CS 61C: Great Ideas in Computer Architecture

Dependability: Parity, RAID, ECC

Instructor: Alan Christopher

Review of Last Lecture

- MapReduce Data Level Parallelism
  - Framework to divide up data to be processed in parallel
  - Handles worker failure and laggard jobs automatically
  - Mapper outputs intermediate (key, value) pairs
  - Optional Combiner in-between for better load balancing
  - Reducer “combines” intermediate values with same key

Agenda

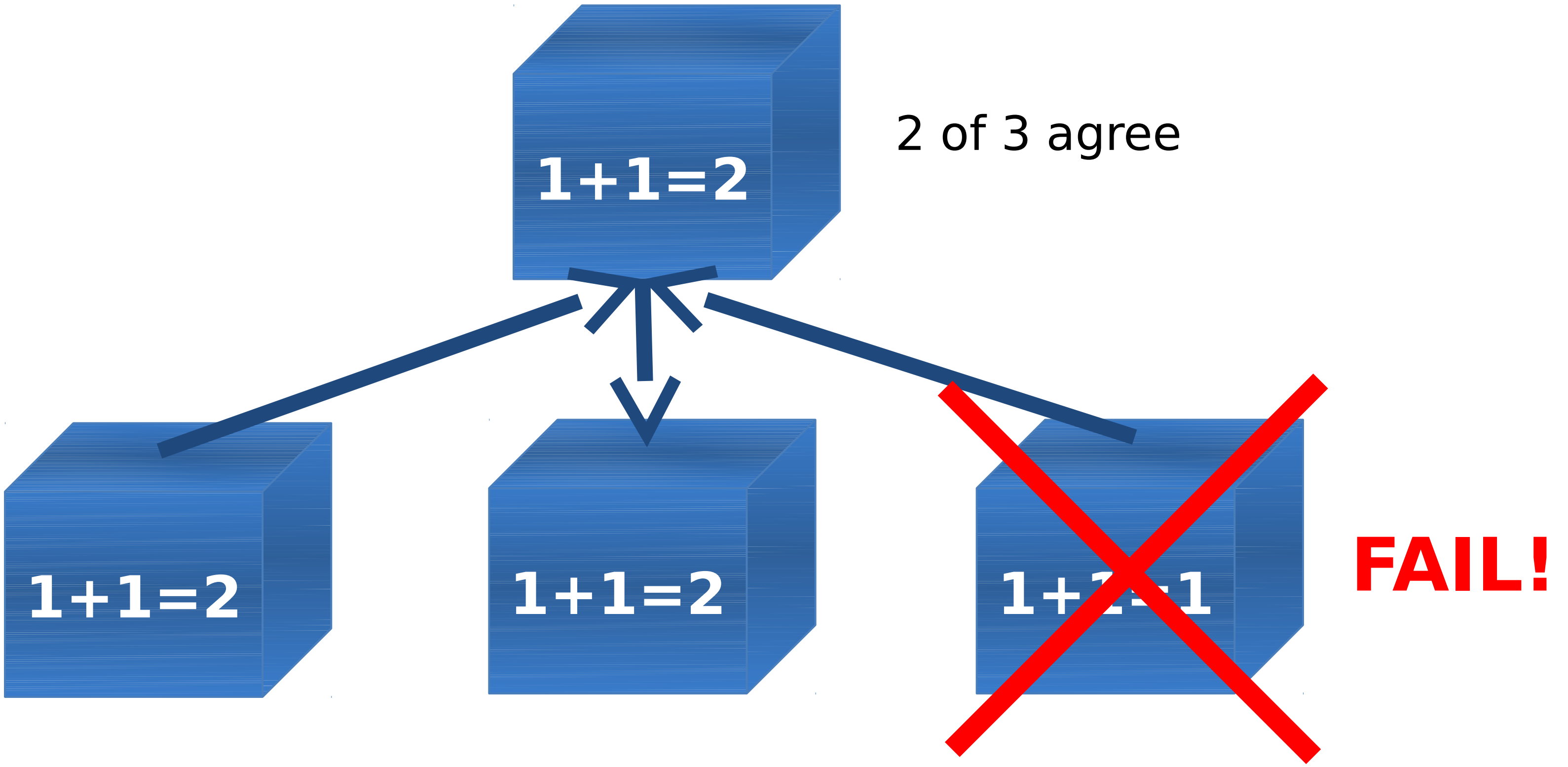
- Dependability
- Administrivia
- RAID
- Error Correcting Codes

Six Great Ideas in Computer Architecture

1. Layers of Representation/Interpretation
2. Technology Trends
3. Principle of Locality/Memory Hierarchy
4. Parallelism
5. Performance Measurement & Improvement
6. Dependability via Redundancy

Great Idea #6: Dependability via Redundancy

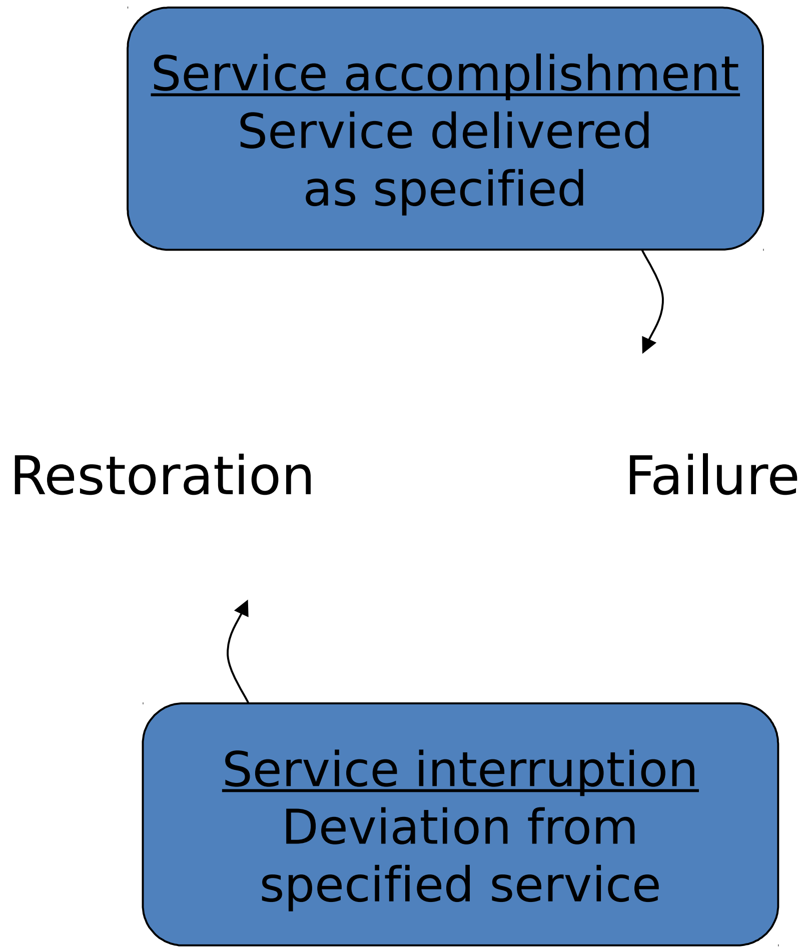
- Redundancy so that a failing piece doesn't make the whole system fail



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
  - Applies to *any* part of the system

