Advanced cache optimizations

ECE 154B
Dmitri Strukov
Advanced Cache Optimization

1) Way prediction
2) Victim cache
3) Pipelined cache
4) Nonblocking cache
5) Multibanked cache
6) Critical word first and early restart
7) Merging write buffer
8) Compiler optimizations
9) Prefetching
#1: Way Prediction

• How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?

• **Way prediction**: keep extra bits in cache to predict the “way,” or block within the set, of next cache access.
  – Multiplexor is set early to select desired block, only 1 tag comparison performed that clock cycle in parallel with reading the cache data
  – Miss $\Rightarrow$ 1st check other blocks for matches in next clock cycle

• **Accuracy** $\approx 85$

• **Drawback**: CPU pipeline is hard if hit takes 1 or 2 cycles
  – Used for instruction caches vs. L1 data caches
  – Also used on MIPS R10K for off-chip L2 unified cache, way-prediction table on-chip
Efficient for thrashing problem in direct mapped caches
Remove 20%-90% cache misses to L1 cache
L1 and Victim cache are exclusive
Miss to L1 but hit in VC; miss in L1 and VC
#3: Pipelining Cache Writes

Address and Store Data From CPU

Tag

Store Data

Delayed Write Addr.

Delayed Write Data

Load/Store

Tags

Data

Load Data to CPU

Hit?

Data from a store hit written into data portion of cache during tag access of subsequent store
#4: Nonblocking Cache: Basic Idea

Stall CPU on miss

Hit under miss

Stall only when result needed

Multiple out-standing misses
Nonblocking Cache

- **Non-blocking cache** or **lockup-free cache** allow data cache to continue to supply cache hits during a miss.
- “**hit under miss**” reduces the effective miss penalty by working during miss vs. ignoring CPU requests.
- “**hit under multiple miss**” or “**miss under miss**” may further lower the effective miss penalty by overlapping multiple misses.
  - Pentium Pro allows 4 outstanding memory misses.
  - (Cray X1E vector supercomputer allows 2,048 outstanding memory misses.)
Non Blocking Cache

Figure 2.5 The effectiveness of a nonblocking cache is evaluated by allowing 1, 2, or 64 hits under a cache miss with 9 SPECINT (on the left) and 9 SPECFP (on the right) benchmarks. The data memory system modeled after the Intel i7 consists of a 32KB L1 cache with a four cycle access latency. The L2 cache (shared with instructions) is 256 KB with a 10 clock cycle access latency. The L3 is 2 MB and a 36-cycle access latency. All the caches are eight-way set associative and have a 64-byte block size. Allowing one hit under miss reduces the miss penalty by 9% for the integer benchmarks and 12.5% for the floating point. Allowing a second hit improves these results to 10% and 16%, and allowing 64 results in little additional improvement.
Nonblocking Cache Implementation

– requires out-of-order execution
– significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
– requires pipelined or banked memory system (otherwise cannot support)
Nonblocking Cache Example

• Maximum number of outstanding references to maintain peak bandwidth for a system?
  - sustained transfer rate 16GB/sec
  - memory access 36ns
  - block size 64 bytes
  - 50% need not be issued
Nonblocking Cache Example

• Maximum number of outstanding references to maintain peak bandwidth for a system?
  - sustained transfer rate 16GB/sec
  - memory access 36ns
  - block size 64 bytes
  - 50% need not be issued

Answer: \( \frac{(16 \times 10)^9}{64} \times 36 \times 10^{-9} \times 2 = 18 \)
#5: Increasing Cache Bandwidth via Multiple Banks

- Rather than treat the cache as a single monolithic block, divide into independent banks that can support simultaneous accesses
  - 4 in L1 and 8 in L2 for Intel core i7
- Banking works best when accesses naturally spread themselves across banks \( \Rightarrow \) mapping of addresses to banks affects behavior of memory system
- Simple mapping that works well is “sequential interleaving”
  - Spread block addresses sequentially across banks
  - E.g., if there 4 banks, Bank 0 has all blocks whose address modulo 4 is 0; bank 1 has all blocks whose address modulo 4 is 1; ...

<table>
<thead>
<tr>
<th>Block address</th>
<th>Bank 0</th>
<th>Block address</th>
<th>Bank 1</th>
<th>Block address</th>
<th>Bank 2</th>
<th>Block address</th>
<th>Bank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
<td>6</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>9</td>
<td></td>
<td>10</td>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>13</td>
<td></td>
<td>14</td>
<td></td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>
#6: Reduce Miss Penalty: Early Restart and Critical Word First

- Don’t wait for full block before restarting CPU
- **Early restart**—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - Spatial locality ⇒ tend to want next sequential word, so not clear size of benefit of just early restart
- **Critical Word First**—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block
  - Long blocks more popular today ⇒ Critical Word 1st Widely used
#7: Merging Write Buffer to Reduce Miss Penalty

- Write buffer to allow processor to continue while waiting to write to memory
- If buffer contains modified blocks, the addresses can be checked to see if address of new data matches the address of a valid write buffer entry
- If so, new data are combined with that entry
- Increases block size of write for write-through cache of writes to sequential words, bytes since multiword writes more efficient to memory
- The Sun T1 (Niagara) processor, among many others, uses write merging
Merging Write Buffer Example

Figure 2.7 To illustrate write merging, the write buffer on top does not use it while the write buffer on the bottom does. The four writes are merged into a single buffer entry with write merging; without it, the buffer is full even though three-fourths of each entry is wasted. The buffer has four entries, and each entry holds four 64-bit words. The address for each entry is on the left, with a valid bit (V) indicating whether the next sequential 8 bytes in this entry are occupied. (Without write merging, the words to the right in the upper part of the figure would only be used for instructions that wrote multiple words at the same time.)

