrecoverable.

The “magic” of erasure coding comes from the final $m$ rows of the Vandermonde matrix - to reiterate, these rows come to be as a result of the particular type of encoding scheme used.

IV High-level introduction to Galois fields ($GF(2^n)$)

In order to ensure parity when decoding, standard arithmetic over the natural numbers cannot preserve data integrity. Therefore, we must operate in a field conducive to these ends - we do this by operations in a Galois field (commonly denoted as $GF(2^n)$, where $2^n$ is the highest order element in the field - note this $n$ is distinct from the $n$ denoting the total number of disks).

Addition and subtraction in Galois fields is equivalent to the XOR operation. Division and multiplication is more complex, requiring many operations in order to compute results - therefore, Jerasure precomputes these values and stores them in lookup tables, so as to ensure amortized constant time when performing such operations.

2 A Broad Overview of the existant Jerasure library utilities

I Types of codes in the previous rendition

Jerasure currently contains several encoding routines renowned in erasure coding for their relative parsimony and reliability.

Foremost among these is the Reed-Solomon encoding method, which has been used commercially in the RAID-6 disk array model. This method works as described in the canonical examples above.

The library also includes the Cauchy-Reed-Solomon (or CRS encoding method, which extends the Reed-Solomon method to work with bitmatrices. Jerasure also contains routines geared towards improving such matrices, depending in part on the number of ones the bitmatrix contains. There also exists an alternative CRS encoding module in Jerasure, which contains
lookup tables for the best CRS matrices as discovered by experimentation by the original authors.

Another module contains the Liber8tion encoding method (a RAID-6 code devised in [Pla08a] specifically for $w = 8$), which utilizes two alternative encoding matrices - the Liber8tion bitmatrices, and Blaum-Roth bitmatrices, both examples of minimum density spanning or MDS codes. InJerasure, these matrices are used almost exclusively in the context of generating schedules: data structures to optimize coding operations (and which in general are beyond the scope of this project, but are here mentioned for completeness).

II Newly-implemented codes

The authors considered one particular class of codes in our project, called Regenerating codes, for inclusion in the new rendition of the Jerasure library. This code was first devised in [DGW+07]. In the course of our research and consideration of these codes, we deemed them incompatible with the Jerasure library in its current form.

i High-level description of Regenerating codes

The concept of Regenerating codes was motivated by the example of very large distributed storage systems which receive massive amounts of data on the global network. These codes maintain that data should not be replicated, and that as much bandwidth as possible should be conserved should the need to restore lost data arise.

The encoding process involves creating fragments of the original data of size $\frac{s}{k}$, then transmitting these fragments from the source to the nodes of the network. When a client connects seeking to download the original source file, it need only connect to a minimum subset of the fragment-holding nodes, download them, and decode the file, effectively reducing the number of bytes transferred in a download.

Should a node of the network fail, the file may be repaired (that is, restored and re-encoded) in a process much like downloading the file, wherein the newcomer node downloads a subset of fragments and subsequently generates its own fragment, preserving the data while minimizing bandwidth.

Regenerating codes are disadvantageous in that they impose higher storage and computational requirements on fragment hosts: in the former case, this entails maintaining matrices of encoding coefficients used to generate the fragments (although this overhead is generally small in practice); in the latter, newcomers and downloaders must first obtain and decode fragments
before they are able to access the source data.

ii Unsuitable for Jerasure

The authors decided against an implementation of Regenerating codes in Jerasure. This was motivated primarily by the fact that such codes are tailored more to network transmission than general-purpose erasure coding, and thus would be better suited to its own application or network coding library than to Jerasure.

3 Analysis and design overview of the updated library

We herein describe our revisions to the library. We will first describe our design concerns in detail, followed by analysis to validate our position, and conclude with a presentation of our design. Where applicable, we will follow this with with implementational details.

I A foreword regarding backward-compatability with prior revisions

An omnipresent concern during the design process was maintaining compatibility with the previous rendition of Jerasure. Many of our designs came down to modifying the existing code in such a way that the return values of the library’s functions would be wholly different than was previously implemented, which naturally endangers the syntax of programs using the previous version of Jerasure (or worse, instills semantic errors that do not arise until late in a program’s execution in the worst case).

Likewise, this concern was present in threading existing code and writing utility functions (which also utilize threads) - our choice for a threading library, pthreads, imposes installation of this library for all programs utilizing Jerasure. Our design simply assumes it exists, in the worst case, to the detriment and increased man-hours of the programmer.

