CSCI-564 Advanced Computer Architecture

Lecture 5: Advanced Cache

Bo Wu
Colorado School of Mines
Write through vs. write back

- When we perform a write, should we just update this cache, or should we also forward the write to the next lower cache?
- If we *do not* forward the write, the cache is “Write back”, since the data must be written back when it’s evicted (i.e., the line can be dirty)
- If we *do* forward the write, the cache is “write through.” In this case, a cache line is never dirty.

Write back advantages

Write through advantages
Write through vs. write back

- When we perform a write, should we just update this cache, or should we also forward the write to the next lower cache?
- If we do not forward the write, the cache is “Write back”, since the data must be written back when it’s evicted (i.e., the line can be dirty)
- If we do forward the write, the cache is “write through.” In this case, a cache line is never dirty.

Write back advantages
- Fewer writes farther down the hierarchy.
- Less bandwidth.
- Faster writes

Write through advantages
Write through vs. write back

- When we perform a write, should we just update this cache, or should we also forward the write to the next lower cache?
- If we *do not* forward the write, the cache is “Write back”, since the data must be written back when it’s evicted (i.e., the line can be dirty)
- If we *do* forward the write, the cache is “write through.” In this case, a cache line is never dirty.
- Write back advantages
  Fewer writes farther down the hierarchy. Less bandwidth. Faster writes
- Write through advantages
  No write back required on eviction.
Write allocate/no-write allocate

- If the cache allocates cache lines on a write miss, it is *write allocate*, otherwise, it is *no write allocate*.
- Write Allocate advantages

- No-write allocate advantages
Write allocate/no-write allocate

- If the cache allocates cache lines on a write miss, it is *write allocate*, otherwise, it is *no write allocate*.
- Write Allocate advantages
  
  Exploits temporal locality. Data written will be read soon, and that read will be faster.

- No-write allocate advantages
Write allocate/no-write allocate

- If the cache allocates cache lines on a write miss, it is *write allocate*, otherwise, it is *no write allocate*.
- Write Allocate advantages
  
  Exploits temporal locality. Data written will be read soon, and that read will be faster.

- No-write allocate advantages

  Fewer spurious evictions. If the data is not read in the near future, the eviction is a waste.
Eviction in associative caches

- We must choose which line in a set to evict if we have associativity
Eviction in associative caches

- We must choose which line in a set to evict if we have associativity
- How we make the choice is called the cache eviction policy
  - Random -- always a choice worth considering.
  - Least recently used (LRU) -- evict the line that was last used the longest time ago.
  - Prefer clean -- try to evict clean lines to avoid the write back.
Eviction in associative caches

- We must choose which line in a set to evict if we have associativity.
- How we make the choice is called *the cache eviction policy*.
  - Random -- always a choice worth considering.
  - Least recently used (LRU) -- evict the line that was last used the longest time ago.
  - Prefer clean -- try to evict clean lines to avoid the write back.
  - Farthest future use -- evict the line whose next access is farthest in the future. This is provably optimal. It is also impossible to implement.
Cache line size

• How big should a cache line be?
• Why is bigger better?

• Why is smaller better?
Cache line size

- How big should a cache line be?
- Why is bigger better?
  - Exploits more spatial locality.
  - Large cache lines effectively \textit{prefetch} data that we have not explicitly asked for.
- Why is smaller better?
Cache line size

- How big should a cache line be?
- Why is bigger better?
  - Exploits more spatial locality.
  - Large cache lines effectively *prefetch* data that we have not explicitly asked for.
- Why is smaller better?
  - Focuses on temporal locality.
  - If there is little spatial locality, large cache lines waste space and bandwidth.
Cache line size

- How big should a cache line be?
- Why is bigger better?
  - Exploits more spatial locality.
  - Large cache lines effectively `prefetch` data that we have not explicitly asked for.
- Why is smaller better?
  - Focuses on temporal locality.
  - If there is little spatial locality, large cache lines waste space and bandwidth.
- In practice 32-64 bytes is good for L1 caches were space is scarce and latency is important.
- Lower levels use 128-256 bytes.
Cache line size
Data vs. instruction cache

- Why have different I and D caches?
  - Different areas of memory
  - Different access patterns
    - I-cache accesses have lots of spatial locality. Mostly sequential accesses.
    - I-cache accesses are also predictable to the extent that branches are predictable
    - D-cache accesses are typically less predictable
  - Not just different, but often across purposes.
    - Sequential I-cache accesses may interfere with the data the D-cache has collected.
Cache misses are our enemy…
Know the Enemy

