CS252
Graduate Computer Architecture
Lecture 24

Error Correction Codes
April 23rd, 2012

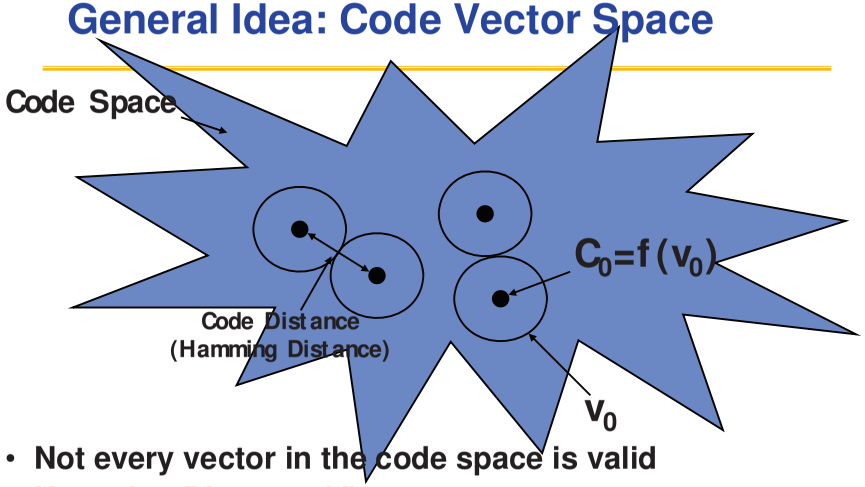
John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs252

Recall: ECC Approach: Redundancy

- Approach: Redundancy
  - Add extra information so that we can recover from errors
  - Can we do better than just create complete copies?
- Block Codes: Data Coded in blocks
  - k data bits coded into n encoded bits
  - Measure of overhead: Rate of Code: K/N
  - Often called an (n,k) code
  - Consider data as vectors in GF(2) [i.e. vectors of bits]
- Code Space is set of all 2^n vectors, Data space set of 2^k vectors
  - Encoding function: *C=f(d)*
  - Decoding function: *d=f(C')*
  - Not all possible code vectors, *C, are valid!*

General Idea: Code Vector Space



- Not every vector in the code space is valid
- Hamming Distance (d):
  - Minimum number of bit flips to turn one code word into another
- Number of errors that we can detect: (d-1)
- Number of errors that we can fix: ½(d-1)

Some Code Types

- Linear Codes: $\vec{C} \forall \vec{G} ! \vec{d}$ $\vec{S} \forall \vec{H} ! \vec{C}$

  Code is *generated* by G and in *null-space* of H
  - (n,k) code: Data space 2^k, Code space 2^n
  - (n,k,d) code: specify distance d as well
- Random code:
  - Need to both identify errors *and* correct them
  - Distance d correct ½(d-1) errors
- Erasure code:
  - Can correct errors if we know which bits/symbols are bad
  - Example: RAID codes, where "symbols" are blocks of disk
  - Distance d correct (d-1) errors
- Error detection code:
  - Distance d detect (d-1) errors
- Hamming Codes
  - d = 3 Columns nonzero, Distinct
  - d = 4 Columns nonzero, Distinct, Odd-weight
- Binary Golay code: based on quadratic residues mod 23
  - Binary code: [24, 12, 8] and [23, 12, 7].
  - Often used in space-based schemes, can correct 3 errors

Hamming Bound, symbols in GF(2)

- Consider an (n,k) code with distance d
  - How do n, k, and d relate to one another?
- First question: How big are spheres?
  - For distance d, spheres are of radius ½ (d-1),
    - » i.e. all error with weight ½ (d-1) or less must fit within sphere
  - Thus, size of sphere is at least:
    1 + Num(1-bit err) + Num(2-bit err) + …+ Num(½(d-1) – bit err)

$$Size = \sum_{e=0}^{\frac{1}{2}(d-1)} \binom{n}{e}$$

- Hamming bound reflects bin-packing of spheres:
  - need 2^k of these spheres within code space

$$2^k \left(\sum_{e=0}^{\frac{1}{2}(d-1)} \binom{n}{e}\right) \le 2^n \qquad 2^k (1 + n) \le 2^n, d = 3$$

How to *Generate* code words?

