EEL-4736/5737 Principles of Computer System Design

Lecture Slides 19

Textbook Chapter 8

Redundancy

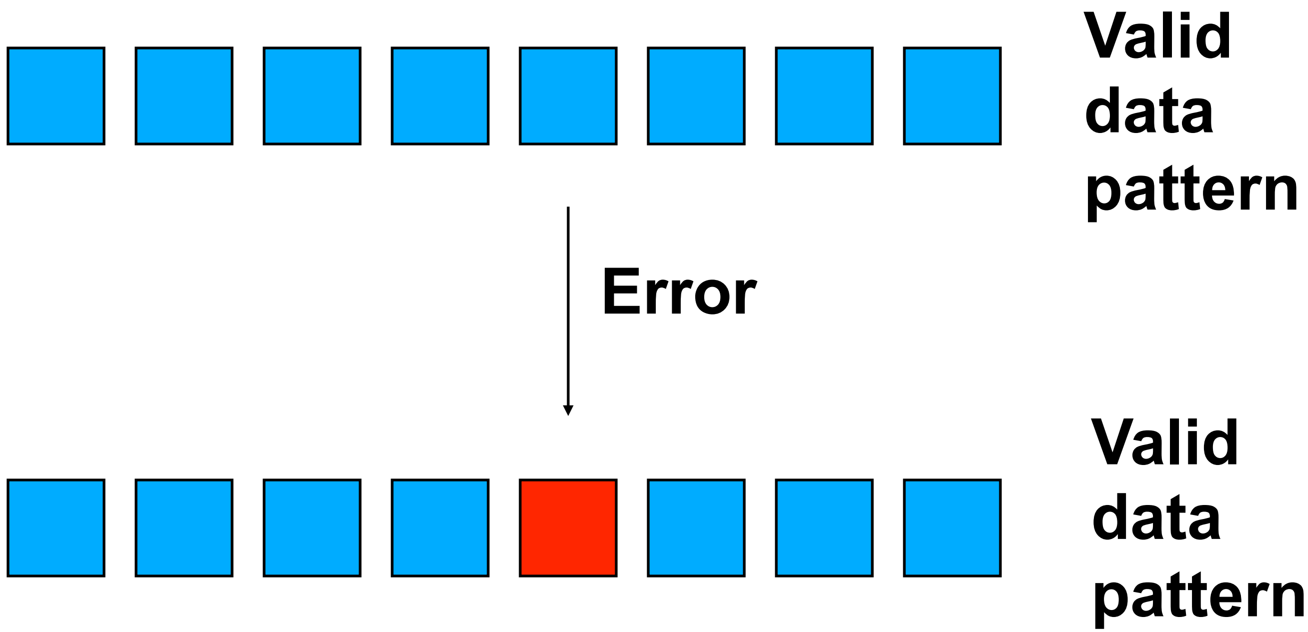
Introduction

- Analog systems – safety margins to cope with variations
- Computer systems – apply redundancy in time and/or space
  - Error correction codes
  - Replication
  - Retry

Coding

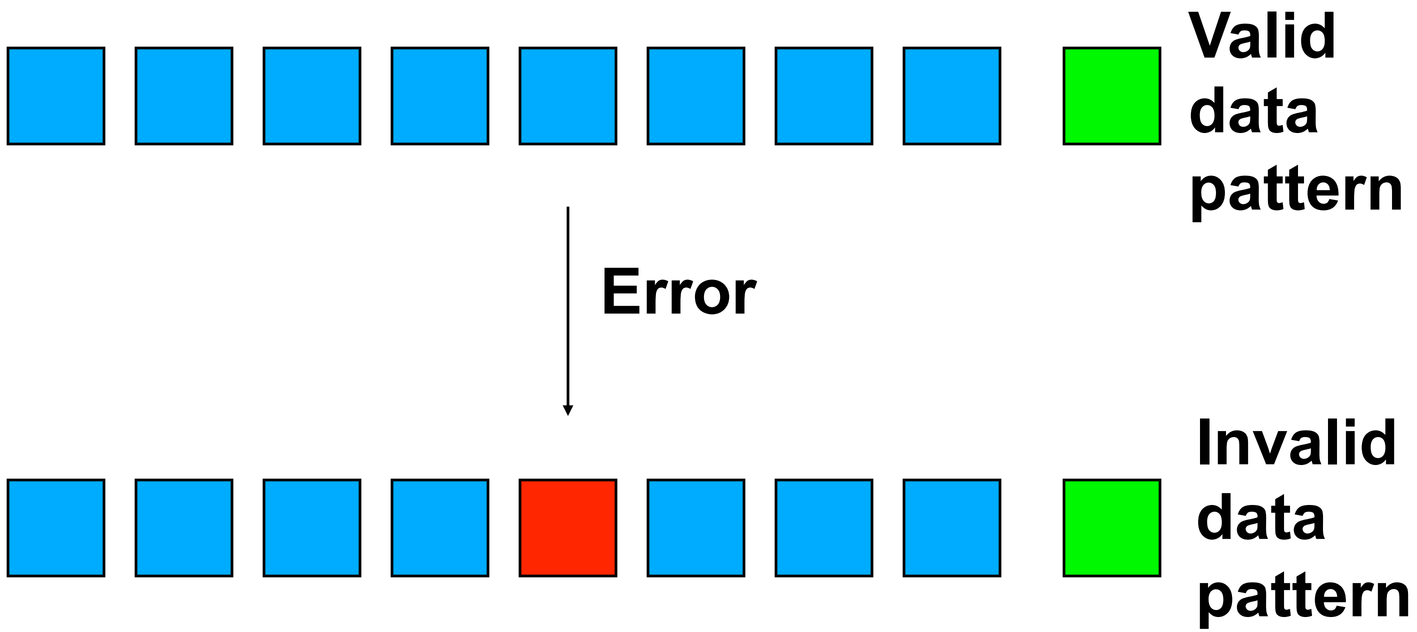
- Incremental redundancy
  - Add information to a message to enable detecting and re-constructing original message if an error occurs

Main idea



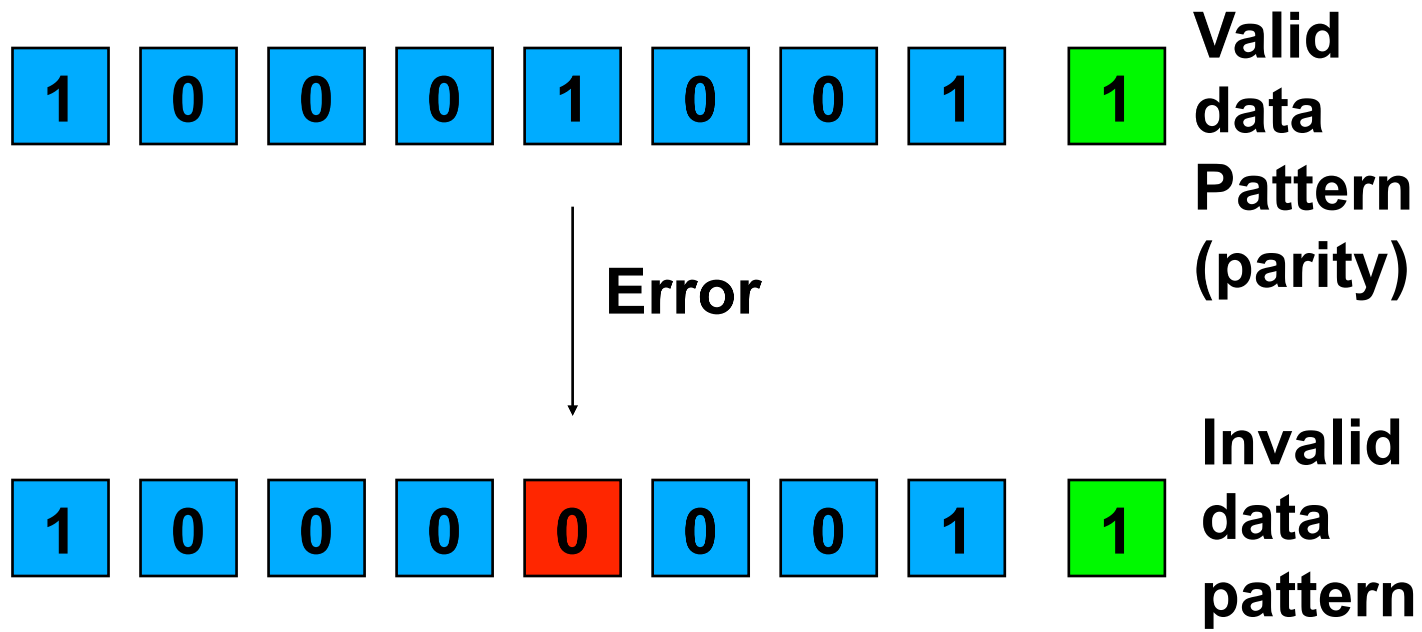
Cannot detect error since it is still a valid data pattern

Main idea



Possible to detect error if error causes an invalid data pattern

Example


1 0 0 0 1 0 0 1 1
Valid data Pattern (parity)
Error
1 0 0 0 0 0 0 1 1
Invalid data pattern

Hamming distance

- Smallest number of bits that must change to transform a legitimate pattern to another legitimate pattern
- Example:

