Reducing Miss Cost

Assume 8 cycles before delivering 2 words/cycle

- $t_{memory} = t_{access} + B \times t_{transfer} = 8 + B \times \frac{1}{2}$
- $B$ is the block size in words
- whole block loaded before CPU gets data

Assume requested word first

- CPU gets data before caches loads it in data array
- $t_{memory} = t_{access} + MB \times t_{transfer} = 8 + 2 \times \frac{1}{2}$
- $MB$ is memory bus width in words
Norm Jouppi’s Victim Cache

Targets conflict misses
Victim cache: a small fully-associative cache capturing “victims”
- victims are conflicts in a set in DM or low-assoc cache
- LRU replacement

A miss in cache + a hit in victim cache
- move line to main cache
- is effectively equal to fast miss handling (or slow hits)
Victim Cache’s Performance

Removing conflict misses
   • even one entry helps some benchmarks
   • helps L-cache more than D-cache

Compared to cache size
   • generally, victim cache helps more for smaller caches

Compared to line size
   • helps more with larger line size (why?)

Multi-Level Caches

```
Proc
  \_ L1I
  \_ \_ L1D
  \_ \_ L2
```
**Why Multi-Level Caches?**

Processors getting faster w.r.t. main memory
- larger caches to reduce frequency of more costly misses
- but larger caches are too slow for processor
- => gradually reduce cost of misses with a multiple cache levels

Exploit today’s technological boundary
- can’t put large caches on chip
- board designer can vary cost/performance
- E.g., IBM Power4’s L3 cache is DRAM-based!

---

**Multi-Level Inclusion**

Multi-level inclusion holds if L2 is always a superset of L1
- handle synonyms (later)
- filter coherence traffic (18-742)
- makes L1 writes simpler
  - for both write-through and write-back

Block sizes can also be different
Multi-Level Inclusion (Cont.)

e.g., local LRU not sufficient
- assume L1 holds two and L2 holds three blocks
- and both use local LRU
- processor references: 1,2,1,3,1,4
- final contents of L1: 1,4
- L1 misses: 1,2,3,4
- final contents of L2: 2,3,4, but not 1

Inclusion takes effort to maintain
- make L2 cache have bits or pointers giving L1 contents
- invalidate from L1 before replacing from L2
- number of pointers per L2 block
  - L2 blocksize/L1 blocksize

Interesting interaction between L1I, L1D, and L2
- Example a matrix multiply program for a large matrix
- What happens if you maintain inclusion for L1I?
Level Two Cache Design

L1 cache design similar to single-level cache
  □ when main memory “faster” w.r.t. CPU
Apply previous experience to L2 design?
L2 “global” miss ratios not significantly altered by L1 presence
  □ if L2 cache size >= 8 x L1 cache size
  □ Przybylski et al., ISCA 1989
But L2 caches bigger than before

Level Two Cache Design

What is miss ratio?
  □ global: L2 misses after L1 references
  □ local: L2 misses after L1/L1 misses
  □ solo: as only cache / references
Level Two Cache Example

recall adding associativity to single-level cache helped if

\[ \text{diff}(t_{\text{cache}}) + \text{diff}(\text{miss}) \times t_{\text{miss}} < 0 \]

\[ \text{diff}(\text{miss}) = -1\%, \ t_{\text{miss}} = 20 \]

\[ \Rightarrow \text{diff}(t_{\text{cache}}) < 0.2 \text{ cycle} \]

Level Two Cache Example

consider doing the same in an L2 cache where

\[ t_{\text{avg}} = t_{\text{cache}1} + \text{miss1} \times t_{\text{cache}2} + \text{miss2} \times t_{\text{memory}} \]

improvement only if

\[ \text{miss1} \times \text{diff}(t_{\text{cache}2}) + \text{diff}(\text{miss2}) \times t_{\text{memory}} < 0 \]

\[ \text{diff}(t_{\text{cache}2}) < \frac{-\text{diff}(\text{miss2})}{\text{miss1}} \times t_{\text{memory}} \]

\[ \text{diff}(t_{\text{cache}2}) < 0.0005/0.05 \times 100 = 1 \text{ cycle} \]
**How Do We Reduce Conflict Misses?**

Why do conflict misses occur?
What are the primary design goals for caches?
Active block frame distribution in sets

How do we redistribute blocks into block frames?

---

**Use Hash Functions**

Current mapping function only uses index bits!
Conflict happens for blocks with identical index bits
Can we use other parts of the address (tag bits) to distinguish?