- Consider a *linear* code. Need a *Generator* Matrix.
  - Let v_i be the data value (k bits), C_i be resulting code (n bits):

$$\vec{C}_i = \mathbf{G}\vec{v}_i$$ ← G must be an n/ k matrix

- Are there 2^k unique code values?
  - Only if the k columns of G are linearly independent!
- Of course, need some way of decoding as well.

$$\vec{v}_i = f_d(\vec{C}_i),$$

  - Is this linear??? Why or why not?
- A code is *systematic* if the data is directly encoded within the code words.
  - Means Generator has form:
  - Can always turn non-systematic code into a systematic one (row ops)

$$\mathbf{G} = \begin{pmatrix} \mathbf{I} \\ \mathbf{P} \end{pmatrix}$$

- But – What is distance of code? Not Obvious!

Implicitly Defining Codes by Check Matrix

- Consider a parity-check matrix H (n/ [n-k])
  - Define valid code words Ci as those that give S_i=0 (null space of H)

$$\vec{S}_i = \mathbf{H}\vec{C}_i = \vec{0}$$

  - Size of null space?
    (null-rank H)=k if (n-k) linearly independent columns in H
- Suppose we transmit code word C with error:
  - Model this as vector E which flips selected bits of C to get R (received):

$$\vec{R} = \vec{C} \oplus \vec{E}$$

  - Consider what happens when we multiply by H:

$$S = \mathbf{H}R = \mathbf{H}(C \oplus E) = \mathbf{H}E$$

- What is distance of code?
  - Code has distance d if no sum of d-1 or less columns yields 0
  - I.e. No error vectors, E, of weight < d have zero syndromes
  - So – Code design is designing H matrix

How to relate G and H (Binary Codes)

- Defining H makes it easy to understand distance of code, but hard to generate code (H defines code implicitly!)
- However, let H be of following form:

$$\mathbf{H} = [\mathbf{P} \mid \mathbf{I}].$$ ← P is (n-k)/ k, I is (n-k)/ (n-k)
Result: H is (n-k)/ n

- Then, G can be of following form (maximal code size):

$$\mathbf{G} = \begin{pmatrix} \mathbf{I} \\ \mathbf{P} \end{pmatrix}$$ ← P is (n-k)/ k, I is k/ k
Result: G is n/ k

- Notice: G generates values in null-space of H *and* has k independent columns so generates 2^k unique values:

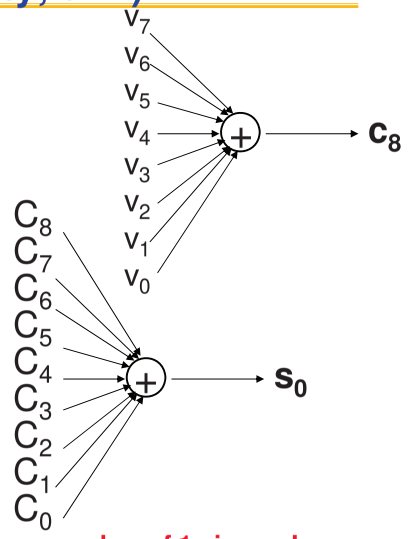
$$\vec{S}_i = \mathbf{H}\mathbf{G}\vec{v}_i, = [\mathbf{P} \mid \mathbf{I}]\begin{pmatrix} \mathbf{I} \\ \mathbf{P} \end{pmatrix}\vec{v}_i = 1 \oplus 0$$

Simple example (Parity, d=2)

- Parity code (8-bits):

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$H = 111111111!$$



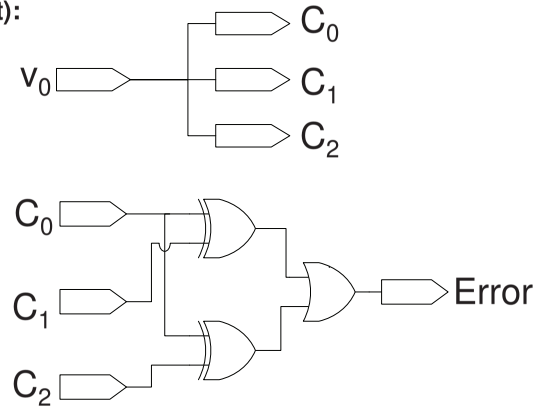
- Note: Complexity of logic depends on number of 1s in row!

