CS 110
Computer Architecture

Dependability and RAID

Instructor:
Sören Schwertfeger

http://shtech.org/courses/ca/

School of Information Science and Technology SIST

ShanghaiTech University

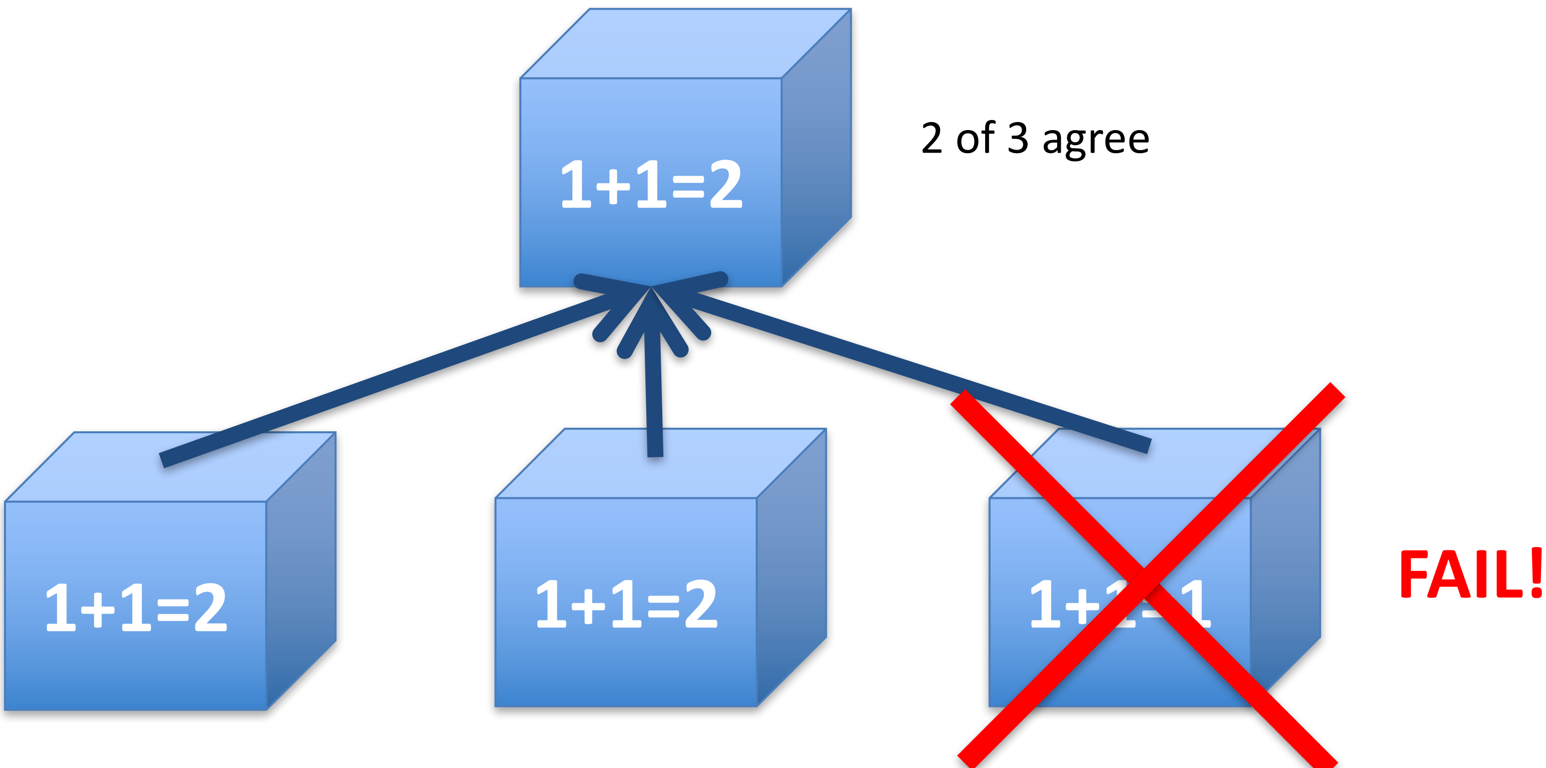
Slides based on UC Berkley's CS61C

Review Last Lecture

- I/O gives computers their 5 senses
- I/O speed range is 100-million to one
- Polling vs. Interrupts
- DMA to avoid wasting CPU time on data transfers
- Disks for persistent storage, replaced by flash

Great Idea #6: Dependability via Redundancy

- Redundancy so that a failing piece doesn't make the whole system fail

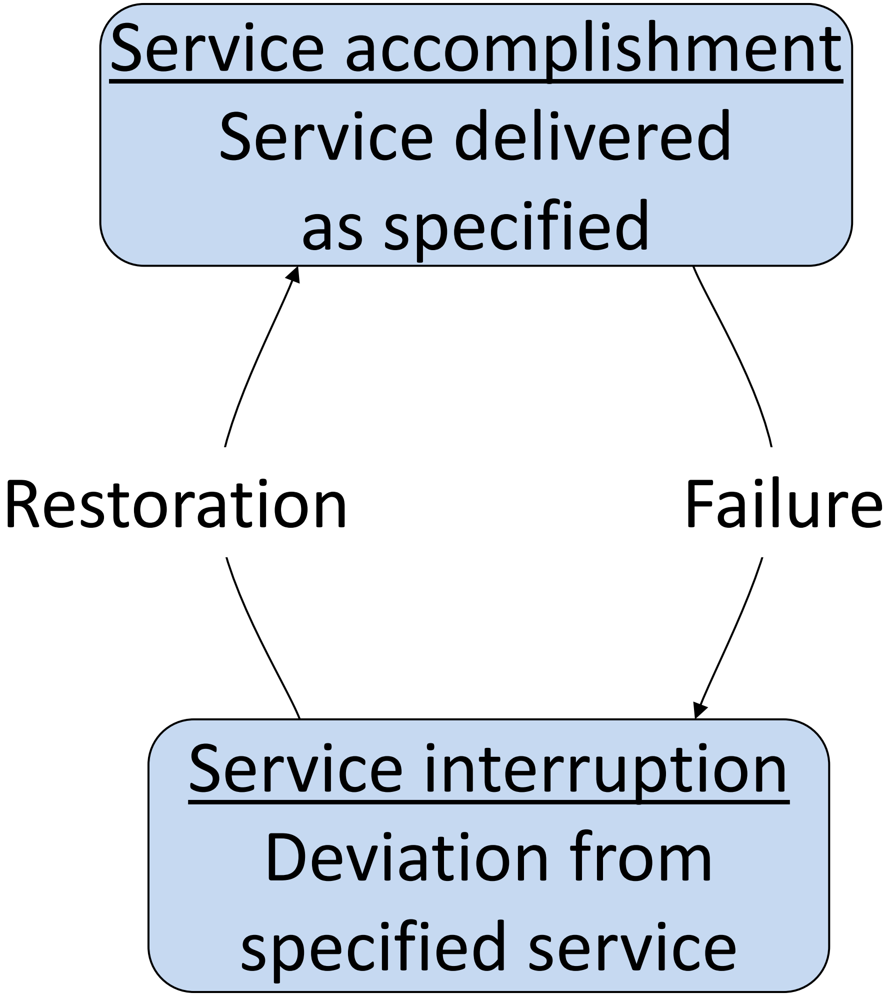


Increasing transistor density reduces the cost of redundancy

Great Idea #6: Dependability via Redundancy

- Applies to everything from datacenters to memory
  - Redundant datacenters so that can lose 1 datacenter but Internet service stays online
  - Redundant routes so can lose nodes but Internet doesn't fail
  - Redundant disks so that can lose 1 disk but not lose data (Redundant Arrays of Independent Disks/RAID)
  - Redundant memory bits of so that can lose 1 bit but no data (Error Correcting Code/ECC Memory)

