Caches
Opening the black boxes that we had labeled with D$ and I$ ...

Readings
- Textbook: Chapter 5
- Recent Research Papers:
  - NUCA (by Kim et al.)
  - Value Prediction (by Lipasti and Shen)

Storage Hierarchy
CPU is small part of system (physically)
most of system is a hierarchy of storage
- closer to CPU = “higher” levels of hierarchy
  - lower capacity (smaller)
  - lower latency (faster)
  - higher bandwidth (more parallelism)
- we’ll study hierarchy top-down
  - 0: registers (already studied)
  - I: caches
  - II: main memory (and virtual memory)
  - III: I/O (disks, buses, network, etc.)

Storage Hierarchy Groups
hierarchy levels can be grouped in different ways
- by ISA visibility
  - registers
  - caches, memory, disk (swap): look like one thing
  - disk (file system)
- by implementation technology
  - registers, caches: SRAM (high-speed circuits)
  - main memory: DRAM (high-density circuits)
  - disk: magnetic iron oxide (electrical/mechanical)
  - we will use this division

A Little Bit of History
“Ideally one would desire an infinitely large memory capacity such that any particular ... word would be immediately available ... We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible”

–BGvN ‘46

(BGvN = Burks, Goldstein, and von Neumann)
Terminology

let $X$ be a structure in storage hierarchy (D$, L2, memory, or disk)

$X$ block: minimum storage unit (usually fixed, e.g., 64 bytes)

$X$ hit: block found in $X$

$X$ miss: block not found in $X$

$X$ miss ratio: fraction of accesses to $X$ that miss (local miss ratio)
  - global ratio = % accesses (incl those that hit above $X$) that miss in $X$

$X$ hit time: time to access $X$

$X$ miss penalty: time to get block into $X$ + time to get data to CPU
  - $X$ access time: time to get requested data (e.g., byte, word)
  - $X$ transfer time: time to get rest of block

Performance

time is the ultimate metric

- miss rate alone does not measure performance (why?)
- performance = function (hit time, miss rate, miss time)
- average (effective) access time is better metric
  
  \[
  t_{\text{avg}} = t_{\text{hit}} + (\%_{\text{miss}} \times t_{\text{miss}})
  \]

- $t_{\text{hit}} = 1$, $t_{\text{miss}} = 20$, $\%_{\text{miss}} = 5$
  
  \[
  t_{\text{avg}} = 1 + 0.05 \times 20 = 2
  \]

misses are much more expensive per instance than hits

Big Concept 1: Locality

hierarchy exploits locality to create illusion of large & fast storage

- temporal locality = locality in time
  - recently referenced data likely to be referenced again soon
  - each level stores most recently touched blocks from lower level

- spatial locality = locality in space
  - neighbors of recently referenced data likely to be referenced soon
  - lower level blocks are bigger, anticipating neighbor accesses

remember: make common case fast (or Amdahl will get you)

- common case: temporal & spatial locality
- fast: smaller, faster memory

Locality is one of the most important ideas in architecture

Big Concept 2: Balance

balance system by adjusting sizes of hierarchy components

- e.g., larger L1 $\Rightarrow$ higher hit rate $\Rightarrow$ lower L2 demand
- e.g., larger memory $\Rightarrow$ less paging $\Rightarrow$ lower I/O demand

- Amdahl's rule: 1 MIPS $\Leftrightarrow$ 1 MB memory $\Leftrightarrow$ 1 Mbit/s I/O
  - if corrected to 1 Mbyte/s of I/O, still valid!

balance example

- IPC = 1.5
  - 30% loads & stores
  - 90% D$ hit rate, 95% I$ hit rate, 32B blocks, no L2 (or L3)

- let's compute memory (i.e., beyond cache) b/w demand
  - mem b/w for instructions: $1.5 \times 1.0 \times 0.05 \times 32 = 2.4$ bytes/clock
  - mem b/w for data: $1.5 \times 0.3 \times 0.10 \times 32 = 1.44$ bytes/clock
  - total = 3.84 bytes/clock (on average - must design for variability)
Typical Storage Hierarchy Specs

