What we want, what we have

- **What we want:**
  - Unlimited memory space
  - Fast, constant, access time
    (UMA: Uniform Memory Access)

- **What we have:**
  - Memories are **getting bigger**
  - **Growing complexity** memory hierarchy
  - **Temporal and spatial locality** issues
    (NUMA: Non-Uniform Memory Access)
Memory speed hasn’t kept up

- Pre-1970s memory faster than processor...

- Now thousands of times slower to access
Multi-core processor chips

- E.g., Intel Core i7 generates up to 2 refs/clock for each of 4 cores @ 3.2GHz
  - 25.6G 64-bit data refs/s
  - 12.8G 128-bit inst. refs/s
Total 409.6GB/s… but DRAM is 25GB/s!
- Multi-port pipelined cache
- Multiple levels of caching
- Logic to support sharing
- Large fraction of area & power budget
Terms

• **Cache line**: one block of data in cache
• **Dirty line**: a block with value different from that block in lower memory levels
• **Hit**: data found here
• **Hit rate or hit ratio**: # hits / # references total
• **Hit time**: RAM access time + check hit/miss
• **Miss**: data not here, must forward read request
• **Miss rate**: # misses / # references total
• **Miss penalty**: time to replace line & deliver data
Basic cache design issues

- Placement (mapping)
- Identification: which line within the set do I want?
- Replacement policy: which line gets kicked-out to make space?
- Write strategy
Placement / Mapping

- Caches are basically hash tables indexed by hash value of the address requested
- **Direct mapped:**
  Each bucket holds one line
- **Set associative:**
  Each bucket holds set-size lines
- **Fully associative:**
  Only one bucket, which holds all lines
Mapping Addresses

• Each address is \{line number\} \{byte offset in line\}; 32B line $\rightarrow$ [4:0] are offset
• Cache set (bucket) index is hash(\{line number\}), often contiguous bits from \{line number\}
• Line tag field holds bits of \{line number\} that are not implied by bucket index

E.g., a 4-way 16KB cache might be:
512 lines, each 32B long, with 128 buckets
Address[4:0] is offset, [11:5] is bucket index
Best replacement policy?

- Direct mapped → no choice
- Random
- Replace a clean (not dirty) line
- LRU (Least Recently Used): mark when line is accessed, replace not accessed recently
- LFU (Least Frequently Used)
- MRU and MFU: Most “”
- Belady’s MIN: replace line not used for the longest time in the future (how to know this?)
- Compiler-driven; e.g., using cache bypass
Best replacement policy?

- Sample comparison of LRU vs. Random
- Miss rate %:

<table>
<thead>
<tr>
<th>Size</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>16KB</td>
<td>5.2</td>
<td>5.7</td>
<td>4.7</td>
</tr>
<tr>
<td>64KB</td>
<td>1.9</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>256KB</td>
<td>1.15</td>
<td>1.17</td>
<td>1.13</td>
</tr>
</tbody>
</table>
Write strategy

• Write through
  – Write always goes to main memory
  – Easy; needed for I/O devices in memory
• Write back
  – Write only when line replaced, saving traffic
  – Could do lazy writes when not busy
  – May need to read on miss to get rest of line
• Write allocate: write back, but don’t wait for line to be read first; aka pre-arrival caching
Write Buffer

• Sort-of like a “level 0 data cache” (faster because no TLB in front of it)
• Buffer can re-group writes to form write to a larger fraction of a line (not just one byte or word)
• Need to be careful about task switches, etc.; may have to flush write buffer often
What causes a miss?

- **Compulsory**
  - Never touched this block before
  - *Shared fetch effect* can avoid these when another process touches what I want first

- **Capacity**
  - Could have been from cache, but didn’t fit

- **Conflict**
  - Could have fit, but cache mapping had a conflict with another line that caused this line to be replaced (e.g., direct mapped)
Cache optimizations

• Larger total cache size
  – Fewer capacity & conflict misses
  – Dumber replacement policy works ok
  – Increases hit time, die space, and power use

• Larger line size
  – Fewer compulsory misses (spatial locality)
  – More capacity & conflict misses
  – Increases miss penalty (block transfer time)
More cache optimizations

• Higher associativity
  – Reduces conflict misses
  – Increases hit time & power use
• More levels of cache
  – Smaller, faster, upper-level caches
  – More complex hardware structure
Still more cache optimizations

• Priority to read misses over writes
  – Reduces miss penalty
  – Modest increase in design complexity
• Avoiding address translation before indexing
  – Reduces hit time
  – Not what operating systems expect
  – Frequent cache flushes or need PID tags
Advanced cache optimizations

Small & simple L1 caches

- Critical timing path is
  Access tags → compare tags → select line
- Direct-mapped can overlap tag compare with transmission of line data
- Lower associativity reduces power
  (fewer comparators, narrower data access)
Advanced cache optimizations

Small & simple L1 caches

Access time vs. L1 size and associativity
Advanced cache optimizations

Small & simple L1 caches

Read energy vs. L1 size and associativity
Advanced cache optimizations

Way prediction

- Used in MIPS R1000, ARM Cortex-A8
  - Helps where tag compares are serialized
  - Mispredict increases hit time
  - Accuracy >90% for 2-way, >80% 4-way
  - Inst. cache more predictable than data
- Way selection predicts line data and tag
Advanced cache optimizations

Pipelined cache

• Improves bandwidth
• Easier to do higher associativity
• Branch mispredict time increases
• Examples:
  Pentium was 1 cycle
  Pentium Pro – III were 2 cycles
  Pentium 4 – Core i7 are 4 cycles
Advanced cache optimizations