Dependability



- Fault: failure of a component
  - May or may not lead to system failure

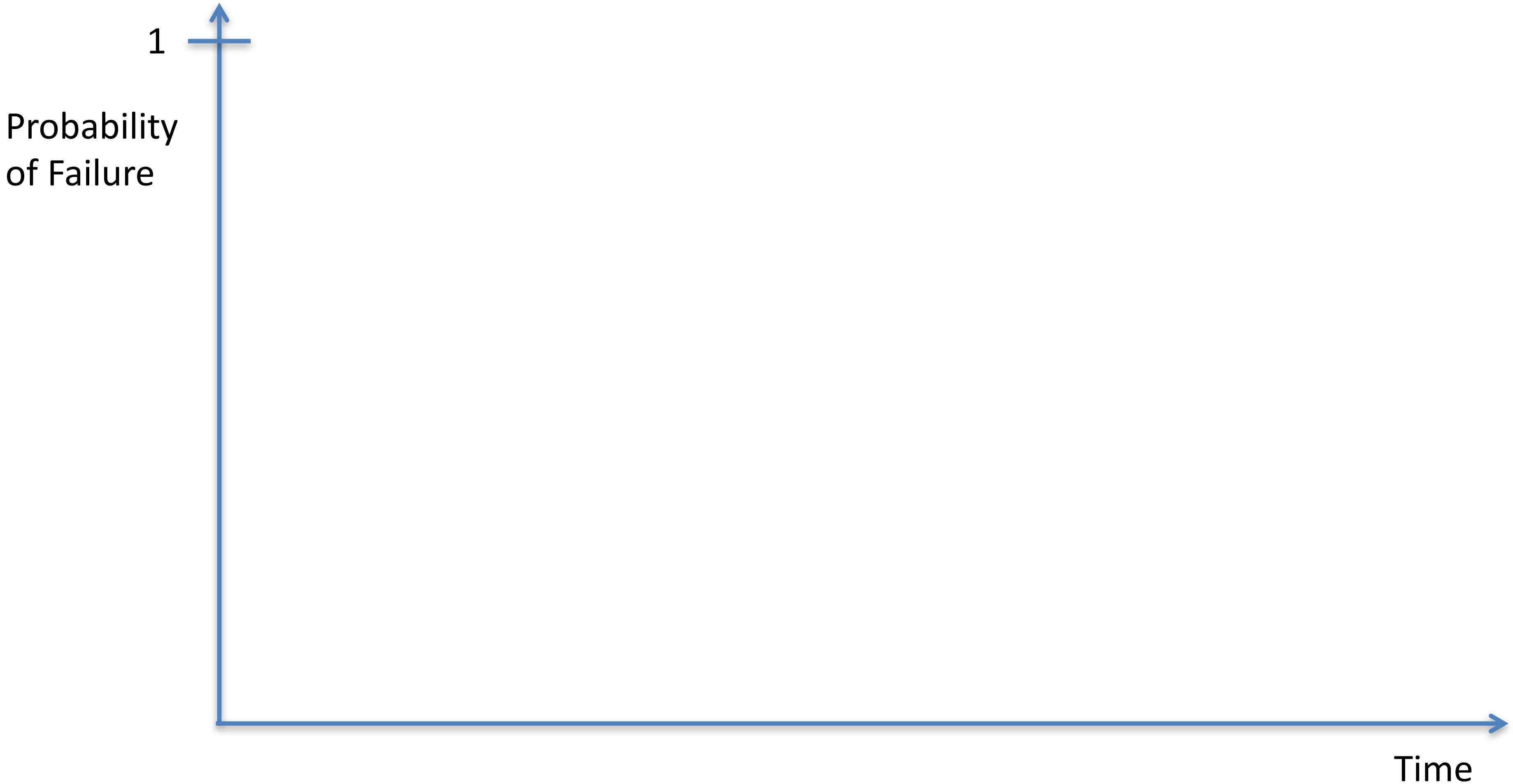
Dependability via Redundancy: Time vs. Space

- *Spatial Redundancy* – replicated data or check information or hardware to handle hard and soft (transient) failures
- *Temporal Redundancy* – redundancy in time (retry) to handle soft (transient) failures

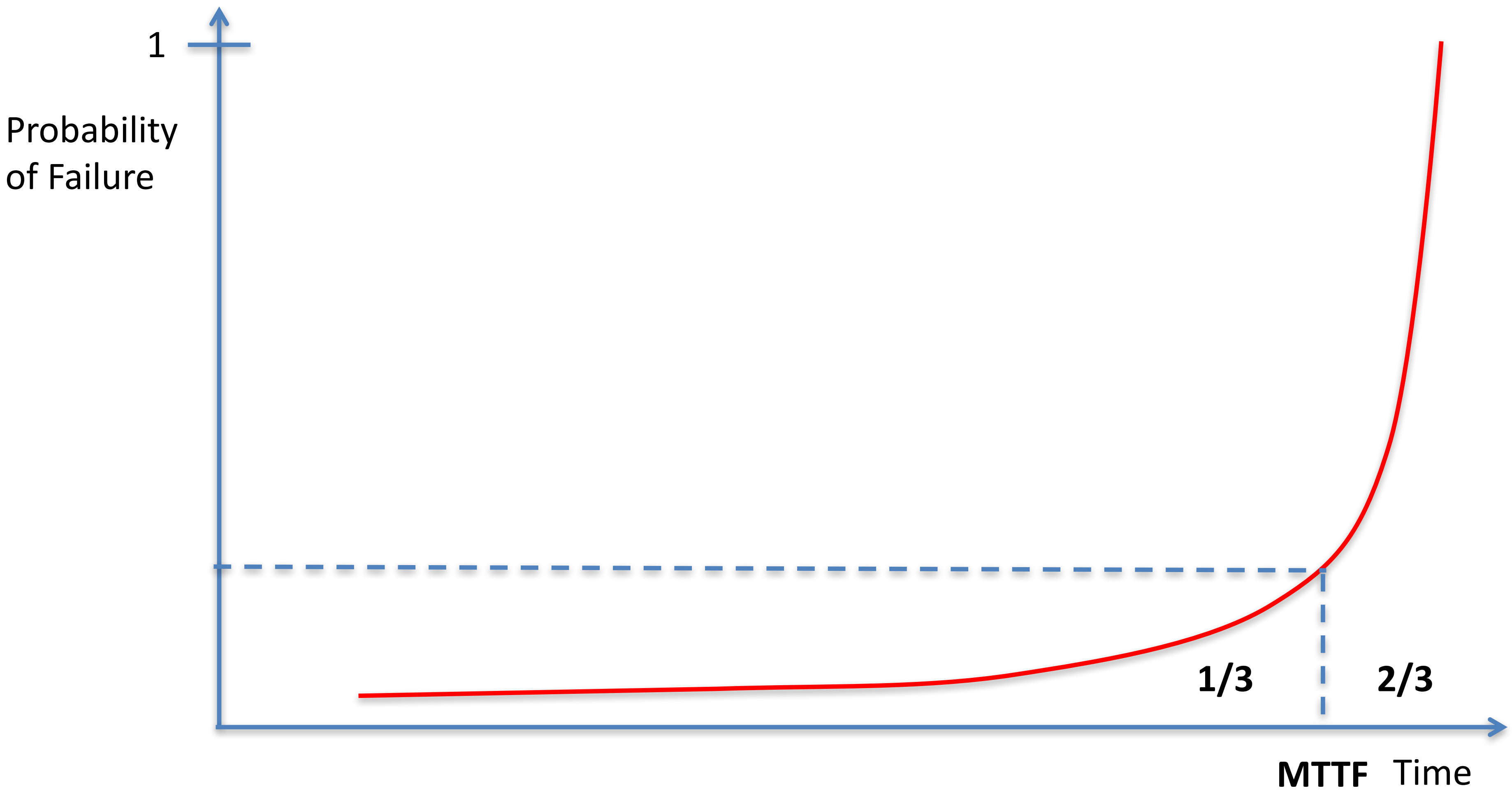
Dependability Measures

- Reliability: Mean Time To Failure (MTTF)
- Service interruption: Mean Time To Repair (MTTR)
- Mean time between failures (MTBF)
  - MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR)
- Improving Availability
  - Increase MTTF: More reliable hardware/software + Fault Tolerance
  - Reduce MTTR: improved tools and processes for diagnosis and repair

Understanding MTTF



Understanding MTTF



Availability Measures

- Availability = MTTF / (MTTF + MTTR) as %
  - MTTF, MTBF usually measured in hours
- Since hope rarely down, shorthand is "number of 9s of availability per year"
- 1 nine: 90% => 36 days of repair/year
- 2 nines: 99% => 3.6 days of repair/year
- 3 nines: 99.9% => 526 minutes of repair/year
- 4 nines: 99.99% => 53 minutes of repair/year
- 5 nines: 99.999% => 5 minutes of repair/year

Reliability Measures

- Another is average number of failures per year: Annualized Failure Rate (AFR)
  - E.g., 1000 disks with 100,000 hour MTTF
  - 365 days * 24 hours = 8760 hours
  - (1000 disks * 8760 hrs/year) / 100,000 = 87.6 failed disks per year on average
  - 87.6/1000 = 8.76% annual failure rate
- Google's 2007 study* found that actual AFRs for individual drives ranged from 1.7% for first year drives to over 8.6% for three-year old drives

research.google**.com/archive/disk_failures.pdf*

Dependability Design Principle

- Design Principle: No single points of failure
  - "Chain is only as strong as its weakest link"
- Dependability Corollary of Amdahl's Law
  - Doesn't matter how dependable you make one portion of system
  - Dependability limited by part you do not improve

Error Detection/ Correction Codes

- Memory systems generate errors (accidentally flipped-bits)
  - DRAMs store very little charge per bit
  - “Soft” errors occur occasionally when cells are struck by alpha particles or other environmental upsets
  - “Hard” errors can occur when chips permanently fail
  - Problem gets worse as memories get denser and larger
- Memories protected against failures with EDC/ECC
- Extra bits are added to each data-word
  - Used to detect and/or correct faults in the memory system
  - Each data word value mapped to unique *code word*
  - A fault changes valid code word to invalid one, which can be detected

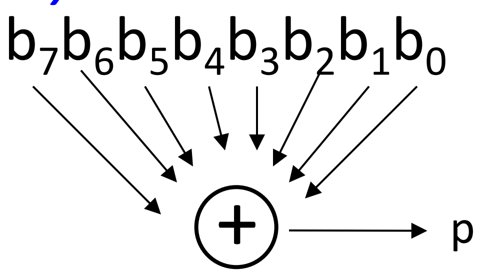
Block Code Principles

- Hamming distance = difference in # of bits
- p = 011011, q = 001111, Ham. distance (p,q) = 2
- p = 011011,
  q = 110001,
  distance (p,q) = ?
- Can think of extra bits as creating a code with the data
- What if minimum distance between members of code is 2 and get a 1-bit error?