In the following sub-sections, we will address this issue as a fundamental requirement for each of our designs, denoting our solution to the problem, and alternative solutions devised during the design process.
II Threading of performance-critical sections

i Introduction

Considering the direction of computer manufacture and architecture over
the past few decades, there is no one concept more emphasized or imple-
mented than the use of multicore in keeping with the computer industry’s
dedication to faster and more efficient computation. Such design choices have
made imperative to the programmer the need to capitalize on this expanded
computational power, and thus fomented the standard data structures and
procedures of parallel programming; with respect to library and applications
development, software utilizes more cores via threads.

There are no indications that a competing paradigm will come to
dominate multicore in the near future, implying that software designed to
harness multiple processors implicitly extends its usefulness and lifespan. In-
deed, the software engineering community at large seems to concur, with
some current projects centered solely on inducing concurrency into existing
software packages, or pre-implementation design focused on utilizing multi-
core.

Given these facts, the authors of this document and the original Jera-
sure library find it beneficial to augment the library at large so as to capitalize
on multicore by enriching certain of its functions to utilize the computational
capabilities that threaded applications afford.

ii Threading Possibilities

For sufficiently large matrices, spawning a thread to compute each dot
product could provide a significant speedup to the process of multiplication.
As was previously indicated, at its core, erasure coding is an exercise in
applied linear algebra; thus, the parallelization of these routines should the-
oretically decrease mean execution time for encoding and decoding routines.

The preeminent use of multicore, as the authors have ascertained,
would fall to convenience functions augmenting the existing library. A prime
example comes from structures integral to Jerasure’s processing; most Jeras-
sure routines rely on table lookups for computation across Galois fields, the
majority of which are initialized as they are used (the lazy initialization de-
sign pattern); it follows that for applications recognizing that they will use
all such tables eventually, initialization of all tables at startup could save the
iii Design Considerations and Concerns

With respect to matrix multiplications, we posit that the complexity of the design herein would warrant a new data structure to encapsulate the matrices, itself containing data structures to prevent race conditions. To ensure that this data structure is simple to use, we would then compose utility functions to handle their operations - for example,

```
matrix *threaded_multiply (matrix m1, matrix m2);
```

where $m1$, $m2$, and the return type represent these data structures. This routine would examine the dimensions of these matrices (ostensibly internal to the structure itself), allocate and initialize a new matrix structure of size $\text{rows}_{m1} \times \text{columns}_{m2}$, and begin spawning threads to compute dot products between the pairwise rows and columns of $m1$ and $m2$ respectively, filling in the newly-allocated structure with the return values of joined threads.

Ensuring backwards compatibility on this front was a contentious issue during the design process. Our final design to this end relies upon the use of C preprocessor flags and definitions to manage compilation. In a mockup implementation of this scheme, we defined a preprocessor constant `JERASURE_2.0` in a separate header file. If this constant is found to be defined in other header files (i.e., it is not commented out or deleted upon inclusion of our header file containing this definition), we issue directives to include `pthreads`, and to define such threaded routines. Otherwise, these libraries are omitted and Jerasure essentially reverts to the previous, non-threaded version. The imposition of editing a single file to ensure this behavior is, we feel, no great imposition on the end user.

There still arises the question, however, of whether the overhead of creating so many threads, particularly for a truly massive matrix, is less than the cost of inline dot products, a matter which will be further discussed in a later section.
III Structure-packing for memory conservation and generality of use

i Introduction

Jerasure extensively utilizes bitmatrices in the encoding and decoding process - indeed, there exists a routine solely to convert existing integer-valued matrices into equivalent bitmatrices. Bitmatrices reduce coding computational complexity in general by turning all matrix operations into a series of xors. The bitmatrix’s most pronounced application in Jerasure, however, is towards creating schedules as was referenced previously in this document - such schedules use heuristics for the coding process, attempting to further minimize the number of required coding operations based upon the content of the matrix itself.

The current implementation of Jerasure, however, defines these structures as whole integers per bit, from which only one of each integer’s thirty-two bits will ever be utilized. For large bitmatrices, the amount of wasted space quickly becomes massive (as we will demonstrate). On the merit of this, we posit a redesign of this aspect of Jerasure so as to maximize the library’s efficiency.

We will describe the structure and composition of such matrices, followed by our proposed solution and implementation.

ii Bitmatrix Structure and Composition

The structure of the bitmatrix is derived from the procedure described in [BKK+95]. Here, we start with a matrix with \( k + m \) rows \( \in GF(2^w) \). Through a series of further expansions, we derive the final matrix to be of size \( w^2 \times k \times m \), remaining functionally equivalent with respect to its generating matrix.