Dependability Measures

- *Reliability:* Mean Time To Failure (MTTF)
- *Service interruption:* Mean Time To Repair (MTTR)
- Mean Time Between Failures (MTBF)
  - MTBF = MTTR + MTTF

- Availability = MTTF / (MTTF + MTTR) = MTTF / MTBF

- Improving Availability
  - Increase MTTF: more reliable HW/SW + fault tolerance
  - Reduce MTTR: improved tools and processes for diagnosis and repair

Reliability Measures

1) MTTF, MTBF measured in hours/failure
   - e.g. average MTTF is 100,000 hr/failure
2) Annualized Failure Rate (AFR)
   - Average rate of failures per year (%)

$$\mathrm{AFR} = \underbrace{\left(\frac{\mathrm{Disks}}{\mathrm{MTTF}} \times 8760\,\frac{\mathrm{hr}}{\mathrm{yr}}\right)}_{\text{Total disk failures/yr}} \times \frac{1}{\mathrm{Disks}} = \frac{8760\ \mathrm{hr/yr}}{\mathrm{MTTF}}$$

Availability Measures

- Availability = MTTF / (MTTF + MTTR) usually written as a percentage (%)
- Common jargon "number of 9s of availability per year" (more is better)
  - 1 nine: 90% => 36 days of repair/year
  - 2 nines: 99% => 3.6 days of repair/year
  - 3 nines: 99.9% => 526 min of repair/year
  - 4 nines: 99.99% => 53 min of repair/year
  - 5 nines: 99.999% => 5 min of repair/year

Dependability Example

- 1000 disks with MTTF = 100,000 hr and MTTR = 100 hr
  - MTBF = MTTR + MTTF = 100,100 hr
  - Availability = MTTF/MTBF = 0.9990 = 99.9%
    - 3 nines of availability!
  - AFR = 8760/MTTF = 0.0876 = 8.76%
- Faster repair to get 4 nines of availability?
  - $0.0001 \times \text{MTTF} = 0.9999 \times \text{MTTR}$
  - MTTR = 10.001 hr

Dependability 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

Question: There's a hardware glitch in our system that makes the Mean Time To Failure (MTTF) *decrease*. Are the following statements TRUE or FALSE?

1) Our system's Availability will *increase*.

2) Our system's Annualized Failure Rate (AFR) will *increase*.

	1	2
(B)	F	F
(G)	F	T
(P)	T	F
(Y)	T	T

Agenda

- Dependability
- Administrivia
- RAID
- Error Correcting Codes

Administrivia

- Project 3 (partners) due Sun 8/10
- Final Review – Sat 8/09, 2-5pm in 2060 VLSB
- Final – Fri 8/15, 9am-12pm, 155 Dwinelle
  - MIPS Green Sheet provided again
  - Two two-sided handwritten cheat sheets
    - Can re-use your midterm cheat sheet!

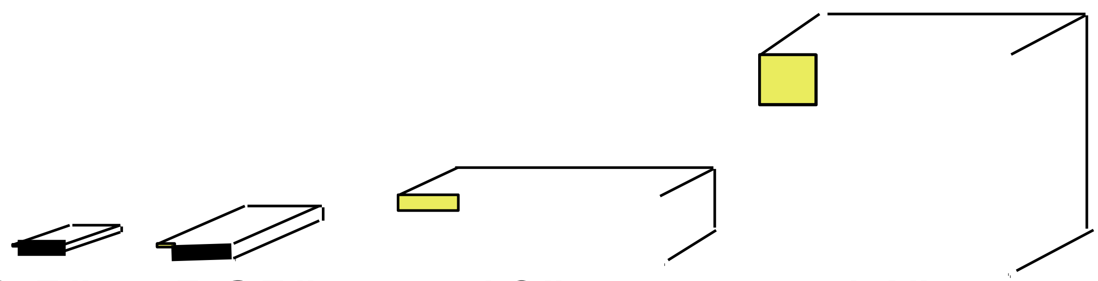
Agenda

- Dependability
- Administrivia
- RAID
- Error Correcting Codes

Arrays of Small Disks

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

Conventional:
4 disk types

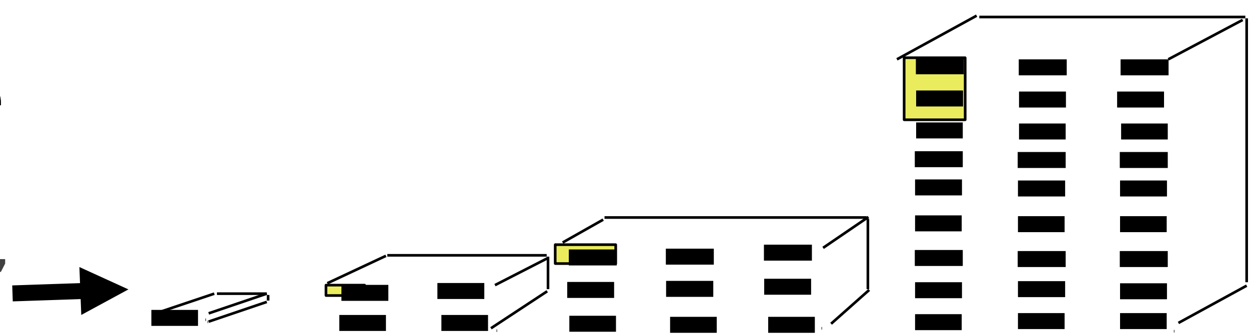


3.5” 5.25” 10” 14”

Low End → High End

Disk Array:
1 disk type

3.5”



Replace Large Disks with Large Number of Small Disks!

(Data from 1988 disks)

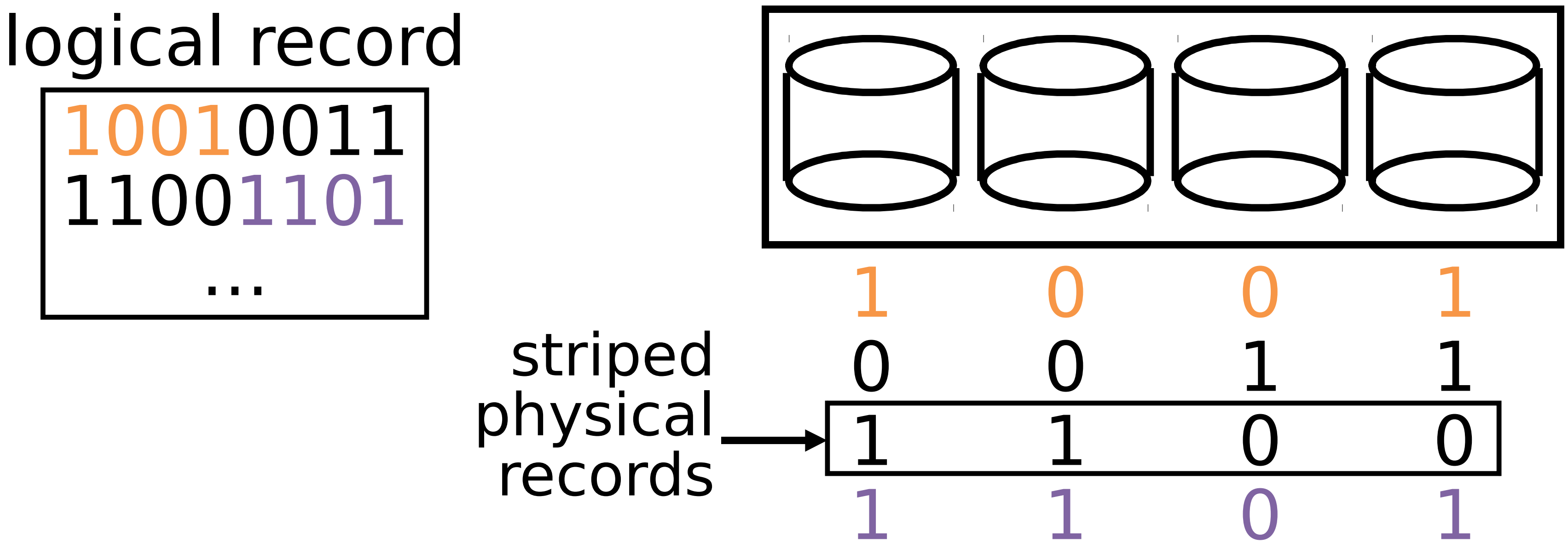
	IBM 3390K	IBM 3.5" 0061	x72	
Capacity	20 GBytes	320 MBytes	23 GBytes	
Volume	97 cu. ft.	0.1 cu. ft.	11 cu. ft.	9X
Power	3 KW	11 W	1 KW	3X
Data Rate	15 MB/s	1.5 MB/s	120 MB/s	8X
I/O Rate	600 I/Os/s	55 I/Os/s	3900 IOs/s	6X
MTTF	250 KHrs	50 KHrs	~700 Hrs	
Cost	$250K	$2K	$150K	