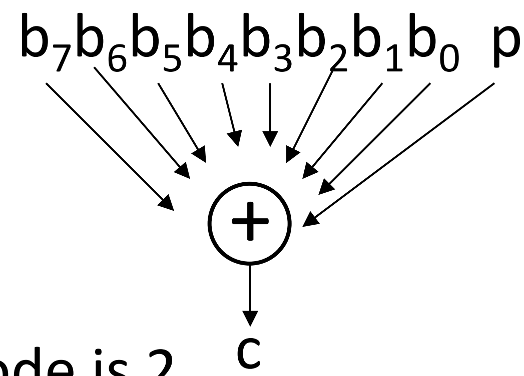
Richard Hamming, 1915-98
Turing Award Winner

Parity: Simple Error-Detection Coding

- Each data value, before it is written to memory is "tagged" with an extra bit to force the stored word to have *even parity*:



- Each word, as it is read from memory is "checked" by finding its parity (including the parity bit).



- Minimum Hamming distance of parity code is 2
- A non-zero parity check indicates an error occurred:
  - 2 errors (on different bits) are not detected
  - nor any even number of errors, just odd numbers of errors are detected

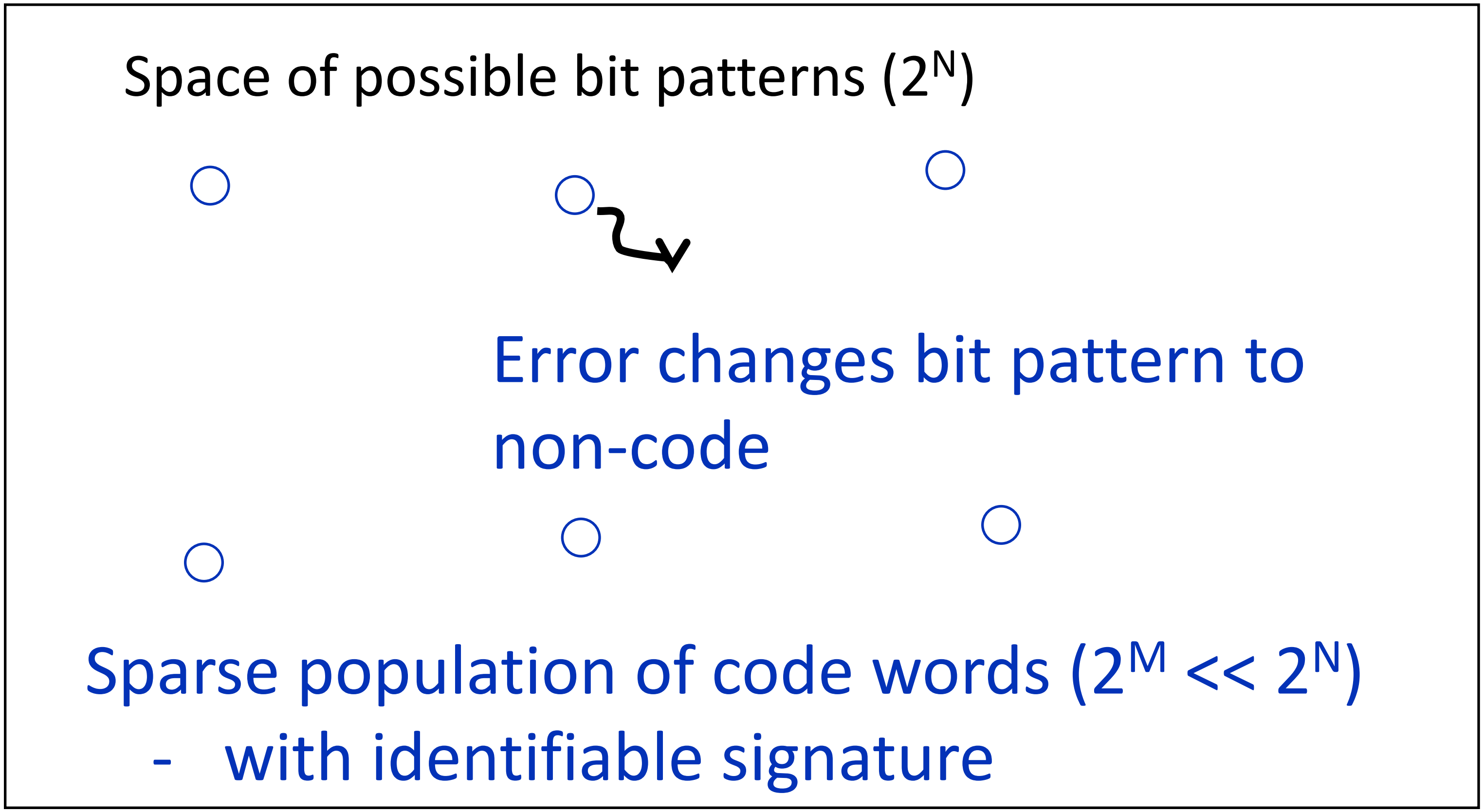
Parity Example

- Data 0101 0101
- 4 ones, even parity now
- Write to memory:
  0101 0101 0
  to keep parity even
- Data 0101 0111
- 5 ones, odd parity now
- Write to memory:
  0101 0111 1
  to make parity even
- Read from memory
  0101 0101 0
- 4 ones => even parity, so no error
- Read from memory
  1101 0101 0
- 5 ones => odd parity, so error
- What if error in parity bit?

Suppose Want to Correct 1 Error?

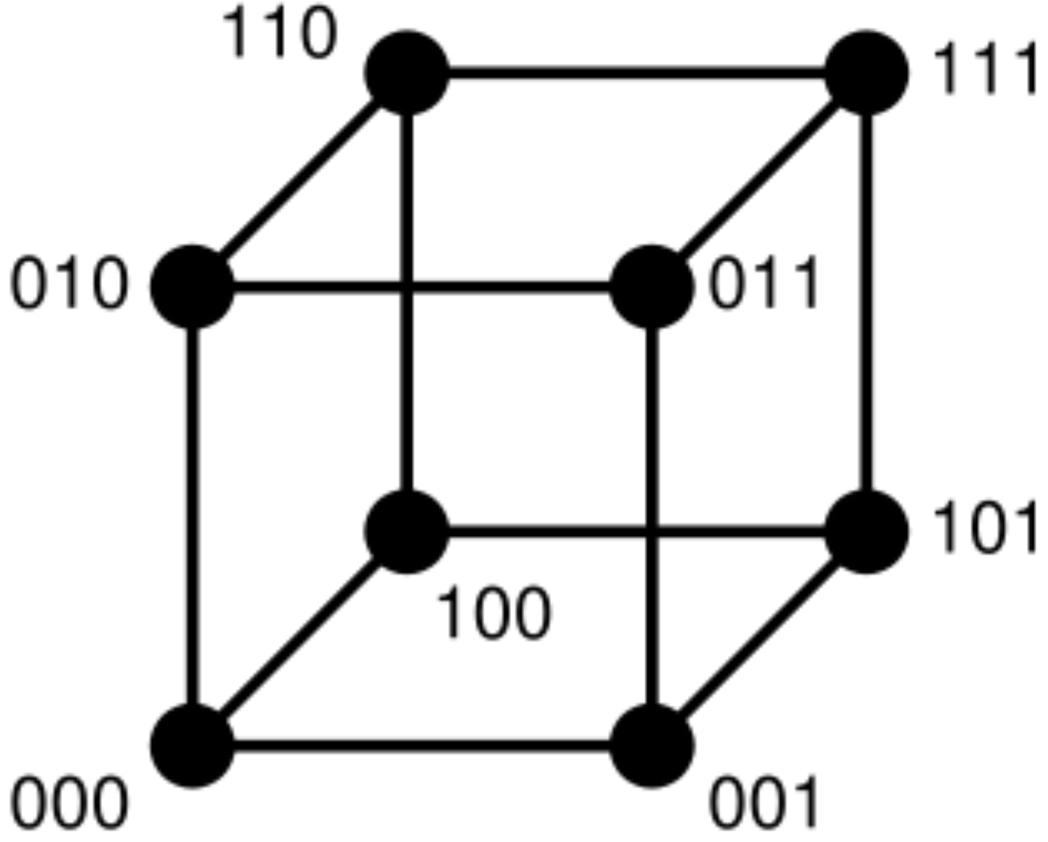
- Richard Hamming came up with simple to understand mapping to allow Error Correction at minimum distance of 3
  - Single error correction, double error detection
- Called “Hamming ECC”
  - Worked weekends on relay computer with unreliable card reader, frustrated with manual restarting
  - Got interested in error correction; published 1950
  - R. W. Hamming, “Error Detecting and Correcting Codes,” *The Bell System Technical Journal*, Vol. XXVI, No 2 (April 1950) pp 147-160.