In practice, \( w \) must be of sufficient size to ensure invertibility (as was previously mentioned) of the coding matrix during decoding - again, non-invertibility implies unrecoverable data loss. We here mention a sufficient field size to remind the reader that, in practice, \( w \geq 16 \) which places an implicit lower bound on memory usage to guarantee system reliability.
iii The Consequent Issue

We have previously stated that Jerasure utilizes integer-type variables for the elements of its bitmatrix. As becomes very clear upon examining the composition of the bitmatrices above, the space complexity of the resultant bitmatrix in terms of bytes is presently $O(4 \times w^2 \times k \times m)$, assuming an integer size of four bytes. Given our previous note on practical field sizes, the asymptotic lower memory bound becomes considerably larger.

Being that each entry will only use a fraction of the total number of bits allocated ($\frac{1}{32}$, to be precise), this implementation has the effect of wasting a large amount of heap memory.

iv Design Considerations and Concerns

The question of a new bitmatrix representation posed us considerable difficulty, being that we wanted both to reduce the memory impact of the current bitmatrix, while, of course, ensuring backwards-compatibility for users of the current Jerasure build.

We derived two approaches. A preliminary approach was to use C’s enigmatic bit field operator as a member of a struct. Structs containing such members resemble the following:

```c
struct packedStruct
{
  <type> packedNum : x;
};
```

where $<\text{type}>$ is a C variable type upon which bitwise operators may be performed (e.g.: `int` or `char`), which is usually further qualified to include or exclude the sign bit to ensure portability; and $x$ is the number of bits to allocate for this variable. Using such a structure would allow the bitmatrices to be exactly the number of bits used in the bitmatrix.

Unfortunately, due to word-alignment requirements imposed by our compilers, this approach still wastes too much space with the most conservative type (`char`), as the full number of bytes are allocated for the type field specified. We could only hope to save 16 bits per matrix over the integer-based implementation (which even then would be under-utilized with respect to the ideal); and moreover, even if this were possible, dereferencing and set-
ting bits would require complex and fragile pointer arithmetic and bitwise
operations.

We then came upon a better solution: using an array of characters. Given that \texttt{sizeof (char)} = 1 byte for our machines, we incur a maximum overhead of 7 unused bits for any bitmatrix in the worst case. Moreover, we may reduce the complexity of indexing appropriate bits to a set of preprocessor macros.

\section*{v Implementational Details}

This portion of our design yielded a more concretized implementation. We first defined a set of utility macros for managing \texttt{char *} bitmatrices.

Let \( b, s, \) and \( d \) be bitmatrices, where \( s \) and \( d \) are source and destination matrices respectively; \( i \), an index into the bitmatrix; and \( x \), a bit to insert into the bitmatrix. We then defined the routines as follows:

\textbf{get\_bit:} To get bit \( i \) from \( b \), we return \( b[i/8] \ll (i\%8) \).

\textbf{set\_bit:} To set bit \( i \) of \( b \) to \( x \), we first check if bit \( i \) is already this value. If so, we simply return. Otherwise, we use \( b[i/8] = (1 \ll (i\%8)) \).

\textbf{zeroing:} This routine sets all bits up to a user-specified size to zero. We do this by iterating up to “size”, calling set\_bit with \( x = 0 \) for each \( i \).

\textbf{copy:} This routine copies “size” bits from \( s \) to \( d \) by iterating up to “size” while calling set\_bit upon \( d \); for each such call, we use index \( i \) and define \( x \) as a call to get\_bit upon \( s \) for the same \( i \).

We then needed only to worry about backwards compatibility. To this end, we used the same scheme as in the section on threading (defining a version number in a special header file) and began recomposing the header and source files such that if the \texttt{JERASURE\_2.0} constant is defined, those functions that accept as input and/or return \texttt{char *} bitmatrices will be included and used as the appropriate function definition in the source code; otherwise, the \texttt{int *} approach will be maintained.

The updated functions required us only to edit the existing source to accept as input and/or return bitmatrices, index bits with our utility functions, and wrap these and the original functions in preprocessor directives such that the appropriate functions are included.
The resultant memory savings we incur by this implementation will be discussed in the results section below.

IV Utility functions added to the library

i Introduction

As this update to Jerasure introduced threading into the core of the library, we incidentally devised a convenience function for threaded initialization of Jerasure’s Galois field tables.

ii Motivation

Entries in each of the Galois field tables are initialized only when they are needed. With calls to the functions in the Galois field arithmetic library, the table entry for the $w$ passed to that function is compared against null - if this comparison returns true, an initialization routine is called to build this portion of the table.