  100101

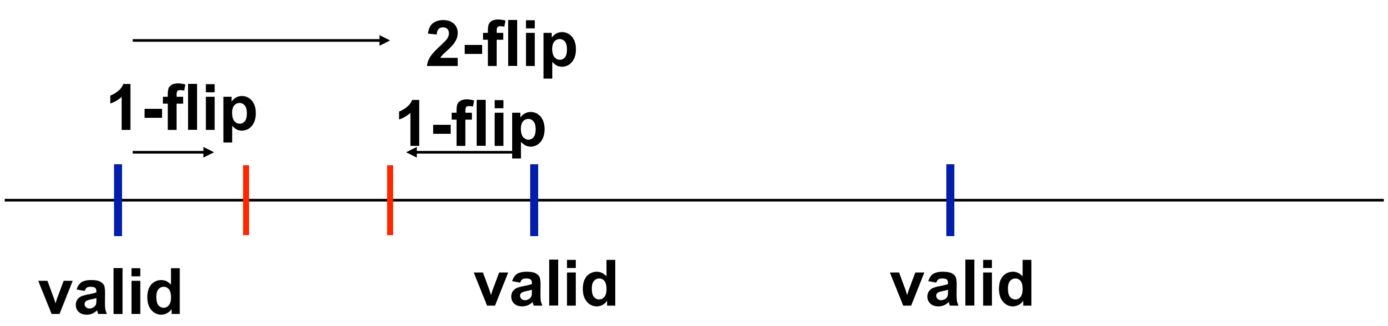
  000111
- Hamming distance of 2
  - Can compute Hamming distance with XOR function and counting number of 1's

Using Hamming codes

- Error detection begins to be possible with distances >=2
- Suppose we find an encoding where the Hamming distance between any two legitimate data patterns is exactly 2
  - A *single* bit flip will cause original data pattern to change such that it has Hamming distance 1
  - When inspecting a pattern, can determine if valid or not since no valid pattern would have the resulting encoding
  - Unfortunately, doesn't help pinpointing the error

Using Hamming codes

- Suppose Hamming distance between valid data patterns is 3
  - A *single* bit flip will cause original data pattern to change such that it has Hamming distances 1 and 2 to nearby valid data patterns
  - When inspecting a pattern, possible to determine if valid or not *and* which pattern is at distance 1
    - Use that pattern as the correct one
  - Unfortunately, if *two bits* flip, will choose incorrect pattern at 1-distance

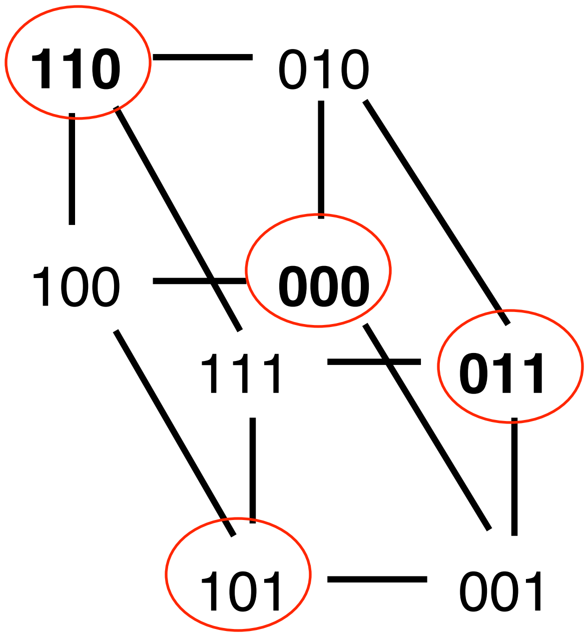


In general

- Hamming code with distance d
  - Can detect d-1 errors
  - Can correct Floor((d-1)/2)) errors
- “Forward” error correction
  - Module generating data creates encoding before transmitting/storing
  - Data can be decoded at destination/reader without contacting creator
  - “Backward” error correction
    - Request from the source, e.g. retry

Example – simple parity

- Hamming distance is 2
- Simple to compute: parity = XOR(data)
- Simple to verify: XOR(data,parity)=0
- Detect up to one error; cannot correct



Example – single bit correction

- Data: 4 bits (P7, P6, P5, P3)
- Code: 3 bits (P4, P2, P1)
  - 2^3 = 8 cases can be encoded
    - No error; error in P1; error in P2; …; error in P7

Choose P_1 so XOR of every other bit $(P_7 \oplus P_5 \oplus P_3 \oplus P_1)$ is 0

Choose P_2 so XOR of every other pair $(P_7 \oplus P_6 \oplus P_3 \oplus P_2)$ is 0

Choose P_4 so XOR of every other four $(P_7 \oplus P_6 \oplus P_5 \oplus P_4)$ is 0

bit	P_7	P_6	P_5	P_4	P_3	P_2	P_1
	$\oplus$		$\oplus$		$\oplus$		$\oplus$
	$\oplus$	$\oplus$			$\oplus$	$\oplus$	
	$\oplus$	$\oplus$	$\oplus$	$\oplus$			

Example

- Data bits:
  - 1001 (P7, P6, P5, P3)
- Code:
  - P4 = P7 xor P6 xor P5 = 1
  - P2 = P7 xor P6 xor P3 = 0
  - P1 = P7 xor P5 xor P3 = 0
- Encoded message:
  - 1001100
- Error detection:
  - Check if P4, P2, P1 invariants hold

Pinpointing error bit

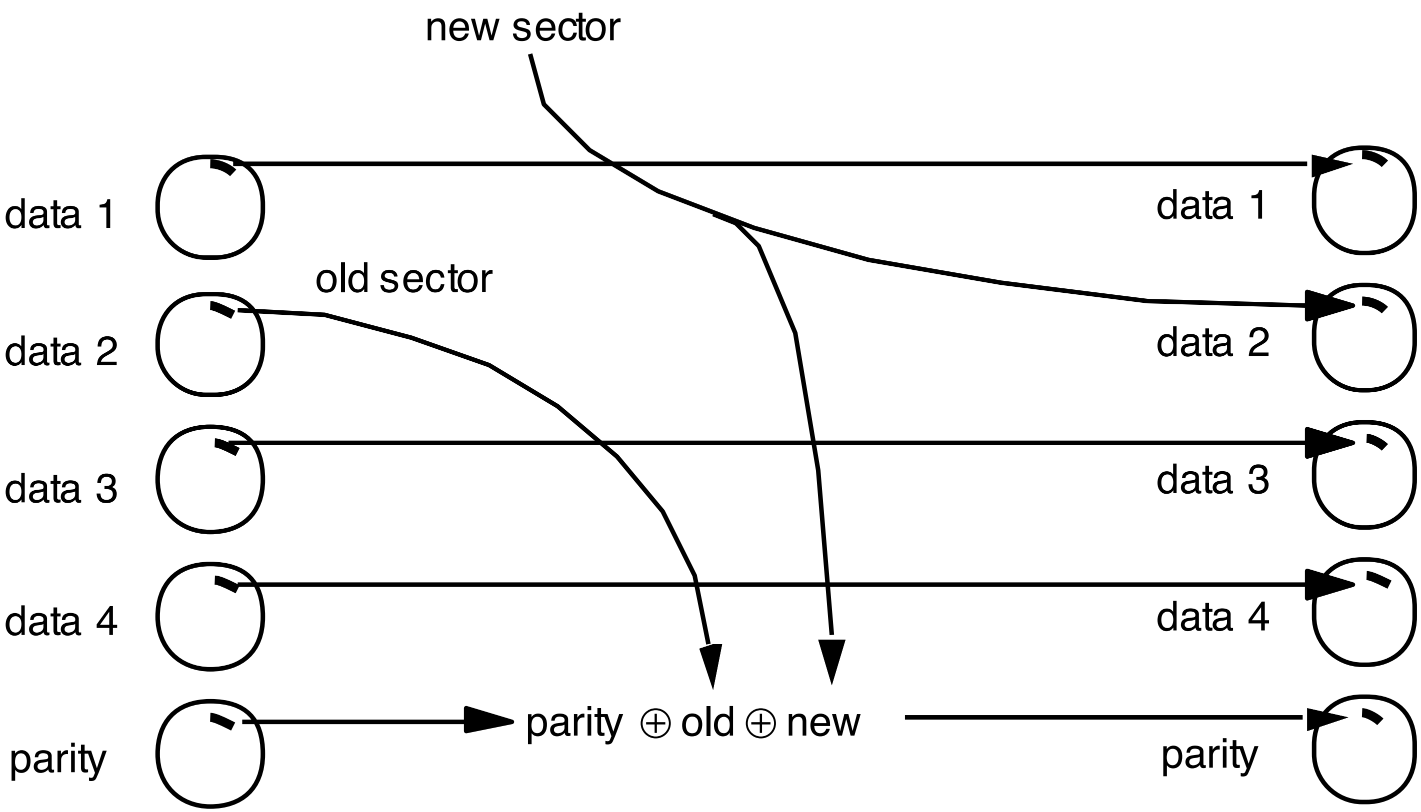
- Encoded message:
  - 1001100
- Error in P7 (i.e. received 0001100)
  - P4, P2, P1 invariants don't hold – error in bit 4+2+1
- Error in P6 (i.e. received 1101100)
  - P4, P2 invariants don't hold – error in bit 4+2
- Error in P5 – P4, P1 invariants don't hold
- Error in P4 – P4 invariant doesn't hold
- Error in P3 – P2, P1 invariants don't hold
- Error in P2 – P2 invariant doesn't hold
- Error in P1 – P1 invariant doesn't hold

Coding to deal with erasures

- If only one item is missing, and system can identify position, parity is sufficient
- Example:
  - Replicated servers - suppose you store 4 consecutive data blocks and parity block:
    - Break 4-block Put(key,blk) into:
      - Put(key_1,blk1), Put(key_2,blk2), Put(key_3,blk3), Put(key_4,blk4)
      - Put(key_p,(blk1 xor blk2 xor blk3 xor blk4)
    - Handle 4-block blk=Get(key) as:
      - Get(key_1); Get(key_2); Get(key_3); Get(key_4)
      - Suppose server for key_2 does not reply:
        - Get(key_p); blk = xor(blk1,blk3,blk4,blkp)