Detecting/Correcting Code Concept

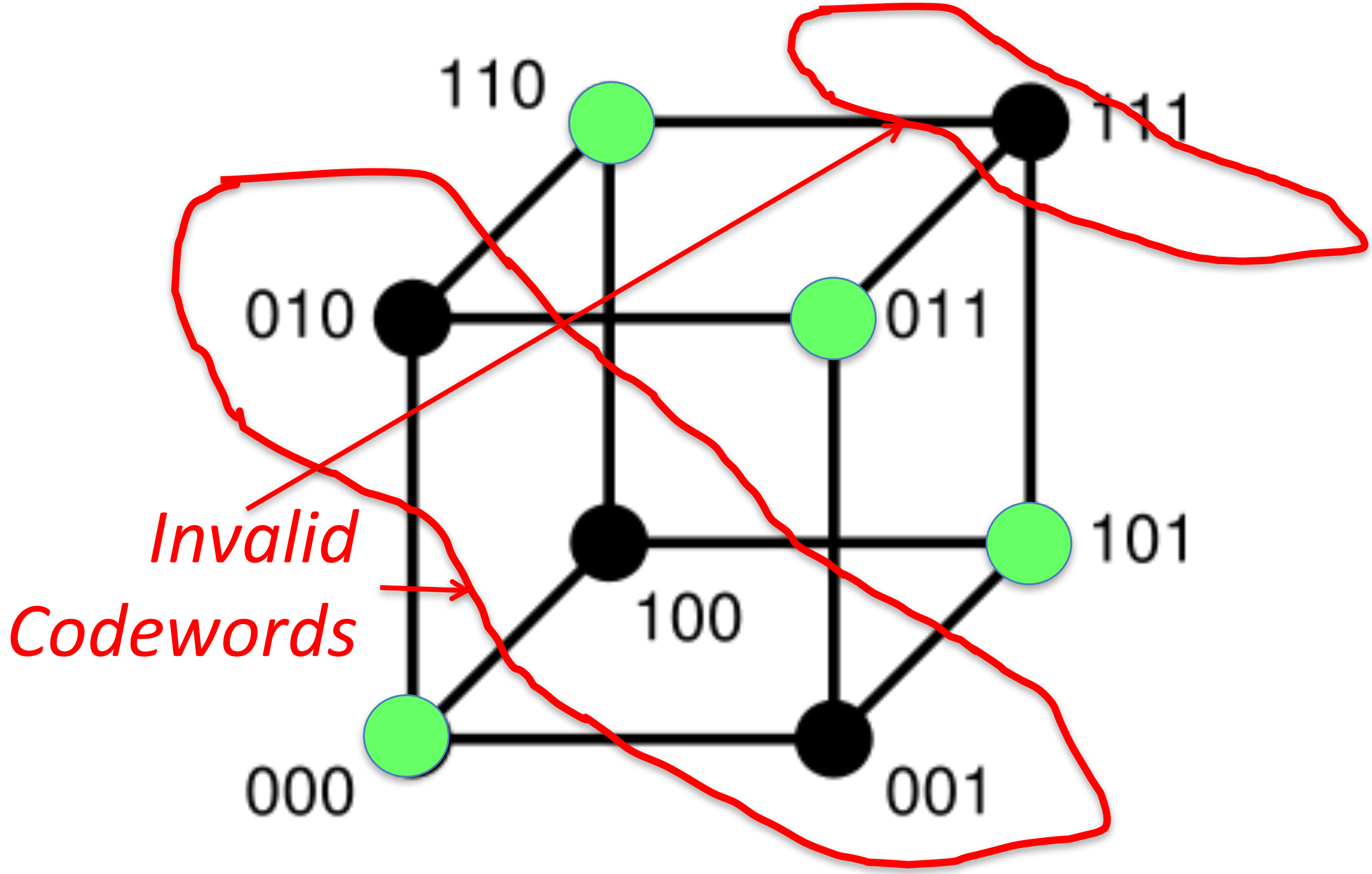


- Detection: bit pattern fails codeword check
- Correction: map to nearest valid code word

Hamming Distance: 8 code words



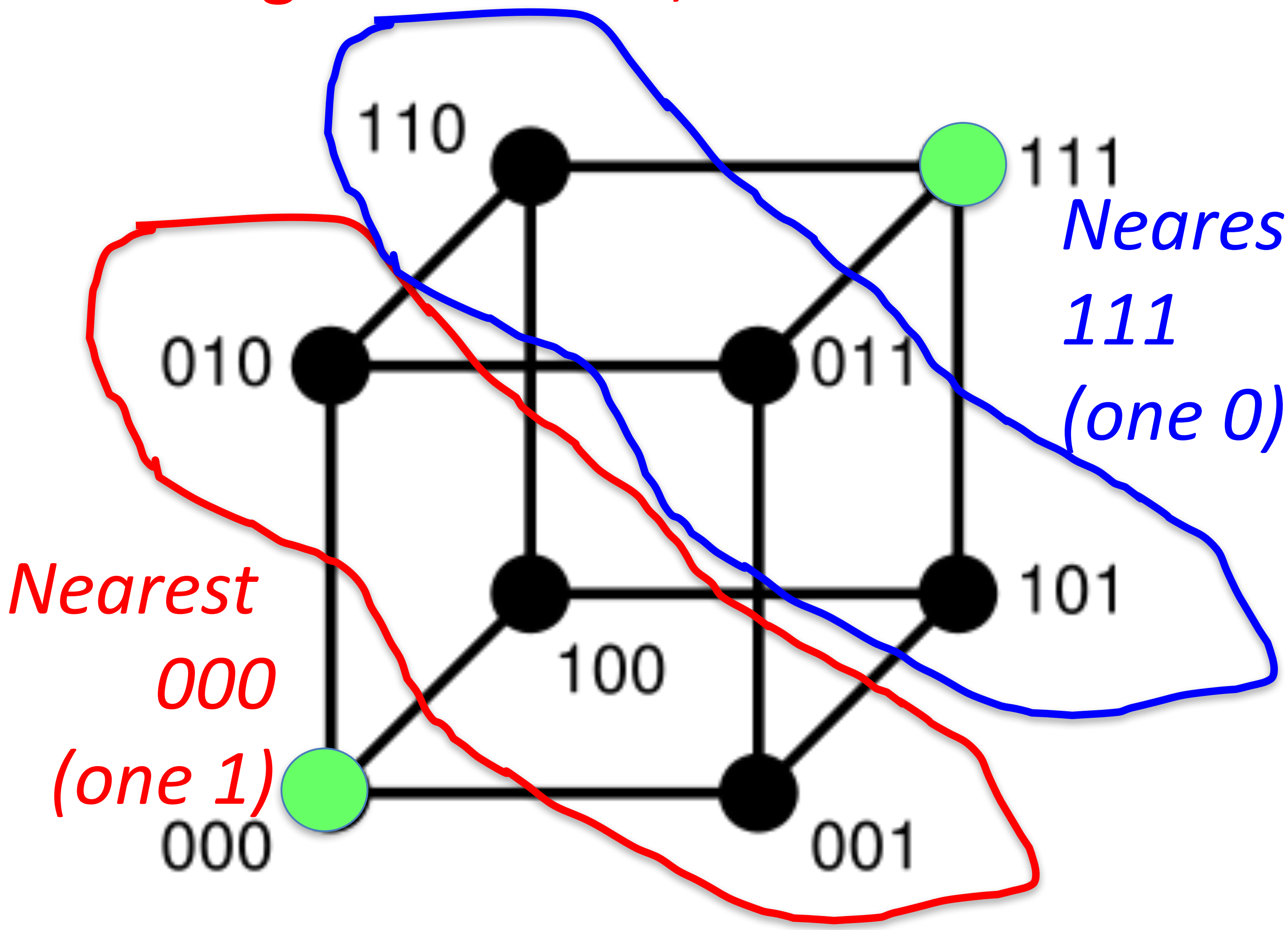
Hamming Distance 2: Detection
Detect Single Bit Errors



- No 1 bit error goes to another valid codeword
- ½ codewords are valid

Hamming Distance 3: Correction

Correct Single Bit Errors, Detect Double Bit Errors



- No 2 bit error goes to another valid codeword; 1 bit error near
- 1/4 codewords are valid

Administrivia

- Final Exam
  - Tuesday, June 26, 2017, 9:00-11:00
  - Location: Teaching Center 301 + 302
  - THREE cheat sheets (MT1, MT2, post-MT2)
    - Hand-written by you, English, A4
- Project 4 published
- HW 7 published

Hamming Error Correction Code

- Use of extra parity bits to allow the position identification of a single error

1. Mark all bit positions that are powers of 2 as parity bits (positions 1, 2, 4, 8, 16, …)
   - Start numbering bits at 1 at left (not at 0 on right)
2. All other bit positions are data bits (positions 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, …)
3. Each data bit is covered by 2 or more parity bits

Hamming ECC

4. The position of parity bit determines sequence of data bits that it checks

- Bit 1 (0001_2): checks bits (1,3,5,7,9,11,...)
  - Bits with least significant bit of address = 1
- Bit 2 (0010_2): checks bits (2,3,6,7,10,11,14,15,…)
  - Bits with 2nd least significant bit of address = 1
- Bit 4 (0100_2): checks bits (4-7, 12-15, 20-23, ...)
  - Bits with 3rd least significant bit of address = 1
- Bit 8 (1000_2): checks bits (8-15, 24-31, 40-47 ,...)
  - Bits with 4th least significant bit of address = 1

Graphic of Hamming Code

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11
Parity bit coverage	p1	X		X		X		X		X		X		X		X
	p2		X	X			X	X			X	X			X	X
	p4				X	X	X	X					X	X	X	X
	p8								X	X	X	X	X	X	X	X

- http://en.wikipedia.org/wiki/Hamming_code

Hamming ECC

5. Set parity bits to create even parity for each group

- A byte of data: 10011010
- Create the coded word, leaving spaces for the parity bits:
- _ _ 1 _ 0 0 1 _ 1 0 1 0