Disk Arrays have potential for large data and I/O rates, high MB/ft3, high MB/KW, *but what about reliability?*

RAID: Redundant Arrays of Inexpensive Disks

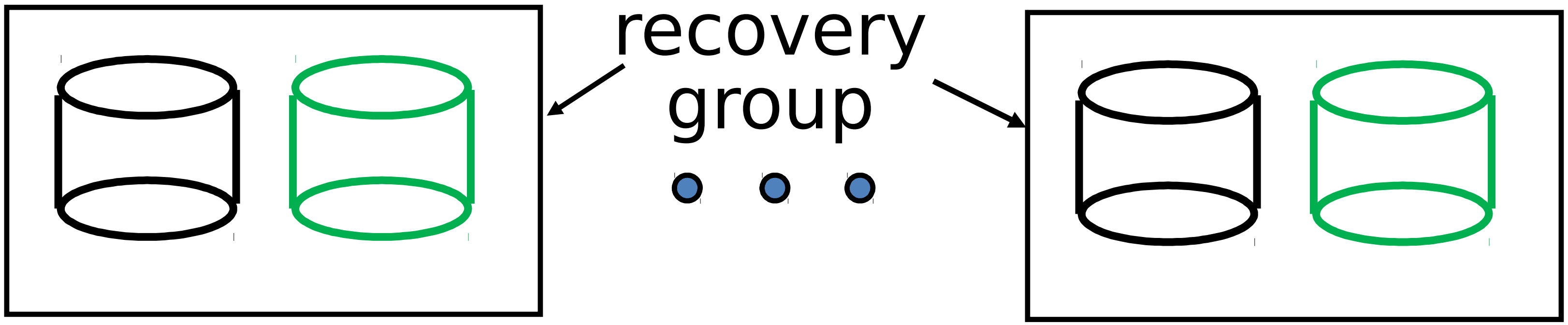
- Files are “striped” across multiple disks
  - Concurrent disk accesses improve throughput
- Redundancy yields high data availability
  - Service still provided to user, even if some components (disks) fail
- Contents reconstructed from data redundantly stored in the array
  - *Capacity penalty* to store redundant info
  - *Bandwidth penalty* to update redundant info

RAID 0: Data Striping



- "Stripe" data across all disks
  - Generally faster accesses (access disks in parallel)
  - No redundancy (really "AID")
  - Bit-striping shown here, can do in larger chunks

RAID 1: Disk Mirroring

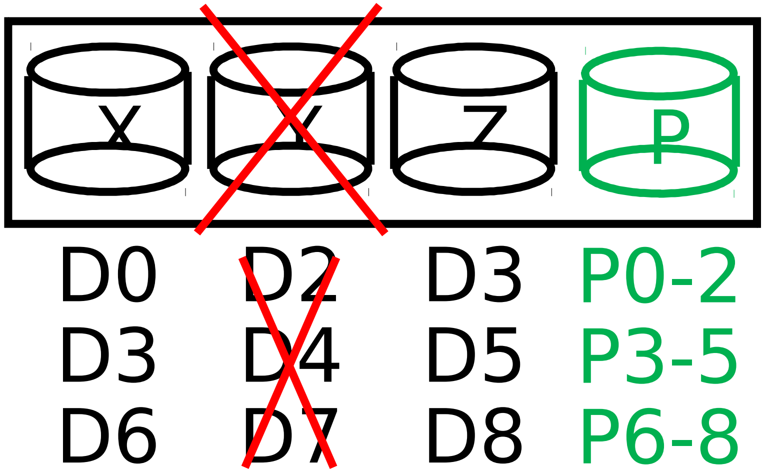


- Each disk is fully duplicated onto its "mirror"
  - Very high availability can be achieved
- Bandwidth sacrifice on write:
  - Logical write = two physical writes
  - Logical read = one physical read
- Most expensive solution: 100% capacity overhead

Parity Bit

- Describes whether a group of bits contains an even or odd number of 1's
  - Define 1 = odd and 0 = even
  - Can use XOR to compute parity bit!
- Adding the parity bit to a group will always result in an even number of 1's ("even parity")
  - 100 Parity: 1, 101 Parity: 0
- If we know number of 1's must be even, can we figure out what a single missing bit should be?
  - 10?11 → missing bit is 1

RAID 3: Parity Disk



- Logical data is byte-striped across disks
- Parity disk P contains parity bytes of other disks
- If any one disk fails, can use other disks to recover data!
  - We have to *know* which disk failed
- Must update Parity data on EVERY write
  - Logical write = min 2 to max N physical reads and writes