Simple example: Repetition (voting, d=3)

- Repetition code (1-bit):

$$G = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

$$H = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$



- Positives: simple
- Negatives:
  - Expensive: only 33% of code word is data
  - Not packed in Hamming-bound sense (only D=3). Could get much more efficient coding by encoding multiple bits at a time

Example: Hamming Code (d=3)

- Binary Hamming code meets Hamming bound
- Recall bound for d=3:

$$2^k \le (1, n) * 2^n + n * 2^{n) k}) 1$$

- So, rearranging:

$$k * 2^c) (c, 1), c \forall n) k$$

- Thus, for:
  - c=2 check bits, k 1 (Repetition code)
  - c=3 check bits, k 4
  - c=4 check bits, k 11, use k=8?
  - c=5 check bits, k 26, use k=16?
  - c=6 check bits, k 57, use k=32?
  - c=7 check bits, k 120, use k=64?
- H matrix consists of all unique, non-zero vectors
  - There are 2^c-1 vectors, c used for parity, so remaining 2^c-c-1

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

$$H = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Example, d=4 code (SEC-DED)

- Design H with:
  - All columns non-zero, odd-weight, distinct
    - » Note that odd-weight refers to *Hamming Weight,* i.e. number of zeros
- Why does this generate d=4?
  - Any single bit error will generate a distinct, non-zero value
  - Any double error will generate a distinct, non-zero value
    - » Why? Add together two distinct columns, get distinct result
  - Any triple error will generate a non-zero value
    - » Why? Add together three odd-weight values, get an odd-weight value
  - So: need four errors before indistinguishable from code word
- Because d=4:
  - Can correct 1 error (*Single Error Correction, i.e.* SEC)
  - Can detect 2 errors (*Double Error Detection, i.e.* DED)
- Example:
  - Note: log size of nullspace will be (columns – rank) = 4, so:
    - » Rank = 4, since rows independent, 4 cols indpt
    - » Clearly, 8 bits in code word
    - » Thus: (8,4) code

$$\begin{pmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix} \forall \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \\ C_7 \end{pmatrix}$$

Tweeks:

- No reason cannot make code shorter than required
- Suppose n-k=8 bits of parity. What is max code size (n) for d=4?
  - Maximum number of unique, odd-weight columns: $2^7 = 128$
  - So, n = 128. But, then k = n – (n – k) = 120. Weird!
  - Just throw out columns of high weight and make (72, 64) code!
- Circuit optimization: if throwing out column vectors, pick ones of highest weight (# bits=1) to simplify circuit
- But – shortened codes like this might have d > 4 in some special directions
  - Example: Kaneda paper, catches failures of groups of 4 bits
  - Good for catching chip failures when DRAM has groups of 4 bits
- What about EVENODD code?
  - Can be used to handle two erasures
  - What about two dead DRAMs? Yes, if you can really know they are dead

Administrivia

- Midterm Results: Almost done. Really!
  - One last problem to grade
- "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design,"
  - Author: Todd M. Austin
  - Use of Checker stage placed after primary computational stage
  - General addition of *dynamic* checking to OOO pipeline
- "Transient Fault Detection via Simultaneous Multithreading,"
  - Authors: Steven K. Reinhardt and Subhendu S. Mukherjee
  - Paired threads duplicating computation to catch transient errors

How to correct errors?

- Consider a parity-check matrix H (n [n-k])
  - Compute the following *syndrome* S_i given code element C_i:

