Advanced Caching Techniques

Approaches to improving memory system performance
- eliminate memory operations
- decrease the number of misses
- decrease the miss penalty
- decrease the cache/memory access times
- hide memory latencies
- increase cache throughput
- increase memory bandwidth

Victim Cache

- small fully-associative cache
- contains the most recently replaced blocks of a direct-mapped cache
- check it on a cache miss
- swap the direct-mapped block and victim cache block
- alternative to 2-way set-associative cache

Handling a Cache Miss the Old Way

1. Send the address & read operation to the next level of the hierarchy
2. Wait for the data to arrive
3. Update the cache entry with data*, rewrite the tag, turn the valid bit on, clear the dirty bit (if data cache)
4. Resend the memory address; this time there will be a hit.

* There are variations:
  - get data before replace the block
  - send the requested word to the CPU as soon as it arrives at the cache (early restart)
  - requested word is sent from memory first; then the rest of the block follows (requested word first)

How do the variations improve memory system performance?

Non-blocking Caches

Non-blocking cache (lockup-free cache)
- allows the CPU to continue executing instructions while a miss is handled
- some processors allow only 1 outstanding miss ("hit under miss")
- some processors allow multiple misses outstanding ("miss under miss")
- miss status holding registers (MSHR)
  - hardware structure for tracking outstanding misses
    - physical address of the block
    - which word in the block
    - destination register number (if data)
    - mechanism to merge requests to the same block
    - mechanism to insure accesses to the same location execute in program order

Handling a Cache Miss the Old Way

1. Send the address & read operation to the next level of the hierarchy
2. Wait for the data to arrive
3. Update the cache entry with data*, rewrite the tag, turn the valid bit on, clear the dirty bit (if data cache)
4. Resend the memory address; this time there will be a hit.

* There are variations:
  - get data before replace the block
  - send the requested word to the CPU as soon as it arrives at the cache (early restart)
  - requested word is sent from memory first; then the rest of the block follows (requested word first)

How do the variations improve memory system performance?

Sub-block Placement

Divide a block into sub-blocks

- sub-block = unit of transfer on a cache miss
  - valid bit/sub-block
  - misses:
    - block-level miss: tags didn't match
    - sub-block-level miss: tags matched, valid bit was clear
  - the transfer time of a sub-block
  - fewer tags than if each block was the size of a block
  - less implicit prefetching

How does sub-block placement improve memory system performance?
Pseudo-set associative Cache

Pseudo-set associative cache

- access the cache
- if miss, invert the high-order index bit & access the cache again
- miss rate of 2-way set associative cache
- access time of direct-mapped cache if hit in the “fast-hit block”
- increase in hit time (relative to 2-way associative) if always hit in the “slow-hit block”
- predict which is the fast-hit block

How does pseudo-set associativity improve memory system performance?

Pipelined Cache Access

Pipelined cache access

- simple 2-stage pipeline
  - access the cache
  - data transfer back to CPU
  - tag check & hit/miss logic with the shorter of the two stages

How do pipelined caches improve memory system performance?

Mechanisms for Prefetching

Stream buffers

- where prefetched instructions/data held
- if requested block in the stream buffer, then cancel the cache access

How do improve memory system performance?

Trace Cache

Trace cache contents

- contains instructions from the dynamic instruction stream
  - fetch statically noncontiguous instructions in a single cycle
  - a more efficient use of “I-cache” space
  - trace is analogous to a cache block wrt accessing

Assessing a trace cache

- trace cache state includes low bits of next addresses (target & fall-through code) for the last instruction in the currently executing trace, which is a branch
- trace cache tag is high branch address bits + predictions for all branches in the trace
- assess trace cache & branch predictor, BTB, I-cache in parallel
- compare high PC bits & prediction history of the current branch instruction to the trace cache tag
- hit: use trace cache & I-cache fetch ignored
- miss: use the I-cache
  start constructing a new trace

Why does a trace cache work?
Cache-friendly Compiler Optimizations

Exploit spatial locality
- schedule for array misses
  - hoist first load to each cache block

Improve spatial locality
- group & transpose
  - makes portions of vectors that are accessed together lie in memory together
- loop interchange
  - so inner loop follows memory layout

Improve temporal locality
- loop fusion
  - do multiple computations on the same portion of an array
- tiling (also called blocking)
  - do all computation on a small block of memory that will fit in the cache

Effect on performance?

Memory Banks

Interleaved memory:
- multiple memory banks
  - word locations are assigned across banks
  - interleaving factor: number of banks
  - send a single address to all banks at once

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Rank 0</th>
<th>Rank 1</th>
<th>Rank 2</th>
<th>Rank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>0</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Memory Banks

Independent memory banks
- different banks can be accessed at once, with different addresses
- allows parallel access, possibly parallel data transfer
- multiple memory controllers & separate address lines, one for each access
- different controllers cannot access the same bank
- less area than dual porting

Effect on performance?

Tiling Example

```c
/* before */
for (i=0; i<n; i=i+1)
  for (j=0; j<n; j=j+1) {
    r = 0;
    for (k=k; k<min(k+n-1, n); k=k+1) {
      r = r + y[i,k] * z[k,j];
    }
    x[i,j] = x[i,j] + r;
  }
/* after */
for (j=0; j<n; j=j+1)
  for (k=k; k<min(k+n-1, n); k=k+1) {
    r = 0;
    for (i=i; i<min(i+n-1, n); i=i+1) {
      r = r + y[i,k] * z[k,j];
    }
    x[i,j] = x[i,j] + r;
  }
```
**Today’s Memory Subsystems**

Look for designs in common:

**Advanced Caching Techniques**

**Approaches to improving memory system performance**

- eliminate memory operations
- decrease the number of misses
- decrease the miss penalty
- decrease the cache/memory access times
- hide memory latencies
- increase cache throughput
- increase memory bandwidth

**Wrap-up**

Victim cache (reduce miss penalty)
TLB (reduce page fault time (penalty))
Hardware or compiler-based prefetching (reduce misses)
Cache-conscious compiler optimizations (reduce misses or hide miss penalty)
Coupling a write-through memory update policy with a write buffer (eliminate store ops/hide store latencies)
Handling the read miss before replacing a block with a write-back memory update policy (reduce miss penalty)
Sub-block placement (reduce miss penalty)
Non-blocking caches (hide miss penalty)
Merging requests to the same cache block in a non-blocking cache (hide miss penalty)
Requested word first or early restart (reduce miss penalty)
Cache hierarchies (reduce misses/reduce miss penalty)
Virtual caches (reduce miss penalty)
Pipelined cache accesses (increase cache throughput)
Pseudo-set associative cache (reduce misses)
Banked or interleaved memories (increase bandwidth)
Independent memory banks (hide latency)
Wider bus (increase bandwidth)