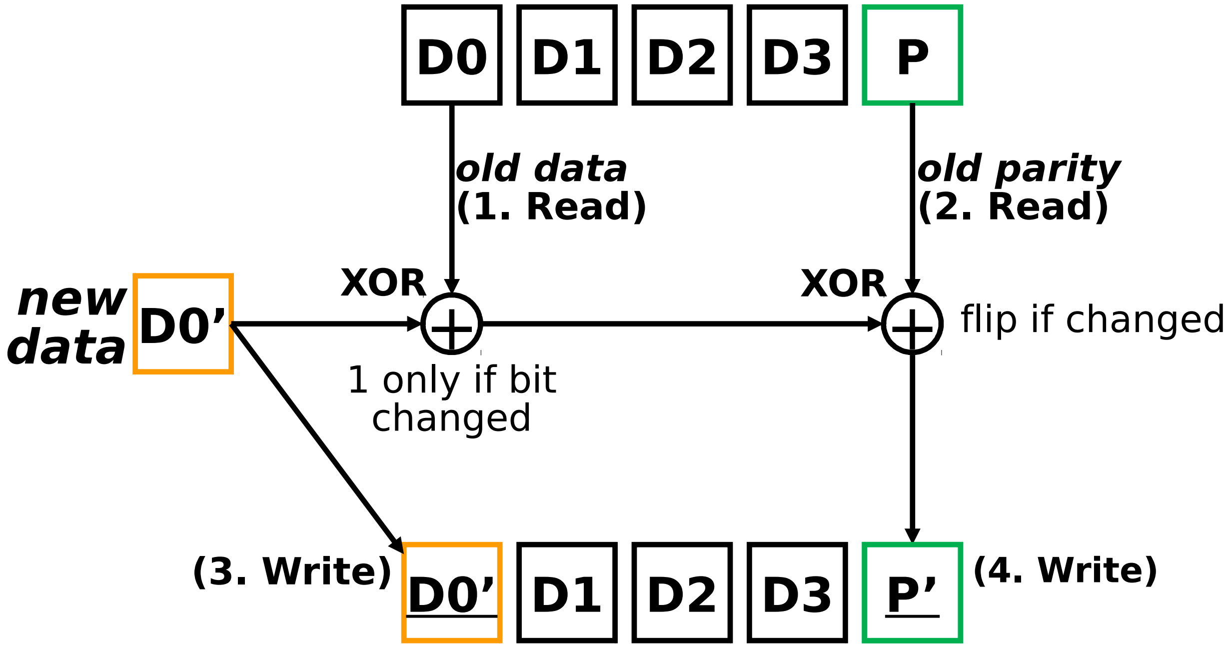
$$\text{parity}_{\text{new}} = \text{data}_{\text{old}} \oplus \text{data}_{\text{new}} \oplus \text{parity}_{\text{old}}$$

Updating the Parity Data

- Examine small write in RAID 3 (1 byte)
  - 1 logical write = 2 physical reads + 2 physical writes
  - Same concept applies for later RAIDs, too

What if writing halfword (2 B)? Word (4 B)?

D0’	D0	P	P’
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



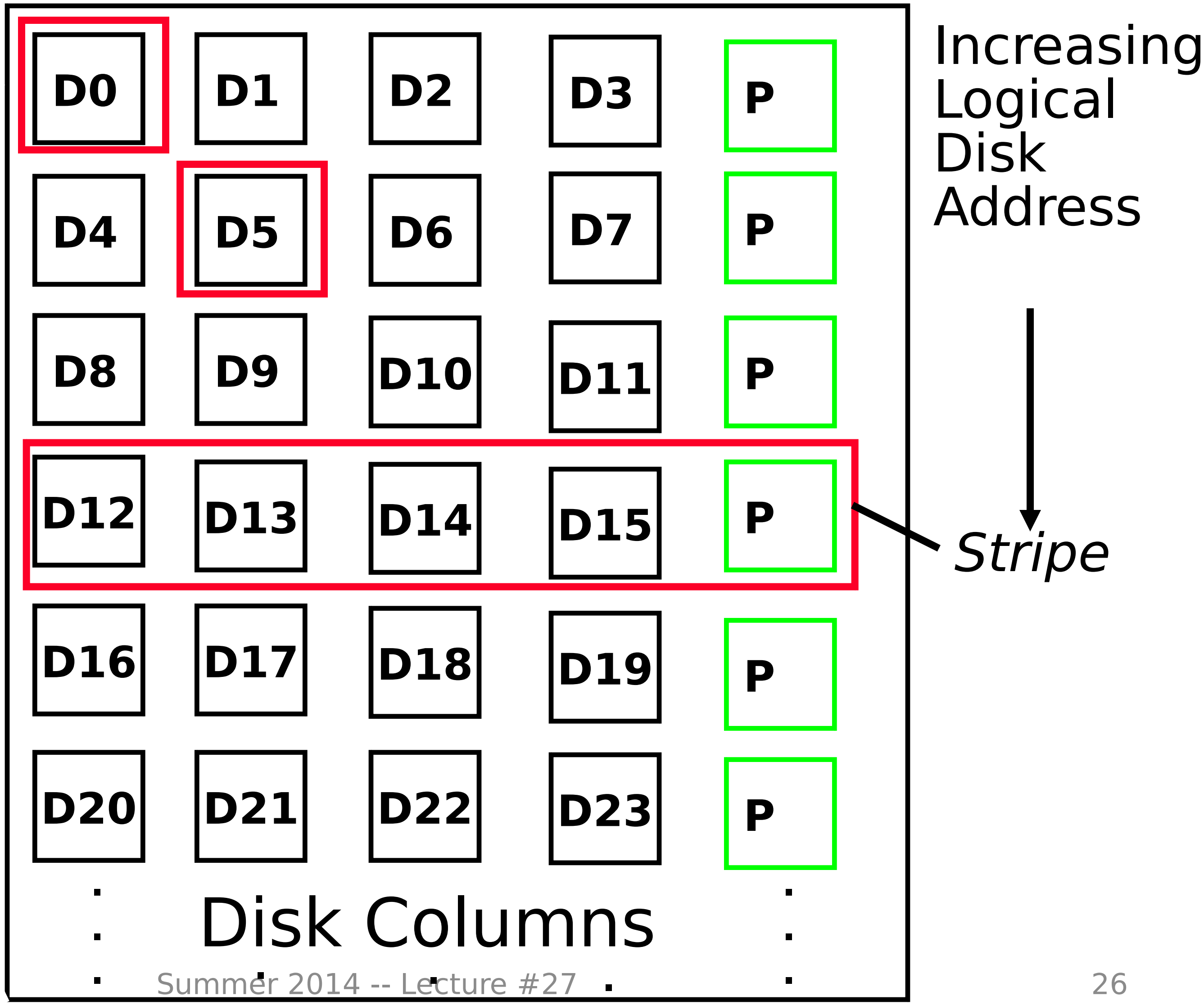
RAID 4: Higher I/O Rate

- Logical data is now *block*-striped across disks
- Parity disk P contains all parity blocks of other disks
- Because blocks are large, can handle small reads in parallel
  - Must be blocks in *different* disks
- Still must update Parity data on EVERY write
  - Logical write = min 2 to max N physical reads and writes
  - Performs poorly on small writes