Example: RAID 4

- One sector stores parity
- Update of a sector requires 2 reads (old sector, parity) and two writes (new sector, new parity)
- Assume that disk devices fail-fast; can identify failed sector and recover information from parity



Dealing with erasures

- Example: dealing with packet loss in networks
  - Send multiple numbered packets, with parity information
  - E.g. send 5 packets, four with data and one with parity as in the previous example
  - If 1 out of 5 packets are lost, can recover without retransmission

Forward vs. backward

- Choice depends on application, environment and expected error rates
  - Broadcast – forward error correction avoids multiple retransmission requests
  - Streaming – forward error correction avoids extra round-trip delay for retransmission request
  - One-way communication or very long delays – e.g. space applications

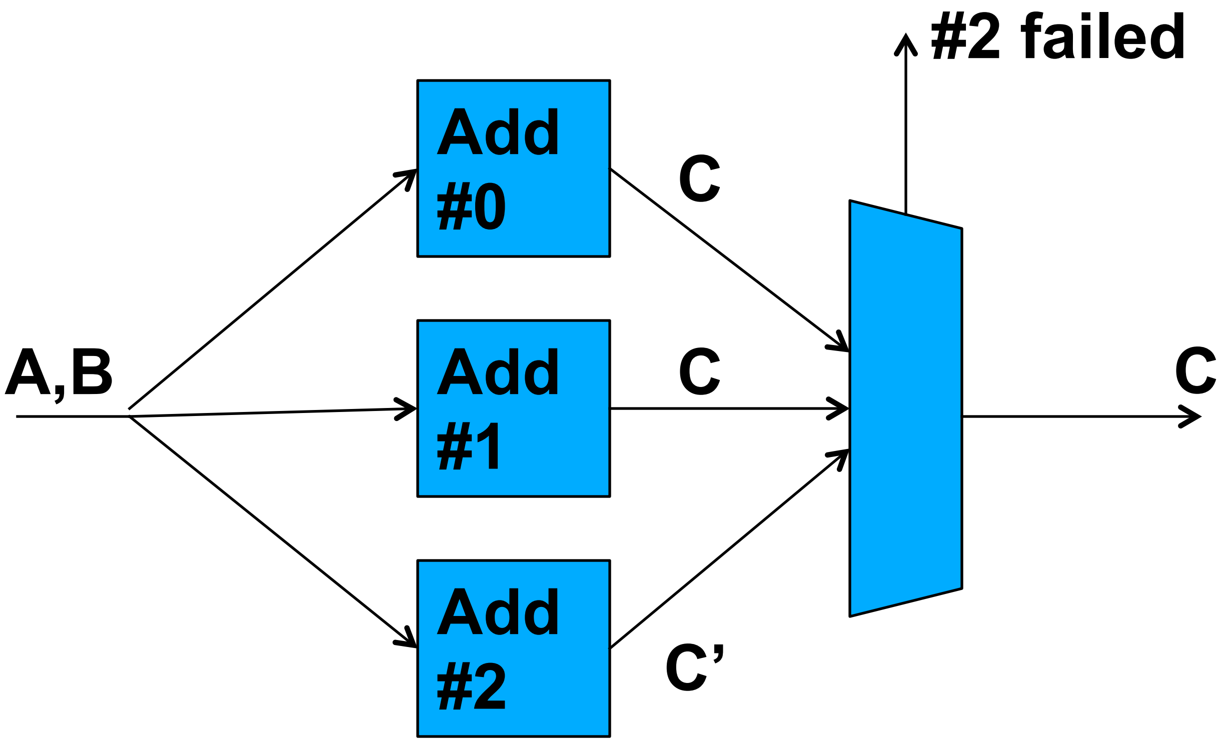
Massive redundancy - replication

- Here, the approach is to use copies (replicas) of components
  - Replicas may pick up the function of a component upon failure
- Creating copies is not sufficient
  - One must interconnect them and provide mechanisms to mask errors if a component fails

Replication in digital logic

- N-modular redundancy (NMR)
  - Example: Triple-modular redundancy (TMR)
  - Substitute a single module with “N” modules which take the same inputs and provide outputs to a “voting” module
    - N replicas plus voting module: “super-module”
  - Example voting module: majority
    - Odd number of replicas, if ((N-1)/2)+1 or more agree, use their output
    - If any replica disagrees, can use this information for maintenance/repair purposes
    - If no majority agrees, report failure
    - Can mask single failure, fail-fast on 2 failures

Example - TMR

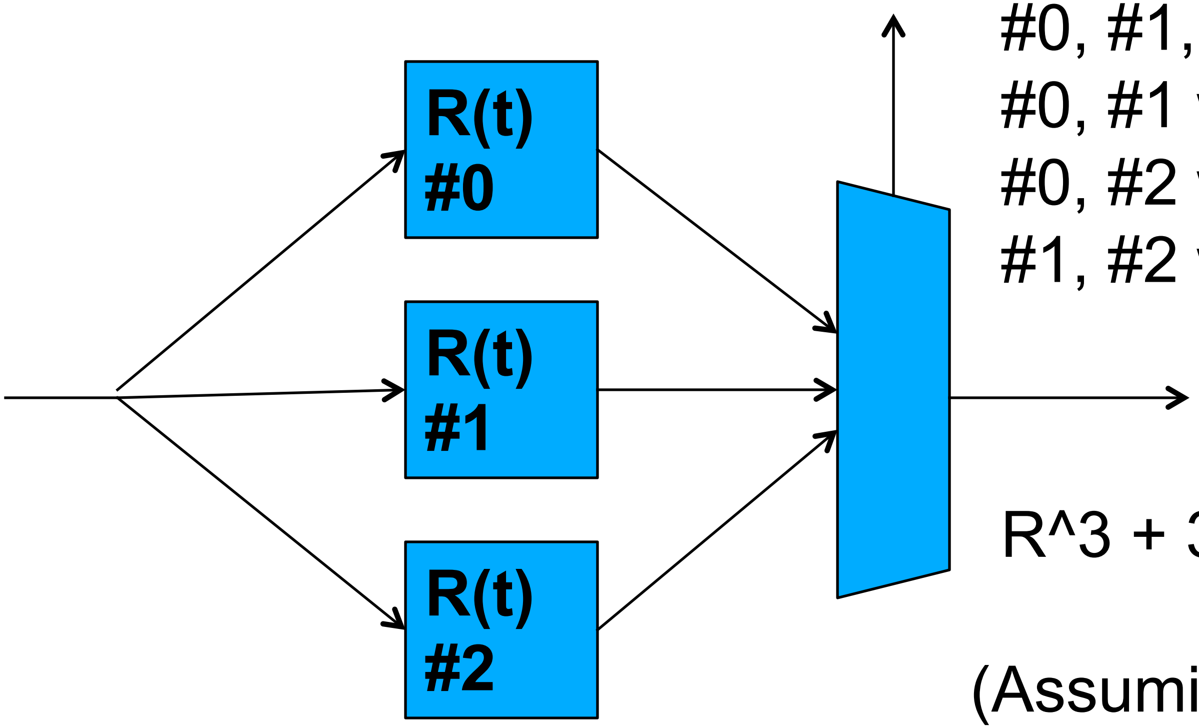


A+B=C

Example: TMR

- Reliability of super-module
  - Single module: reliability R
  - Assume voter is completely reliable
  - Super-module works if all 3 modules work (R^3), or if 2 modules work and one module does not: R^2(1-R)
    - 3 combinations of 2 modules working, one not
    - $R_{supermodule}$ = R^3 + 3*(R^2(1-R)) = 3R^2 – 2R^3
  - Note: possible that two modules provide the same wrong answer and supermodule is not fail-fast

Example - TMR



Supermodule works if:
#0, #1, #2 working, or
#0, #1 working, #2 not, or
#0, #2 working, #1 not, or
#1, #2 working, #0 not

R^3 + 3*(R^2*(1-R))

(Assuming independent failures; and vote, interconnects don't fail)

R(t) = Prob (module not yet failed at time t)