  0 0 0 0 0 0 0 0 0 1 1 1

  1 2 3 4 5 6 7 8 9 0 1 2
- Calculate the parity bits

Hamming ECC

- Position 1 checks bits 1,3,5,7,9,11 (bold):
  ? _ 1 _ 0 0 **1** _ **1** 0 **1** 0. set position 1 to a _:
  _ _ **1** _ **0** 0 **1** _ **1** 0 **1** 0
- Position 2 checks bits 2,3,6,7,10,11 (bold):
  0 **? 1** _ 0 **0 1** _ 1 **0 1** 0. set position 2 to a _:
  0 _ **1** _ 0 **0 1** _ 1 **0 1** 0
- Position 4 checks bits 4,5,6,7,12 (bold):
  0 1 1 **? 0 0 1** _ 1 0 1 **0**. set position 4 to a _:
  0 1 1 _ **0 0 1** _ 1 0 1 **0**
- Position 8 checks bits 8,9,10,11,12:
  0 1 1 1 0 0 1 **? 1 0 1 0**. set position 8 to a _:
  0 1 1 1 0 0 1 _ **1 0 1 0**

Hamming ECC

- Position 1 checks bits 1,3,5,7,9,11:
? _ **1** _ **0** 0 **1** _ **1** 0 **1** 0. set position 1 to a 0:
0 _ **1** _ **0** 0 **1** _ **1** 0 **1** 0
- Position 2 checks bits 2,3,6,7,10,11:
0 **? 1** _ 0 **0 1** _ 1 **0 1** 0. set position 2 to a 1:
0 **1 1** _ 0 **0 1** _ 1 **0 1** 0
- Position 4 checks bits 4,5,6,7,12:
0 1 1 **? 0 0 1** _ 1 0 1 **0**. set position 4 to a 1:
0 1 1 **1 0 0 1** _ 1 0 1 **0**
- Position 8 checks bits 8,9,10,11,12:
0 1 1 1 0 0 1 **? 1 0 1 0**. set position 8 to a 0:
0 1 1 1 0 0 1 **0 1 0 1 0**

Hamming ECC

- Final code word: <u>01</u>1<u>1</u>001<u>0</u>1010
- Data word: 1 001 1010

Hamming ECC Error Check

- Suppose receive
  011100101110

0 1 1 1 0 0 1 0 1 1 1 0

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11
Parity bit coverage	p1	X		X		X		X		X		X		X		X
	p2		X	X			X	X			X	X			X	X
	p4				X	X	X	X					X	X	X	X
	p8								X	X	X	X	X	X	X	X

Hamming ECC Error Check

- Suppose receive
  0 1 1 1 0 0 1 0 1 1 1 0

Hamming ECC Error Check

- Suppose receive
  0 1 1 1 0 0 1 0 1 1 1 0
  0 1 0 1 1 1 √
  1 1 0 1 1 1 X-Parity 2 in error
  1 0 0 1 0 √
  0 1 1 1 0 X-Parity 8 in error
- *Implies position 8+2=10 is in error*
  0 1 1 1 0 0 1 0 1 **1** 1 0

Hamming ECC Error Correct

- Flip the incorrect bit …
  0 1 1 1 0 0 1 0 1 0 1 0

Hamming ECC Error Correct

- Suppose receive

  0 1 1 1 0 0 1 0 1 0 1 0

  0 1 0 1 1 1 √

  1 1 0 1 0 1 √

  1 0 0 1 0 √

  0 1 0 1 0 √

Hamming Error Correcting Code

- Overhead involved in single error-correction code
- Let p be total number of parity bits and d number of data bits in $p + d$ bit word
- If p error correction bits are to point to error bit ($p + d$ cases) + indicate that no error exists (1 case), we need:

  $$2^p >= p + d + 1,$$

  thus $p >= \log(p + d + 1)$

  for large d, p approaches $\log(d)$
- *8 bits data => $d = 8$, $2^p = p + 8 + 1 => p = 4$*
- *16 data => 5 parity,*
  32 data => 6 parity,
  64 data => 7 parity

Hamming Single-Error Correction, Double-Error Detection (SEC/DED)

- Adding extra parity bit covering the entire word provides double error detection as well as single error correction

  1 2 3 4 5 6 7 8

  p_1 p_2 d_1 p_3 d_2 d_3 d_4 p_4

- Hamming parity bits H (p_1 p_2 p_3) are computed (even parity as usual) plus the even parity over the entire word, p_4:

  H=0 p_4=0, no error

  H≠0 p_4=1, correctable single error (odd parity if 1 error => p_4=1)

  H≠0 p_4=0, double error occurred (even parity if 2 errors=> p_4=0)

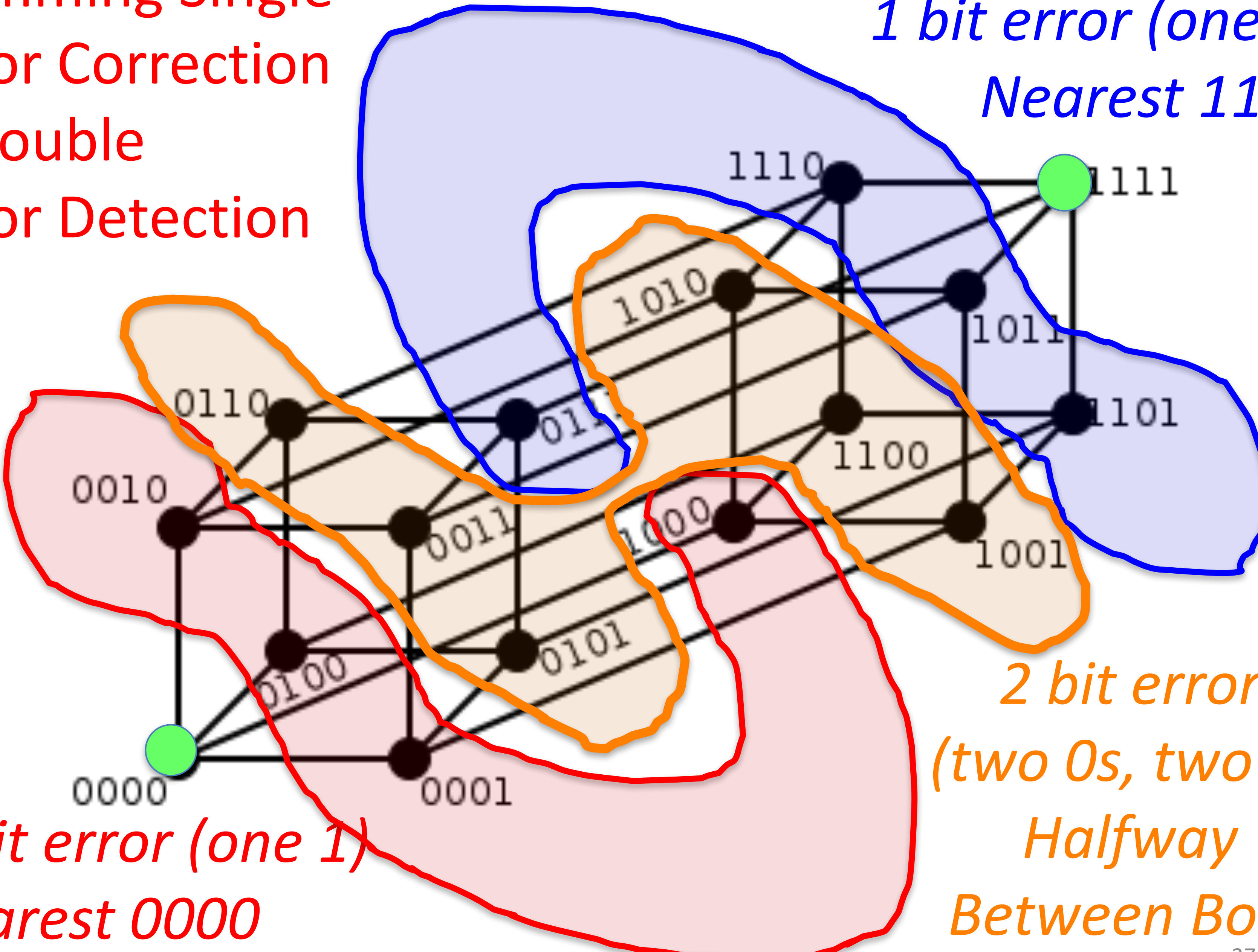
Typical modern codes in DRAM memory systems:

64-bit data blocks (8 bytes) with 72-bit code words (9 bytes).

Hamming Distance = 4

Hamming Single Error Correction + Double Error Detection

1 bit error (one 0) Nearest 1111



1 bit error (one 1) Nearest 0000

2 bit error (two 0s, two 1s) Halfway Between Both

What if More Than 2-Bit Errors?

- Network transmissions, disks, distributed storage common failure mode is bursts of bit errors, not just one or two bit errors
  - Contiguous sequence of *B* bits in which first, last and any number of intermediate bits are in error
  - Caused by impulse noise or by fading in wireless
  - Effect is greater at higher data rates

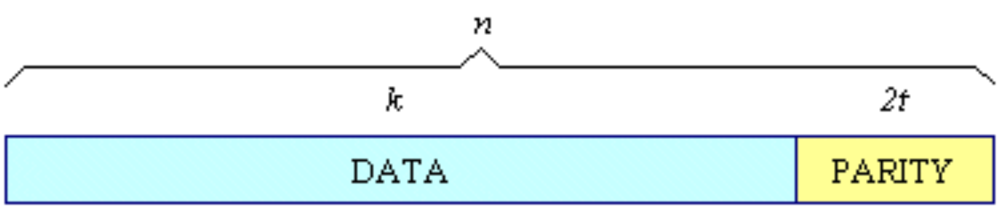
Cyclic Redundancy Check

Simple example: Parity Check Block

Data	10011010		10011010
	01101100		01101100
	11110000		11110000
	00101101		00000000
	11011100		11011100
	00111100		00111100
	11111100		11111100
	00001100		00001100
Check	00111011		00111011
	00000000	0 = Check!	00101101 Not 0 = Fail!

