Computer Architecture

Spring 2016

Lecture 08: Caches III

Shuai Wang

Department of Computer Science and Technology

Nanjing University

Improve Cache Performance

- Average memory access time (AMAT):

$$AMAT = Hit\ Time + Miss\ Rate \times Miss\ Penalty$$

- Approaches to improve cache performance
  - Reduce the miss penalty
  - Reduce the miss rate
  - Reduce the miss penalty or miss rate via parallelism
  - Reduce the hit time

Reducing Misses

- Classifying Misses: 3 Cs
  - Compulsory—The first access to a block is not in the cache, so the block must be brought into the cache. Also called cold start misses or first reference misses.

    [Misses in even an Infinite Cache]
  - Capacity—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.

    [Misses in Fully Associative Size X Cache]
  - Conflict—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called collision misses or interference misses.

    [Misses in N-way Associative, Size X Cache]
- More recent, 4th "C":
  - Coherence - Misses caused by cache coherence.

Reduce Miss Rate

- Larger Block Size
- Larger Cache
- Higher Associativity
- Prefetching

Reduce Miss Rate: Compiler Optimizations

- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- Instructions
  - Reorder procedures in memory so as to reduce conflict misses
  - Profiling to look at conflicts (using tools they developed)
- Data
  - Merging Arrays: improve spatial locality by single array of compound elements vs. 2 arrays
  - Loop Interchange: change nesting of loops to access data in order stored in memory
  - Loop Fusion: Combine 2 independent loops that have same looping and some variables overlap
  - Blocking: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows

Merging Arrays Example

```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];
/* After: 1 array of stuctures */
struct merge {
        int val;
        int key;
};
struct merge merged_array[SIZE];
```

- Reducing conflicts between val & key; improve spatial locality

Loop Interchange Example

```
/* Before */
for (k = 0; k < 100; k = k+1)
        for (j = 0; j < 100; j = j+1)
                for (i = 0; i < 5000; i = i+1)
                        x[i][j] = 2 * x[i][j];
/* After */
for (k = 0; k < 100; k = k+1)
        for (i = 0; i < 5000; i = i+1)
                for (j = 0; j < 100; j = j+1)
                        x[i][j] = 2 * x[i][j];
```

- Sequential accesses instead of striding through memory every 100 words; improved spatial locality

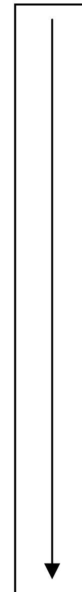
Loop Fusion Example

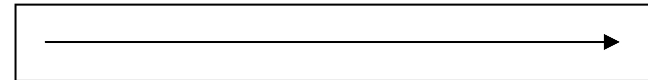
```
/* Before */
for (i = 0; i < N; i = i+1)
        for (j = 0; j < N; j = j+1)
                a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
        for (j = 0; j < N; j = j+1)
                d[i][j] = a[i][j] + c[i][j];
/* After */
for (i = 0; i < N; i = i+1)
        for (j = 0; j < N; j = j+1)
        {       a[i][j] = 1/b[i][j] * c[i][j];
                d[i][j] = a[i][j] + c[i][j];     }
```

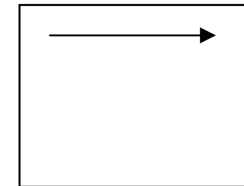
- 2 misses per access to a & c vs. one miss per access; improve spatial locality

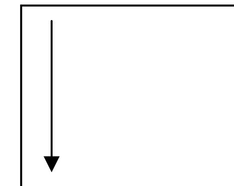
Blocking Example

```
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
    {
        r = 0;
        for (k = 0; k < N; k = k+1)
        {
            r = r + y[i][k]*z[k][j];
        }
            x[i][j] = r;
    }
```









- Two Inner Loops:
  - Read all NxN elements of z[]
  - Read N elements of 1 row of y[] repeatedly
  - Write N elements of 1 row of x[]
- Capacity Misses a function of N & Cache Size:
  - $2N^3 + N^2$ => (assuming no conflict; otherwise …)
- Idea: compute on BxB submatrix that fits