Redundancy and MTTF

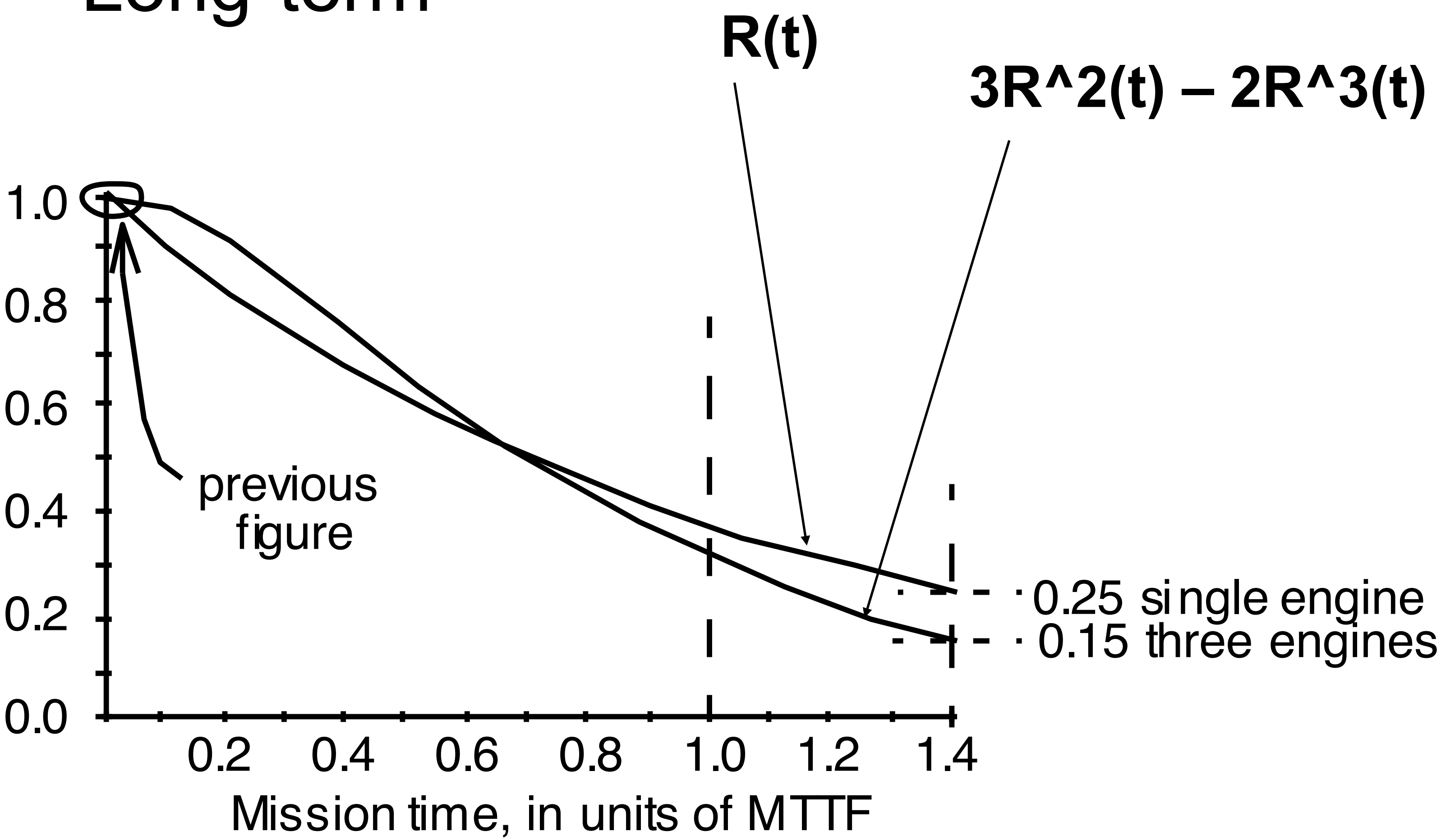
- Keep in mind that redundancy adds components to the system, and more components may lead to lower MTTF
- Example: airplane with three independent, same-model engines
  - MTTF = 6000 hours
- Assume airplane can fly with 2 or 3 engines

Redundancy and MTTF

- Single-engine plane: MTTF 6000 hours
- 3-engine redundant plane
  - Engines fail independently – as a system with 3 engines, mean time to *first* failure is 2000 hours
  - Mean time to second failure: as a system with 2 engines, MTTF 3000 hours
  - MTTF: 5000 hours
- What is missing here?
  - Planes do not fly for that many hours without repair

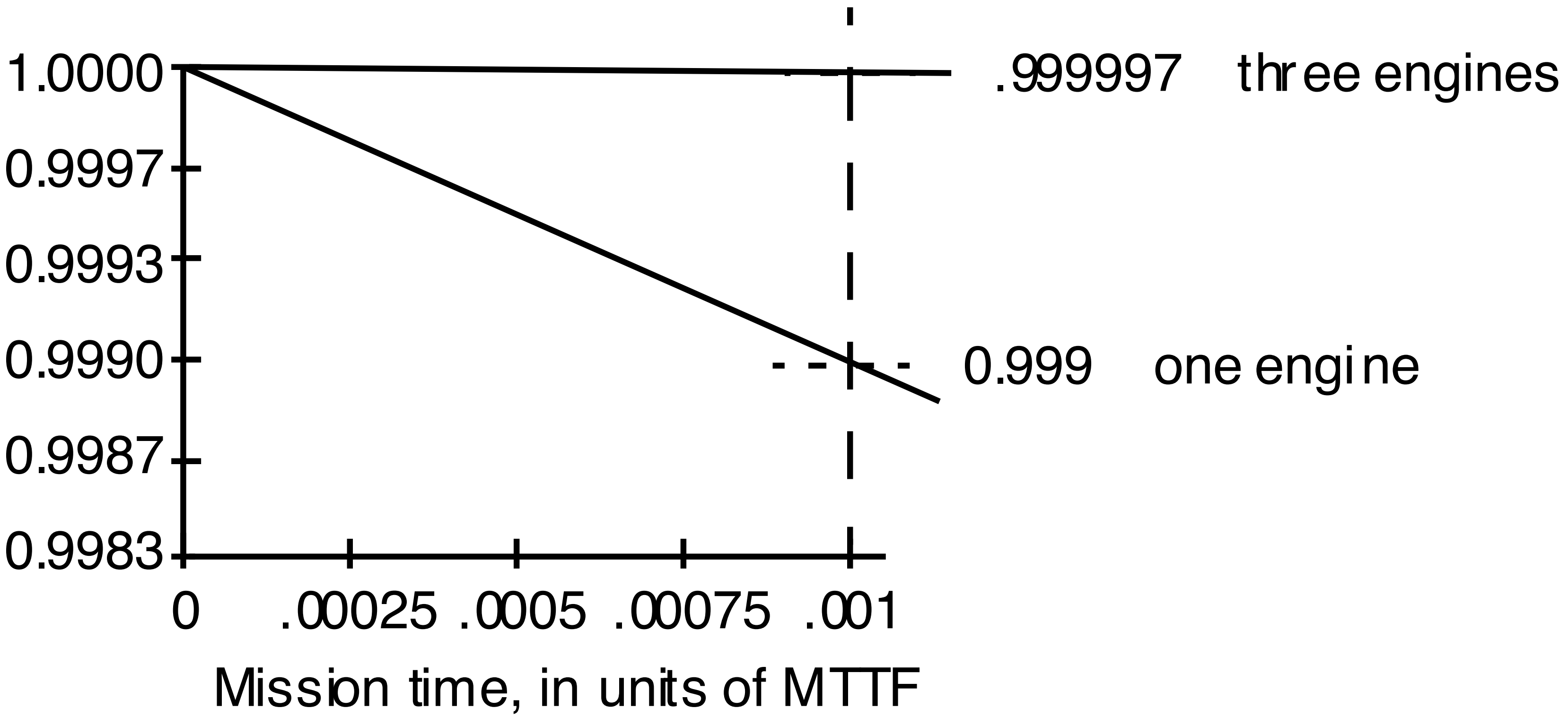
Redundancy and MTTF

- Long-term



Redundancy and MTTF

- Short-term reliability



NMR with repair

- In systems where mission time is long compared to MTTF of replica
  - Simple replication escalates cost and provides little benefit
- However, if modular redundancy is used, and modules that are detected as failed can be repaired
  - The overall MTTF can be substantially improved

NMR with Repair

- Example – triple modular redundancy
  - One module fails; voter is able to detect
  - Initiate repair procedure
    - Fix, or replace
  - Mean probability of super-module failure while repairing
    - Rate of failure of 1 replica: 1/MTTF failures per unit of time
      - 2 replicas: 2/MTTF
    - Time to repair: MTTR
    - Pr,failure = 2*MTTR/MTTF

MTTF

- TMR/vote super-module with memory-less failure and repair processes
  - MTTFsupermodule = (MTTFreplica)^2 / 6*MTTRreplica
    - (No need to memorize/derive this)
- Example: 3-disk array, each disk with MTTF of 5 years. Disks can be repaired/ replaced in 10 hours
  - MTTFsupermodule = 3650 years

Caveats

- Assumptions:
  - Independent failures
    - Correlated failures not uncommon: heat, flood, power, …
  - Memory-less failures
    - Constant failure rate vs ‘bathtub curve’
  - Memory-less repairs
    - Out of stock in spare parts?
  - Perfect repair

Failures in running threads

- Assuming that failures will happen
  - To tolerate failures in software, important to track state of running programs
  - State is distributed - multiple modules
    - Registers, memory, storage, client, server, …
  - Useful classification:
    - “soft state” that can safely abandon upon failure
    - State whose integrity the system should preserve despite failure
  - Sweeping simplification:
    - Thread state (registers, stack, memory): abandonable

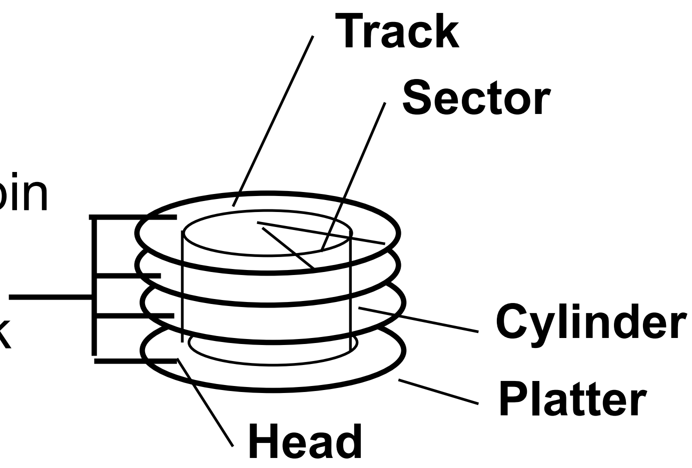
Separating state

- Error containment boundary between module that holds soft state and module that needs to preserve data integrity
- Example:
  - Volatile memory: fine-grain random-access
    - Fast operation; errors can propagate fast, but still within boundary
  - Non-volatile storage: coarse-grain blocks
    - Slow operation; goal is to detect errors before they cross the boundary from volatile to non-volatile

