ECE 4750
Computer Architecture
Prof. José F. Martínez
Improving Cache Performance

• Use better technology
• Decrease Miss Rate
• Decrease Miss Penalty
• Decrease Hit Time
Increase Block Size

- Larger block size better exploits spatial locality, but
  - larger block size means larger miss penalty
    - takes longer time to transfer the block
  - if block size is too big
    - average access time goes up
    - temporal locality is reduced when the replaced data would have been reused (too few lines in cache)
Higher Associativity

- Reduce the number of conflict misses
  - more places to put data

- Two general rules of thumb (empirical):
  - an 8-way set-associative cache performs near fully associative
  - direct mapped cache of size N has same MR as a 2-way set associative cache of size N/2 (2:1 cache rule of thumb)

- Tradeoff is increased hit time

- Commonly see high associativity in 2^{nd}-level caches
  - less common in 1^{st}-level caches
Victim Cache

• Small fully-associative cache between real cache and its refill path
  • contains only blocks replaced on recent misses (*victims*)
• On a miss:
  • check victim cache
  • if present, swap victim and cache entry
  • else fetch as usual, put new victim in victim cache
• Shown effective for small direct-mapped caches
  • trade-off is area and additional control complexity
• Does not trade-off hit time
  • miss penalty?
Pseudo Set-Associative Caches

• Combine advantages of direct-mapped and set-associative caches
• Perform cache access as in a direct-mapped cache
  • if hit, done
  • if miss, check another cache entry!
  • one scheme: invert MSB of index and check there
• Effectively a second set
  • but, no parallel comparators or muxes (less HW)
  • one set is fast access, one is slow access (extra cycle)
  • want fast hits and not the slow hits
• Need some form of way prediction
  • can result in lower AMAT than both DM and SA caches
  • variable hit times complicate pipelines – use in lower levels
Hardware Prefetching

• A technique to improve cold and capacity misses
• Have hardware fetch extra lines on a miss
  • Can store in cache, or a separate stream buffer
• E.g., i-cache fetch 2 blocks on an instruction miss
• Can do the same for data cache, even multiple buffers
  • Modern prefetchers learn non-unit strides
• Scheme relies on excess available memory bandwidth
  • Can hurt performance if it interferes with demand misses
Software Prefetching

- Compiler-directed
  - analyze code and know where misses occur
- Insert a special *prefetch* instruction into the code stream
  - most useful when it is a *non-binding prefetch*
    - turns into a *nop* on an exception
  - don’t prefetch everything – too much instruction overhead!
  - ideally just prefetch the misses
  - sophisticated compiler analysis in general case
- Requires the existence of *lockup-free* (non-blocking) caches
- Subsequent load to same cache line will
  - hit in cache if prefetch is back from memory system
  - miss, but not issue, if prefetch still outstanding
Compiler Optimizations

• Reduce miss rates without changing the hardware!
• Code is easily re-ordered
  • *cording* rearranges procedures to reduce conflict misses
  • use profiled information
• Data is more interesting (and harder)
  • still, can to re-arrange data accesses to improve locality
• Examples:
  • array merging
  • loop interchange
  • loop fusion
  • blocking
Array Merging

• Some weak programmers produce code like:

```c
int val[SIZE];
int key[SIZE];
```

• ...and then proceed to reference `key` and `val` in lockstep

• What’s the problem?
Array Merging

• Some weak programmers produce code like:
  
```c
int val[SIZE];
int key[SIZE];
  
• ...and then proceed to reference `key` and `val` in lockstep

• Danger is that these accesses may interfere w/ each other

• Solution?
Array Merging

• Some weak programmers produce code like:
  ```c
  int val[SIZE];
  int key[SIZE];
  • ...and then proceed to reference key and val in lockstep
  • Danger is that these accesses may interfere w/ each other
  • Solution: merge the arrays into a single array of records:
  ```
  ```c
  struct merge {
      int val;
      int key;
  };
  struct merge merged_array[SIZE];
  ```
Loop Interchange

• Some weak programmers produce code like:

```c
for (j=0; j < 100; j++)
    for (k=0; k < 100; k++)
        x[k][j] = 2 * x[k][j];
```

• What’s the problem?
Loop Interchange

• Some weak programmers produce code like:

```c
for (j=0; j < 100; j++)
    for (k=0; k < 100; k++)
        x[k][j] = 2 * x[k][j];
```

• C is a row-major language (Fortran is column-major)
  • This code has a stride of 100 words, not 1
  • No spatial locality, poor hit rates

• Solution?
Loop Interchange

• Some weak programmers produce code like:
  
  ```c
  for (j=0; j < 100; j++)
      for (k=0; k < 100; k++)
          x[k][j] = 2 * x[k][j];
  ```

• C is a *row-major* language (Fortran is *column-major*)
  
  • This code has a *stride* of 100 words, not 1
  • No spatial locality, poor hit rates

• Solution: interchange the loops!
  
  ```c
  for (k=0; k < 100; k++)
      for (j=0; j < 100; j++)
          x[k][j] = 2 * x[k][j];
  ```

  • Does not affect number of instructions, just more hits!
Loop Fusion

• Some weak programmers produce code like:

```c
for (j=0; j < N; j++)
    for (k=0; k < N; k++)
        a[j][k] = 1/b[j][k] * c[j][k];
for (j=0; j < N; j++)
    for (k=0; k < N; k++)
        d[j][k] = a[j][k] + c[j][k];
```

• What’s the problem?
Loop Fusion

• Some weak programmers produce code like:

```c
for (j=0; j < N; j++)
    for (k=0; k < N; k++)
        a[j][k] = 1/b[j][k] * c[j][k];
for (j=0; j < N; j++)
    for (k=0; k < N; k++)
        d[j][k] = a[j][k] + c[j][k];
```

• No temporal locality if arrays are big enough
  • Codes takes misses to a and c arrays twice

• Solution?
Loop Fusion

- Some weak programmers produce code like:
  ```c
  for (j=0; j < N; j++)
    for (k=0; k < N; k++)
      a[j][k] = 1/b[j][k] * c[j][k];
  for (j=0; j < N; j++)
    for (k=0; k < N; k++)
      d[j][k] = a[j][k] + c[j][k];
  ```
- No temporal locality if arrays are big enough
  - Codes takes misses to a and c arrays twice
- Solution: fuse the loops!
  ```c
  for (j=0; j < N; j++)
    for (k=0; k < N; k++) {
      a[j][k] = 1/b[j][k] * c[j][k];
      d[j][k] = a[j][k] + c[j][k];
    }
  ```
Improving Cache Performance

- Use better technology
- Decrease Miss Rate
- **Decrease Miss Penalty**
- Decrease Hit Time
Read Priority

• Processor need not wait for (isolated) writes
  • but what if we want to read – RAW through memory
• Reads do stall CPU – give priority to reads
  • but serialize/forward if overlap with earlier write
Fill Before Spill

- In writeback caches
- If line is Dirty on a read/write miss, need to write it back
- This increases miss penalty for the demand miss
- Solution: *spill buffer*
  - fetch demand miss from memory
  - spill dirty line into on-chip spill buffer
  - write spill buffer to memory in background after demand miss
- Subsequent misses wait for spill buffer to empty
  - or even snoop
Early Restart

• Decrease miss penalty with no new hardware
  • well, okay, with some more complicated control

• Strategy: impatience!

• There is no need to wait for entire line to be fetched

• *Early Restart* – as soon as the requested word (or double word) of the cache block arrives, let the CPU continue execution

• If CPU references another cache line or a later word in the same line: stall

• Early restart is often combined with the next technique...
Critical Word First

- Improvement over early restart
  - request missed word first from memory system
  - send it to the CPU as soon as it arrives
  - CPU consumes word while rest of line arrives

- Even more complicated control logic
  - memory system must also be changed
  - block fetch must wrap around

- Example: 32B block (8 words), miss on address 20
  - words return from memory system as follows: 20, 24, 28, 0, 4, 8, 12, 16
    - other sequences possible
Lockup-Free Caches

• The CPU need not stall on cache misses
  • dynamically scheduled processors can hide memory latency
  • caches must be non-blocking or lockup-free

• Hit under miss schemes allow data cache to supply data for other lines during a cache miss

• Extensions include “hit under multiple miss” (overlap misses)
  • Significantly complicates cache control: Miss Handling Table (MHT)
2nd-Level Caches

- Add another level of cache between CPU and main memory
  - allows first-level cache to remain small and fast
  - second-level is slower but much larger (MBs)
- Reduces overall miss penalty, complicated perf analysis
  - $AMAT = \text{Hit time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$
  - $\text{Miss Penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}$
  - What is 2nd-level miss rate?
    - local miss rate – number of cache misses / cache accesses
    - global miss rate – number of cache misses / CPU memory refs
- Local miss rate can be large...why?
- Global miss rate is more useful measure
Second-Level Cache Design

• Speed of 2\textsuperscript{nd} level cache typ. affects only miss penalty, not CPU clock
  • will it lower the AMAT portion of the CPI?
  • how much does it cost?
• Size of 2\textsuperscript{nd} level cache >> first level
• Most capacity misses go away, leaving conflict misses
• 2\textsuperscript{nd}-level caches therefore
  • typically have some degree of associativity > 1
  • have large block sizes
  • emphasis shifts from fast hits to fewer misses
Improving Cache Performance

• Use better technology
• Decrease Miss Rate
• Decrease Miss Penalty
• Decrease Hit Time
Improving Cache Performance

• Use better technology
• Decrease Miss Rate
• Decrease Miss Penalty
• **Decrease Hit Time**
  • Use better or faster technology
  • Simplify design (e.g., direct-mapped)
  • Avoid or concurrentize translations (e.g., virtually-indexed)