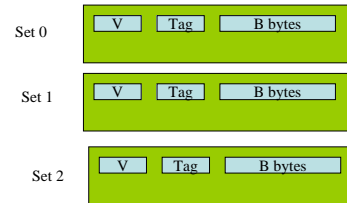


CS211 Computer Architecture Cache Memories

- Topics
 - Cache organization
 - Caching performance

Direct Mapped caches

- Each set has exactly 1 line ($E=1$)
- Cache has an array of $S=2^s$ cache sets
- Each line contains a data block of size $B=2^b$ bytes
- Memory address $M = 2^m$ has m bits to specify an address
- Cache size $C=S \times E \times B$

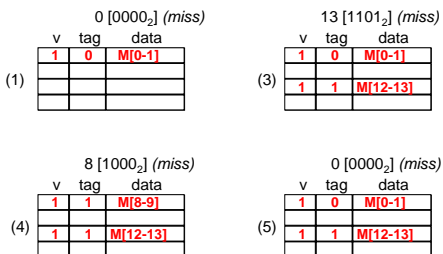


Direct-Mapped Cache Simulation

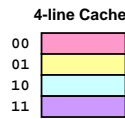
$t=1$	$s=2$	$b=1$
x	xx	x

**M=4 bit addresses, B=2 bytes/block,
S=4 sets (s=2), E=1 entry/set**

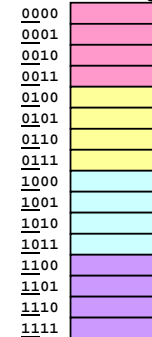
**Address trace (reads):
0 [0000₂], 1 [0001₂], 13 [1101₂], 8 [1000₂], 0 [0000₂]**



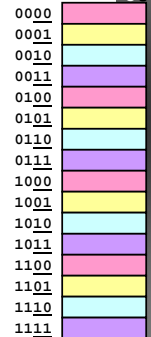
Why Use Middle Bits as Index?



**High-Order
Bit Indexing**



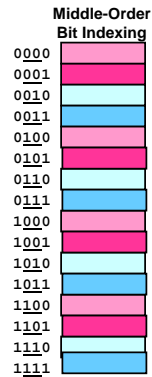
**Lower-Order
Bit Indexing**



- High-Order Bit Indexing
 - Adjacent memory lines would map to same cache entry
 - Poor use of spatial locality
- Lower-Order Bit Indexing
 - Consecutive memory lines map to different cache lines
 - Can hold C-byte region of address space in cache at one time

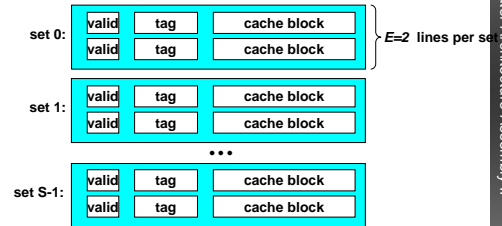
Why index with middle bits?

- Lower bits correspond to blocks within each cache line
- Middle bits correspond to the set
- Higher order bits stored in tags
- A cache can hold a contiguous set of words = $S \times B$ in cache



Set Associative Caches

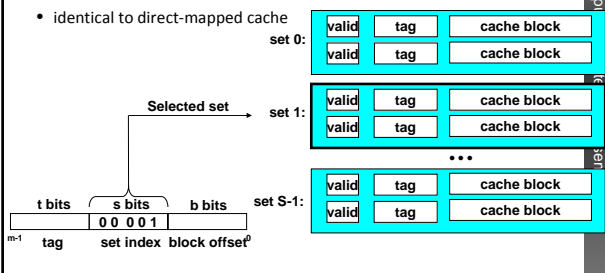
- Characterized by more than one line per set



Accessing Set Associative Caches

- Set selection

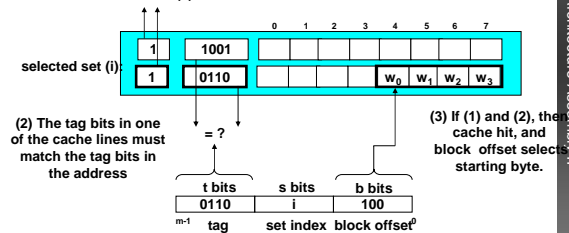
- identical to direct-mapped cache



Accessing Set Associative Caches

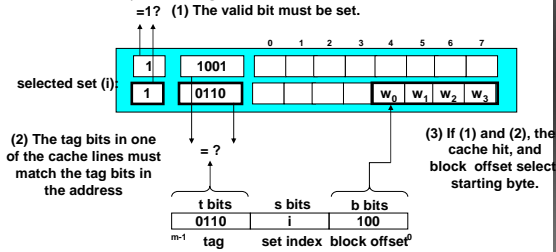
- Line matching and word selection

- must compare the tag in each valid line in the selected set.