RAID 4: Higher I/O Rate

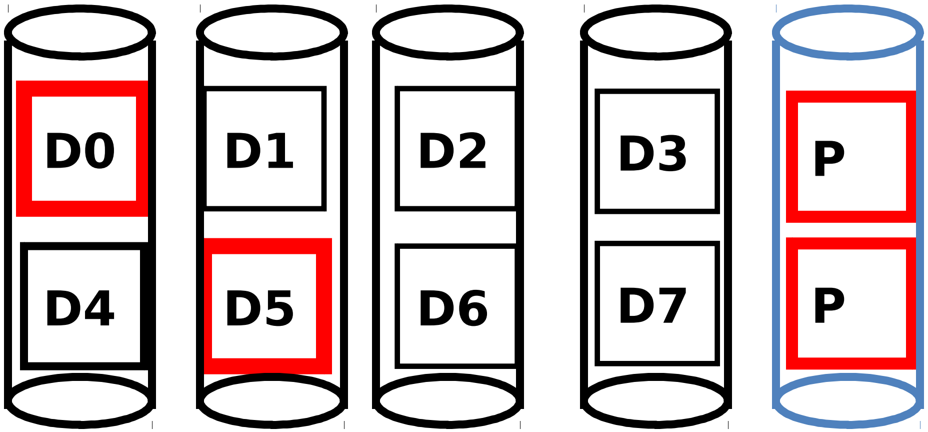
Insides of 5 disks

Example: small read D0 & D5, large write D12-D15



Inspiration for RAID 5

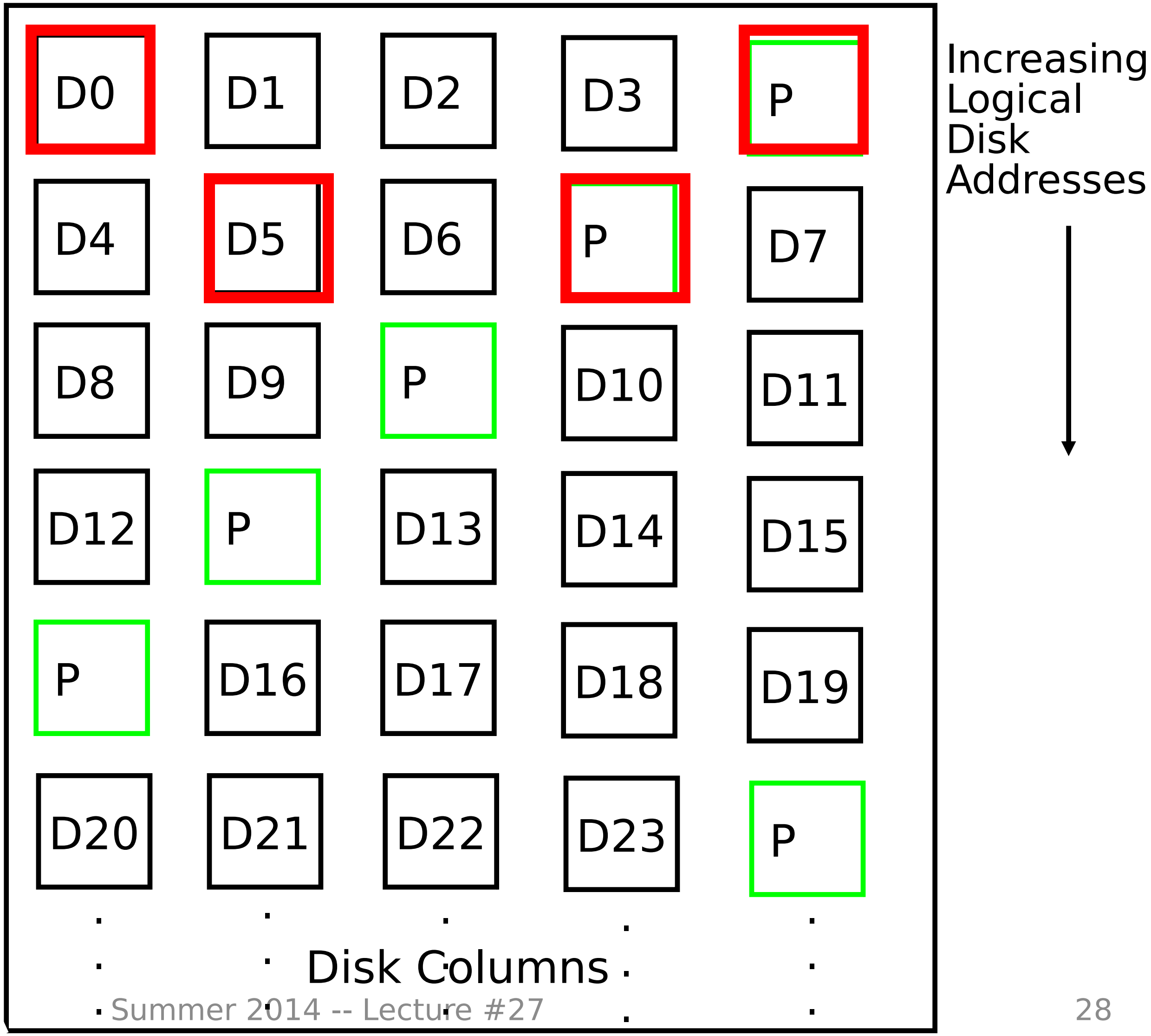
- When writing to a disk, need to update Parity
- Small writes are bottlenecked by Parity Disk: Write to D0, D5 both also write to P disk