General approach

- Prepare for failure, recognizing that all volatile state in threads can be lost without warning
  - Upon failure, start new threads that restore state in non-volatile storage to a consistent state; recovery
- Protect data in non-volatile storage using replication – durable storage

Non-volatile Magnetic Disks



- One or more platters with magnetic media spin
  - Thousands of RPMs
- Read/write heads – step motors "seek" track
  - Cylinder: all the tracks under the head at a given point on all surface
- A track has multiple sectors – smallest read/write block
  - E.g. 512 Bytes
- Read/write data is a three-stage process:
  - Seek time: position the arm over the proper track
  - Rotational latency: wait for the desired sector to rotate under the read/write head
  - Transfer time: transfer a block of bits (sector) under the read-write head

Magnetic disk fault modes

- Manufacturing defects on surface
  – Sectors that do not reliably store data
  – May also decay over time
  – Sources of hard errors
- Transient mechanical/electrical errors
  – Dust particles; move and retry
- Environmental hazards
  – Bump, ‘head-crash’ – several correlated damaged sectors (hard errors), dust
- Faults in arm positioning system
  – Seek errors

System faults

- Loss of power mid write cycle
  - Partial update of a sector
- Data in volatile memory may be corrupted due to a software fault
  - E.g. runaway O/S code corrupts a buffer of memory that is about to be DMA'ed in to disk

Durable storage systems

- A layered approach to handle faults
  - Lowest layer: *raw storage layer*
  - Second layer: hardware/firmware of disk controller; can detect failures in raw storage layer
    - *Fail-fast storage layer*
  - Third layer: takes advantage of detection capabilities of second layer to create a more reliable storage system
    - Careful storage layer
  - Fourth: *durable storage layer*

Raw disk storage layer

- Simple interface:
  - RAW_SEEK(track)
  - RAW_PUT(data)
  - RAW_GET(data)
- Untolerated errors:
  - Particle dust; incorrect data read/written
  - Defective surface
  - Stored information decayed, no longer correct
  - Correct read/write, but seek error takes to wrong track
  - Power fails during RAW_PUT, partial track update
  - O/S fails during RAW_PUT, corrupted data written

Fail-fast storage layer

- Disk controller divides tracks: sectors
  - Sectors relatively small, and include error detection code, and fixed-size space for track and sector numbers
  - Error detection code allows detection of whether data read was correct or not
  - Writes: verify data integrity by reading from media and comparing with buffer
  - Track/sector numbers allow detection of whether seek ended up in right position

Fail-fast storage layer

- Interface:
  - Status <- FAIL_FAST_SEEK(track)
  - Status <- FAIL_FAST_PUT(data,sector#)
  - Status <- FAIL_FAST_GET(data,sector#)
- Detected errors:
  - FAIL_FAST_GET: code (checksum) check does not verify
    - Could be a soft or hard error
    - No attempt to distinguish; return status=BAD

Fail-fast storage layer

- Detected errors:
  - FAIL_FAST_PUT writes, reads, and buffer does not match – return status=BAD
  - FAIL_FAST_SEEK: read the track number written on the track, does not match – return status=BAD
  - FAIL_FAST_PUT during a power outage – partial write, will not return any value. But a later FAIL_FAST_GET will find a bad sector checksum and return status=BAD
    - Disk may also have sufficient back-up power to complete a write cycle

Fail-fast storage layer

- Untolerable errors
  - O/S fails during RAW_PUT, corrupted data written
  - Data in sector has been corrupted (decay, partial writes) but checksum verification passes
    - Careful in design of verification code so this probability is negligible

Careful storage layer

- Fail-fast detects but does not mask errors – left to the next layer
- Careful layer:
  - Check value of status from each operation
  - Retry to cope with soft errors; re-seek to dislodge dust particles
- Interface:
  - Status <- CAREFUL_SEEK(track)
  - Status <- CAREFUL_PUT(data,sector#)
  - Status <- CAREFUL_GET(data,sector#)

Careful storage layer

- Tolerated errors:
  - Soft errors on reads, writes; seek errors
    - Retry until lower layer returns OK
- Detected errors:
  - Hard errors – persistent after multiple retries; fail with status=BAD, leave to upper layer to deal with
    - Also possible to re-vector/re-map; more later
  - Fail during PUT – checksum from lower layer returns BAD
- Untolerated errors:
  - O/S fails during RAW_PUT, corrupted data written
  - Data in sector has been corrupted (decay, partial writes) but checksum verification passes

Re-vectoring

- Careful layer keeps spare sectors and a mapping table to use in case of hard errors
  - Writes that fail due to a hard error can be retried on spare re-mapped sector and be tolerable
  - Reads of data stored in a block that decays or has a hard fault will be detected, but not masked

Durable layer

- Example: RAID 1
  – Redundant Array of Inexpensive Disks
  – RAID level 1: full replication (mirroring)
- Status <- DURABLE_PUT(data, v_sector#)
- Status <- DURABLE_GET(data, v_sector#)
- PUT writes multiple replicas
  – In separate devices
  – Example: write data to same sector in two replicas
- GET: try to retrieve from any replica
  – If careful layer fails, reads from other replicas
- Repair:
  – Replace failed disk; mirror data into it

Durable layer

- Decay on sectors that store mirrored data would cause an untolerated failure
- One approach: pro-actively check and "refresh" sectors
  - Read sectors every period Td
    - If unavailable, copy from replica
  - Reduce likelihood that bad sectors will go unnoticed until it is not possible to recover
    - Can design Td such that likelihood is negligible

Corrupt data in O/S crashes

- This scenario cannot be handled by the disk subsystem alone
- End-to-end argument – this is better handled by an application
  - Compute and store application-layer checksums on writes
  - Check on reads
- Sector-layer checksum on lower layer has its role – detection/masking of errors related to storage media

Durable storage systems

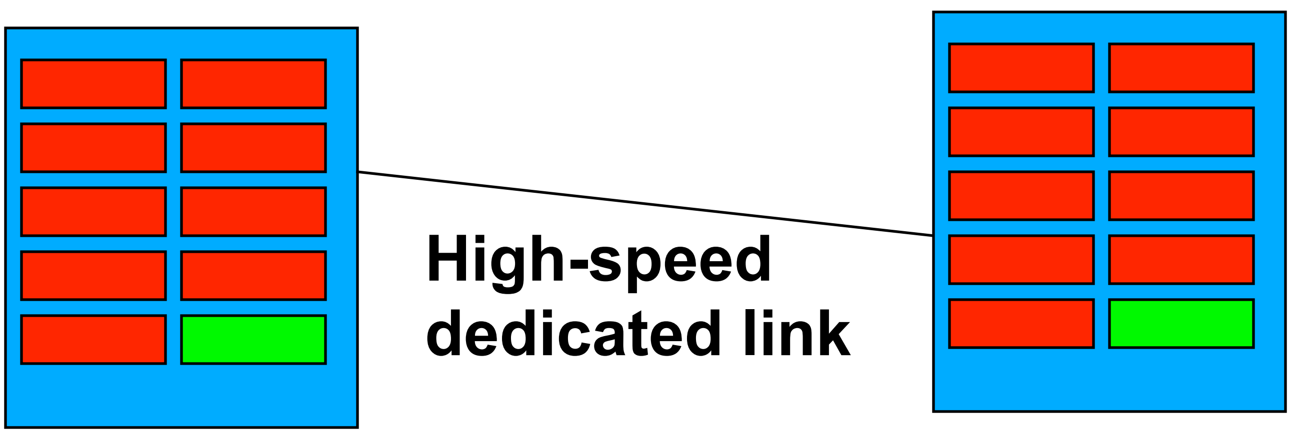
- Early magnetic disks provided the raw storage layer; software dealt with remaining layers
- Modern hard drives provide raw, fail-fast and careful layers in firmware
  - Buffers; retries; sector failure maps;
- RAID systems aggregate multiple hard drives and provide durable layer
  - “Software-RAID” also used

Summary

	raw layer	fail-fast layer	careful layer	durable layer	more durable layer
soft read, write, or seek error	failure	detected	masked		
hard read, write error	failure	detected	detected	masked	
power failure interrupts a write	failure	detected	detected	masked	
single data decay	failure	detected	detected	masked	
multiple data decay spaced in time	failure	detected	detected	detected	masked
multiple data decay within T_d	failure	detected	detected	detected	failure*
undetectable decay	failure	failure	failure	failure	failure*
system crash corrupts write buffer	failure	failure	failure	failure	detected

Enterprise storage

- Replication with RAID4/5
  - Mask single failures with parity blocks
- Spares in disk array spun up “hot-swap” on failure
  - Reduced MTTR
- Geographical replication
  - Avoid correlated failures; disaster recovery