Accessing Set Associative Caches

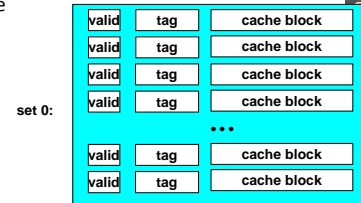
- Line matching and word selection
 - must compare the tag in each valid line in the selected set.



Why all of this extra work? Any other issues? Policy?

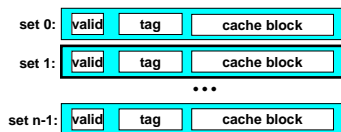
Fully Associative Caches

- Set selection is trivial.
- Accessing a line is the same as a set associative cache
 - Difference in scale



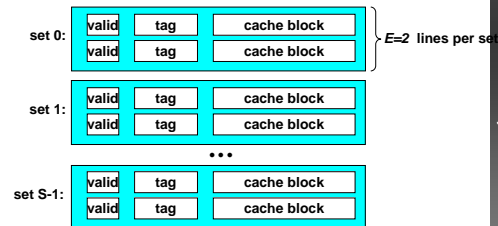
Example: Direct mapped cache

- 32 bit address, 64KB cache, 32 byte block
- How many sets, how many bits for the tag, how many bits for the offset?



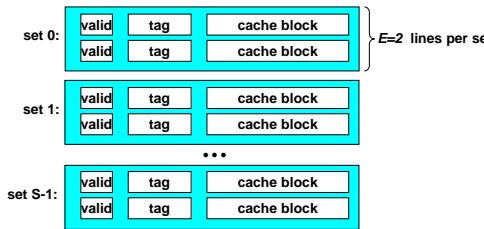
Example: 2-way associative cache

- 32 bit address, 64KB cache, 32 byte block
- How many sets (lines), how many bits for the tag, how many bits for the offset?



Example: 2-way associative cache

- 32 bit address, 32KB cache, 16 byte block
- How many sets, how many bits for the tag, how many bits for the offset?



Write-through vs write-back

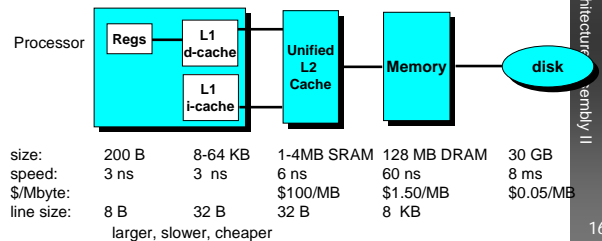
- What to do when an update occurs?
- Write-through: immediately
 - Simple to implement, synchronous write
 - Uniform latency on misses
- Write-back: write when block is replaced
 - Requires additional dirty bit or modified bit
 - Asynchronous writes
 - Non-uniform miss latency
 - Clean miss: read from lower level
 - Dirty miss: write to lower level and read (fill)

Writes and Cache

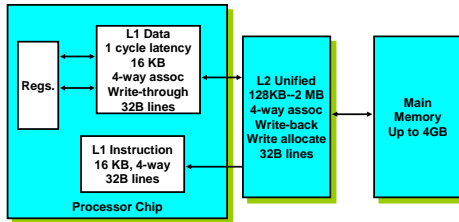
- Reading information from a cache is straight forward.
- What about writing?
 - What if you're writing data that is already cached (**write-hit**)?
 - What if the data is not in the cache (**write-miss**)?
- Dealing with a write-hit.
 - **Write-through** - immediately write data back to memory
 - **Write-back** - defer the write to memory for as long as possible
- Dealing with a write-miss.
 - **write-allocate** - load the block into memory and update
 - **no-write-allocate** - writes directly to memory
- Benefits? Disadvantages?
- **Write-through** are typically **no-write-allocate**.
- **Write-back** are typically **write-allocate**.