Blocking Example

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
        for (j = jj; j < min(jj+B-1,N); j = j+1){
                r = 0;
                for (k = kk; k < min(kk+B-1,N); k = k+1) {
                        r = r + y[i][k]*z[k][j];}
                x[i][j] = x[i][j] + r;}
```

- B called Blocking Factor
- Capacity Misses from $2N^3 + N^2$ to $N^3/B+2N^2$
- Conflict Misses Too?

Summary: Miss Rate Reduction

- 3 Cs: Compulsory, Capacity, Conflict
  - Reduce Misses via Larger Block Size
  - Reduce misses via Larger Cache Size
  - Reduce Misses via Higher Associativity
  - Reduce Misses via Prefetching
  - Reduce Misses via Compiler Optimizations

Improve Cache Performance

- Average memory access time (AMAT):

$$AMAT = Hit\ Time + Miss\ Rate \times Miss\ Penalty$$

- Approaches to improve cache performance
  - Reduce the miss penalty
  - Reduce the miss rate
  - Reduce the miss penalty or miss rate via parallelism
  - Reduce the hit time

Nonblocking Caches to Reduce Stalls on Cache Misses

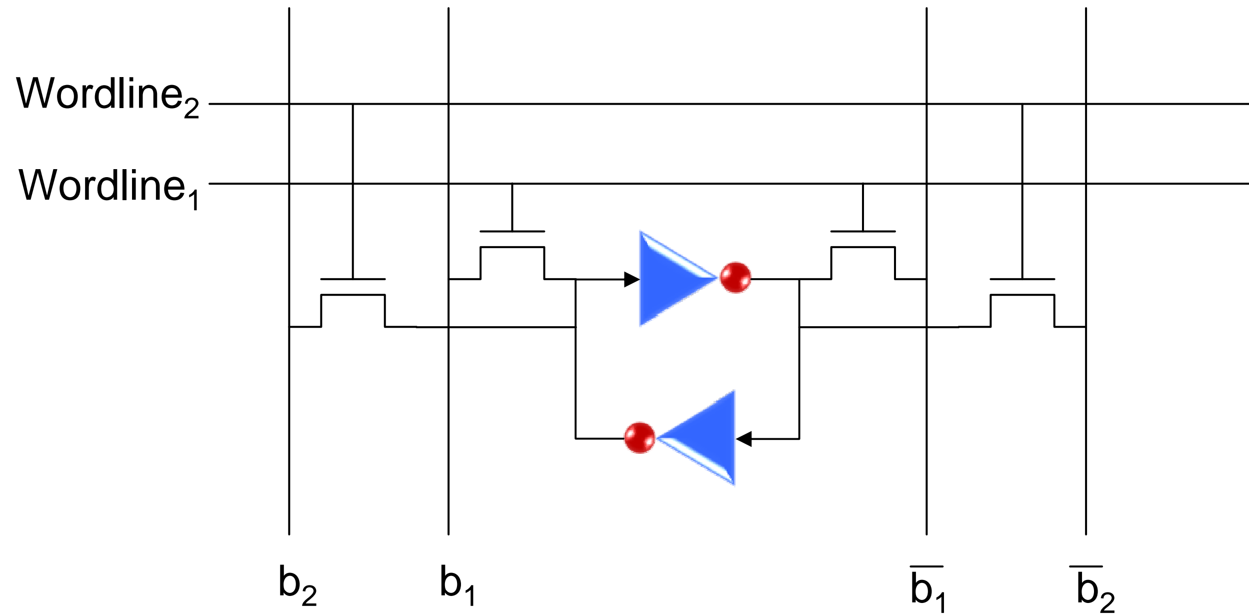
- Non-blocking cache or lockup-free cache allow data cache to continue to supply cache hits during a miss
  - requires F/E bits on registers or out-of-order execution
  - requires multi-bank memories
- "hit under miss" reduces the effective miss penalty by working during miss vs. ignoring CPU requests
- "hit under multiple miss" or "miss under miss" may further lower the effective miss penalty by overlapping multiple misses
  - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
  - Requires multiple memory banks (otherwise cannot support)