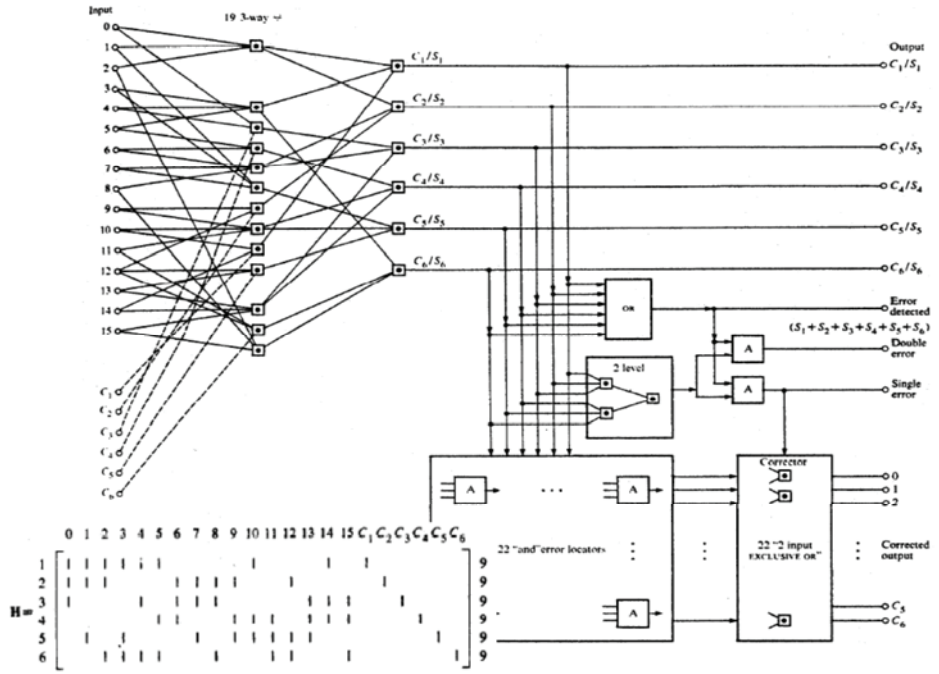
$$\overline{S}_i \# \mathbf{H} \overline{\forall C}_i \# \mathbf{H} \overline{\forall E}$$

- Suppose that two ***correctable*** error vectors E_1 and E_2 produce same syndrome:

$$\mathbf{H} \overline{\forall E}_1 \# \mathbf{H} \overline{\forall E}_2 \ni \mathbf{H} \overline{\forall E}_1 \& \overline{E}_2 \# 0$$

$$\ni \overline{E}_1 \& \overline{E}_2 \text{ has d or more bits set}$$

- But, since both E_1 and E_2 have ! (d-1)/2 bits set, $E_1 + E_2$! d-1 bits set so this conclusion cannot be true!
- So, syndrome is unique indicator of correctable error vectors



Galois Field

- Definition: *Field:* a complete group of elements with:
  - Addition, subtraction, multiplication, division
  - Completely *closed* under these operations
  - Every element has an additive inverse
  - Every element except zero has a multiplicative inverse
- Examples:
  - Real numbers
  - Binary, called GF(2) Galois Field with base 2
    - » Values 0, 1. Addition/subtraction: use xor. Multiplicative inverse of 1 is 1
  - Prime field, GF(p) Galois Field with base p
    - » Values 0 … p-1
    - » Addition/subtraction/multiplication: modulo p
    - » Multiplicative Inverse: every value except 0 has inverse
    - » Example: GF(5): 1! 1 ∀1 mod 5, 2! 3 ∀1mod 5, 4! 4 ∀1 mod 5
  - General Galois Field: GF(p^m) base p (prime!), dimension m
    - » Values are vectors of elements of GF(p) of dimension m
    - » Add/subtract: vector addition/subtraction
    - » Multiply/divide: more complex
    - » Just like real numbers but finite!
    - » Common for computer algorithms: GF(2^m)

Specific Example: Galois Fields GF(2^n)

- Consider *polynomials* whose coefficients come from GF(2).
- Each term of the form x^n is either present or absent.
- Examples: $0, 1, x, x^2$, and $x^7 + x^6 + 1$

$$= 1 \cdot x^7 + 1 \cdot x^6 + 0 \cdot x^5 + 0 \cdot x^4 + 0 \cdot x^3 + 0 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0$$

- With addition and multiplication these form a "ring" (not quite a field – still missing division):
- "Add": XOR each element individually with no carry:

$$\begin{array}{r} x^4 + x^3 + \quad + x + 1 \\ \hline + \quad x^4 + \quad + x^2 + x \\ \hline x^3 + x^2 \quad + 1 \end{array}$$

- "Multiply": multiplying by x is like shifting to the left.

$$\begin{array}{r|l} & x^2 + x + 1 \\ \hline ! & x + 1 \\ \hline & x^2 + x + 1 \\ x^3 + x^2 + x & \\ x^3 \quad + 1 & \end{array}$$

So what about division (mod)