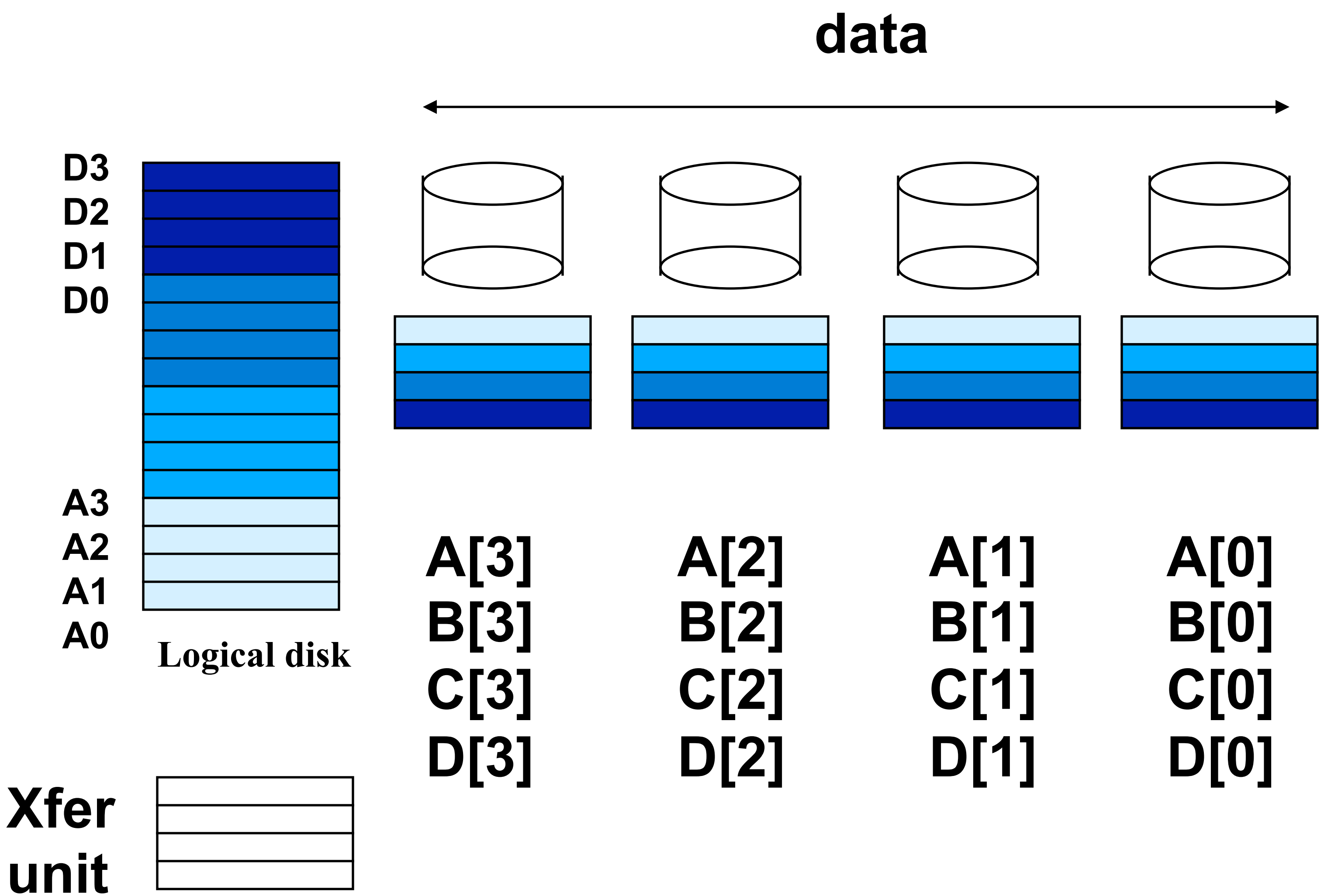
RAID and performance

- Several "levels" of RAID can be implemented and configured in a given controller
  - Tradeoffs in controller complexity, fault tolerance and performance
- RAID0
  - No redundancy – plain disk array (N disks)
    - Best performance, simplest, but not fault-tolerant
- RAID1
  - Mirroring (2*N disks)
    - Good performance, simple, but costly in terms of storage

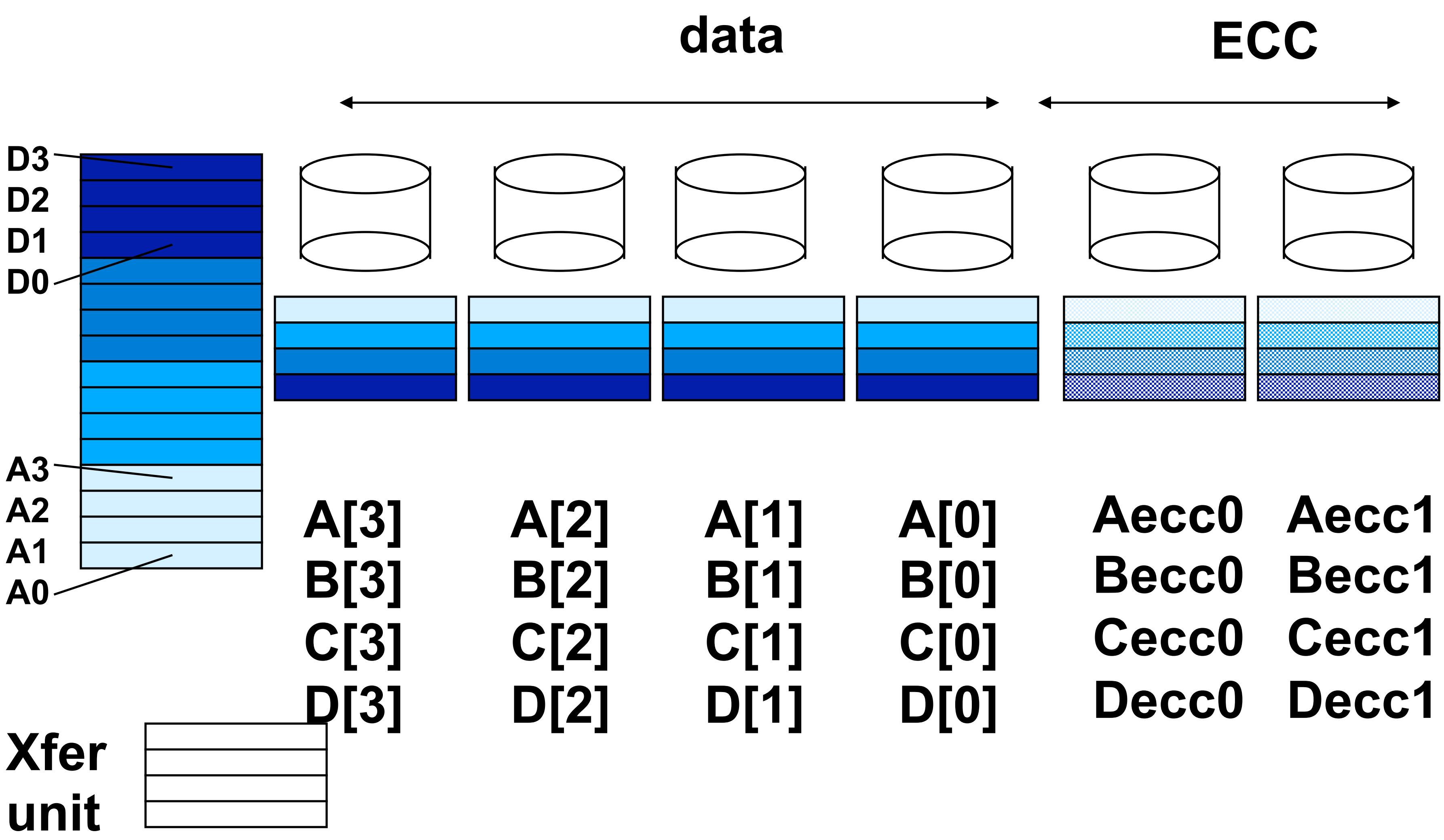
RAID2

- “Bit-interleaving” – data from single block spread across disks
  - Interleaving granularity can vary; bytes
  - RAID3 also bit-interleaved
    - RAID 0, 4, 5 keep a block in a single disk
- Add error correcting codes to data
  - Hamming codes
  - E.g. double error detection, single error correction (DED/SEC)
- N+k disks