  - Pentium Pro allows 4 outstanding memory misses

Multiple Accesses per Cycle

- Need high-bandwidth access to caches
  - Core can make multiple access requests per cycle
  - Multiple cores can access LLC at the same time

- Must either delay some requests, or…
  - Design SRAM with multiple ports
    - Big and power-hungry
  - Split SRAM into multiple banks
    - Can result in delays, but usually not

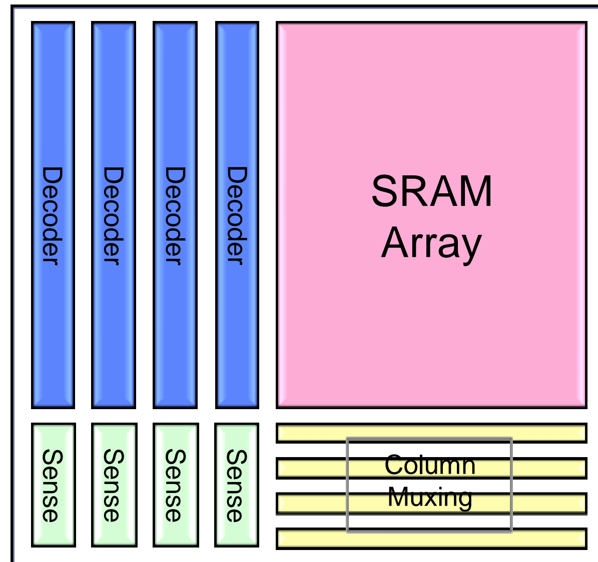
Multi-Ported SRAMs

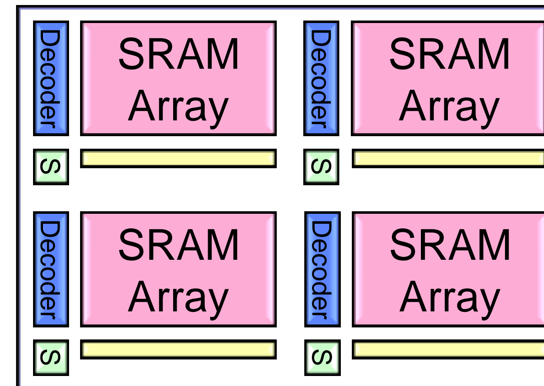


Wordlines = 1 per port
Bitlines = 2 per port

→ Area = $O(\text{ports}^2)$

Multi-Porting vs Banking





4 ports
Big (and slow)
Guarantees concurrent access

4 banks, 1 port each
Each bank small (and fast)
Conflicts (delays) possible

Bank Conflicts

- Banks are *address interleaved*
  - For block size *b cache with N* banks…
  - Bank = (Address / b) % N
    - More complicated than it looks: just low-order bits of index
  - Modern processors perform *hashed cache indexing*
    - May randomize bank and index
    - XOR some low-order tag bits with bank/index bits (why XOR?)

- Banking can provide high bandwidth
- But only if all accesses are to different banks
  - For 4 banks, 2 accesses, chance of conflict is 25%

Improve Cache Performance

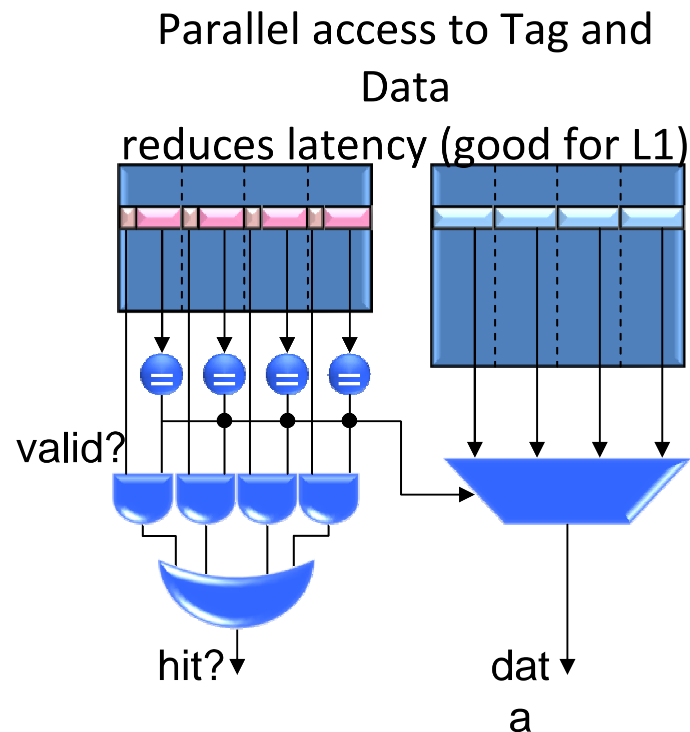
- Average memory access time (AMAT):

$$AMAT = Hit\ Time + Miss\ Rate \times Miss\ Penalty$$

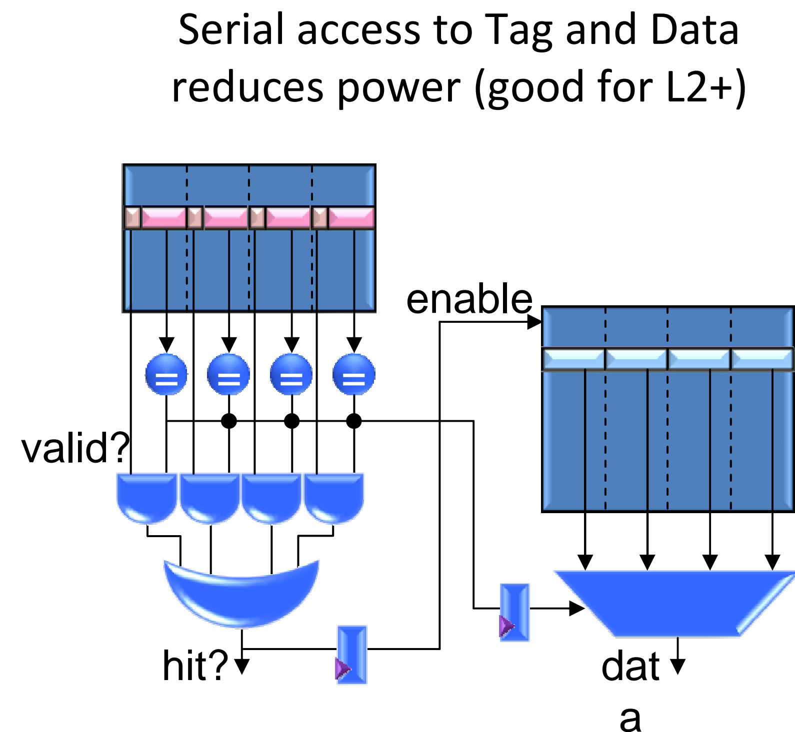
- Approaches to improve cache performance
  - Reduce the miss penalty
  - Reduce the miss rate
  - Reduce the miss penalty or miss rate via parallelism
  - Reduce the hit time

Reducing Hit Time: Parallel vs Serial Caches