$\frac{x^4 + x^2}{x} = x^3 + x$ with remainder 0

$\frac{x^4 + x^2 + 1}{x + 1} = x^3 + x^2$ with remainder 1

$$\begin{array}{r|l} & x^3 + x^2 + 0x + 0 \\ \hline x + 1 & x^4 + 0x^3 + x^2 + 0x + 1 \\ & x^4 + x^3 \\ \hline & x^3 + x^2 \\ & x^3 + x^2 \\ \hline & 0x^2 + 0x \\ & 0x + 1 \end{array}$$

Remainder 1

Producing Galois Fields

- These polynomials form a Galois (finite) field if we take the results of this multiplication modulo a prime polynomial p(x)
  - A prime polynomial cannot be written as product of two non-trivial polynomials q(x)r(x)
  - For any degree, there exists at least one prime polynomial.
  - With it we can form GF(2^n)
- Every Galois field has a primitive element, #, such that all non-zero elements of the field can be expressed as a power of #
  - Certain choices of $p(x)$ make the simple polynomial x the primitive element. These polynomials are called *primitive*
- For example, $x^4 + x + 1$ is primitive. So # = x is a primitive element and successive powers of # will generate all non-zero elements of GF(16).
- *Example on next slide.*

Galois Fields with primitive $x^4 + x + 1$

$0 = 1$
$1 = x$
$2 = x^2$
$3 = x^3$
$4 = x + 1$
$5 = x^2 + x$
$6 = x^3 + x^2$
$7 = x^3 + x + 1$
$8 = x^2 + 1$
$9 = x^3 + x$
$10 = x^2 + x + 1$
$11 = x^3 + x^2 + x$
$12 = x^3 + x^2 + x + 1$
$13 = x^3 + x^2 + 1$
$14 = x^3 + 1$
$15 = 1$

$4 = x^4 \bmod x^4 + x + 1$
$= x^4 \text{ xor } x^4 + x + 1$
$= x + 1$

- Primitive element = x in GF(2^n)
- In general finding primitive polynomials is difficult. Most people just look them up in a table, such as:

Primitive Polynomials

$x^2 + x + 1$	$x^{12} + x^6 + x^4 + x + 1$	$x^{22} + x + 1$
$x^3 + x + 1$	$x^{13} + x^4 + x^3 + x + 1$	$x^{23} + x^5 + 1$
$x^4 + x + 1$	$x^{14} + x^{10} + x^6 + x + 1$	$x^{24} + x^7 + x^2 + x + 1$
$x^5 + x^2 + 1$	$x^{15} + x + 1$	$x^{25} + x^3 + 1$
$x^6 + x + 1$	$x^{16} + x^{12} + x^3 + x + 1$	$x^{26} + x^6 + x^2 + x + 1$
$x^7 + x^3 + 1$	$x^{17} + x^3 + 1$	$x^{27} + x^5 + x^2 + x + 1$
$x^8 + x^4 + x^3 + x^2 + 1$	$x^{18} + x^7 + 1$	$x^{28} + x^3 + 1$
$x^9 + x^4 + 1$	$x^{19} + x^5 + x^2 + x + 1$	$x^{29} + x + 1$
$x^{10} + x^3 + 1$	$x^{20} + x^3 + 1$	$x^{30} + x^6 + x^4 + x + 1$
$x^{11} + x^2 + 1$	$x^{21} + x^2 + 1$	$x^{31} + x^3 + 1$
		$x^{32} + x^7 + x^6 + x^2 + 1$

Galois Field	Hardware
Multiplication by x	shift left
Taking the result mod $p(x)$	XOR-ing with the coefficients of $p(x)$ when the most significant coefficient is 1.
Obtaining all 2^n-1 non-zero elements by evaluating x^k for $k = 1, \ldots, 2^n$-1	Shifting and XOR-ing 2^n-1 times.