Interesting issue with conflicting design objectives, i.e. ejecting as soon as possible vs. keeping longer for merging
#8: Compiler Optimizations

- Restructuring code affects the data block access sequence
  - Group data accesses together to improve spatial locality
  - Re-order data accesses to improve temporal locality

- Prevent data from entering the cache
  - Useful for variables that will only be accessed once before being replaced
  - Needs mechanism for software to tell hardware not to cache data (instruction hints or page table bits)

- Kill data that will never be used again
  - Streaming data exploits spatial locality but not temporal locality
  - Replace into dead cache locations
Loop Interchange

```c
for (j=0; j < N; j++) {
    for (i=0; i < M; i++) {
        x[i][j] = 2 * x[i][j];
    }
}
```

```c
for (i=0; i < M; i++) {
    for (j=0; j < N; j++) {
        x[i][j] = 2 * x[i][j];
    }
}
```

What type of locality does this improve?
Loop Fusion

for(i=0; i < N; i++)
    for(j=0; j < M; j++)
        a[i][j] = b[i][j] * c[i][j];

for(i=0; i < N; i++)
    for(j=0; j < M; j++)
        d[i][j] = a[i][j] * c[i][j];

for(i=0; i < M; i++)
    for(j=0; j < N; j++)
        a[i][j] = b[i][j] * c[i][j];
    d[i][j] = a[i][j] * c[i][j];

What type of locality does this improve?
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
        r = 0;
        for (k = 0; k < N; k++)
            r = r + y[i][k] * z[k][j];
        x[i][j] = r;
    }
Blocking

for(jj=0; jj < N; jj=jj+B) 
  for(kk=0; kk < N; kk=kk+B) 
    for(i=0; i < N; i++) 
      for(j=jj; j < min(jj+B,N); j++) {
        r = 0;
        for(k=kk; k < min(kk+B,N); k++)
          r = r + y[i][k] * z[k][j];
        x[i][j] = x[i][j] + r;

What type of locality does this improve?
#9: Prefetching

- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses

- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes

- What types of misses does prefetching affect?
Issues in Prefetching

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

- Instruction prefetch in Alpha AXP 21064
  - Fetch two blocks on a miss; the requested block and the next consecutive block
  - Requested block placed in cache, and next block in instruction stream buffer
Hardware Data Prefetching

• Prefetch-on-miss:
  – Prefetch b + 1 upon miss on b

• One Block Lookahead (OBL) scheme
  – Initiate prefetch for block b + 1 when block b is accessed
  – Why is this different from doubling block size?
  – Can extend to N block lookahead

• Strided prefetch
  – If sequence of accesses to block b, b+N, b+2N, then prefetch b+3N etc.
Software Prefetching

```c
for(i=0; i < N; i++) {
    prefetch( &a[i + 1] );
    prefetch( &b[i + 1] );
    SUM = SUM + a[i] * b[i];
}
```

- What property do we require of the cache for prefetching to work?
Software Prefetching Issues

- **Timing is the biggest issue, not predictability**
  - If you prefetch very close to when the data is required, you might be too late
  - Prefetch too early, cause pollution
  - Estimate how long it will take for the data to come into L1, so we can set P appropriately
  - *Why is this hard to do?*

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

Must consider cost of prefetch instructions
## Summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>Hit time</th>
<th>Bandwidth</th>
<th>Miss penalty</th>
<th>Miss rate</th>
<th>Power consumption</th>
<th>Hardware cost/complexity</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small and simple caches</td>
<td>+</td>
<td></td>
<td>–</td>
<td>+</td>
<td>0</td>
<td>Trivial; widely used</td>
<td></td>
</tr>
<tr>
<td>Way-predicting caches</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td>1</td>
<td>Used in Pentium 4</td>
<td></td>
</tr>
<tr>
<td>Pipelined cache access</td>
<td>–</td>
<td>+</td>
<td></td>
<td></td>
<td>1</td>
<td>Widely used</td>
<td></td>
</tr>
<tr>
<td>Nonblocking caches</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td>3</td>
<td>Widely used</td>
<td></td>
</tr>
<tr>
<td>Banked caches</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td>1</td>
<td>Used in L2 of both i7 and Cortex-A8</td>
<td></td>
</tr>
<tr>
<td>Critical word first and early restart</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>Widely used</td>
<td></td>
</tr>
<tr>
<td>Merging write buffer</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>Widely used with write through</td>
<td></td>
</tr>
<tr>
<td>Compiler techniques to reduce cache misses</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>Software is a challenge, but many compilers handle common linear algebra calculations</td>
<td></td>
</tr>
<tr>
<td>Hardware prefetching of instructions and data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 instr., 3 data</td>
<td>Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware</td>
<td></td>
</tr>
<tr>
<td>Compiler-controlled prefetching</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td>3</td>
<td>Needs nonblocking cache; possible instruction overhead; in many CPUs</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.11** Summary of 10 advanced cache optimizations showing impact on cache performance, power consumption, and complexity. Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.
Acknowledgements

Some of the slides contain material developed and copyrighted by Arvind, Emer (MIT), Asanovic (UCB/MIT) and instructor material for the textbook