Certainly for most applications, Jerasure is correct in using lazy initialization, since with all probability a distinct subset of the tables will be used - ergo, initializing all tables would waste a non-trivial amount of space. However, we considered it probable that the end user would perhaps want to utilize a greater range of $w$, and would not want to waste time creating these tables while the calling function stalls until initialization is complete. Moreover, given that threads are implicit in our revision, we hoped to realize a speedup in table creation based upon the fact that we could distribute these computations to multiple cores.

iii Design Considerations and Concerns

For the utilities themselves, backwards compatibility became less a concern than in our other revisions: the functions required to implement this portion were separated into their own module; inclusion of the related header file was wrapped in preprocessor directives contingent upon the definition of JERASURE_2_0 once again. This revision did require modifications to the Galois arithmetic module which we will describe momentarily.

The design challenges herein stemmed from two sources: the first was ensuring that no table entry would be doubly-initialized. If the early
initialization routines are running, and concurrently a function is called that requires these tables, Jerasure could invoke the existing initialization routine upon that table entry, making for memory leaks or worse, a race condition between the initialization thread and the initialization function. To compensate for this, we defined a static array of mutexes from the pthreads library, which were used to secure table entries. We then augmented the procedures in the Galois module to check this lock and proceed with initialization were it able to acquire the mutex, and blocking on the mutex until the initialization routine finished otherwise.

The second was how to control return values from initialization routines in a multithreaded environment. Typically, the Galois field routines will return 1 for success and 0 on failure (generally when a call to malloc fails). Being that we are returning anything at all implicitly disallowed our detaching the initialization threads from the thread routine itself. This, of course, foments a bottleneck should the user want to initialize all the tables at once.

In the course of authoring this document, we find ourselves unset-tled as to what to do on this point. One potential workaround to this was to create a master thread responsible for spawning the early initialization routines. This thread would then be told by the user whether it should be concerned about the return values from the Galois initialization threads - if not, it could simply ignore them; else, it could cache them in a user-provided array and return them upon joining. This provides a measure of flexibility in that the thread may be joined after other tasks have been completed, and its return values measured for errors. We realize, however, that this idea need be further cultivated before implementation.

Another fix, which is admittedly lacking in elegance, would be to define static arrays matching the size of each table, for each of the tables. These arrays would be sentinelized to show no initialization, and filled in with the return value of the initialization call as it occurs. Such arrays could be made thread-safe by using the same mutexes we defined to protect the tables. The natural drawback to this method is that it requires a good deal of overhead per table, even if we use chars as the data type for these arrays.

iv Implementational Details

In large part, our design for this section coincided with the implementation - therefore, much of the detail has been laid out in the previous section.
We have defined a few functions in achieving early initialization. First of all, we have defined a “master function” to determine what the user wants initialized, and to set up and run threads if this has been requested. This function takes as input a preprocessor defined, bitmasked integer corresponding to which tables the user wants initialized, and a pointer to an integer \textit{wlist} and an integer \textit{wsize} to determine which specific table entries should be created, and how large this list is, respectively. Of course, the routine may be sent a the address of a stack allocated integer - this is fine, so long as \textit{wsize} is set to one. In fact, this is required to tell the routine to initialize all possible tables, as the method provides this behavior upon seeing a $-1$ in \textit{wlist}.

This routine then calls a suite of subroutines in setting up the appropriate tables. Such routines have been defined for every table type so defined in the Galois arithmetic module. These routines are also set up to handle multiple values of \textit{w}, so that they may iterate the list and create tables as intended. This is accomplished in general by spawning a thread for each \textit{w}, where the function for each thread acts like the initialization routines in the original Galois module.

Of note is the fact that certain of the Galois tables require other Galois tables to be set up (for instance, the division tables must use the results of the log tables for the equivalent value of \textit{w}). Therefore, the master function “intelligently” parses the integer code, calling for the creation of independent tables before dependent ones. It is hoped that, in doing so, the independent tables will have had at least some time to make progress on their calculations, thus minimizing the wait for the dependent tables. We acknowledge that a thread’s gaining control of the processor cannot be readily ascertained by the user, but suggest this approach to be at least in concept better than heedlessly spawning table creation threads.