Reed-Solomon Codes

- Galois field codes: code words consist of *symbols*
  - Rather than bits
- Reed-Solomon codes:
  - Based on polynomials in GF(2^k) (I.e. k-bit symbols)
  - Data as coefficients, code space as values of polynomial:
  - $P(x)=a_0+a_1x^1+\ldots a_{k-1}x^{k-1}$
  - Coded: P(0),P(1),P(2)….,P(n-1)
  - Can recover polynomial as long as get *any* k of n
- Properties: can choose number of check symbols
  - Reed-Solomon codes are "maximum distance separable" (MDS)
  - Can add d symbols for distance d+1 code
  - Often used in "erasure code" mode: as long as no more than n-k coded symbols erased, can recover data
- Side note: Multiplication by constant in GF(2^k) can be represented by k∀k matrix: a#k
  - Decompose unknown vector into k bits: $x=x_0+2x_1+\ldots+2^{k-1}x_{k-1}$
  - Each column is result of multiplying a by 2^i

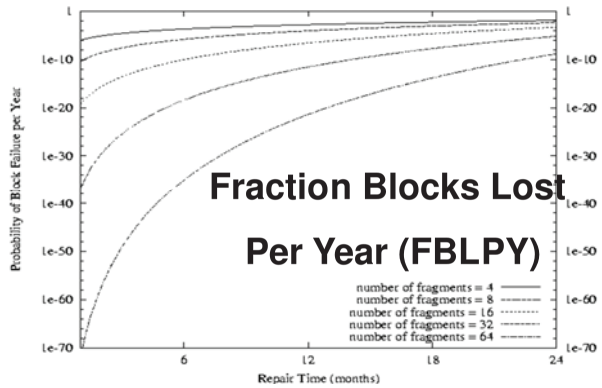
Reed-Solomon Codes (con't)

- Reed-solomon codes (Non-systematic):
  - Data as coefficients, code space as values of polynomial:
  - $P(x)=a_0+a_1x^1+\ldots a_6x^6$
  - Coded: P(0),P(1),P(2)….,P(6)
- Called Vandermonde Matrix: maximum rank
- Different representation (This H' and G not related)
  - Clear that all combinations of two or less columns independent d=3
  - Very easy to pick whatever d you happen to want: add more rows
- Fast, Systematic version of Reed-Solomon:
  - Cauchy Reed-Solomon, others

$$G * \begin{pmatrix} 1^0 & 1^1 & 1^2 & 1^3 & 1^4 \\ 2^0 & 2^1 & 2^2 & 2^3 & 2^4 \\ 3^0 & 3^1 & 3^2 & 3^3 & 3^4 \\ 4^0 & 4^1 & 4^2 & 4^3 & 4^4 \\ 5^0 & 5^1 & 5^2 & 5^3 & 5^4 \\ 6^0 & 6^1 & 6^2 & 6^3 & 6^4 \\ 7^0 & 7^1 & 7^2 & 7^3 & 7^4 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix}$$

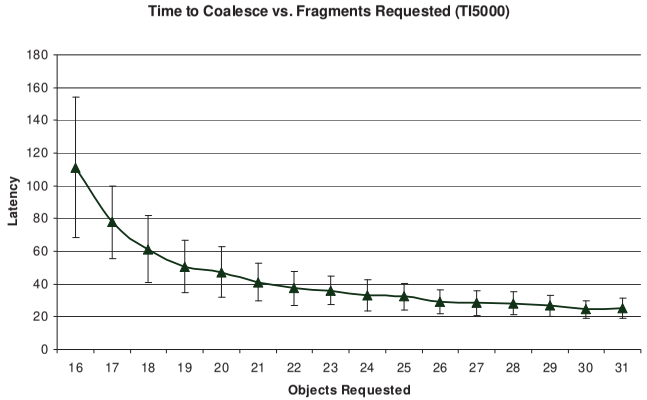
$$H' * \begin{pmatrix} 1^0 & 2^0 & 3^0 & 4^0 & 5^0 & 6^0 & 7^0 \\ 1^1 & 2^1 & 3^1 & 4^1 & 5^1 & 6^1 & 7^1 \end{pmatrix}$$

Aside: Why erasure coding? High Durability/overhead ratio!



- Exploit law of large numbers for durability!
- 6 month repair, FBLPY:
  - Replication: 0.03
  - Fragmentation: 10^{-35}

Statistical Advantage of Fragments



- Latency *and* standard deviation reduced:
  - Memory-less latency model
  - Rate ½ code with 32 total fragments

Conclusion

- ECC: add redundancy to correct for errors
  - (n,k,d) n code bits, k data bits, distance d
  - Linear codes: code vectors computed by linear transformation
- Erasure code: after identifying "erasures", can correct
- Reed-Solomon codes
  - Based on GF(p^n), often GF(2^n)
  - Easy to get distance d+1 code with d extra symbols
  - Often used in erasure mode