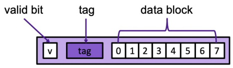
Checkin 3 Prep

Mon. Oct. 30

Caching:

Cache Lines



- **data block:** cached data (i.e., copy of bytes from memory)
- **tag:** uniquely identifies which data is stored in the cache line
- **valid bit:** indicates whether or not the line contains meaningful information

address ō data (after hex to bin conversion):

tag	index	offset
The rest ō ẽ bits	log(n° ō cache lines) bits	log(block size) bits

Spacial locality: close items in memory...

Temporal locality: referenced item tend to be referenced again soon

K-way set Associative Cache

K cache lines per set

index becomes log(n° ō sets)

On a cache miss, if we're not using a direct mapped cache, we have to choose the line that gets evicted. most caches evict ẽ least recently used line.

Caching Organization Summarized

- A cache consists of lines
- A **line** contains
  - A **block** of bytes, the data values from memory
  - A **tag**, indicating where in memory the values are from
  - A **valid bit**, indicating if the data are valid
- Lines are organized into sets
  - **Direct-mapped cache:** one line per set
  - **k-way associative cache:** k lines per set
  - **Fully associative cache:** all lines in one set → no index, check all lines
- Caches handle both reads and writes
  - **write-through:** write to both cache and memory
  - **write-back:** write only to cache, write to memory on evict,
  - **write-allocate:** alloc on any miss
  - **no-write allocate:** alloc only on read miss

Cache size: total n° ō bytes that can be stored in ẽ cache

Avg access time = hit time + miss rate * miss-penalty

Category ō misses

- Compulsory: 1st reference to a blo
- Capacity: Cache is too small
- Conflict: Collisions in a specific set

Caching and Writes

- What to do on a write-hit?
  - **Write-through:** write immediately to memory (update both copies : in cache & in memory)
  - **Write-back:** defer write to memory until replacement of line → only update cache
    - Need a dirty bit (line different from memory or not)
- What to do on a write-miss?
  - **Write-allocate:** load into cache, update line in cache
    - Good if more writes to the location follow
  - **No-write-allocate:** writes straight to memory, does not load into cache → only update to mem
- Typical
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate** → if it's there you only update cache, otherwise, brings data into cache & then only update copy in cache.

Optimization

Summary of Matrix Multiplication

```
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;
  }
}
```

ijk (& jik):
- 2 memory accesses (2 reads, 0 write)
- misses/iter = 1.25

```
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];
 }
}
```

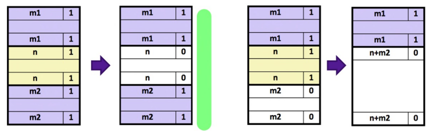
kij (& ikj):
- 3 memory accesses (2 reads, 1 write)
- misses/iter = 0.5

```
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;
 }
}
```

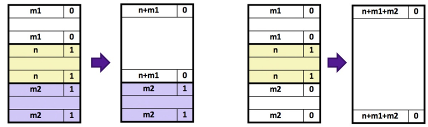
jki (& kji):
- 3 memory accesses (2 reads, 1 write)
- misses/iter = 2.0

Constant-Time Coalescing

Case 1: Prev and next block allocated   Case 2: Prev block free, next block allocated



Case 2: Prev block allocated, next block free   Case 4: Prev and next block free



Machine Independent Optimization

- Compilers optimize assembly code
  - Dead code elimination
  - Code motion
  - Factoring out common subexpressions
  - Loop elimination
  - Reduction in Strength
- Optimization blockers:
  - Aliasing
    - Use local variables
  - Procedure calls
    - Move them yourself

Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses / accesses)
  - Typically 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
  - Typically 4 clock cycles for L1, 10 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)

Allocator Goals

- **Throughput:** number of requests completed per time unit
  - Make allocator efficient
  - Example: if your allocator processes 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds then throughput is 1,000 operations/second
- **Memory Utilization:** fraction of heap memory allocated
  - Minimize wasted space
  - Peak Memory Utilization $U_t = \frac{\max_{i \le t} space\ allocated\ at\ time\ i}{size\ of\ heap\ at\ time\ t}$
- These goals are often conflicting

Summary of Matrix Multiplication

```
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;
  }
}
```

ijk (& jik):
- 2 memory accesses (2 reads, 0 write)
- misses/iter = 1.25

```
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];
 }
}
```

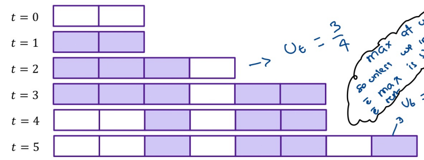
kij (& ikj):
- 3 memory accesses (2 reads, 1 write)
- misses/iter = 0.5

```
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;
 }
}
```

jki (& kji):
- 3 memory accesses (2 reads, 1 write)
- misses/iter = 2.0

Exercise: Memory Utilization

- Recall that Peak Memory Utilization $U_t = \frac{\max_{i \le t} space\ allocated\ at\ time\ i}{size\ of\ heap\ at\ time\ t}$



→ $U_t = 3/4$

max at $U_3 = 5$ so unless we increase it max is still 5 for ...

$U_5 = 5/8$

- What is the Peak Memory Utilization at time $t = 2$?
- What is the Peak Memory Utilization at time $t = 5$?