• Misses happen for different reasons
• The three C’s (types of cache misses)
  • Compulsory: The program has never requested this data before. A miss is mostly unavoidable.
  • Conflict: The program has seen this data, but it was evicted by another piece of data that mapped to the same “set”
  • Capacity: The program is actively using more data than the cache can hold.
Reducing Compulsory Misses

- Increase cache line size so the processor requests bigger chunks of memory
- This only works if there is good spatial locality, otherwise you are bringing in data you don’t need
  - If you are reading a few bytes here and a few bytes there (i.e., no spatial locality) this will hurt performance
  - But it will help in cases like this

```c
for(i = 0; i < 1000000; i++) { 
  sum += data[i];
}
```
Prefetching

• Speculate on future instruction and data accesses and fetch them into cache
  • Instruction accesses easier to predict than data accesses

• Varieties of prefetching
  • Hardware prefetching
  • Software prefetching
  • Mixed schemes
Reducing Compulsory Misses

- **Hardware Prefetching**

  ```
  for(i = 0; i < 1000000; i++) {
      sum += data[i];
  }
  ```

- In this case, the processor could identify the pattern and proactively prefetch data the program will ask for.

- **Pattern**: `nextAddr = curAddr + 4`
Hardware Prefetching
Hardware Prefetching

- Prefetch-on-miss
  - prefetch b+1 upon miss on b
Hardware Prefetching

- Prefetch-on-miss
  - prefetch b+1 upon miss on b
- One block lookahead scheme
  - initiate prefetch for block b+1 when block b is accessed
  - Can extend to N-block lookahead
Hardware Prefetching

• Prefetch-on-miss
  • prefetch b+1 upon miss on b

• One block lookahead scheme
  • initiate prefetch for block b+1 when block b is accessed
  • Can extend to N-block lookahead

• Strided prefetch
  • If observe sequence of accesses to block b, b+N, b+2N, then prefetch b+3N etc.
Issues in Prefetching

- Usefulness – should produce hits
- Timeliness – not late and not too early
- Cache and bandwidth pollution
Software Prefetching

for(i=0; i < N; i++) {
    prefetch( &a[i + P] );
    prefetch( &b[i + P] );
    SUM = SUM + a[i] * b[i];
}

• Timing is the biggest issue, not predictability
  • If you prefetch very close to when the data is requested, you may be too late
  • Prefetch too early, cause pollution
  • Estimate how long it will take for the data to come into l1 cache, so we can set P appropriately
  • Why is this hard to do?
**Hardware Instruction Prefetching**

- Fetch two blocks on a miss; the requested block (i) and the next consecutive block (i+1)
- Requested block placed in cache, and the next block in instruction stream buffer
- If miss in cache but hit in stream buffer, move stream buffer block into cache and prefetch next block (i+2)
Restructuring

```c
struct Atom {
    double v,
    double f,
    double3 p
};

Atom atoms[N];
```

```c
for (i=0; i<N; ++i)
    ...
    = atoms[i].f + ...

for (i=0; i<N; ++i)
    ...
    = atoms[i].v - ...

for (i=0; i<N; ++i)
    ...
    = atoms[i].p + ...
```
Restructuring

```c
struct Atom {
    double v,
    double f,
    double3 p
};

Atom atoms[N];

structure Atom {
    double v,
    double f,
    double3 p
};

Atom atoms[N];

double vs[N];
double fs[N];
double3 ps[N];

for (i=0; i<N; ++i)
    ... = atoms[i].f + ...

for (i=0; i<N; ++i)
    ... = atoms[i].v - ...

for (i=0; i<N; ++i)
    ... = atoms[i].p + ...
```
Conflict Misses

• Conflict misses occur when the data we need was in the cache previously but got evicted.

• Evictions occur because:
  • Direct mapped: Another request mapped to the same cache line.
  • Associative: Too many other requests mapped to the same set.

```c
while(1) {
    for(i = 0; i < 1024*1024; i += 4096) {
        sum += data[i];
    } // Assume a 4 KB Cache
}
```
Colliding threads and data

- The stack and the heap tend to be aligned to large chunks of memory (maybe 128MB).
Colliding threads and data

- The stack and the heap tend to be aligned to large chunks of memory (maybe 128MB).
- Threads often run the same code in the same way.
- This means that thread stacks will end up occupying the same parts of the cache.
Colliding threads and data

- The stack and the heap tend to be aligned to large chunks of memory (maybe 128MB).
- Threads often run the same code in the same way.
- This means that thread stacks will end up occupying the same parts of the cache.
- Randomize the base of each thread's stack.
Colliding threads and data