Non-blocking cache

• Allows hits before previous misses complete
  – “Hit under miss”
  – “Hit under multiple miss”
• Required for L2 caches
• Processors can’t hide L2 miss penalty
Advanced cache optimizations

Non-blocking cache
Advanced cache optimizations

Multi-banked cache

- Fragment cache into independent banks
  - ARM Cortex-A8 supports 1-4 banks of L2
  - Intel i7 supports 4-bank L1, 8-bank L2
- Banks commonly interleave on low bits of line address (sequential interleaving):
Advanced cache optimizations

Critical word first

- Requested word in line is fetched first
- Requested word is returned immediately upon arrival at the cache – don’t wait for full line
- Other words of line fetched in some order
- Can use rotated order
  - Start at word k, ith fetch is \((k+i)\%n\)
  - Sort-of like assuming sequential prefetch
Advanced cache optimizations

**Early restart**

- Requests words in normal order
- Requested word is returned immediately upon arrival at cache – don’t wait for full line
- Potentially simpler than critical word first, probably not as effective…
Advanced cache optimizations

Merging write buffer

• This is the write buffer described earlier… essentially a FIFO of lines
• Does NOT always treat write buffer as a FIFO
  – Each line tracks which fields are “present”
  – Collects word writes into the same line entry
  – I/O addresses must still be FIFO
• Increases effective size of write buffer
• If entire line is present, cache doesn’t need to read the missing parts
Linker optimization

- Changing link order can change caching by changing which addresses conflict in cache
  - If f() calls g(), different buckets for f() and g()
  - Profiling to detect conflict pattern
- Same idea can be used to pick addresses for data structures
Compiler optimizations

• Restructure code to change data access pattern
  – Group data (data layout)
  – Reorder accesses (loop transformations)
• Prevent cache pollution
  – Why cache what you get from a register?
  – Often double-map: cache / bypass
• Avoid saving data that isn’t used again
Compiler optimizations

Merging/splitting arrays

• Array elements accessed together can be grouped together to enhance spatial locality
• Also separate those not accessed together

E.g., suppose a[i] and c[i] accessed together:

```c
int a[N], b[N], c[N];
struct { int a, b, c; } abc[N];
struct { int a, c; } ac[N]; int b[N];
```
Compiler optimizations

Loop interchange

- Loop nest traversal order matches data layout
- Improves spatial locality

E.g., if \( a[0][0] \) is next to \( a[0][1] \):

```
for (i=0; i<N; ++i)
  for (j=0; j<M; ++j) a[i][j] = 0;
for (j=0; j<M; ++j)
  for (i=0; i<N; ++i) a[i][j] = 0;
```
Compiler optimizations

Loop fusion

• Fuse loops that work on similar data
• Improves spatial locality

```c
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<N; ++i)
    for (j=0; j<M; ++j) {
        a[i][j] = b[i][j] + c[i][j];
        d[i][j] = a[i][j] * c[i][j];
    }
```
Compiler optimizations

Loop blocking/stenciling

• Iterate in pattern that maximizes reuse

```c
for (i=0; i<N; ++i)
    for (j=0; j<N; ++j) {
        r = 0;
        for (k=0; k<N; ++k)
            r += y[i][k] * z[k][j];
        x[i][j] = r;
    }
```
Compiler optimizations

Loop blocking/stenciling

• Iterate in pattern that maximizes reuse

```c
for (jj=0; jj<N; jj+=B)
    for (kk=0; kk<N; 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 += y[i][k] * z[k][j];
                x[i][j] += r; }
```
Prefetching

• Software (by compiler)
  – Hoist load to earlier position in program
  – *Suggest* hardware load into cache

• Hardware
  – Assume or recognize reference pattern and request expected next early
  – Line +/-1, strided, other patterns

• Works better for instructions than data
• Generally can abort a prefetch to cache, Prefetches can’t fault (no exceptions)
Prefetching

- Fetch line and next line on a miss (Pentium 4)
<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>+</td>
<td>0</td>
<td>Trivial; widely used</td>
</tr>
<tr>
<td>Way-predicting caches</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td>+</td>
<td>1</td>
<td>Used in Pentium 4</td>
</tr>
<tr>
<td>Pipelined cache access</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td>1</td>
<td>Widely used</td>
</tr>
<tr>
<td>Nonblocking caches</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td></td>
<td></td>
<td>3</td>
<td>Widely used</td>
</tr>
<tr>
<td>Banked caches</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td>1</td>
<td>Used in L2 of both i7 and Cortex-A8</td>
</tr>
<tr>
<td>Critical word first and early restart</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
<td>Widely used</td>
</tr>
<tr>
<td>Merging write buffer</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td>1</td>
<td>Widely used with write through</td>
</tr>
<tr>
<td>Compiler techniques to reduce cache misses</td>
<td></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>
</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>3 instr., 3 data</td>
<td>Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware.</td>
</tr>
<tr>
<td>Compiler-controlled prefetching</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>Needs nonblocking cache; possible instruction overhead; in many CPUs</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.
Consistency Models

• The *volatile* keyword in C/C++ gives potential memory order constraints
• **Strict**: everybody sees result at next tick
• **Sequential**: everybody sees things as if they happened in a sequential order
• **Weak Ordering**: memory barriers/fences force ordering of before vs. after
Cache Coherence

• How one maintains consistency
• What to do when something writes?
  – **Invalidate**: mark/discard old entries
  – **Update**: use the write data to update
• Who to notify?
  – **Snooping**: everybody watches
  – **Ownership**: only talk to owner
  – **Directory**: permissions, who to notify
• **MESI Protocol**: Modified (dirty), Exclusive, Shared (clean), Invalid – 4 line states