4 Analysis and consideration of updates

We herein offer an analysis of the benefits of each of our design considerations, in the order in which they were presented previously.
I Prior to updates concerning time and memory usage

i Analysis of the Multicore Benefit

Through extensive research of the code at large, we have determined, unfortunately, that threading seems to serve Jerasure little to none. The ultimate and underlying reason for this determination is due to Jerasure’s core methodology: encoding and decoding operations boil down to simple dot products. Given compilation optimization of Jerasure’s routines, the speedup would be negligible at best for most matrices that could conceivably fit in main memory. Already, the code as it is currently authored has been optimized to maximize the speed of matrix operations. Testing done to this end has showed exceptionally minimal slowdown as matrix size increases.

We also bring into question how great the overhead of creating a thread is versus simply doing a dot product, often using exceptionally quick bitwise operations. It would seem that the time necessary to set up the thread and begin its execution would be quite immense, particularly with large matrices (indeed, the very problems we hoped by threads to solve). This, however, requires further testing and implementation in the library at large before this issue may be resolved.

ii Analysis of Structure-packing

As has been made clear in the related section, structure-packing in Jerasure is heavily warranted, particularly considering its current configuration. As was previously mentioned, by reducing the structure down to a char * array upon which we perform bitwise operations, we may strip a large constant factor from the memory usage of bitmatrices.

Certainly this is not without its price: the operations required to do this are inherently more complex than simply iterating an $r \times c$ matrix and masking out its values. By the virtue of our scheme to ensure backwards compatibility on this front, the source code has inevitably become bloated, certainly with many redundant instructions.

However, we feel that software vying for contention in a global market with a host of alternatives might only be taken seriously if it is elite in every way. Assuredly this is not so, and such alterations as we have put forth would rectify this tremendously.
iii Analysis of Utility Functions

The functionality provided by this portion of our design is somewhat debatable. On one hand, a key point in software development is to make more convenient a task that was previous less or not so. On the other, one must ask how much, if any, utility would be garnered from such functionality.

Admittedly, the number of tables in the Galois arithmetic table are not exceedingly large; moreover, their initialization in the general case is a series of generally trivial computations. To this end we wonder, much like when considering the benefits of multicore at large, whether the overhead of threading, coupled with the additional data structures to stave the many consequences of using threads on shared data, will in the long run be a greater detriment than the current setup. To this end, much more testing must be done on these functions to determine their uses.

On the subject of convenience, a particular gap in our knowledge while designing this functionality was, for whom are we providing such functionality, and for what purpose is the functionality being utilized? We might extend these questions to the library itself, in all reality. It is a common occurrence for the computer scientist to regard a piece of software as interesting, but in the real world have it be ignored or abhorred. A lack of knowledge of the target market fomented the development of this portion with nothing but good intentions - it shall remain to be seen if these were to good ends.

5 Future Directions

Herein, we would like to posit future work we have considered in the process of our design. We present this as such with the hope that the library’s maintainers after us may consider these possibilities in subsequent renditions of Jerasure.

I Expansion of Galois field size beyond 32 bit

Jerasure’s galois fields have an absolute upper bound of $w = 32$. This has occurred in part because table computations require domain knowledge of certain primitive polynomials. At the original authoring of the library, the higher-end polynomials grew quite massive, threatening to exceed storage limits for an integer type.
Over time, however, such limitations have been relaxed. With the advent of the long long int data type, for example, the range of supportable polynomials has increased dramatically. Consequently, so could the maximum $w$ of Jerasure.

II Implementation of SSE

Streaming SIMD Extensions are instruction set additions for the x86 instruction set, created to make vector calculations lightning-fast. SSE has gone through several versions to date, with a host of new instructions added per version, geared towards this end.

Developers of the instruction set have created a C API, by which the user may directly instruct the process to do these rapid calculations on data he or she provides. Given, as we have previously stated, that Jerasure functions largely as a result of vector calculations, one might witness enhanced library performance were it augmented with these calls.

III Object-oriented Jerasure?

The domain of systems programming is, with good reason, a constant companion of the C language for its general purpose speed and power. In keeping with these traits was Jerasure devised, and with good reason: the library, compiled with maximum optimization, is incredibly fast for even the most impatient user, and consequently so powerful.

While acknowledging this, it would be interesting to see an implementation of the library in a object-oriented language. The functions of each module of Jerasure might be coalesced into a distinct class, and each class given matrices upon which to operate. Gauging the community response for such an implementation might also be an interesting research topic: has tolerance for such languages among systems programmers increased over the years?

6 Conclusion

We here conclude our Senior Design Project. In many ways, our work and research has only scratched the surface insofar as the future of the library is concerned. It is certain that the Jerasure library will remain a top contender among erasure coding packages in the years to come. It is our sincere hope that we have had a hand in the direction of development for
this software, and that it will continue to be a prime example of excellence in erasure coding.
7 Bibliography