Bit/byte interleaving



RAID2: bit interleaving with ECC



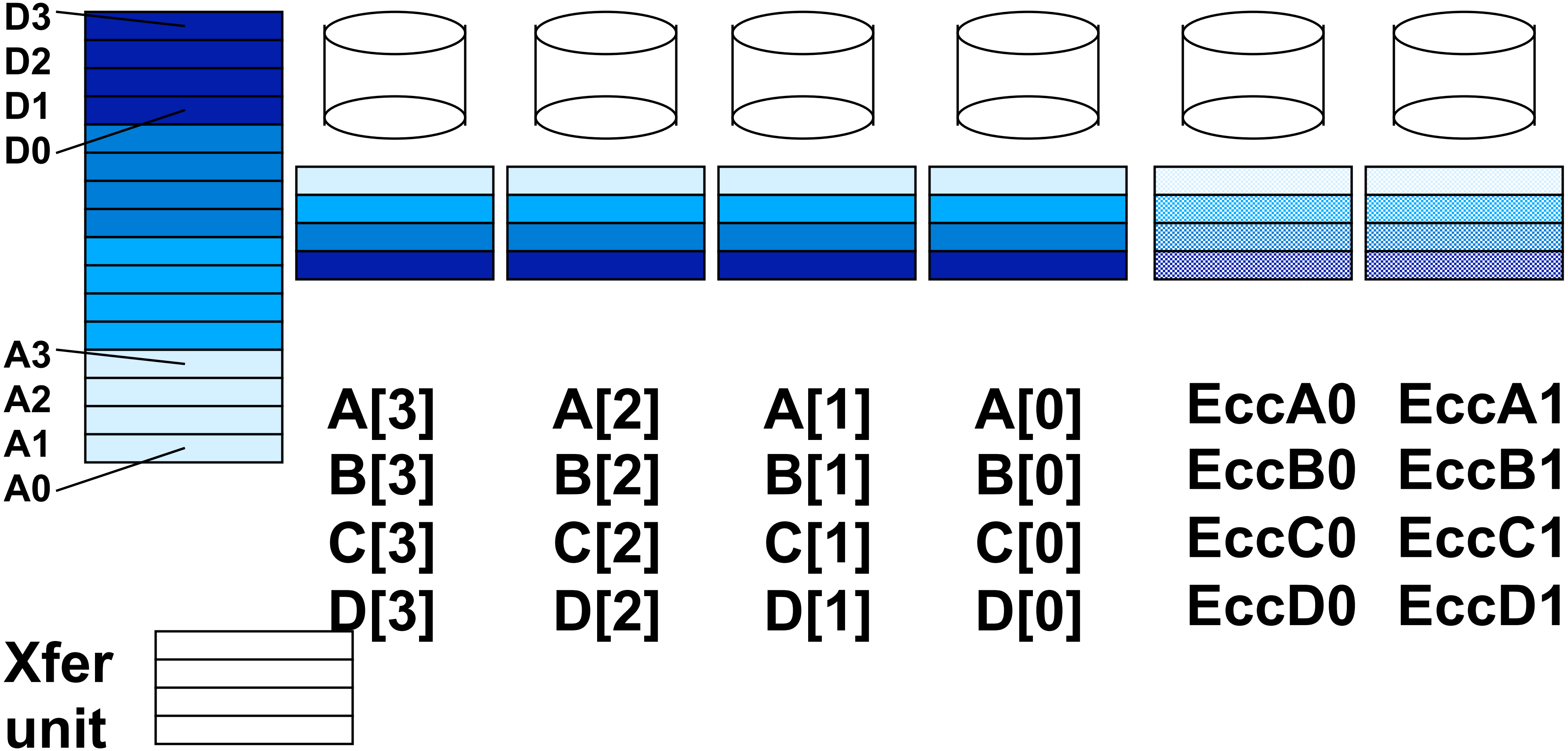
RAID2: reads and writes

- Writes:
  - N bits: generate k-word ECC, write N+k
    - Access all N+k disks on a write
  - 0<M<N: read N-M, generate k-word, write N+k
    - Read, modify, write; again access all disks
- Reads:
  - 0<M≤N: read N+k, check ECC
    - Access all disks on a read too

RAID2 performance

"small" read/write (e.g. of A[ ])

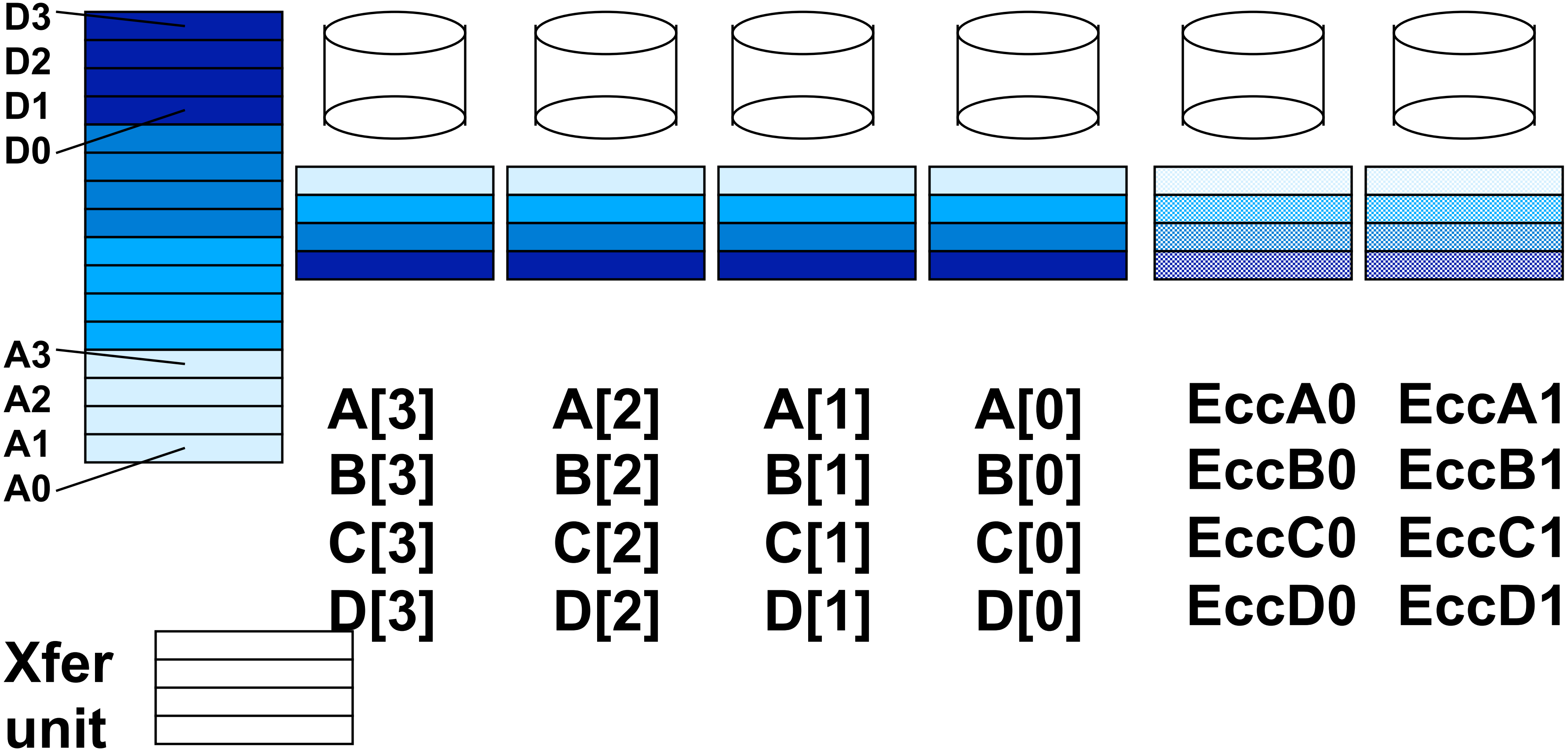
- need to access all disks
- compete for disk with other "small" accesses



RAID2 performance

"large" read/write (e.g. of A[ ] B[ ] C[ ] and D[ ])

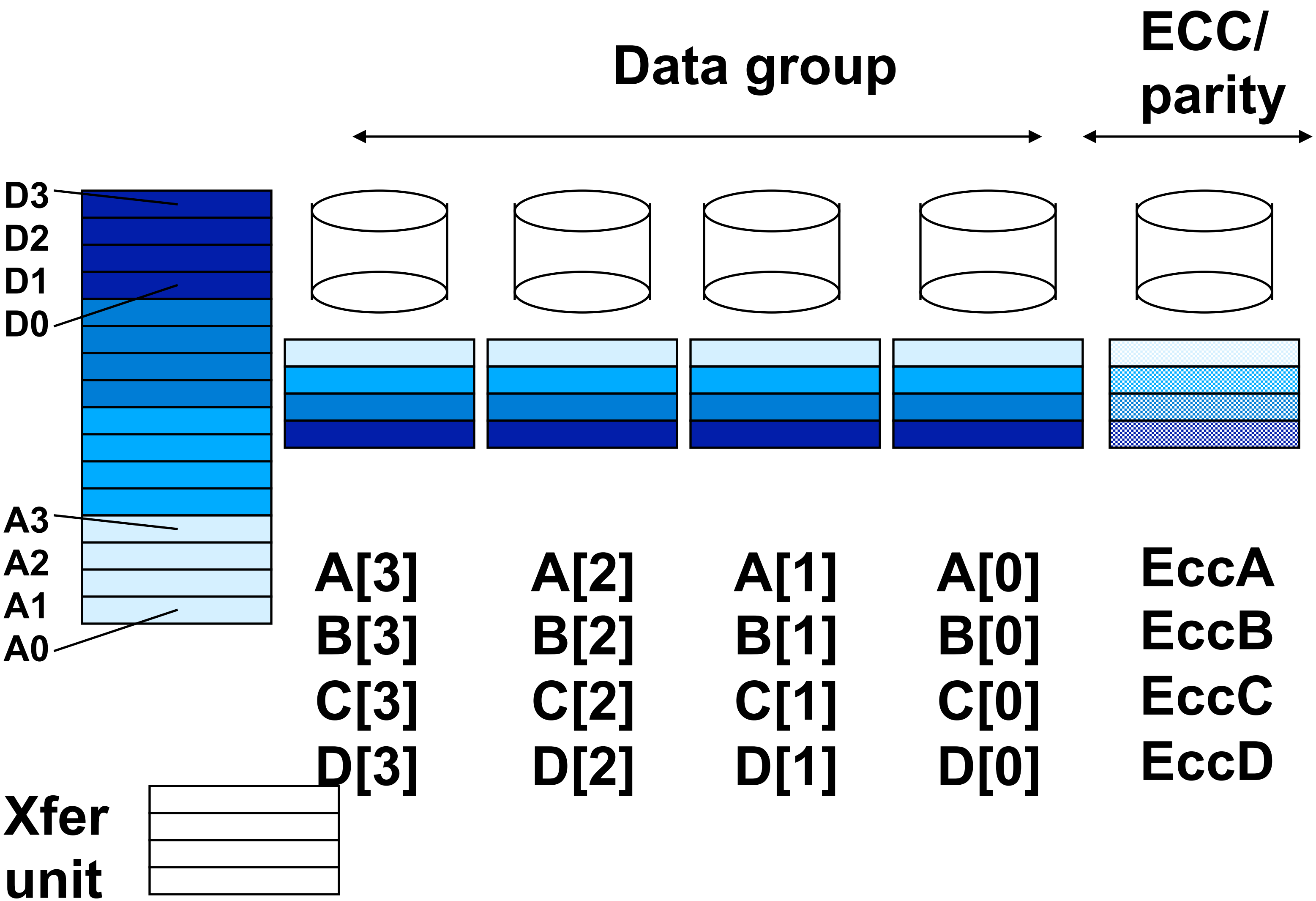
- still need to access all disks
- More data transferred than "small" case, same time -> higher throughput