<table>
<thead>
<tr>
<th>Type</th>
<th>Capacity</th>
<th>Latency</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>&lt; 2 KB</td>
<td>1 ns</td>
<td>150 GB/s</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>&lt; 64 KB</td>
<td>4 ns</td>
<td>50 GB/s</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>&lt; 8 MB</td>
<td>10 ns</td>
<td>25 GB/s</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>&lt; 64 MB</td>
<td>20 ns</td>
<td>10 GB/s</td>
</tr>
<tr>
<td>Memory</td>
<td>&lt; 4 GB</td>
<td>50 ns</td>
<td>4 GB/s</td>
</tr>
<tr>
<td>Disk</td>
<td>&gt; 10 GB</td>
<td>10 ms</td>
<td>10 MB/s</td>
</tr>
</tbody>
</table>

Storage Hierarchy I: Caches

caches: upper levels of memory hierarchy
- managed by hardware (almost always)
- implemented in SRAM
- recently: any locality exploiting buffer
  - e.g., file cache, Netscape page cache

preview of coming attractions:
- answers to four questions
- ABCs: associativity, block size, capacity
- 3C (4C) miss model
- advanced cache techniques
- later in course: cache coherence

Some Perspective

| 15% | 85% |

SRAM (Static Random Access Memory)

- “logic” (CPU process, registers are SRAM)
- store bits in flip-flops (cross-coupled NORs)
  - not very dense (six transistors per bit)
  - fast
  - doesn’t need to be “refreshed” (data stays as long as power is on)
Basic Cache

structure
- collection of “frames” for holding blocks of memory
- frame can hold: data + tag + state bits
  - tag: identifies the address of the block (why is this needed?)
  - state bits: valid (tag/data there), dirty (data has been written by us)

Basic Cache Organization

logical organization
- frames grouped into sets
- number of frames (ways) in each set is the associativity
  - if one frame per set, cache is direct-mapped
  - but how do we know where to look in the cache?

Mapping Addresses to Frames

divide address into offset, index, tag
- offset: specifies the word within a cache block
  - O-bit offset \(\Rightarrow 2^O\)-byte block size
- index: specifies the set containing a block frame
  - N-bit index \(\Rightarrow 2^N\) sets in cache
  - direct-mapped cache: index finds frame directly
- tag: remaining bits not implied by block frame, must match
  - Q: but how do you find the way within a set?
Cache Access Example

- 32-bit machine
- 4KB cache, 16B block, direct-mapped
  - 16B blocks ⇒ 4 offset bits
- 4KB cache / 16B blocks ⇒ 256 frames
- 256 frames / 1-way (DM) ⇒ 256 sets ⇒ 8 index bits
- 32-bit address – 4 offset bits – 8 index bits ⇒ 20 tag bits
- memory: @0x1400fa20: 007CFFFF 1200F01C 1200F448 00000001

ldl R4, 0x1400fa24 (0001 0010 0000 0000 1111 1010 0010 0100)
- offset = 0x4 (0100)
- block address = 0x1400fa20
- set index = 0xa2 (1010 0010)
- tag = 0x1400f (0001 0010 0000 0000 1111)
- R4 = 0x1200F01C (we’re assuming big endian, so start from left)

Cache Example: Tag Array Size

- 32-bit machine
- 64KB cache, 32B block, 2-way set-associative cache
- compute total size of tag array (not including state bits)
  - 64KB cache / 32B block ⇒ 2K blocks
  - 2K blocks / 2-way set-associative ⇒ 1K sets
  - 32B blocks ⇒ 5 offset bits
  - 1K sets ⇒ 10 index bits
  - 32-bit addresses – 5 offset bits – 10 index bits ⇒ 17 tag bits
  - 17 tag bits * 2K blocks ⇒ 34Kb ⇒ 4.25KB

The Four Questions

1. where can a block be placed? block placement
2. how is a block found? block identification
3. which block is replaced on a miss? block replacement
4. what happens on a write (store)? write strategy

Block Placement + Identification

placement
- invariant: block always goes in exactly one set
- fully-associative: cache is one set, block goes anywhere
- direct-mapped: block goes in exactly one frame
- set-associative: block goes in any one of a few frames

identification
- find set
- search ways in parallel (compare tags, check valid bits)
  - higher associativity = more hardware, slower access time
Block Replacement

some options

• least recently used (LRU)
  • optimized for temporal locality, complicated to implement
• (pseudo) random
  • nearly as good as LRU, much simpler to implement
• not most recently used (NMRU)
  • track MRU, random select from others, good compromise
• optimal, but not implementable (Belady’s algorithm)
  • replace block used furthest ahead in time (theoretical limit)

replacement is not an issue for direct-mapped caches

Write Policies (2 separate questions)

