Classifying Misses: 3C Model (Hill)

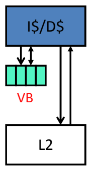
- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - **Would miss even in infinite cache**
  - **Capacity**: miss caused because cache is too small
    - **Would miss even in fully associative cache**
    - Identify? Consecutive accesses to block separated by access to at least N other distinct blocks (N is number of entries in cache)
  - **Conflict**: miss caused because cache associativity is too low
    - Identify? **All other misses**
  - **(Coherence)**: miss due to external invalidations
    - Only in shared memory multiprocessors (later)
- Calculated by multiple simulations
  - Simulate infinite cache, fully-associative cache, normal cache
  - Subtract to find each count

Miss Rate: ABC

- Why do we care about 3C miss model?
  - So that we know what to do to eliminate misses
  - If you don't have conflict misses, increasing associativity won't help
- **Associativity**
  - + Decreases conflict misses
  - – Increases latency$_{hit}$
- **Block size**
  - – Increases conflict/capacity misses (fewer entries)
  - + Decreases compulsory/capacity misses (spatial locality)
  - No significant effect on latency$_{hit}$
- **Capacity**
  - + Decreases capacity misses
  - – Increases latency$_{hit}$

Reducing Conflict Misses: Victim Buffer

- Conflict misses: not enough associativity
  - High-associativity is expensive, but also rarely needed
    - 3 blocks mapping to same 2-way set and accessed (XYZ)+
- **Victim buffer (VB)**: small fully-associative cache
  - Sits on I$/D$ miss path
  - Small so very fast (e.g., 8 entries)
  - Blocks kicked out of I$/D$ placed in VB
  - On miss, check VB: hit? Place block back in I$/D$
  - 8 extra ways, shared among all sets
    - + Only a few sets will need it at any given time
  - + Very effective in practice
  - Does VB reduce %$_{miss}$ or latency$_{miss}$?



Overlapping Misses: Lockup Free Cache

- **Lockup free**: allows other accesses while miss is pending
  - Consider: `load [r1]→r2; load [r3]→r4; add r2,r4 →r5`
  - **Handle misses in parallel**
    - "memory-level parallelism"
  - Makes sense for…
    - Processors that can go ahead despite D$ miss (out-of-order)
  - Implementation: **miss status holding register (MSHR)**
    - Remember: miss address, chosen entry, requesting instruction
    - When miss returns know where to put block, who to inform
  - Common scenario: "hit under miss"
    - Handle hits while miss is pending
    - Easy
  - Less common, but common enough: "miss under miss"
    - A little trickier, but common anyway
    - Requires multiple MSHRs: search to avoid frame conflicts

Software Restructuring: Data

- Capacity misses: poor spatial or temporal locality
  - Several code restructuring techniques to improve both
  - – Compiler must know that restructuring preserves semantics

Loop interchange: spatial locality
- Example: row-major matrix: `X[i][j]` followed by `X[i][j+1]`
- Poor code: `X[i][j]` followed by `X[i+1][j]`

```
for (j = 0; j<NCOLS; j++)
   for (i = 0; i<NROWS; i++)
      sum += X[i][j];    // say
```

- Better code

```
for (i = 0; i<NROWS; i++)
   for (j = 0; j<NCOLS; j++)
      sum += X[i][j];
```

Software Restructuring: Data

- **Loop blocking**: temporal locality
  - Poor code

```
for (k=0; k<NITERATIONS; k++)
   for (i=0; i<NELEMS; i++)
      sum += X[i];    // say
```

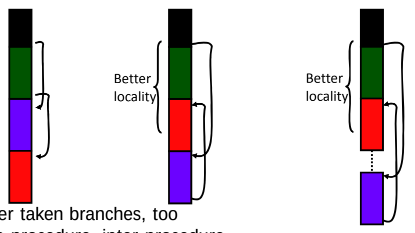
  - Better code
    - Cut array into CACHE_SIZE chunks
    - Run all phases on one chunk, proceed to next chunk

```
for (i=0; i<NELEMS; i+=CACHE_SIZE)
   for (k=0; k<NITERATIONS; k++)
      for (ii=0; ii<i+CACHE_SIZE-1; ii++)
         sum += X[ii];
```

  - – Assumes you know **CACHE_SIZE**, do you?
  - Loop fusion: similar, but for multiple consecutive loops