Cyclic Redundancy Check

- Parity codes not powerful enough to detect long runs of errors (also known as *burst errors*)
- Better Alternative: *Reed-Solomon Codes*
  - Used widely in CDs, DVDs, Magnetic Disks
  - RS(255,223) with 8-bit symbols: each codeword contains 255 code word bytes (223 bytes are data and 32 bytes are parity)



  - For this code: n = 255, k = 223, s = 8, 2t = 32, t = 16
  - Decoder can correct any errors in up to 16 bytes anywhere in the codeword

Cyclic Redundancy Check

14 data bits 3 check bits 17 bits total

```
11010011101100 000 <--- input right padded by 3 bits
1011               <--- divisor
01100011101100 000 <--- result
 1011              <--- divisor
00111011101100 000
  1011
00010111101100 000
   1011
00000001101100 000 <--- skip leading zeros
       1011
00000000110100 000
        1011
00000000011000 000
         1011
00000000001110 000
          1011
00000000000101 000
           101 1
-----------------
00000000000000 100 <--- remainder
```

3 bit CRC using the polynomial $x^3 + x + 1$
(divide by 1011 to get remainder)

Cyclic Redundancy Check

- For block of *k* bits, transmitter generates an *n-k* bit frame check sequence
- Transmits *n* bits exactly divisible by some number
- Receiver divides frame by that number
  - If no remainder, assume no error
  - Easy to calculate division for some binary numbers with shift register
- Disks detect *and correct* blocks of 512 bytes with called Reed Solomon codes ≈ CRC

(In More Depth: Code Types)

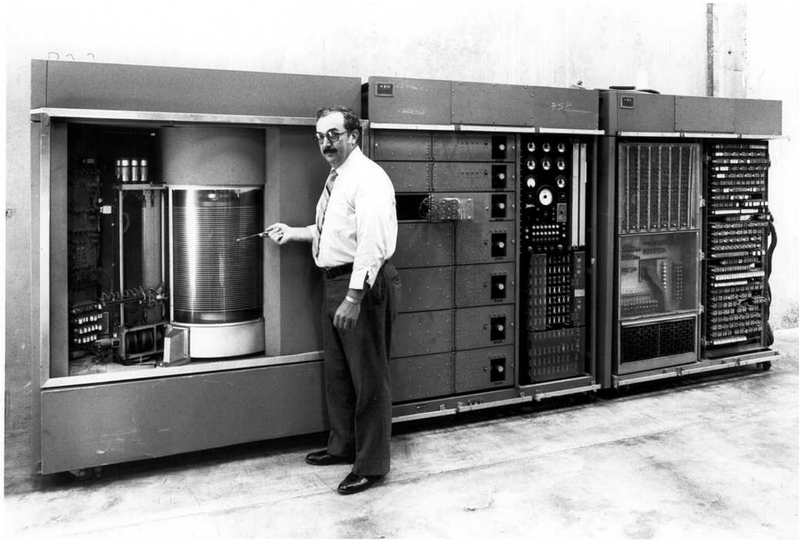
- Linear Codes:
  This image cannot currently be displayed. This image cannot currently be displayed.
  Code is *generated* by G and in *null-space* of H
- Hamming Codes: Design the H matrix
  - d = 3 $\Rightarrow$ Columns nonzero, Distinct
  - d = 4 $\Rightarrow$ Columns nonzero, Distinct, Odd-weight
- 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
  - Alternatively: 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 $k\times k$ matrix: $a\cdot x$
  - 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

Hamming ECC on your own

- Test if these Hamming-code words are correct. If one is incorrect, indicate the correct code word. Also, indicate what the original data was.
- 110101100011
- 111110001100
- 000010001010

Evolution of the Disk Drive

IBM 3390K, 1986



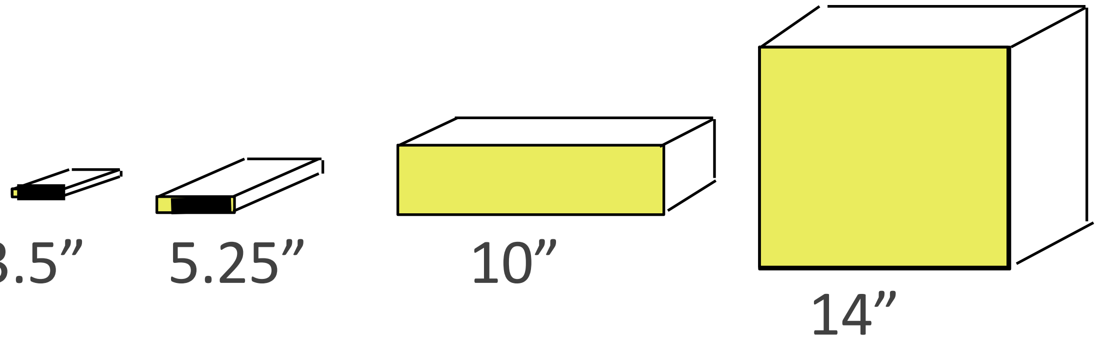
IBM RAMAC 305, 1956

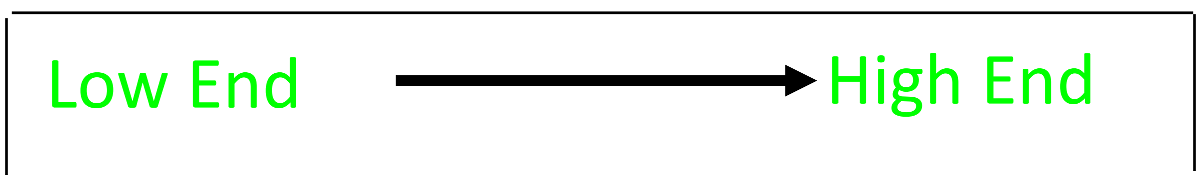
Apple SCSI, 1986

Arrays of Small Disks

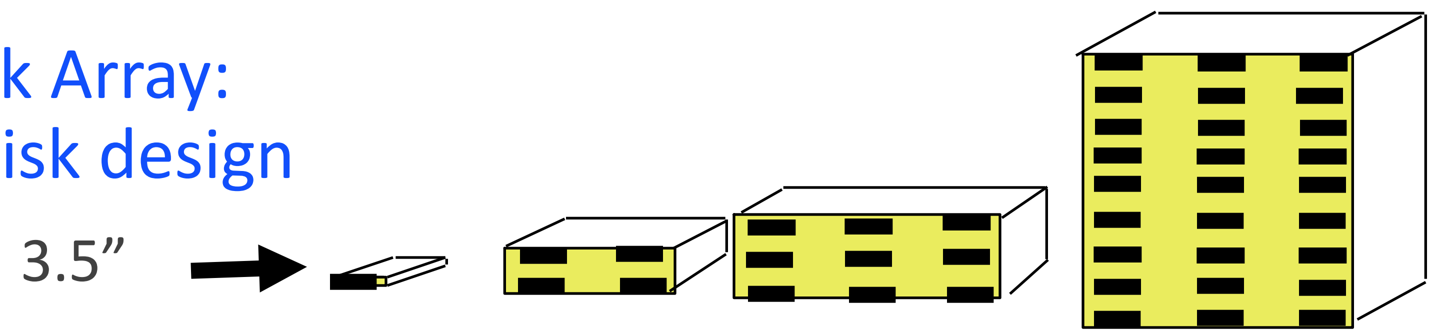
Can smaller disks be used to close gap in performance between disks and CPUs?

Conventional:
4 disk designs





Disk Array:
1 disk design



Replace Small Number of Large Disks with Large Number of Small Disks! (1988 Disks)

	IBM 3390K	IBM 3.5" 0061	x70	
Capacity	20 GBytes	320 MBytes	23 GBytes	
Volume	97 cu. ft.	0.1 cu. ft.	11 cu. ft.	9X
Power	3 KW	11 W	1 KW	3X
Data Rate	15 MB/s	1.5 MB/s	120 MB/s	8X
I/O Rate	600 I/Os/s	55 I/Os/s	3900 IOs/s	6X
MTTF	250 KHrs	50 KHrs	??? Hrs	
Cost	$250K	$2K	$150K	

Disk Arrays have potential for large data and I/O rates, high MB per cu. ft., high MB per KW, but what about reliability?