Q1: do we propagate the new value to level below (i.e., update)?
  • yes: write-through (propagate all writes to level below)
    • update traffic is independent of cache hit rate (bad)
    • updates per cache reference = \( f_{write} \times \text{access size} \)
  • no: write-back (update only on block replacement)
    • set dirty bit on write, replace “clean” blocks without update
      + less traffic for larger caches (higher hit rates)
      + multiple writes to same line combined into 1 update
    • updates per reference = \( f_{dirty} \times \%\text{miss} \times \text{block size} \)

Q2: on a miss, allocate a cache block?
  • yes: write-allocate (usually with write-back)
  • no: no-write-allocate (usually with write-through)

Fundamental Parameters: ABCs of Caches

• associativity
• block size
• capacity

We’re going to look at these in reverse order ...

Capacity (Cache Size)

total data (not including tag) capacity of cache
  • bigger not ALWAYS better

too small
  – doesn’t exploit temporal locality well
  – useful data prematurely replaced

too large
  – too slow
  • longer access may slow clock or increase cycles (more likely)
Block Size

minimal data size
  • associated with a tag / transferred from memory
too small
  – doesn’t exploit spatial locality well
  – inordinate tag overhead (= tag storage / data storage)
too large
  – useless data transferred (i.e., not enough spatial locality)
  – premature replacement (since fewer blocks in cache)

Associativity

number of frames (i.e., ways) in each set
  • typical values: 1, 2, 3, 4, 5, 8, 16
large associativity
  + lower miss rate
  + less variability among programs
small associativity
  + faster hit time (perhaps)
    • for given $t_{hit}$, can build bigger DM cache than SA cache
    • remember why is DM faster than SA?

Tag/Data Access

in parallel (common organization)
  + faster
in series (first tag, then data)
  + uses less power
    • don’t fire up data array on miss
    • SA? fire up only one data array

Write Buffers

parallel data access/tag check only for reads!!
  • cannot be done for writes
solution: write buffer (WB) (WB != STQ)
  • small buffer for committed writes
  • allows writes to be pipelined
    • check tag
    • write store data into WB
    • write data from WB to cache (tags OK now)
      • in parallel!
  • problem: read to data in write buffer
    • match read addresses to WB addresses
      • stall or bypass
**Write Buffers**

- can keep writes in write buffer for a while
  - reads can still proceed (as described)
  - coalesce (combine) writes to same block
    - fewer writes, fast burst transfer (transfer of entire block)
  - also good for write-thru caches (don’t stall for memory)

**Writeback Buffers**

- like a write buffer, but between cache and memory
  + allows replacement to be performed before writeback
    - replacement is more latency critical (someone is waiting)

![Diagram]

**Advanced Cache Topics**

- evaluation methods
- reducing miss rate
- reducing miss penalty
- reducing hit time
- increasing bandwidth (for superscalar processors)

**Evaluation Methods**

- hardware counters
  + accurate, realistic workloads (system, user, everything)
    - machine must exist, can’t vary cache parameters, non-deterministic
- analytical models (mathematical expressions)
  + fast, good insight (can vary parameters)
    - tough to model cache behaviors, finding parameters is difficult
- trace-driven simulation (get $\%_{\text{miss}}$, plug in $t_{\text{hit}}$ and $t_{\text{miss}}$)
  + experiments repeatable, can be accurate
    - time consuming, don’t model speculative execution
- full processor simulation
  + true performance (parallel misses, prefetches, speculation effects)
    - complicated simulation model, very time consuming
Miss Classification: 3C’s (4C’s)
taxonomy due to Mark Hill and Alan J. Smith

- **compulsory (cold-start):** first ever access to a block
  - would miss even in an infinite-sized cache
- **capacity:** miss because cache not big enough
  - would miss even in fully associative cache
- **conflict:** miss because of low associativity
  - remaining misses
- **coherence:** misses due to external invalidations
  - only in systems where other components can access memory (e.g., multithreaded processors, multiprocessors, systems with I/O)

purpose of classification?
- different techniques for attacking different misses

Miss Classification Example

32B cache, 8B blocks, direct-mapped
- 4 blocks, 4 sets

<table>
<thead>
<tr>
<th>contents</th>
<th>address</th>
<th>miss? which?</th>
</tr>
</thead>
<tbody>
<tr>
<td>010, 020, 030, 040</td>
<td>044</td>
<td>hit</td>
</tr>
<tr>
<td>010, 020, 030, 040</td>
<td>140</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>010, 020, 030, 140</td>
<td>210</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>210, 020, 030, 140</td>
<td>010</td>
<td>conflict miss</td>
</tr>
<tr>
<td>010, 020, 030, 140</td>
<td>220</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>010, 220, 030, 140</td>
<td>040</td>
<td>capacity miss</td>
</tr>
</tbody>
</table>