- Tag and Data usually separate (tag is smaller & faster)
  - State bits stored along with tags
    - *Valid* bit, “LRU” bit(s), …



Parallel access to Tag and Data reduces latency (good for L1)



Serial access to Tag and Data reduces power (good for L2+)

Way-Prediction

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?
- Way prediction: extra bits are kept in cache to predict which way to try on the next cache access
- Correct prediction delivers access latency as for a direct-mapped cache
- Otherwise, other ways are tries in subsequent clock cycles

Physically-Indexed Caches

- Core requests are VAs
- Cache index is PA[15:6]
  - VA passes through TLB
  - D-TLB on critical path
- Cache tag is PA[63:16]
- If index size < page size
  - Can use VA for index

Virtual Address

tag[63:14] | index[13:6] | block offset[5:0]

virtual page[63:13] | page offset[12:0]

D-TLB

index[6:0]

physical index[7:0]

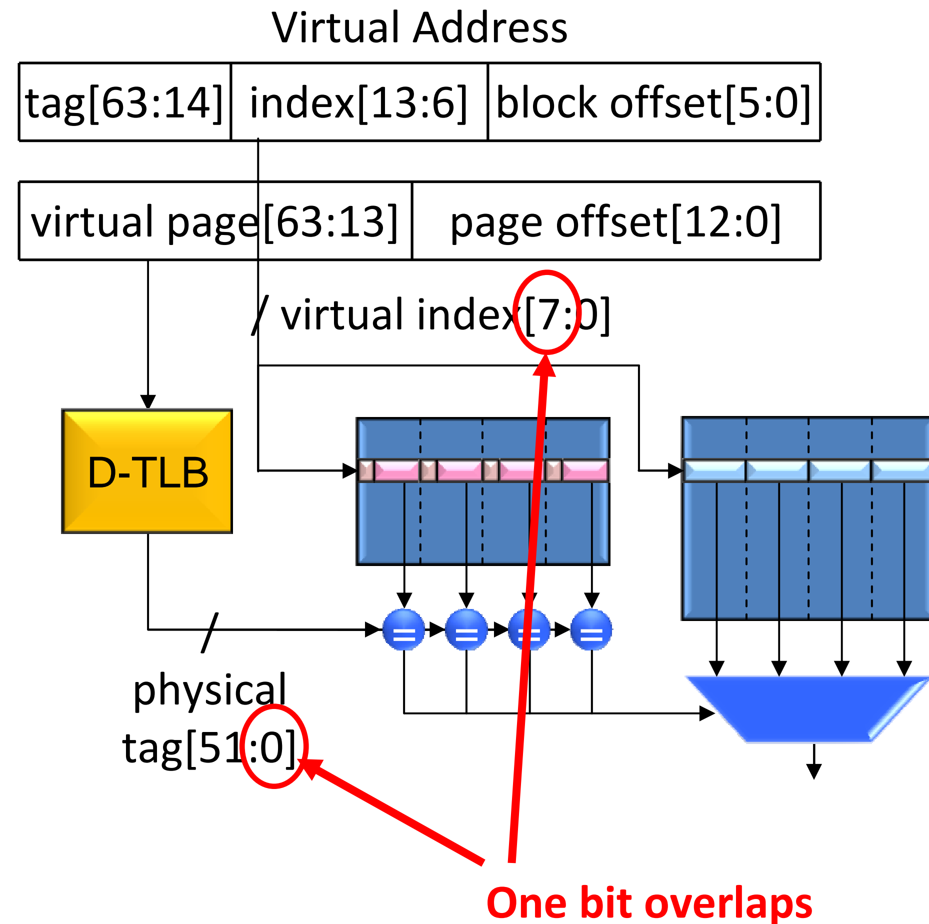
physical index[0:0]

physical tag[51:1]

Virtually-Indexed Caches

- Core requests are VAs
- Cache index is VA[15:6]
- Cache tag is PA[63:16]

- Why not tag with VA?
  - Cache flush on ctx switch
- Virtual aliases
  - Ensure they don't exist
  - … or check all on miss

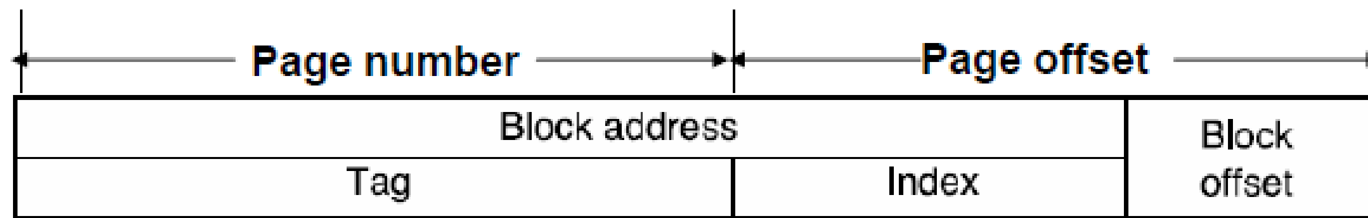


Reducing Hit Time: Avoiding Address Translation during Indexing of the Cache

- Send virtual address to cache? Called Virtually Addressed Cache or just Virtual Cache vs. Physical Cache
  - Every time process is switched logically must flush the cache; otherwise get false hits
    - Cost is time to flush + "compulsory" misses from empty cache