For $f_{\text{index}}(\text{addr0}) = f_{\text{index}}(\text{addr1})$ can chose $f_{\text{hash}}$ so that

$f_{\text{hash}}(\text{addr0}) \neq f_{\text{hash}}(\text{addr1})$

I.e., use more address bits than just index bits
What is the disadvantage of complex mapping functions?
Regular Set-Associative Cache

Bank 1

For all addresses with similar index

Bank 2

Tag  Index  Block xx

Seznec's Skewed-Associative Cache

Bank 1

For all addresses

Bank 2

A3  A2  A1  A0
**Seznec’s Skewed-Associative Cache**

For 4-way skewed-associative cache consider:
- bank0: A1 xor A2
- bank1: shuffle(A1) xor A2
- bank2: shuffle(shuffle(A1)) xor A2
- bank3: shuffle(shuffle(shuffle(A1))) xor A2

Implementation only adds xor’s to cache access path

---

**Blocking Algorithm**

Blocking intended to reduce conflict misses
- □ block loops so that arrays are operated on “blocks”
  - ❌ do not confuse the algorithm’s blocks with cache blocks
- ❌ but there are still self-conflicts in a block
- ❌ there can be conflicts among different arrays
- ❌ AND
- ❌ array sizes may be unknown at compile/programming time

Copying block to “work space”
- ❌ may be the only way to avoid conflicts
**Skewed-Associative Cache: Performance**

Simple experiment - too simple?
- What was the experiment?

Trace-driven simulation - what is wrong with this?
DM, SA, blocked, blocked+copy, Skewed

Results:
- high variability in DM, SA with and without blocked
- blocked+copy is more stable but perf can degrade!
- skewed is stable for all

---

**Norm Jouppi’s Victim Cache**

Targets conflict misses

Victim cache: a small fully-associative cache capturing “victims”
- victims are conflicts in a set in DM or low-assoc cache
- LRU replacement

A miss in cache + a hit in victim cache
- move line to main cache
- is effectively equal to fast miss handling (or slow hits)
Norm Jouppi's Victim Cache

Victim Cache's Performance

Removing conflict misses
- even one entry helps some benchmarks
- helps I-cache more than D-cache

Compared to cache size
- generally, victim cache helps more for smaller caches

Compared to line size
- helps more with larger line size (why?)
Software Restructuring

If column-major
- $x[i+1,j]$ follows $x[i,j]$  
- $x[i,j+1]$ long after $x[i,j]$

Poor code
- for $i = 1$, rows
- for $j = 1$, columns
  - sum = sum + $x[i,j]$

Better code
- for $j = 1$, columns
- for $i = 1$, rows
  - sum = sum + $x[i,j]$

Optimizations: must check for validity
- loop interchange
- merging arrays
- loop fusion
- blocking

Superscalar Caches

Increasing issue width => wider caches
Parallel cache access is harder than parallel FUs
- fundamental difference: caches have state, FUs don’t
- one port affects future for other ports

Several approaches used
- true multi-porting
- multiple cache copies
- virtual multi-porting
- multi-banking (interleaving)
- line buffers
Lock-up Free Caches

Normal caches stall while a miss is pending

Lock-up free caches
- [Kroft ISCA'81, Sohi ASPLOS'91]
- handle nbits while 1 miss is pending
- handle hits and misses while k misses pending

Potential benefits
- overlap misses with useful work (hits)
- overlap misses with each other

Lock-up Free Caches (Cont.)

Only makes sense if processor
- can handle multiple pending references
- often can do useful work under a miss
- has misses that can be overlapped

Key implementation problems
- handle reads to pending miss
- handle writes to pending miss
- keep multiple requests straight
Lock-up Free Caches (Cont.)

MSHRs: miss status holding registers
1. Is there already a miss to the same block?
2. Route data back to CPU

- Valid bit and tag: associatively compared on each miss
- Status & pointer to block frame
- What happens on a miss?
- Tag L1 requests to allow out-of-order service from L2
- Split-transaction/pipelined L1-L2 interface
- Associative MSHRs could become bottleneck

True Multi-Porting

Would be ideal
- Increase cache area
  - more chip area
  - slower access
  - difficult to pipeline access
**Multiple Cache Copies**

- Used in 21164
- Independent load paths
- Single shared store path
  - bottleneck
  - not scalable

---

**Virtual Multi-Porting**

- Used in Power2 and Alpha 21264
  - 21264 uses wave pipelining
- Time-share a single port
  - may require access to be faster than clock
  - probably not scalable beyond 2
Multi-Banking (Interleaving)

Used in Intel P6
Need routing network
Must deal with bank conflicts
Extra delays can be pipelined

Line Buffers (LO)

Pipe
Addr to a block

Allows multiple ops to/from a block
Increases bandwidth for wide issue
Juan et al.'s Simulation Study

What did they say?