Multi-Level Caches

- Options: separate **data** and **instruction caches**, or a **unified cache**



Intel Pentium Cache Hierarchy



Cache Performance Metrics

- **Miss Rate**
 - Fraction of memory references not found in cache (misses/references)
 - Typical numbers:
 - 3-10% for L1
 - can be quite small (e.g., < 1%) for L2, depending on size, etc.
- **Hit Time**
 - Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
 - Typical numbers:
 - 1 clock cycle for L1
 - 3-8 clock cycles for L2
- **Miss Penalty**
 - Additional time required because of a miss

Writing Cache Friendly Code

- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
- Examples:
 - cold cache, 4-byte words, 4-word cache blocks

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = $1/4 = 25\%$

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = **100%**

Matrix Multiplication Example

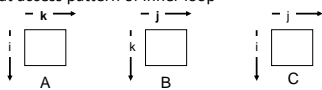
- Major Cache Effects to Consider
 - Total cache size
 - Exploit temporal locality and keep the running total (e.g., `sum`)
 - Block size
 - Exploit spatial locality
- Description:
 - Multiply $N \times N$ matrices
 - $O(N^3)$ total operations
 - Accesses
 - N reads per source element
 - N values summed per destination
 - but may be able to hold in register

```
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Variable sum held in register

Miss Rate Analysis for Matrix Multiply

- Assume:
 - Line size = 32BYTES (big enough for 4 64-bit words)
 - Matrix dimension (N) is very large
 - Approximate $1/N$ as 0.0
 - Cache is not even big enough to hold multiple rows
- Analysis Method:
 - Look at access pattern of inner loop



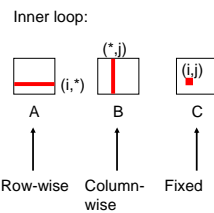
Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
 - each row in contiguous memory locations
- Stepping through columns in one row:
 - for (i = 0; i < N; i++)
 - sum += a[0][i];
 - accesses successive elements
 - if block size (B) > 4 bytes, exploit spatial locality
 - compulsory miss rate = 4 bytes / B
- Stepping through rows in one column:
 - for (i = 0; i < n; i++)
 - sum += a[i][0];
 - accesses distant elements
 - no spatial locality!
 - compulsory miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```

/* ijk */
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
    
```



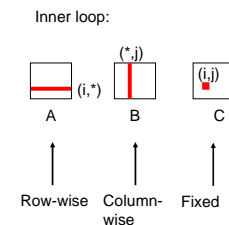
- Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```

/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
    
```



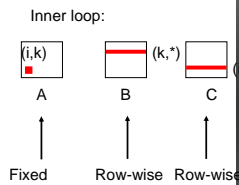
- Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```

/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
    
```



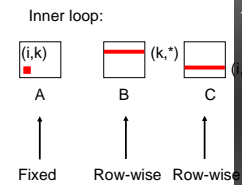
• Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```

/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
    
```



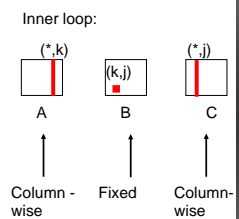
• Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```

/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
    
```



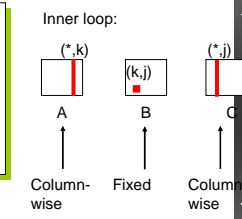
• Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```

/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
    
```



• Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = **1.25**

kij (& ikj):

- 2 loads, 1 store
- misses/iter = **0.5**

jki (& kji):

- 2 loads, 1 store
- misses/iter = **2.0**

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

```
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

```
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

Concluding Observations

- Programmer can optimize for cache performance
 - How data structures are organized
 - How data are accessed
 - Nested loop structure
 - Blocking is a general technique
- All systems favor “cache friendly code”
 - Getting absolute optimum performance is very platform specific
 - Cache sizes, line sizes, associativities, etc.
 - Can get most of the advantage with generic code
 - Keep working set reasonably small (temporal locality)
 - Use small strides (spatial locality)