  - Dealing with aliases (sometimes called synonyms);
    - Two different virtual addresses map to same physical address
  - I/O must interact with cache, so need virtual address
- Solution to aliases
  - HW guarantees covers index field & direct mapped, they must be unique; called page coloring
- Solution to cache flush
  - Add process identifier tag that identifies process as well as address within process: can't get a hit if wrong process

Reducing Hit Time: Avoiding Address Translation during Indexing of the Cache

- Virtually indexed, physically tagged caches
  - If index is physical part of address, can start tag access in parallel with translation so that can compare to physical tag



- Limits cache to page size: what if want bigger caches and uses same trick?
  - Higher associativity moves barrier to right
  - Page coloring

Reducing Hit Time: Pipelined Cache Access

- Pipelining cache access in multiple cycles
  - Gives fast clock cycle time, but slow cache hits
- Disadvantages of pipelined cache access
  - Increase the pipeline depth
  - Increase the misprediction penalty
  - Increase the clock cycles between the issue of the load and use of the data
- Pipelining increases the bandwidth of instructions

Reducing Hit Time: Trace Caches

- Wider superscalar datapath demands higher instruction fetch bandwidth
- Instruction fetch bandwidth is limited inst. Cache bandwidth, branch predictor, dynamic control flow
- Inst. Cache stores static code sequences generated by compiler, different from the dynamic execution sequences
- The idea of trace cache: capture and store dynamic inst. Sequences in trace cache