- first time see block ⇒ compulsory
- 4 (N) distinct blocks used since last access ⇒ capacity
- everything else ⇒ conflict

Miss Rate: Work with Cache Parameters

key: think about which kinds of misses you are reducing

- increase capacity (obviously)
  + monotonically reduce capacity misses (approx: size/2 ⇒ %miss/2)
    - may slow hit time
- increase block size (while fixing capacity & associativity)
  + reduce compulsory & capacity misses (spatial prefetching)
    - increase capacity & conflict misses (fewer frames)
    - typically, miss rate drops up to 32/64B, increases afterwards
    - increase memory traffic
- increase associativity (fixed capacity & block size)
  + reduce associativity (monotonically, up to fully-associative)
    - hard to implement, may slow hits

Let's look at some other ways to attack each type of miss

Conflict Misses: Skewed Associativity

observation: if addresses conflict in 1 way, conflict in all
- e.g., 3 addresses with same index “thrash” in 2-way cache
- solution: different mapping function for each way! [Seznec]
- e.g., 2-way skewed
  - divide index+tag into two sets of bits (a1, a2)
  - mapping functions: way 0 ⇒ f0(a1,a2), way 1 ⇒ f1(a1,a2)
- performance
  - not much higher, but stable (no pathologic cases)
- upshot
  - not implemented - hardware is too messy

(This is not a topic I’ll be testing you on)
**Conflict Misses: Victim Buffer**

**observation:** high associativity needed infrequently, dynamically

- **solution:** victim buffer [by Jouppi]
  - add a small fully associative buffer between Level_1 and Level_{i+1}
  - holds victims replaced by cache
  - miss in cache + hit in victim buffer ⇒ move line into cache
- **performance**
  - even one entry VB helps some benchmarks (wow!)
  - I-cache helped more than D-cache (why?)
  - generally helps more for smaller caches (why?)
  - helps more with larger line size (why?)
- **upshot**
  - commonly implemented

---

**Capacity Misses: Software Restructuring**

re-order program accesses to improve locality

- e.g., loop-interchange for column-major matrix
  - assume rows in matrix are in consecutive memory addresses
  - poor code (doesn't exploit spatial locality)
    
    ```
    for (i=0; i<ROWS; i++)
    for (j=0; j<COLS; j++)
    sum += x[i][j]
    ```
  - better code
    
    ```
    for (j=0; j<COLS; j++)
    for (i=0; i<ROWS; i++)
    sum += x[i][j]
    ```

- do this automatically in compiler? must check if valid (hard)

---

**Software Restructuring**

- **loop blocking**
  - cut array into cache-size chunks
  - run all phases on one chunk, proceed to next chunk
    
    ```
    for (k=0; k<ITERATIONS; k++)
    for (i=0; i<ELEMS;i++)
    // do something
    ```

    ```
    for (i=0; i<ELEMS; i+=CACHE_SIZE)
    for (k=0; k<ITERATIONS; k++)
    for (ii=i; i<i+CACHE_SIZE-1; ii++)
    // do something
    ```

- **loop fusion**
  - similar

---

**Capacity Misses: Prefetching**

even "demand fetching" prefetches other words in block

- **spatial prefetching**

prefetching should...

- get data before it is referenced
- avoid late prefetching
- not get data that will not be used
- avoid useless prefetching
- not prematurely replace prefetched data
- avoid early prefetching

prefetching uses extra bandwidth (if it isn’t done perfectly)

prefetching is a classic latency/bandwidth tradeoff
Software Prefetching

- **binding**: prefetch into register
  - no ISA support, use normal loads
  - need more registers, what about faults?
  - upshot: not used much (except for software pipelining)

- **non-binding**: prefetch into cache (e.g., below)
  - need ISA support (non-binding, non-faulting loads)
  - simpler semantics, preferred

```c
for (j=0; j<COLS; j++)
  for (i=0; i<ROWS; i+=BLOCK_SIZE)
    prefetch (&x[i][j]+BLOCK_SIZE);
  for (ii=i; ii<i+BLOCK_SIZE-1; ii++)
    sum += x[ii][j];
```

Hardware Prefetching

**options for what to prefetch**

- simplest: one block ahead (spatially)
  - works well for instructions and sequential data (arrays)

- more complex: use “address prediction”
  - needed for non-sequential data (e.g., linked list)

**options for when to prefetch**

- on every reference
- on a miss (i.e., effectively double block size)
  - better performance than doubling block size (why?)
- when resident block becomes dead [ISCA'01]
  - when no one will use it anymore (how do we know this?)