- The stack and the heap tend to be aligned to large chunks of memory (maybe 128MB).
- Threads often run the same code in the same way.
- This means that thread stacks will end up occupying the same parts of the cache.
- Randomize the base of each threads stack.
Colliding threads and data

- The stack and the heap tend to be aligned to large chunks of memory (maybe 128MB).
- Threads often run the same code in the same way.
- This means that thread stacks will end up occupying the same parts of the cache.
- Randomize the base of each threads stack.
- Large data structures (e.g., arrays) are also often aligned. Randomizing malloc() can help here.
Capacity Misses
Capacity Misses

- Capacity misses occur because the processor is trying to access too much data
  - Working set: The data that is currently important to the program
  - If the working set is bigger than the cache, you are going to miss frequently
Capacity Misses

• Capacity misses occur because the processor is trying to access too much data
  • Working set: The data that is currently important to the program
  • If the working set is bigger than the cache, you are going to miss frequently

• Capacity misses are a bit hard to measure
  • Easiest definition: non-compulsory miss rate in an equivalently-sized fully-associative cache
  • Intuition: Take away the compulsory misses and the conflict misses, and what you have left are the capacity misses
Reducing Capacity Misses
Reducing Capacity Misses

- Increase capacity
- More associativity or more associative “sets”
  - Costs area and makes the cache slower
- Cache hierarchy do this implicitly already
  - if the working set “falls out” of the L1, you start using L2
- In practice, you make the L1 as big as you can within your cycle time and the L2 and L3 as big as you can while upping it on chip
Reducing capacity misses: the compiler

- Tiling
  - We need to make several passes over a large array
  - Doing each pass in turn will “blow out” our cache
  - “Blocking” or “tiling” the loops will prevent the blow out
  - Whether this is possible depends on the structure of the loop
- You can tile hierarchically, to fit into each level of the memory hierarchy.
Reducing capacity misses: the compiler

- Tiling
  - We need to make several passes over a large array
  - Doing each pass in turn will “blow out” our cache
  - “blocking” or “tiling” the loops will prevent the blow out
  - Whether this is possible depends on the structure of the loop
- You can tile hierarchically, to fit into each level of the memory hierarchy.
More discussion about prefetching

- Affect capacity misses
- Affect conflict misses
Write Performance
Reducing Write Time

• Problem: Writes take two cycles; One for tag check and the other for writing data

• Solution 1
  • Step 1: Tag check, buffering old data and write data
  • Step 2: if tag check fails, write old data back

• Solution 2
  • Pipelining the writes
Reducing Miss Penalty

Evicted dirty lines for writeback cache
OR
All writes in writethrough cache
Multi-level Caches

Problem: A memory cannot be large and fast
Solution: Increasing sizes of cache at each level

Local miss rate = misses in cache / accesses to cache
Global miss rate = misses in cache / CPU memory accesses
Misses per instruction = misses in cache / number of instructions
Presence of L2 Influences L1 Design

- Use smaller L1 if there is also L2
  - Trade increased L1 miss rate for reduced L1 hit time and reduce L1 miss penalty
  - Reduce average access energy
- Use simpler write-through L1 with on-chip L2
  - Write-back L2 absorbs write traffic, doesn’t go off-chip
Inclusion Policy

• Inclusive multilevel cache
  • e.g., L2 cache holds copies of data in L1 cache

• Exclusive multilevel cache
  • e.g., L2 cache may hold data not in L1 cache
Inclusion Policy

• Inclusive multilevel cache
  • e.g., L2 cache holds copies of data in L1 cache

• Exclusive multilevel cache
  • e.g., L2 cache may hold data not in L1 cache

Why choose one type or the other?
Victim Cache

- Say a set has $W$ ways, but $W+2$ lines are mapped to each set
Increasing Cache Bandwidth
Increasing Cache Bandwidth
Multiport Caches

- Large area increase (could be double for 2-port)
- Hit time increase (can be made small)
Banked Caches

- Partition address space into multiple banks

- Benefits
  - higher throughput

- Challenges
  - Bank conflicts
  - Extra wiring
  - Uneven utilization
Critical Word First

Basic Blocking Cache:

- CPU Time
- Miss Penalty
- Order of fill: 0, 1, 2, 3, 4, 5, 6, 7

Blocking Cache with Critical Word first:

- CPU Time
- Miss Penalty
- Order of fill: 3, 4, 5, 6, 7, 0, 1, 2
Early Restart

Basic Blocking Cache:
- CPU Time
- Miss Penalty
  Order of fill: 0, 1, 2, 3, 4, 5, 6, 7

Blocking Cache with Early Restart:
- CPU Time
- Miss Penalty
  Order of fill: 0, 1, 2, 3, 4, 5, 6, 7