Software Restructuring: Code

- Compiler can layout code for temporal and spatial locality
  - If (a) { **code1;** } else { **code2;** } **code3;**
  - But, code2 case never happens (say, error condition)

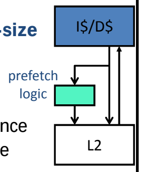


- Fewer taken branches, too
- Intra-procedure, inter-procedure

Prefetching

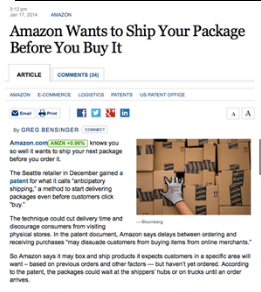
Prefetching: put blocks in cache proactively/speculatively

- Key: anticipate upcoming miss addresses accurately
  - Can do in software or hardware
- Simple example: **next block prefetching**
  - Miss on addr **X** → anticipate miss on **X+block-size**
  - + Works for insns: sequential execution
  - + Works for data: arrays



- **Timeliness**: initiate prefetches sufficiently in advance
- **Coverage**: prefetch for as many misses as possible
- **Accuracy**: don't pollute with unnecessary data
  - It evicts useful data

Prefetching in the Real World!



3:12 pm
Jan 17, 2014 AMAZON

Amazon Wants to Ship Your Package Before You Buy It

ARTICLE COMMENTS (34)

AMAZON E-COMMERCE LOGISTICS PATENTS US PATENT OFFICE

By GREG BENSINGER

Amazon.com AMZN +0.96% knows you so well it wants to ship your next package before you order it.

The Seattle retailer in December gained a patent for what it calls "anticipatory shipping," a method to start delivering packages even before customers click "buy."

— Bloomberg

The technique could cut delivery time and discourage consumers from visiting physical stores. In the patent document, Amazon says delays between ordering and receiving purchases "may dissuade customers from buying items from online merchants."

So Amazon says it may box and ship products it expects customers in a specific area will want – based on previous orders and other factors — but haven't yet ordered. According to the patent, the packages could wait at the shippers' hubs or on trucks until an order arrives.

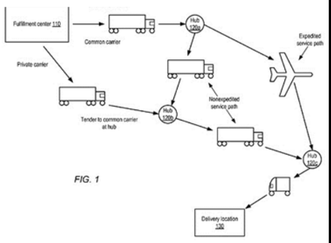


FIG. 1

Software Prefetching

- Use a special "prefetch" instruction
  - Tells the hardware to bring in data, doesn't actually read it
  - Just a hint
- Inserted by programmer or compiler

Example:

```
for (i = 0; i<NROWS; i++)
    for (j = 0; j<NCOLS; j+=BLOCK_SIZE) {
        __builtin_prefetch(&X[i][j]+BLOCK_SIZE);
        for (jj=j; jj<j+BLOCK_SIZE-1; jj++)
            sum += x[i][jj];
    }
```

- Multiple prefetches bring multiple blocks in parallel
  - Using lockup-free caches
  - "Memory-level" parallelism

Hardware Prefetching

- What to prefetch?
  - **Stride-based sequential prefetching**
    - Can also do N blocks ahead to hide more latency
    - + Simple, works for sequential things: insns, array data
    - + Works better than doubling the block size
  - **Address-prediction**
    - Needed for non-sequential data: lists, trees, etc.
    - Use a hardware table to detect strides, common patterns
- When to prefetch?
  - On every reference?
  - On every miss?

More Advanced Address Prediction

- "Next-block" prefetching is easy, what about other options?
- **Correlating predictor**
  - Large table stores (miss-addr → next-miss-addr) pairs
  - On miss, access table to find out what will miss next
    - It's OK for this table to be large and slow
- Content-directed or dependence-based prefetching
  - Greedily chases pointers from fetched blocks
- Jump pointers
  - Augment data structure with prefetch pointers
- Make it easier to prefetch: cache-conscious layout/malloc
  - Lays lists out serially in memory, so they look like arrays
- Active area of research

Write Issues

- So far we have looked at reading from cache
  - Instruction fetches, loads
- What about writing into cache
  - Stores, not an issue for instruction caches (why they are simpler)
- Several new issues
  - Tag/data access
  - Write-through vs. write-back
  - Write-allocate vs. write-not-allocate
  - Hiding write miss latency

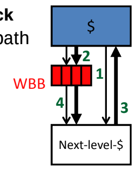
Tag/Data Access

- Reads: read tag and data in parallel
  - Tag mis-match → data is garbage (OK, stall until good data arrives)
- Writes: read tag, write data in parallel?
  - Tag mis-match → clobbered data (oops)
  - For associative caches, which way was written into?
- Writes are a pipelined two step (multi-cycle) process
  - Step 1: match tag
  - Step 2: write to matching way
  - Bypass (with address check) to avoid load stalls
  - May introduce structural hazards

Write Propagation

When to propagate new value to (lower level) memory?