Hardware prefetching is and has been a hot research topic

Hardware Prefetching: Stream Buffers

*stream buffers*: same paper as victim buffers [Jouppi]

- prefetch into buffers, NOT into cache
  - on miss: start filling stream buffer with successive lines
  - on access: check both cache and stream buffer
  - SB hit ⇒ move line to cache, miss both ⇒ clear/refill SB

- performance
  - very effective for I$
  - less so for D$ (why?)
  - fix with multiple stream buffers

what behavior is this design targeting?

- i.e., why not just prefetch into the cache?
- how do we fairly evaluate SB?
  - hint: SB uses more transistors than just a cache ...

Hardware Prefetching: Address Prediction

**address-prediction**

- easy for arrays, harder for pointer-based data structures

some options for pointer-based code (e.g., linked lists)

- cache conscious layout/malloc
  - lays lists out serially in memory, makes them look like arrays

- correlated predictors
  - large tables (maintain <miss, next miss> pairs)

- dependence based prefetching (pre-execution)
  - FSM greedily chases pointers from fetched blocks

- jump-pointers (software assistance)
  - augment data structure with prefetch pointers
Miss Cost: Early Restart/Critical Word First

**Observation:** \( t_{\text{miss}} = t_{\text{access}} + t_{\text{transfer}} \)
- \( t_{\text{access}} \): time to get first word
- \( t_{\text{transfer}} \): time to get rest of block
- implies whole block loaded before data returned to CPU!

**Optimization:** early restart/critical word first
- **Critical word first:** requested word returned first
  - must arrange for this to happen (bus & memory must cooperate)
- **Early restart:** send requested word to CPU immediately
  - get rest of block, load into cache in parallel

Miss Cost: Sub-Blocking

**Observation:** not all parts of block equally important
- to reduce memory required for tags
  - make tags smaller
  - the resulting blocks are large
- To reduce miss penalty
  - don’t load full block on a miss
  - have bits per subblock to indicate valid

Miss Cost: Lock-up Free Caches

Normal cache stalls while a miss is pending

**Lock-up free caches** [Kroft'81] (aka “non-blocking cache”)
- handle hits while miss is pending
  - “hit under miss” (very common)
- handle misses while miss is pending
  - overlapping misses (less common, but miss serialization costly)
- only makes sense in following situations:
  - if processor can do useful work under a miss (dynamic scheduled)
  - if processor/program has misses that can be overlapped
  - for data cache (why?)
- implementation: MSHR (miss status holding register)
- catch? bus must be pipelined/split transaction

Miss Cost: L2 Caches

**Observation:** CPU getting faster w.r.t memory (memory gap)
- miss to memory is more costly
- larger L1 may make hits too slow, or may not fit on chip

**Solution:** second-level (L2) cache
- reduces frequency of misses to memory
- L1 hits (common case) still fast
- exploit technological boundaries
  - on-chip vs. off-chip (tags on-chip, data off-chip)
  - if off-chip, can have multiple sizes (within same processor family)
**Measuring L2 Cache Performance**

Q: how is $t_{avg}$ affected by the addition of an L2?

A: depends on how we define $%_{miss-L2}$ and $t_{miss-L1}$
- **local L2 miss rate**: L2 misses / L2 accesses
  - "filtered" miss rate, used to calculate $t_{miss-L1}$
  - $t_{avg} = t_{hit-L1} + (%_{miss-L1} \times (t_{hit-L2} + (%_{miss-local-L2} \times t_{miss-L2})))$
- **global L2 miss rate**: L2 misses / total references
  - "unfiltered" miss rate, used directly
  - $t_{avg} = t_{hit-L1} + (%_{miss-L1} \times t_{hit-L2}) + (%_{miss-global-L2} \times t_{miss-L2})$
- **solo L2 miss rate**: pretend L2 is the only cache
  - similar to global miss rate for large L2s ( $> 8 \times L1$)

upshot: choose definition that fits situation

---

**Designing an L2 Cache**

apply L1 principles to L2 design? not necessarily
- low latency, high b/w less important (why?)
- low miss rate very important (why?)