RAID 5: Interleaved Parity

Independent writes possible because of interleaved parity

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



Agenda

- Dependability
- Administrivia
- RAID
- Error Correcting Codes

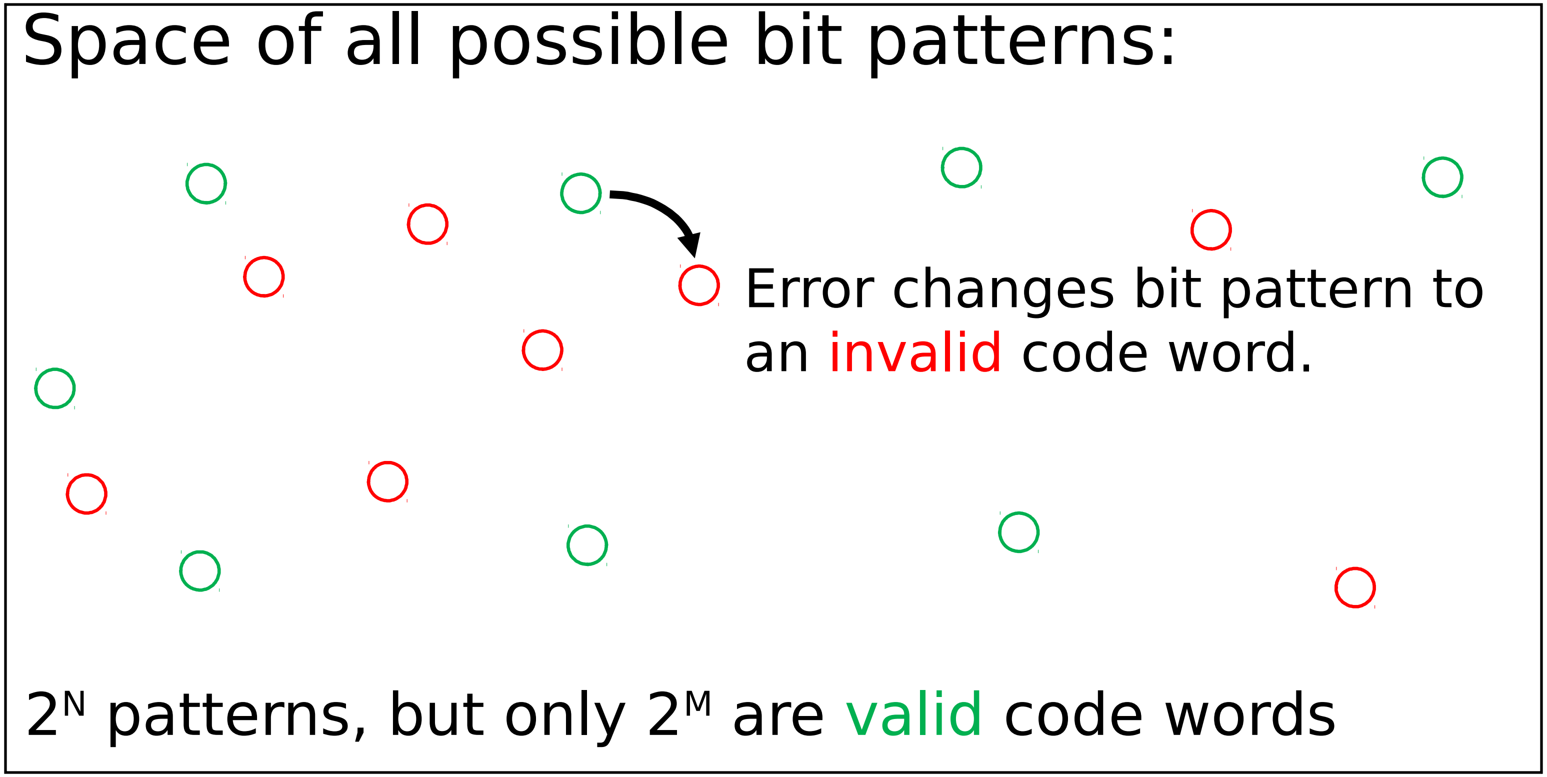
Error Detection/Correction Codes

- Memory systems generate errors (accidentally flipped-bits)
  - DRAMs store very little charge per bit
  - “Hard” errors occur when chips permanently fail
  - “Soft” errors occur occasionally when cells are struck by alpha particles or other environmental upsets
  - Problem gets worse as memories get denser and larger

Error Detection/Correction Codes

- Protect against errors with EDC/ECC
- Extra bits are added to each M-bit data chunk to produce an N-bit “code word”
  - Extra bits are a function of the data
  - Each data word value is mapped to a valid code word
  - Certain errors change valid code words to invalid ones (i.e. can tell something is wrong)

Detecting/Correcting Code Concept



- *Detection:* fails code word validity check
- *Correction:* can map to nearest valid code word

Hamming Distance

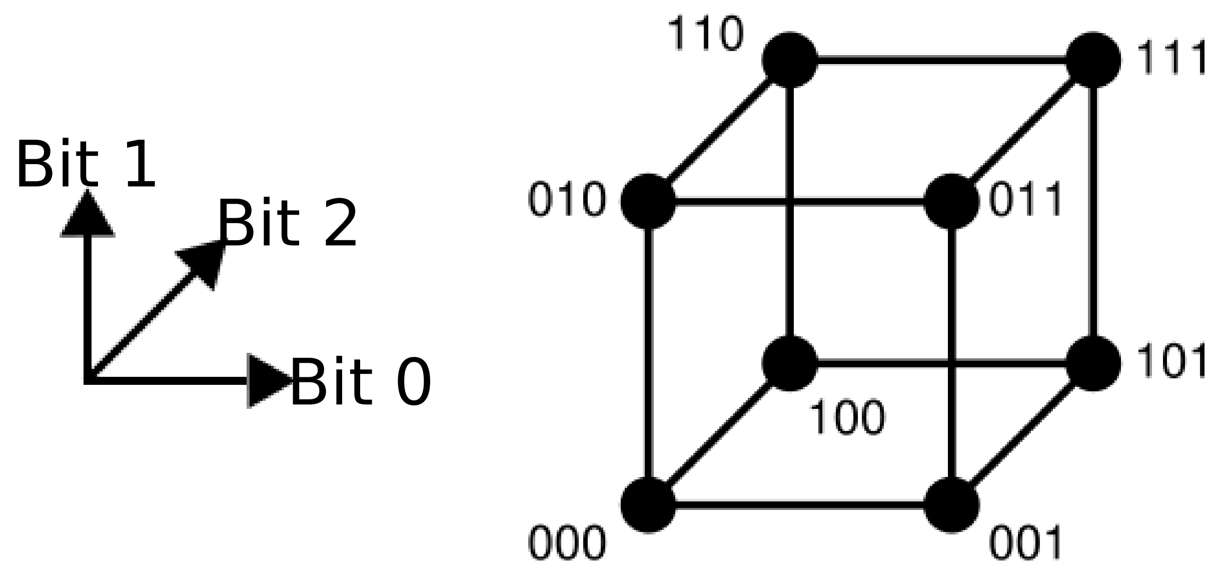
- Hamming distance = # of bit changes to get from one code word to another
- p = 0<u>1</u>1<u>0</u>11,
  q = 0<u>0</u>1<u>1</u>11, H_{dist}(p,q) = 2
- p = 011011,
  q = 110001, H_{dist}(p,q) = 3

Richard Hamming (1915-98)
Turing Award Winner

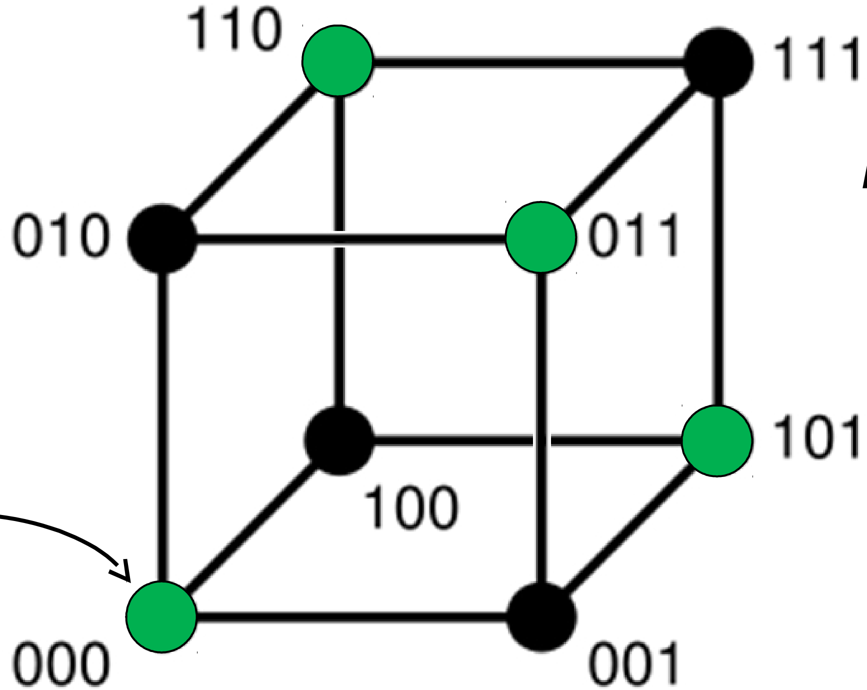
- If all code words are valid, then *min H_{dist} between valid code words is 1*
  - Change one bit, at another valid code word

3-Bit Visualization Aid

- Want to be able to *see* Hamming distances
  - Show code words as *nodes*, H_{dist} of 1 as *edges*
- For 3 bits, show each bit in a different dimension:



