Dependability: Parity, ECC, RAID

Instructor: Suvansh Sanjeev
Agenda

• Dependability
• Administrivia
• Error Correcting Codes (ECC)
• RAID
Great Idea: Dependability via Redundancy

• Redundancy so that a failing piece doesn’t make the whole system fail
Great Idea: 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

Service accomplishment
Service delivered as specified

Service interruption
Deviation from specified service

Restoration → Failure → Restoration
Dependability Measures

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

• **Availability** = \( \frac{MTTF}{MTTF + MTTR} \) = \( \frac{MTTF}{MTBF} \)

• 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 (%)

\[
\text{AFR} = \left( \frac{\text{Disks}}{\text{MTTF}} \times 8760 \text{ hr/yr} \right) \times \frac{1}{\text{Disks}} = \frac{8760 \text{ hr/yr}}{\text{MTTF}}
\]

Total disk failures/yr
Availability Measures

- Availability = MTTF / (MTTF + MTTR) usually written as a percentage (%)
- Want high availability, so categorize by “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 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%
  - AFR = 8760/MTTF = 0.0876 = 8.76%
Calculating MTTR Example

- Faster repair to get 4 nines of availability?
  - $0.9999 = \frac{MTTF}{MTTF + MTTR}$
  - $0.9999 \times (MTTF + MTTR) = MTTF$
  - $0.9999 \times MTTF + 0.9999 \times MTTR = MTTF$
  - $0.9999 \times MTTR = 0.0001 \times MTTF$
    - Plug in $MTTF = 100,000 \text{ hr}$
  - $MTTR = 10.001 \text{ hr}$
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 because dependability is limited by the 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*.

(A) F F
(B) F T
(C) T F
(D) T T
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*.

(A) F F
(B) F T
(C) T F
(D) T T
 Administrivia

• Proj4 due on Friday (8/03)
  – Hold off on submissions for now
• HW7 due 8/06
• Guerilla Session today @Cory 540AB, 4-6p
• Regrade requests are open for MT2 until Friday
• The final will be 8/09 7-10PM @VLSB 2040/2060!
  – If you have a conflict, look out for an email with details
Agenda

• Dependability
• Administrivia
• Error Correcting Codes (ECC)
• RAID
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 occur when chips permanently fail
  – Problem gets worse as memory systems 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. you can tell something is wrong)
Detecting/Correcting Code Concept

Space of all possible bit patterns:

2^N patterns, but only 2^M are valid code words

- 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

- $p = 0\underline{110}11$, $q = 0\underline{011}11$, $Hdist(p,q) = 2$

- $p = 01\underline{110}11$, $q = 1\underline{1100}01$, $Hdist(p,q) = 3$

- If all code words are valid, then $\min Hdist$ between valid code words is 1
  
  - Change one bit, at another valid code word

Richard Hamming
(1915-98)
Turing Award Winner
3-Bit Visualization Aid

• Want to be able to see Hamming distances
  – Show code words as *nodes*, Hdist of 1 as *edges*
• For 3 bits, show each bit in a different dimension:
Minimum Hamming Distance 2

Let 000 be 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

Half the available code words are valid
Minimum Hamming Distance 3

• 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!

Let 000 be valid

Nearest 000 (one 1)

Nearest 111 (one 0)

Only a quarter of the available code words are valid
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
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
   • What if error in parity
     bit?
     - Can detect!
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, …)
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 \((0001_2)\) checks bits 1,3,5,7, … \((XXX1_2)\)
     - Bit 2 \((0010_2)\) checks bits 2,3,6,7, … \((XX1X_2)\)
     - Bit 4 \((0100_2)\) checks bits 4-7, 12-15, … \((X1XX_2)\)
     - Bit 8 \((1000_2)\) checks bits 8-15, 24-31, … \((1XXX_2)\)
### Graphic of Hamming Code

<table>
<thead>
<tr>
<th>Bit position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Encoded data bits</strong></td>
<td>p1</td>
<td>p2</td>
<td>d1</td>
<td>p4</td>
<td>d2</td>
<td>d3</td>
<td>d4</td>
<td>p8</td>
<td>d5</td>
<td>d6</td>
<td>d7</td>
<td>d8</td>
<td>d9</td>
<td>d10</td>
<td>d11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parity bit coverage</th>
<th>p1</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>p8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hamming ECC Example (1/4)

• A byte of data: 10011010
• Create the code word, leaving spaces for the parity bits:

  _1 _2 _3 _4 5 6 7 _8 9 10 11 12
Hamming ECC Example (2/4)