RAID3

- Key insight:
  - Can detect failures, at per-sector granularity, using the disk layers we have studied
  - Can reduce redundancy overhead by storing a single parity ECC disk (on a “per-group” basis, ~ 10 disks)
    - Still byte interleaving

RAID3


Data group
ECC/
parity
D3
D2
D1
D0
A3
A2
A1
A0
A[3] B[3] C[3] D[3]
A[2] B[2] C[2] D[2]
A[1] B[1] C[1] D[1]
A[0] B[0] C[0] D[0]
EccA
EccB
EccC
EccD
Xfer
unit

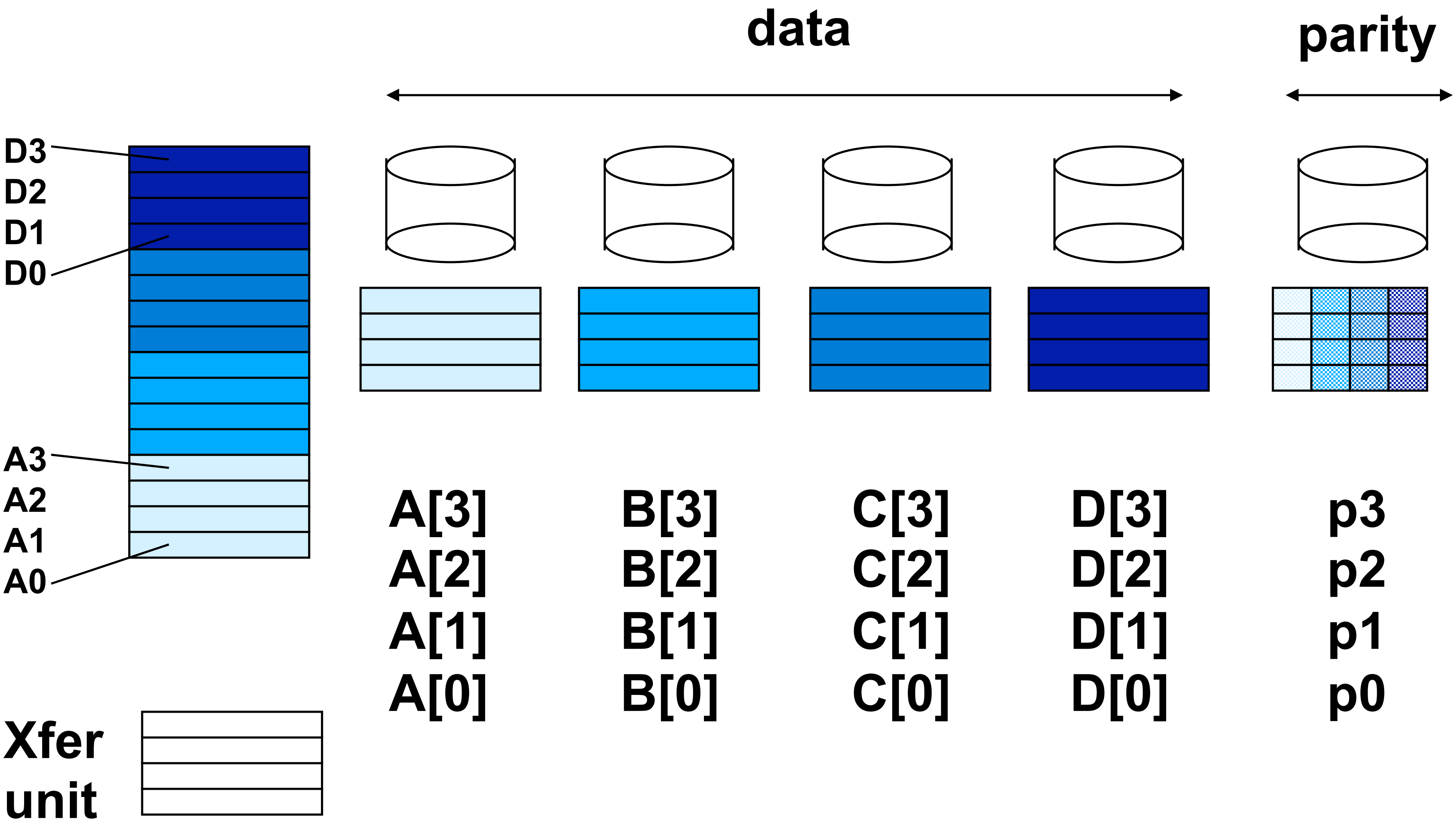
RAID3 vs. RAID2

- RAID3 Reduces storage overhead (1 vs. k) with respect to RAID2, but similar performance
- Both designs have good performance for “large” reads/writes
  – All disks are accessed, but to transfer large amounts of data – throughput/bandwidth
  – Applications: data-intensive (media, supercomputers)
- Transaction processing: “small” R/W
  – All disks are accessed; not all data is used

RAID 4 and 5

- Improve upon RAID2 and RAID3 for “small” accesses
- Compute parity differenly
  - Block-interleaving (sector(s))
  - Avoiding small read/write accesses going to all disks
- RAID 4 and 5 differ in how parity is stored in the disk array

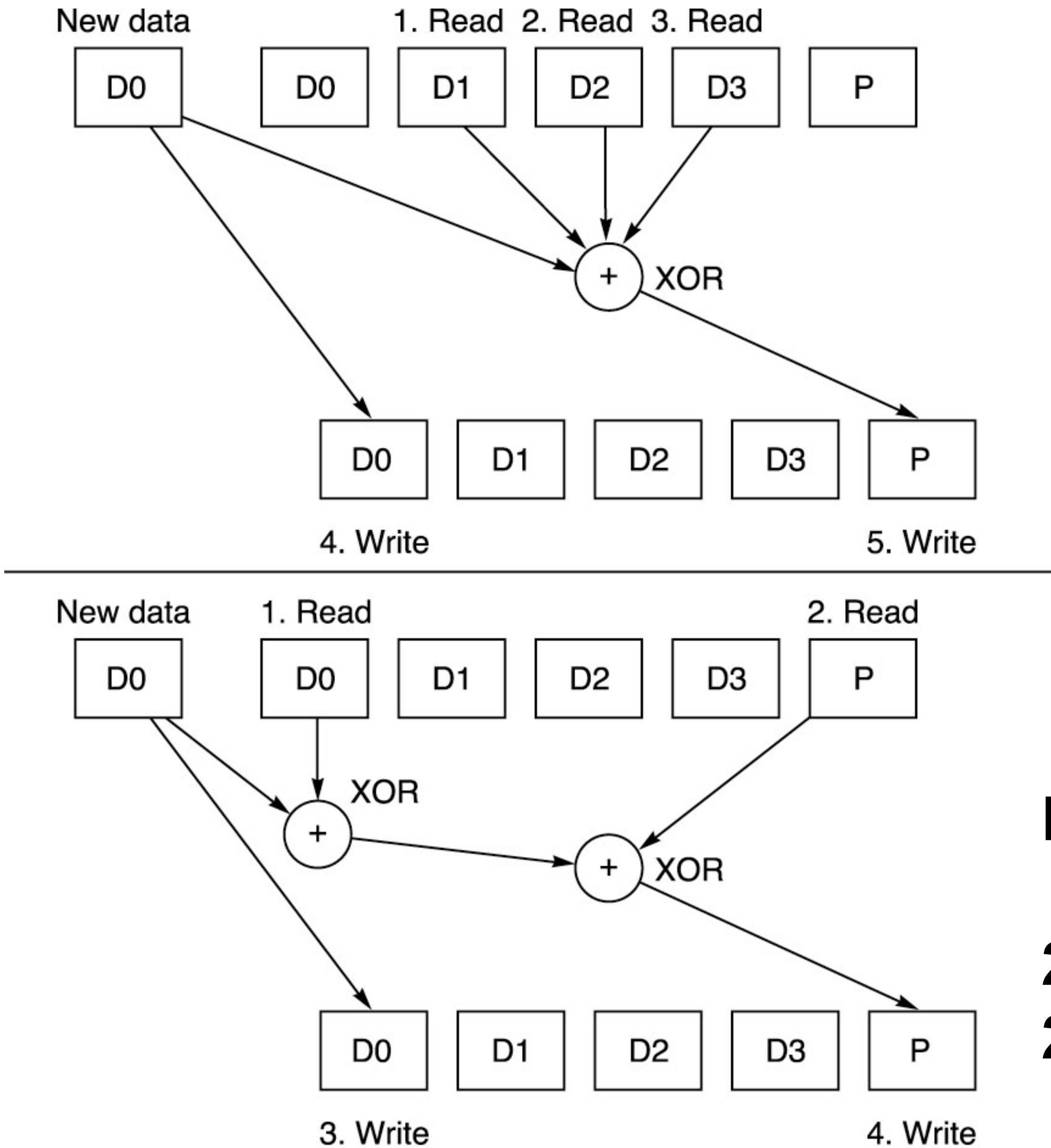
Block interleaving



Small reads/writes

- Small read:
  - Disks have mechanisms to detect errors in a sector
  - Do not access parity disk unless an error has been detected (not the common case)
    - One access to one disk
- Small write:
  - As in RAID 3, must access parity disk to update it
  - However, no need to access other data disks to compute parity

Parity in RAID 4,5



RAID4,5 write:

2 disk reads
2 disk writes

RAID4 vs RAID5

- Where to place the parity information?
  - RAID4: single disk
    - All small writes will serialize in the parity disk
  - RAID5: parity itself is distributed
    - Hope to distribute small writes across multiple disks so they can proceed in parallel

RAID 4 vs. RAID 5