Minimum Hamming Distance 2

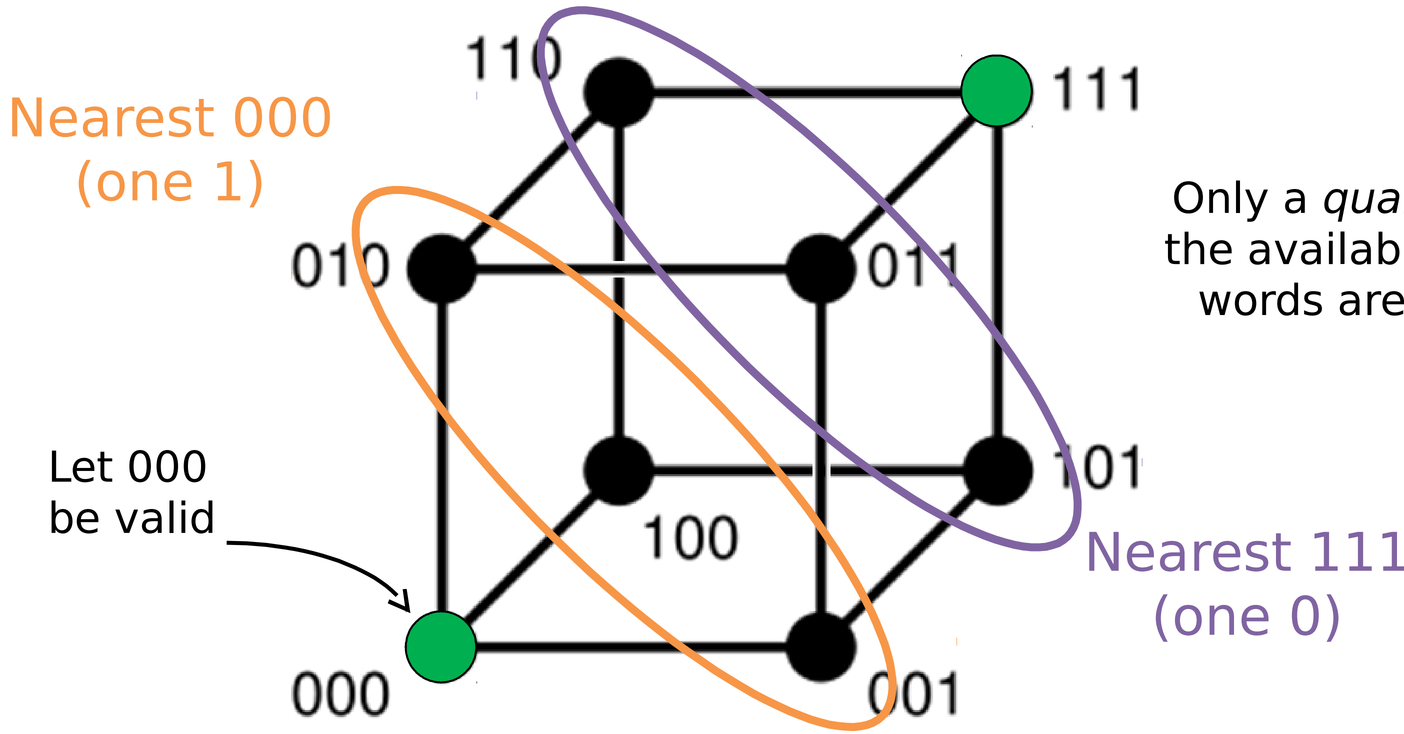


Let
000 be
valid

Half the available
code words
are valid

- If 1-bit error, is code word still valid?
  - No! So can *detect*
- If 1-bit error, know which code word we came from?
  - No! Equidistant, so cannot *correct*

Minimum Hamming Distance 3

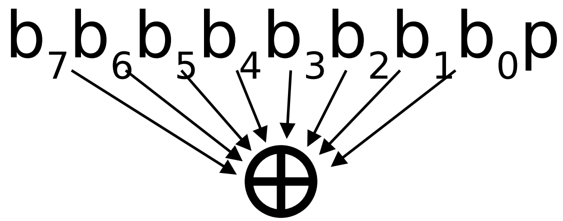


Only a *quarter* of the available code words are valid

- How many bit errors can we detect?
  - Two! Takes 3 errors to reach another valid code word
- If 1-bit error, know which code word we came from?
  - Yes!

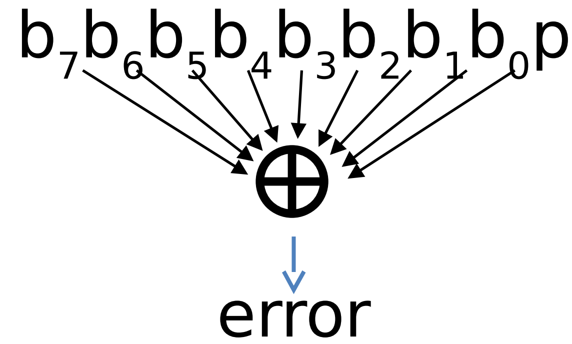
Parity: Simple Error Detection Coding

Add parity bit when writing block of data:



Check parity on block read:

- Error if odd number of 1s
- Valid otherwise



- Minimum Hamming distance of parity code is 2
- Parity of code word = 1 indicates an error occurred:
  - 2-bit errors not detected (nor any even # of errors)
  - Detects an odd # of errors

Parity Examples

1) Data 0101 0101
   - 4 ones, even parity now
   - Write to memory 0101 0101 0 to *keep* parity even

2) Data 0101 0111
   - 5 ones, odd parity now
   - Write to memory: 0101 0111 1 to *make* parity even

3) Read from memory 0101 0101 0
   - 4 ones → even parity, so no error

4) Read from memory 1101 0101 0
   - 5 ones → odd parity, so error

5) What if error in parity bit?
   - Can detect!

Technology Break

Agenda

- Dependability
- Administrivia
- RAID
- Error Correcting Codes (Cont.)

How to Correct 1-bit Error?

- **Recall:** Minimum distance for correction?
  - Three
- Richard Hamming came up with a mapping to allow Error Correction at min distance of 3
  - Called Hamming ECC for Error Correction Code

Hamming ECC (1/2)

- Use *extra parity bits* to allow the position identification of a single error
  - Interleave parity bits within bits of data to form code word
  - **Note:** Number bits starting at 1 from the left

1) Use *all* bit positions in the code word that are powers of 2 for parity bits (1, 2, 4, 8, 16, ...)
2) All other bit positions are for the data bits (3, 5, 6, 7, 9, 10, ...)

Hamming ECC (2/2)

3) Set each parity bit to create even parity for *a group* of the bits in the code word
   - The position of each parity bit determines the group of bits that it checks
   - Parity bit p checks every bit whose position number in binary has a 1 in the bit position corresponding to p
     - Bit 1 (00012) checks bits 1,3,5,7, … (XXX12)
     - Bit 2 (00102) checks bits 2,3,6,7, … (XX1X2)
     - Bit 4 (01002) checks bits 4-7, 12-15, … (X1XX2)
     - Bit 8 (10002) checks bits 8-15, 24-31, … (1XXX2)

Hamming ECC Example (1/3)

- A byte of data: 10011010
- Create the code word, leaving spaces for the parity bits:
  $\_\_{1}\ \_\_{2}\ 1_{3}\ \_\_{4}\ 0_{5}\ 0_{6}\ 1_{7}\ \_\_{8}\ 1_{9}\ 0_{10}\ 1_{11}\ 0_{12}$

Hamming ECC Example (2/3)

- Calculate the parity bits:
  - Parity bit 1 group (1, 3, 5, 7, 9, 11):

    ? _ **1** _ **0** 0 **1** _ **1** 0 **1** 0 → 0
  - Parity bit 2 group (2, 3, 6, 7, 10, 11):

    0 **? 1** _ 0 **0 1** _ 1 **0 1** 0 → 1
  - Parity bit 4 group (4, 5, 6, 7, 12):

    0 1 1 **? 0 0 1** _ 1 0 1 **0** → 1
  - Parity bit 8 group (8, 9, 10, 11, 12):

    0 1 1 1 0 0 1 **? 1 0 1 0** → 0

Hamming ECC Example (3/3)

- Valid code word: $\underline{01}1\underline{1}001\underline{0}1010$
- Recover original data: 1 001 1010

Suppose we see $0_1 1_2 1_3 1_4 0_5 0_6 1_7 0_8 1_9 1_{10} 1_{11} 0_{12}$ instead – fix the error!