- Improve inst. Cache hit time by fetching multiple noncontinuous basic blocks in one fetch cycle, which would require multiple fetch cycles in the conventional cache.

Write Policies

- Writes are more interesting
  - On reads, tag and data can be accessed in parallel
  - On writes, needs two steps
  - Is access time important for writes?

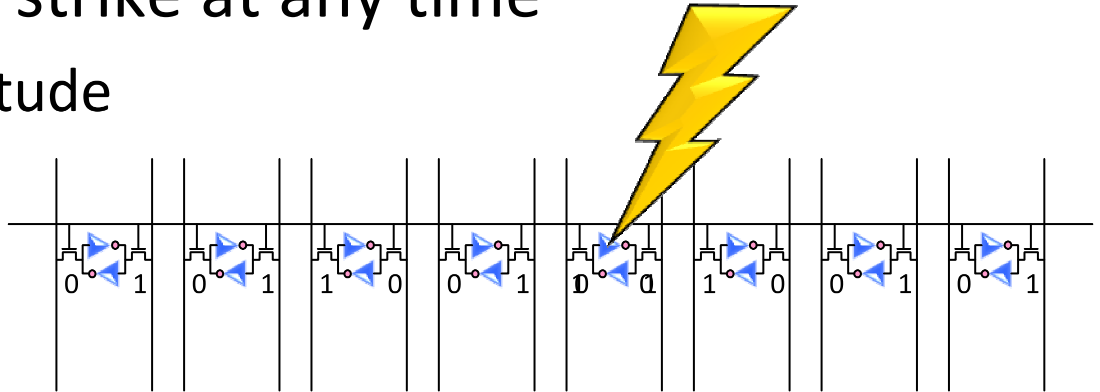
- Choices of Write Policies
  - On write hits, update memory?
    - Yes: *write-through* (higher bandwidth)
    - No: *write-back* (uses *Dirty* bits to identify blocks to write back)
  - On write misses, allocate a cache block frame?
    - Yes: *write-allocate*
    - No: *no-write-allocate*

Inclusion

- Core often accesses blocks not present on chip
  - Should block be allocated in L3, L2, and L1?
    - Called *Inclusive* caches
    - Waste of space
    - Requires forced evict (e.g., force evict from L1 on evict from L2+)
  - Only allocate blocks in L1
    - Called *Non-inclusive* caches (why not "exclusive"?)
    - Must write back clean lines
- Some processors combine both
  - L3 is inclusive of L1 and L2
  - L2 is non-inclusive of L1 (like a large victim cache)

Parity & ECC

- Cosmic radiation can strike at any time
  - Especially at high altitude
  - Or during solar flares



- What can be done?
  - *Parity*
    - 1 bit to indicate if sum is odd/even (detects single-bit errors)
  - *Error Correcting Codes (ECC)*
    - 8 bit code per 64-bit word
    - Generally *SECDED (Single-Error-Correct, Double-Error-Detect)*
- Detecting errors on clean cache lines is harmless
  - Pretend it's a cache miss