- **Option #1: Write-through**: immediately
  - On hit, update cache
  - Immediately send the write to the next level
- **Option #2: Write-back**: when block is replaced
  - Requires additional "dirty" bit per block
    - Replace **clean** block: **no extra traffic**
    - Replace **dirty** block: **extra "writeback" of block**
  - + **Writeback-buffer (WBB)**: keep it off critical path
    1. Send "fill" request to next-level
    2. While waiting, write dirty block to buffer
    3. When new blocks arrives, put it into cache
    4. Write buffer contents to next-level



Write Propagation Comparison

- **Write-through**
  – Requires additional bus bandwidth
    - Consider repeated write hits
  – Next level must handle small writes (1, 2, 4, 8-bytes)
  + No need for dirty bits in cache
  + No need to handle "writeback" operations
    - Simplifies miss handling (no write-back buffer)
  - Sometimes used for L1 caches (for example, by IBM)
- **Write-back**
  + Key advantage: uses less bandwidth
  - Reverse of other pros/cons above
  - Used by Intel and AMD
  - 2nd-level and beyond are generally write-back caches

Write Miss Handling

How is a write miss handled?

Write-allocate: fill block from next level, then write it

+ Decreases read misses (next read to block will hit)

– Requires additional bandwidth

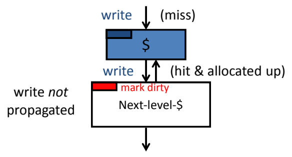
- Commonly used (especially with **write-back** caches)

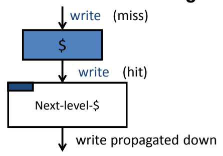
Write-non-allocate: just write to next level, no allocate

– Potentially more read misses

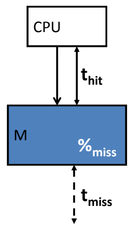
+ Uses less bandwidth

- Use with **write-through**





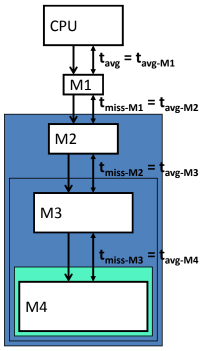
Memory Performance Equation



- **Access**: read or write to M
- **Hit**: desired data found in M
- **Miss**: desired data not found in M
  - Must get from another (slower) component
- **Fill**: action of placing data in M
- $\%_{miss}$ (miss-rate): #misses / #accesses
- t_{hit}: time to read data from (write data to) M
- t_{miss}: time to read data into M
- Performance metric
  - t_{avg}: average access time

$$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$$

Hierarchy Performance


CPU
$t_{avg} = t_{avg\text{-}M1}$
M1
$t_{miss\text{-}M1} = t_{avg\text{-}M2}$
M2
$t_{miss\text{-}M2} = t_{avg\text{-}M3}$
M3
$t_{miss\text{-}M3} = t_{avg\text{-}M4}$
M4

t_{avg} =
$t_{avg\text{-}M1}$
$t_{hit\text{-}M1} + (\%_{miss\text{-}M1} * t_{miss\text{-}M1})$
$t_{hit\text{-}M1} + (\%_{miss\text{-}M1} * t_{avg\text{-}M2})$
$t_{hit\text{-}M1} + (\%_{miss\text{-}M1} * (t_{hit\text{-}M2} + (\%_{miss\text{-}M2} * t_{miss\text{-}M2})))$
$t_{hit\text{-}M1} + (\%_{miss\text{-}M1} * (t_{hit\text{-}M2} + (\%_{miss\text{-}M2} * t_{avg\text{-}M3})))$
…