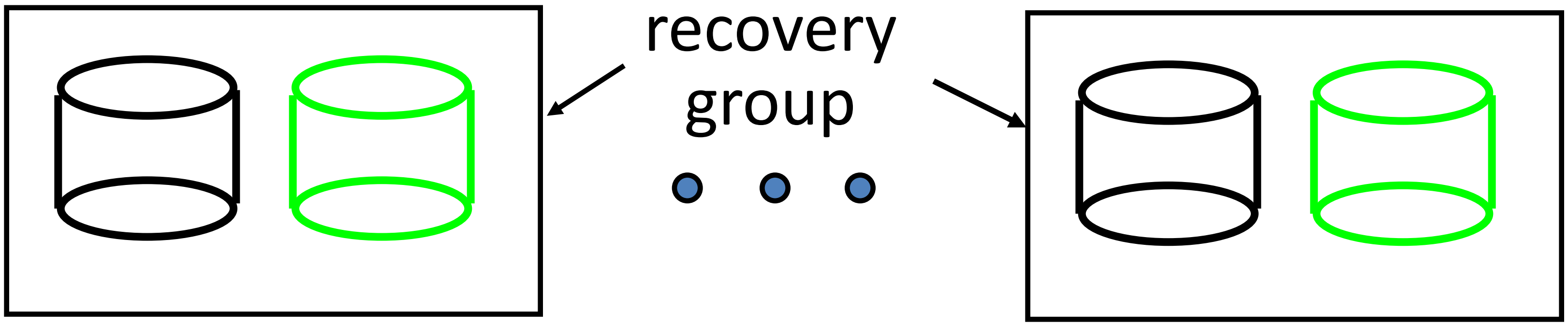
RAID: Redundant Arrays of (Inexpensive) Disks

- Files are "striped" across multiple disks
- Redundancy yields high data availability
  - Availability: service still provided to user, even if some components failed
- Disks will still fail
- Contents reconstructed from data redundantly stored in the array

  => Capacity penalty to store redundant info

  => Bandwidth penalty to update redundant info

Redundant Arrays of Inexpensive Disks RAID 1: Disk Mirroring/Shadowing



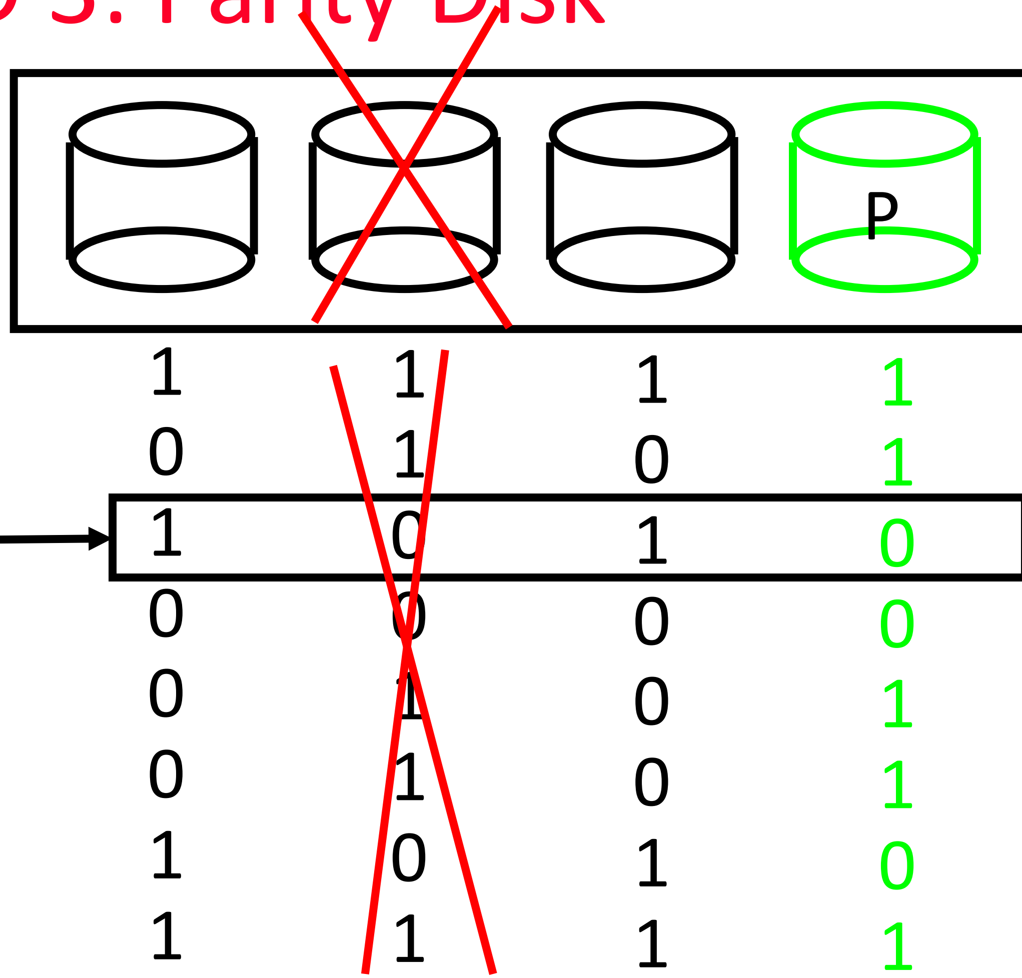
- Each disk is fully duplicated onto its "mirror"
  - Very high availability can be achieved
- Bandwidth sacrifice on write:
  - Logical write = two physical writes
  - Reads may be optimized
- Most expensive solution: 100% capacity overhead

Redundant Array of Inexpensive Disks
RAID 3: Parity Disk

10010011
11001101
10010011
. . .

logical record

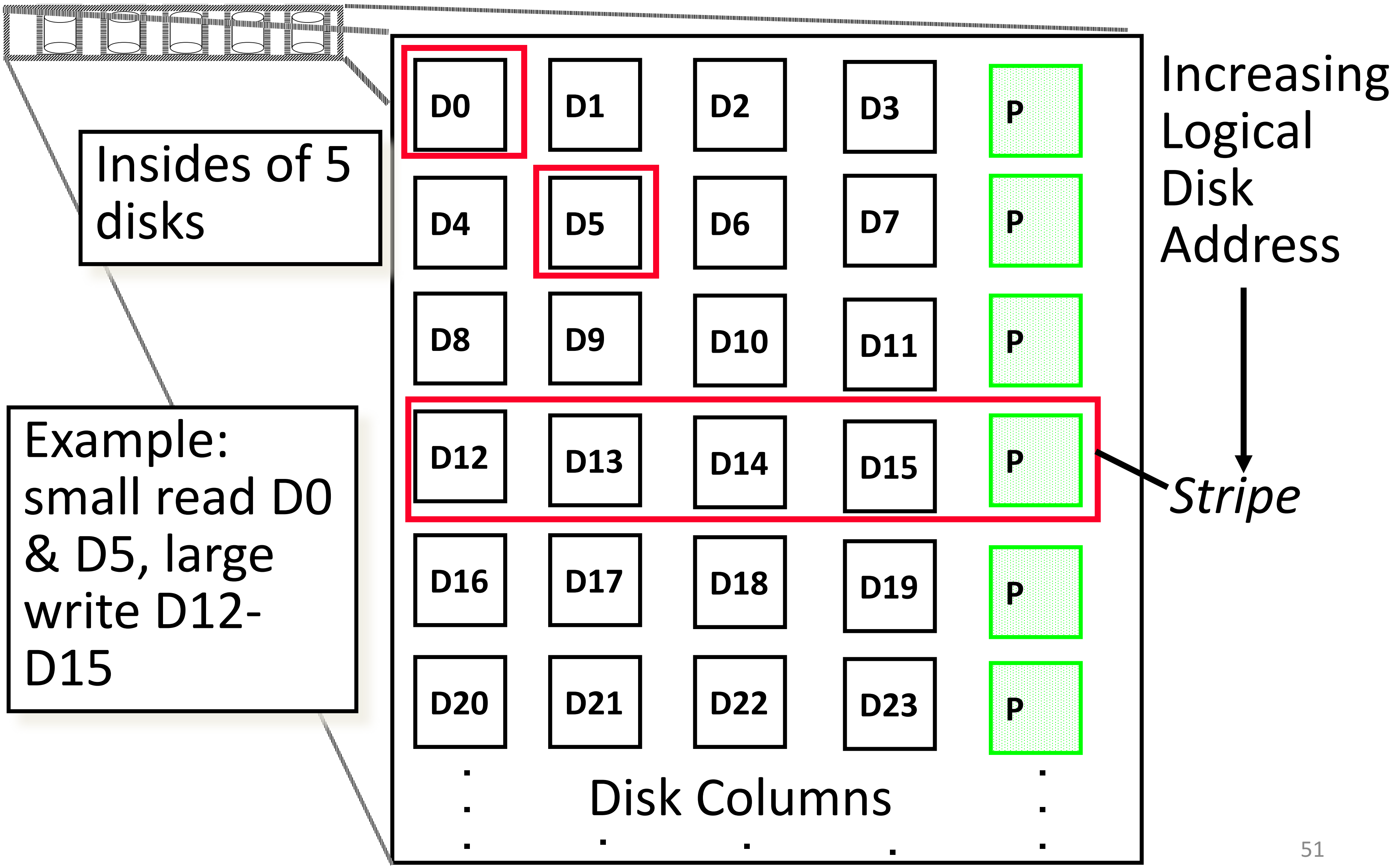
Striped physical records



P contains sum of other disks per stripe mod 2 ("parity")

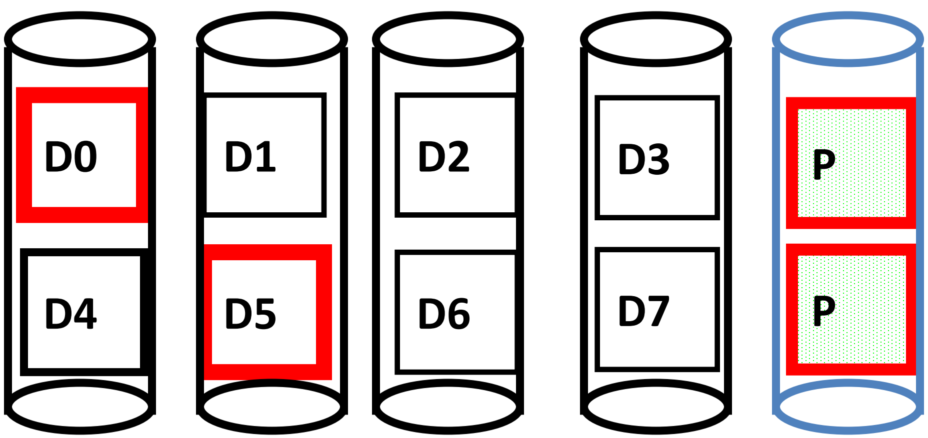
If disk fails, subtract P from sum of other disks to find missing information