- Check each parity group
  - Parity bits 2 and 8 are incorrect
  - As 2+8=10, bit position 10 is the bad bit, so flip it!
- Corrected value: $011100101\underline{0}10$

Hamming ECC "Cost"

- Space overhead in single error correction code
  - Form $p + d$ bit code word, where p = # parity bits and d = # data bits
- Want the p parity bits to indicate either "no error" or 1-bit error in one of the $p + d$ places
  - Need $2p \geq p + d + 1$, thus $p \geq \log_2(p + d + 1)$
  - For large d, p approaches $\log_2(d)$
- **Example:** $d = 8 \rightarrow p = \lceil \log_2(p+8+1) \rceil \rightarrow p = 4$
  - $d = 16 \rightarrow p = 5$; $d = 32 \rightarrow p = 6$; $d = 64 \rightarrow p = 7$

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

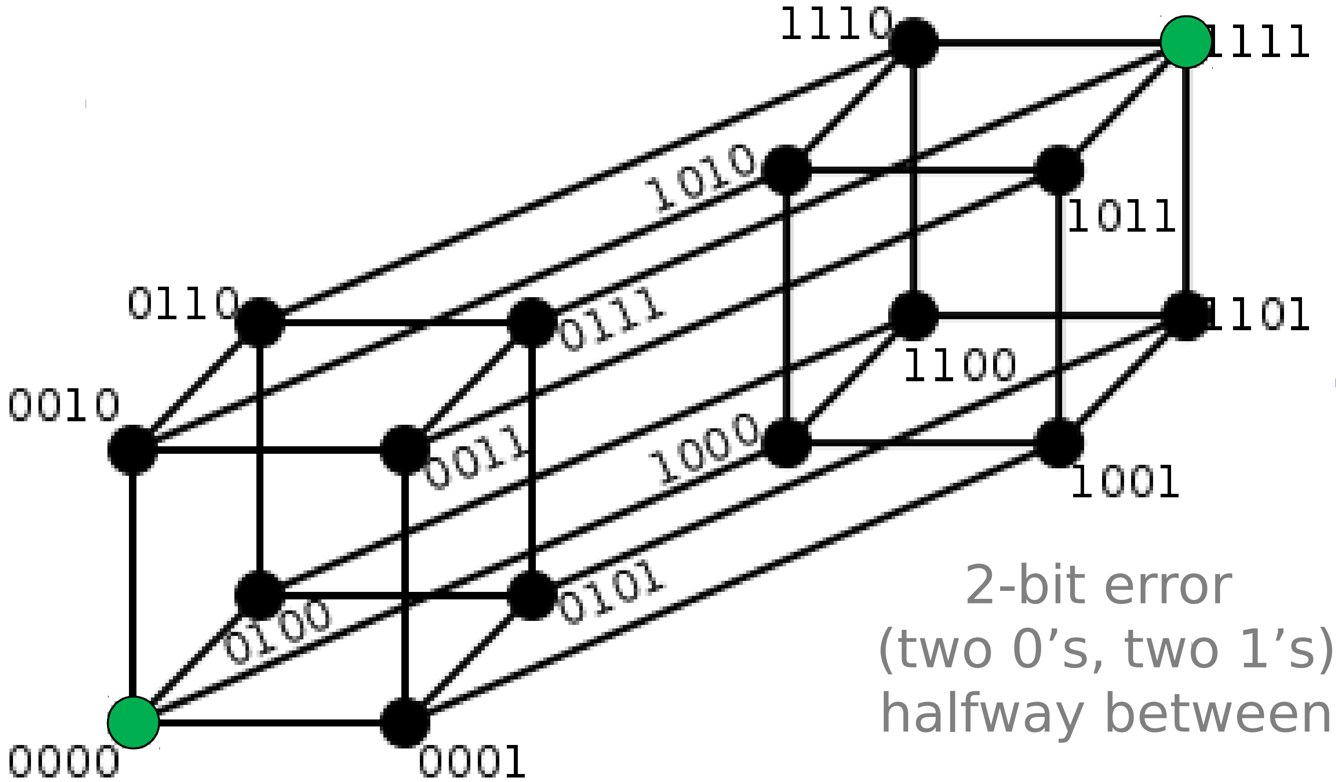
- Adding extra parity bit covering the entire SEC code word provides *double error detection* as well!

1	2	3	4	5	6	7	8
p_1	p_2	d_1	p_3	d_2	d_3	d_4	p_4

- Let H be the position of the incorrect bit we would find from checking p_1, p_2, and p_3 (0 means no error) and let P be parity of complete code word
  - H=0 P=0, no error
  - H≠0 P=1, correctable single error (p_4=1 → odd # errors)
  - H≠0 P=0, double error detected (p_4=0 → even # errors)
  - H=0 P=1, an error occurred in p_4 bit, not in rest of word

SEC/DED: Hamming Distance 4

1-bit error (one 0)
Nearest 1111



2-bit error
(two 0’s, two 1’s)
halfway between

1-bit error (one 1)
Nearest 0000

Modern Use of RAID and ECC (1/2)

- Typical modern code words in DRAM memory systems:
  - 64 bit data blocks (8 B) with 72 bit codewords (9 B)
  - $D = 64 \rightarrow p = 7$, +1 for DED
- RAID 6: Recovering from two disk failures!
  - RAID 5 with an extra disk's amount of parity blocks (also interleaved)
  - Extra parity computation more complicated than Double Error Detection (not covered here)
  - When useful?
    - Operator replaces wrong disk during a failure
    - Disk bandwidth is growing more slowly than disk capacity, so MTTR a disk in a RAID system is increasing (increases the chances of a 2nd failure during repair)

Modern Use of RAID and ECC (2/2)

- Common failure mode is bursts of bit errors, not just 1 or 2
  - Network transmissions, disks, distributed storage
  - Contiguous sequence of bits in which first, last, or any number of intermediate bits are in error
  - Caused by impulse noise or by fading signal strength; effect is greater at higher data rates
- **Other tools:** cyclic redundancy check, Reed-Solomon, other linear codes

Summary

- Great Idea: Dependability via Redundancy
  - Reliability: MTTF & Annual Failure Rate
  - Availability: % uptime = MTTF/MTBF
- RAID: Redundant Arrays of Inexpensive Disks
  - Improve I/O rate while ensuring dependability
  - http://www.accs.com/p_and_p/RAID/BasicRAID.html
- Memory Errors:
  - Hamming distance 2: Parity for Single Error Detect
  - Hamming distance 3: Single Error Correction Code + encode bit position of error
  - Hamming distance 4: SEC/Double Error Detection