• Calculate the parity bits:
  – Parity bit 1 group (1, 3, 5, 7, 9, 11):
    \[
    ? \_ 1 \_ 0 0 1 \_ 1 0 1 0 \rightarrow 0
    \]
  – Parity bit 2 group (2, 3, 6, 7, 10, 11):
    \[
    \_ \_ 1 \_ 0 0 1 \_ 1 0 1 0 \rightarrow 1
    \]
  – Parity bit 4 group (4, 5, 6, 7, 12):
    \[
    \_ \_ 1 \_ 0 0 1 \_ 1 0 1 0 \rightarrow 1
    \]
  – Parity bit 8 group (8, 9, 10, 11, 12):
    \[
    \_ \_ 1 \_ 0 0 1 \_ 1 0 1 0 \rightarrow 0
    \]
Hamming ECC Example (3/4)

• Valid code word: 01110010101010
• Recover original data: 1 001 1010
Suppose we see 01110010111010 instead – fix the error!
But how would we figure out where the error is if we just see the code word? Hmm….
• Let’s examine the parity bits that are dependent on bit 10
  – Maybe we can figure out a pattern?
Looks like $p_2$ and $p_8$ were responsible

- $p_2$ & $p_8$ will be the only incorrect parity bits IFF bit 10 is incorrect
- Notice that $2 + 8 = 10$
- Seems like incorrect parity bits tell us where the error is... :3

<table>
<thead>
<tr>
<th>Bit position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoded data bits</td>
<td>p1</td>
<td>p2</td>
<td>d1</td>
<td>p4</td>
<td>d2</td>
<td>d3</td>
<td>d4</td>
<td>p8</td>
<td>d5</td>
<td>d6</td>
<td>d7</td>
<td>d8</td>
<td>d9</td>
<td>d10</td>
<td>d11</td>
</tr>
</tbody>
</table>

- Parity bit coverage:
  - $p_1$: X X X X X X X X X X X X
  - $p_2$: X X X X X X X X X X X X
  - $p_4$: X X X X X X X X X X X X
  - $p_8$: X X X X X X X X X X X X

- Notice that $2 + 8 = 10$
Hamming ECC Example (3/4)

• Valid code word: 01110010101010
• Recover original data: 1 001 1010

Suppose we see 01110010111110 instead – fix the error!

How to figure out where the error is? Hmm….

• Check the parity bits responsible for bit 10
  – Parity bits 2 and 8 are incorrect
  – As 2+8=10, bit position 10 is the bad bit, so flip it!
• Corrected value: 01110010101010
OMFG
The parity bits form a number—the bit position in the code word!
• This tells you which bit of the code word is incorrect!

<table>
<thead>
<tr>
<th>Bit position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Encoded data bits</strong></td>
<td>p1</td>
<td>p2</td>
<td>p4</td>
<td>d1</td>
<td>d2</td>
<td>d3</td>
<td>d4</td>
<td>p8</td>
<td>p5</td>
</tr>
<tr>
<td><strong>Parity bit coverage</strong></td>
<td>p1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>p2</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>p4</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>p8</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Coverage</th>
<th>p1</th>
<th>p2</th>
<th>p4</th>
<th>p8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Parity</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Encoded</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>data</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Replace all the X’s with 1’s Everywhere else put a 0.
Going in reverse:
We see 0111001011110

Figure out which bit is wrong:

\[ p_1 : 0101111 \text{ = even number of 1's : 0} \]
\[ p_2 : 1101111 \text{ = odd number of 1's : 1} \]
\[ p_4 : 10010 \text{ = even number of 1's : 0} \]
\[ p_8 : 01110 \text{ = odd number of 1's : 1} \]

Incorrect code bit:
0b \ p_8 \ p_4 \ p_2 \ p_1 = 0b 1010 = 10! So flip bit 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 $2^p \geq p + d + 1$, thus $p \geq \log_2(p + d + 1)$
  – For large $d$, $p$ approaches $\log_2(d)$
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!

\[ \begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
p_1 & p_2 & d_3 & p_4 & d_5 & d_6 & d_7 & p_8 \\
\end{array} \]

- Let \( H \) be the position of the incorrect bit we would find from checking \( p_1, p_2, \) and \( p_4 \) (0 means no error) and let \( P \) be parity of complete code word (p’s & d’s)
  - \( H=0 \ P=0, \) no error
  - \( H\neq0 \ P=1, \) correctable single error (\( P=1 \rightarrow \) odd # errors)
  - \( H\neq0 \ P=0, \) double error detected (\( P=0 \rightarrow \) even # errors)
  - \( H=0 \ P=1, \) an error occurred in \( p_8 \) bit, not in rest of word
Question: We saw that when the minimum hamming distance of codewords was 3, we get single error detection and correction. If we had a hamming distance of 4, we’d have

1) single error detection
2) single error correction
3) double error detection
4) double error correction

A) 1, 2
B) 1, 3
C) 1, 2, 3
D) 1, 2, 3, 4
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
Summary

• Great Idea: Dependability via Redundancy
  – Reliability: MTTF & Annual Failure Rate
  – Availability: % uptime = MTTF/MTBF

• 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

• We’ll talk about RAID next lecture!