Redundant Arrays of Inexpensive Disks
RAID 4: High I/O Rate Parity



Inspiration for RAID 5

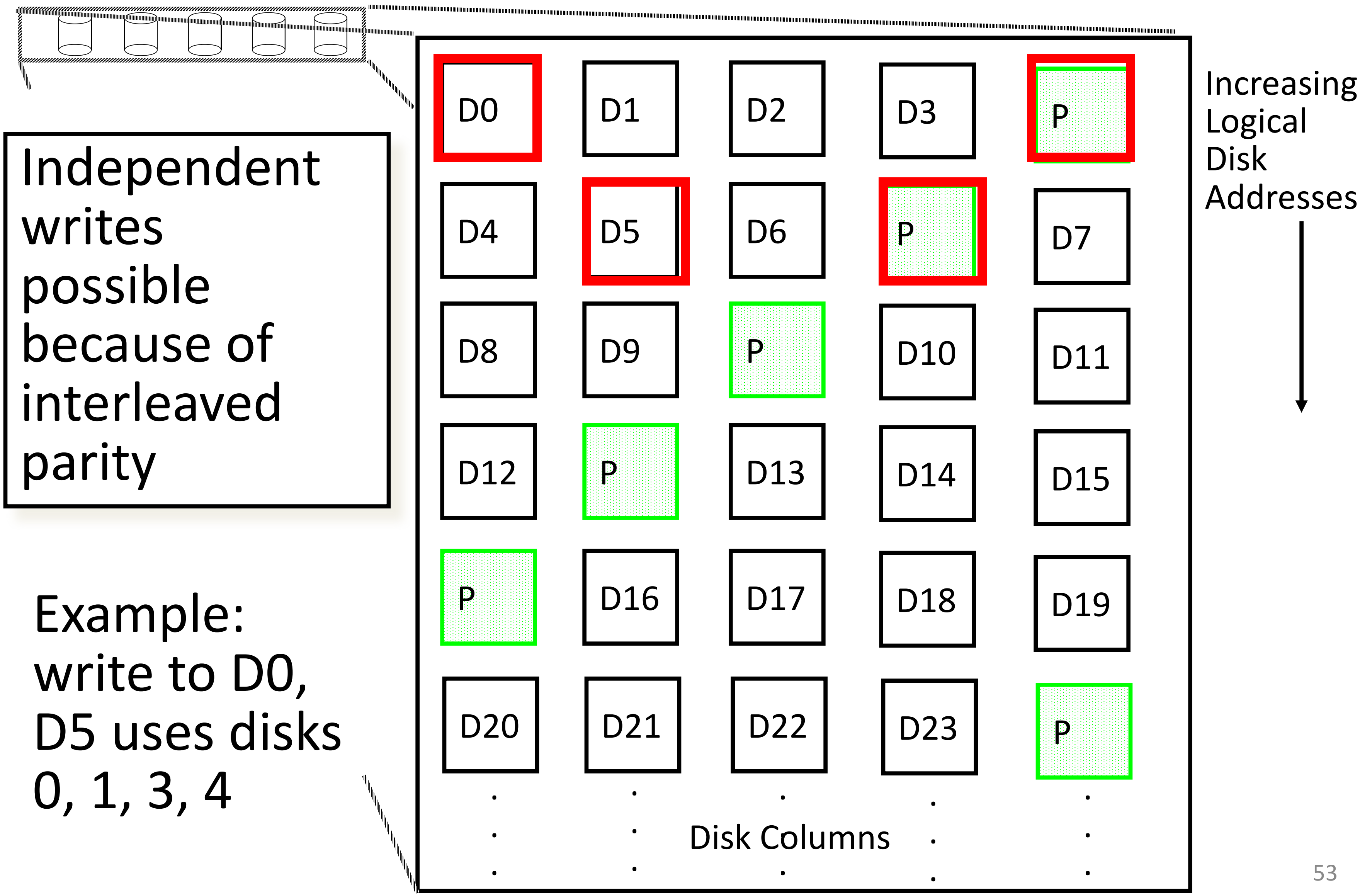
- RAID 4 works well for small reads
- Small writes (write to one disk):
  - Option 1: read other data disks, create new sum and write to Parity Disk
  - Option 2: since P has old sum, compare old data to new data, add the difference to P
- Small writes are limited by Parity Disk: Write to D0, D5 both also write to P disk



RAID 5: High I/O Rate Interleaved Parity

Independent writes possible because of interleaved parity

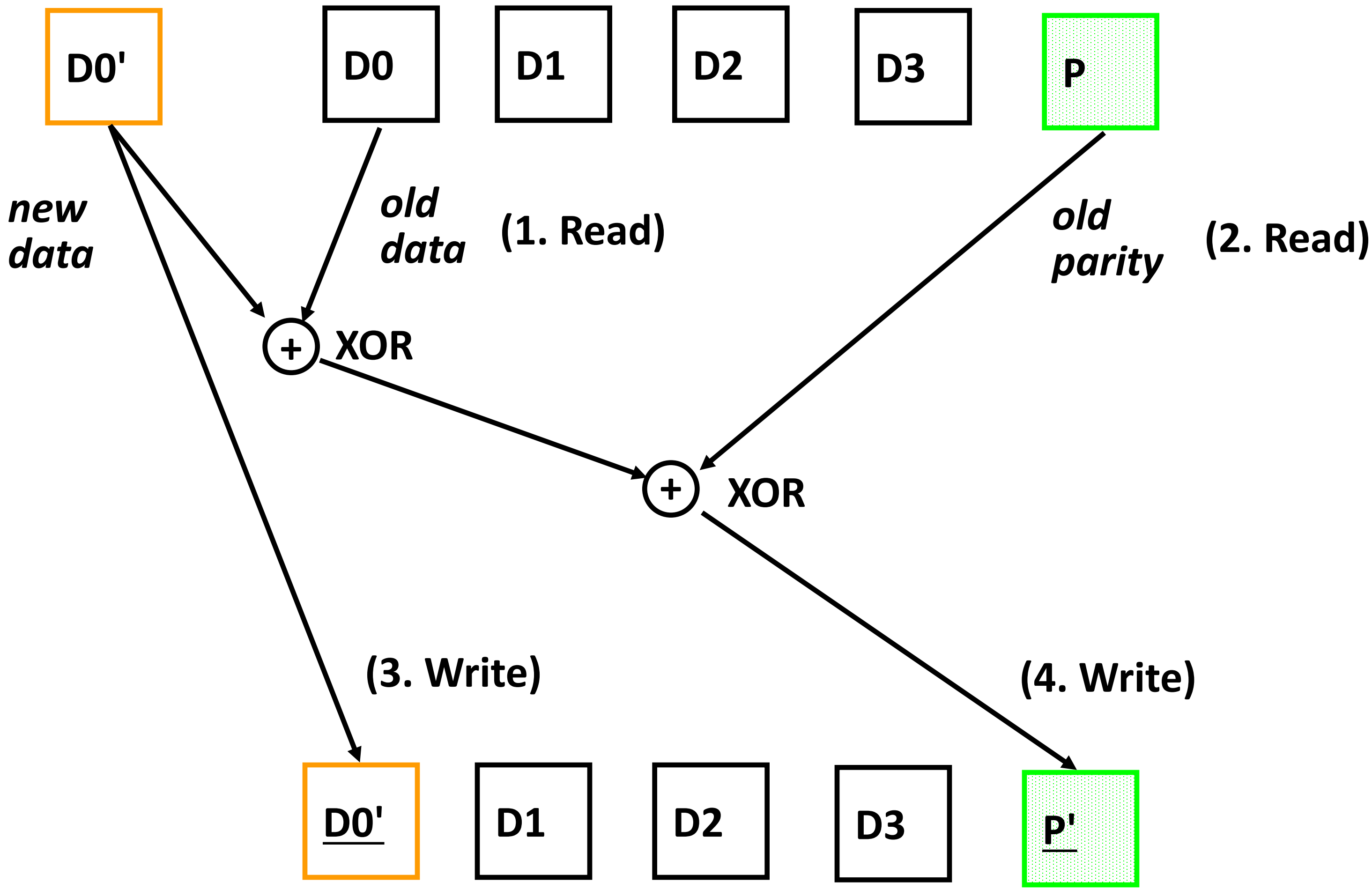
Example: write to D0, D5 uses disks 0, 1, 3, 4



Problems of Disk Arrays: Small Writes

RAID-5: Small Write Algorithm

1 Logical Write = 2 Physical Reads + 2 Physical Writes



And, in Conclusion, …

- Great Idea: Redundancy to Get Dependability
  - Spatial (extra hardware) and Temporal (retry if error)
- Reliability: MTTF & Annualized Failure Rate (AFR)
- Availability: % uptime (MTTF-MTTR/MTTF)
- Memory
  - Hamming distance 2: Parity for Single Error Detect
  - Hamming distance 3: Single Error Correction Code + encode bit position of error
- Treat disks like memory, except you know when a disk has failed—erasure makes parity an Error Correcting Code
- RAID-2, -3, -4, -5: Interleaved data and parity