Performance Calculation with $ Hierarchy

- **Parameters**
  - Reference stream: all loads
  - D$: t_{hit} = 1ns, $\%_{miss}$ = 5%
  - L2: t_{hit} = 10ns, $\%_{miss}$ = 20% (local miss rate)
  - Main memory: t_{hit} = 50ns
- **What is $t_{avgD\$}$ without an L2?**
  - $t_{missD\$}$ =
  - $t_{avgD\$}$ =
- **What is $t_{avgD\$}$ with an L2?**
  - $t_{missD\$}$ =
  - t_{avgL2} =
  - $t_{avgD\$}$ =

Performance Calculation with $ Hierarchy

- **Parameters**
  - Reference stream: all loads
  - D$: t_{hit} = 1ns, $\%_{miss}$ = 5%
  - L2: t_{hit} = 10ns, $\%_{miss}$ = 20% (local miss rate)
  - Main memory: t_{hit} = 50ns
- **What is $t_{avgD\$}$ without an L2?**
  - $t_{missD\$} = t_{hitM}$
  - $t_{avgD\$} = t_{hitD\$} + \%_{missD\$} * t_{hitM}$ = 1ns+(0.05*50ns) = 3.5ns
- **What is $t_{avgD\$}$ with an L2?**
  - $t_{missD\$} = t_{avgL2}$
  - $t_{avgL2} = t_{hitL2} + \%_{missL2} * t_{hitM}$ = 10ns+(0.2*50ns) = 20ns
  - $t_{avgD\$} = t_{hitD\$} + \%_{missD\$} * t_{avgL2}$ = 1ns+(0.05*20ns) = 2ns

Designing a Cache Hierarchy

- For any memory component: t_{hit} vs. $\%_{miss}$ tradeoff

- Upper components (I$, D$) emphasize low t_{hit}
  - Frequent access → t_{hit} important
  - t_{miss} is not bad → $\%_{miss}$ less important
  - Low capacity/associativity (to reduce t_{hit})
  - Small-medium block-size (to reduce conflicts)

- Moving down (L2, L3) emphasis turns to $\%_{miss}$
  - Infrequent access → t_{hit} less important
  - t_{miss} is bad → $\%_{miss}$ important
  - High capacity/associativity/block size (to reduce $\%_{miss}$)

Memory Hierarchy Parameters

Parameter	I$/D$	L2	L3	Main Memory
t_{hit}	2ns	10ns	30ns	100ns
t_{miss}	**10ns**	**30ns**	**100ns**	**10ms (10M ns)**
Capacity	8KB–64KB	256KB–8MB	2–16MB	1-4GBs
Block size	16B–64B	32B–128B	32B-256B	NA
Associativity	1–4	4–16	4-16	NA

- Some other design parameters
  - Split vs. unified insns/data
  - Inclusion vs. exclusion vs. nothing
  - On-chip, off-chip, or partially on-chip?

Split vs. Unified Caches

Split I$/D$: insns and data in different caches
- To minimize structural hazards and t_{hit}
- Larger unified I$/D$ would be slow, 2nd port even slower
- Optimize I$ for wide output (superscalar), no writes

Unified L2, L3: insns and data together
- To minimize $\%_{miss}$
- + Fewer capacity misses: unused insn capacity used for data
- – More conflict misses: insn/data conflicts
  - A much smaller effect in large caches
- Insn/data structural hazards are rare: simultaneous I$/D$ miss
- Go even further: unify L2, L3 of multiple cores in a multi-core

Hierarchy: Inclusion versus Exclusion

- **Inclusion**
  - A block in the L1 is always in the L2
  - Good for write-through L1s (why?)
- **Exclusion**
  - Block is either in L1 or L2 (never both)
  - Good if L2 is small relative to L1
    - Example: AMD's Duron 64KB L1s, 64KB L2
- **Non-inclusion**
  - No guarantees

Summary

- **Average access time** of a memory component
  - $latency_{avg} = latency_{hit} + \%_{miss} * latency_{miss}$
  - low $latency_{hit}$ and $\%_{miss}$ in one structure = hard → hierarchy
- **Memory hierarchy**
  - Cache (SRAM) → memory (DRAM) → swap (Disk)
  - Smaller, faster, more expensive → bigger, slower, cheaper
- Cache ABCs (**associativity, block size, capacity**)
  - 3C miss model: compulsory, capacity, conflict
- **Performance optimizations**
  - $\%_{miss}$: prefetching
  - $latency_{miss}$: victim buffer, critical-word-first, lockup-free design
- **Write issues**
  - Write-back vs. write-through
  - write-allocate vs. write-no-allocate