design L2 for low miss rates
- unified: better frame utilization
- large size: 256KB to 8MB (latency be damned)
- large block size: 64B or 128B (more spatial prefetching)
- high associativity: 4, 8, 16 (again, latency be damned)
- blocking? (multiple L2 misses less common)

important: design L1 and L2 together (match b/w & latencies)

---

**Multi-Level Inclusion?**

multi-level inclusion: L2 cache contents **always** superset of L1
- + filters coherence traffic (later)
- + makes L1 writes simpler (why?)
  - local LRU (L1&L2 independently) can't guarantee it (why?)
    - e.g., 2-block L1, 3-block L3, local LRU, references: 1,2,1,3,1,4
    - final L1 contents: 1,4, final L2 contents: 2,3,4
  - takes effort to maintain [Wang,Baer,Levy]
    - L2 blocksize/L1blocksize pointers per L2 block give L1 contents
    - invalidate from L1 before replacing from L2

options other than inclusion
- exclusion (if in L1, then NOT in L2, and vice versa)
- neither (neither inclusion nor exclusion)

---

**Hit Latency Reduction Techniques**

obvious/have seen/later
- small/simple caches
- parallel tag/data for fast reads
- pipelining (write buffers) for fast writes
- avoid address translation during indexing (VM, later)
Cache Bandwidth

Superscalar processors need multiple cache accesses per cycle
- Parallel cache accesses harder than parallel ALU ops
- Difference? Caches have state
- Operation thru one port affects future operations thru others

Bandwidth Techniques

- True multiporting (N = number of ports)
  - Multiple decoders, multiple r/w lines for every SRAM bit
  - No bandwidth loss due to conflicts (any combination of accesses)
  - Cache area: \(O(N^2)\), access time = wire length: \(O(N^2)\)
- Virtual multiporting (time multiplexing)
  - Pipeline a single port (time share on clock edges)
  - Not scalable beyond 2 ports
  - Alpha 21264, HP PA-8X00: wave pipelining (no latches)
  - Multiple values on wire at precise intervals (circuit magic)
- Multiple cache copies (don't laugh)
  - Replicate tag/data arrays (Alpha 21164)
  - Independent load ports, single shared store port (why?)
  - No load b/w loss, smaller than true multiporting (why?)
  - No added store bandwidth, not scalable beyond 2 paths

Bandwidth: Multibanking (Interleaving)

- Divide cache into banks
- Each bank usually a subset of the sets (why?)
- Pre-determine which bank to access (using address bits)
- Allow parallel access to different banks
- Two accesses to same bank \(\Rightarrow\) conflict (one must stall)
- Low area, few conflicts if sufficient banks
- Observe: requests to same bank often to same block! exploit?

Miss Rate + Bandwidth: Unified vs. Split

Q: Instructions and data together or in separate caches?

- Unified I and D
  + Cheaper to build
  + Higher utilization (dynamic load balance)
  + Handles writes to I-stream (self-modifying code, JIT)
    - I/D conflicts (both access and block)
- Split I and D (Harvard)
  + 2x bandwidth, place close to I/D ports
  + No I/D conflicts
    - Self-modifying code can cause problems
- Bottom line: Split if simultaneous I and D accesses frequent
  + Split L1
  + Unified L2 (and beyond)
Cache Performance Summary

\[ t_{avg} = t_{hit} + \%_{miss} \cdot t_{miss} \]

- reducing miss rate (\%_{miss})
- large block size, higher associativity, victim buffers
- software restructuring, prefetching
- reducing miss penalty (t_{miss})
- early restart/critical word first, sub-blocking
- non-blocking caches, L2 caches
- reducing hit time (t_{hit})
- small and simple caches
- parallel tag/data for reads, pipelining (write buffer) for writes
- increasing bandwidth
- banking/interleaving, multiporting
- tradeoff: more b/w => less latency

Memory Hierarchy Summary

L1, L2, L3 have different design constraints

- L1: low latency, high b/w (primary), low miss rate
- L2: low miss rate, low latency
- L3: starting to appear, very low miss rate

Design hierarchy together to match capacities, bandwidths

next up: main memory + virtual memory
then: I/O (disks, network, etc.)