Memory Caches

Contents

Supplement to material in section 5.2.

Includes notation presented in class.

Not yet complete.

Terminology

Address Space

The set of addresses defined in the ISA.

Effective addresses of loads and stores are in an address space.

Size usually specified by the number of bits in an address' binary representation.

Symbol used in class: a.

Typical values:

DLX and older ISAs, $a = 32$,

DEC Alpha, Sun SPARC V9, $a = 64$.

Typical Address Spaces

Real Address Space and *Real Addresses*

Addresses used by main memory.

Given a real address, can open up computer's case and point to data.

Virtual Address Space and *Virtual Addresses*

Addresses (usually) used by programs.

Data can be in many locations ...
... this chip, that chip, this disk, that disk, ...
... or nowhere at all (if never written).

To access data virtual addresses must be translated to real addresses.

Each process (running program) gets its own virtual address space.

Only real addresses used in this set.

Unit of Addressability

What a memory address refers to; defined by the ISA.

Sometimes called a character.

Symbol used in class: c

In most systems, $c = 8$ bits.

Bus Width

The number of bits brought into the processor in a single access.

The number of bits accessed by an instruction may be less, the other bits are ignored.

This is an implementation feature.

Symbol used in class w.

In any reasonable system w is a multiple of c.

Typical values, $w = 64$ bits.

Example

```
! Data in memory:
! 0x1000:  5,  0x1001: 6,  0x1002: 7,  0x1003: 8.
! 0x1004 to 0x1007: 0x11111111
addi r1, r0, 0x1000
lb r10, 0(r1)  ! Eff. addr = 0x1000,   r10 <-  0x00000005
lb r11, 1(r1)  ! Eff. addr = 0x1001,   r11 <-  0x00000006
lb r12, 2(r1)  ! Eff. addr = 0x1002,   r12 <-  0x00000007
lb r13, 3(r1)  ! Eff. addr = 0x1003,   r13 <-  0x00000008
lh r15, 0(r1)  ! Eff. addr = 0x1000,   r15 <-  0x00000506
lh r16, 2(r1)  ! Eff. addr = 0x1002,   r16 <-  0x00000708
lw r20, 0(r1)  ! Eff. addr = 0x1000,   r20 <-  0x05060708
```


Relationship Between Address and Data on Bus

If $w = c$ then data on bus is for a single address.

If $w/c = 2$ then data on bus is for two consecutive addresses.

If $w/c = D$ then data on bus is for D consecutive addresses.

When load instructions refer to less than w/c addresses ...
... the processor hardware extracts bits needed ...
... using an *alignment network*.

Alignment Network

Bus width fixed at w bits ...
... but instructions may expect fewer bits.

Alignment network extracts bits that instruction needs ...
... and pads unused high-order bits with zeros or sign.

Example: Suppose $w = 128\,\text{b} = 16\,\text{B}$:

```
! r1 = 0x1003
! MEM[0x1000..0x100F] = 0x80 0x81 0x82 ... 0x8f
 lb r2, 0(r1)   ! Bus contents: 0x80 0x81...0x8f, r2<-0xffffff83
 lu r3, 0(r1)   ! Bus contents: 0x80 0x81...0x8f, r3<-0x83
 lw r4, 1(r1)   ! Bus contents: 0x80 0x81...0x8f, r4<-0x84858687
```

For `lb` alignment network extracts bits $24:31$ from bus ...
... padding high-order bits with 1 (MSB of 0x83).

For `lu` alignment network extracts bits $24:31$ from bus ...
... padding high-order bits with 0 (unsigned load).

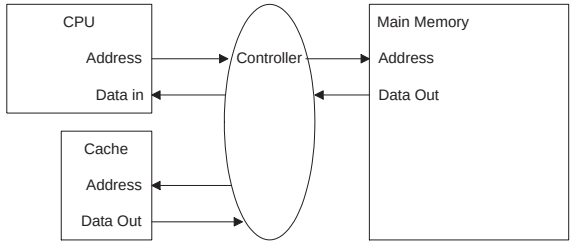
For `lw` alignment network extracts bits $32:63$ from bus ...
... no padding since register and data are both 32 bits.

Cache Motivation

Loads and stores are performed frequently.

Fast memory devices are expensive and size limited and ...
... cheap memory devices are slow.

By organizing these devices into a *cache* performance will be ...
... almost as fast as expensive devices ...
... almost as cheap as slow devices.



EE 4720 Lecture Transparency. Formatted 17:05, 1 December 1999 from lsli13.
13-9

Cache Idea

Use cheap devices to implement (real part of) address space.

Use expensive devices to duplicate parts of address space.

Hit

If accessed data in expensive devices, data returned quickly.

Called a *hit*.

Miss

If accessed data not in expensive devices ...
... data copied from cheap to expensive devices ...
... and passed to processor (taking much more time than a hit).

Called a *miss*.

EE 4720 Lecture Transparency. Formatted 17:05, 1 December 1999 from lsli13.
13-10

Organization of Simple Cache

Uses two memories, called *data* and *tag*. (Both expensive.)

Data memory holds copies of data held by cheap devices.

Tag memory holds information about data held in data memory.

Block (also called line)

The unit of storage in a cache.

Consists of one *or more* bus-width's worth of data.

On a miss an entire block's worth of data copied from main memory.

The size of a block usually given in terms of characters.

Symbols used in class: L block size in characters, $l = \log_2 L$.

EE 4720 Lecture Transparency. Formatted 17:05, 1 December 1999 from lsli13.
13-11

Set

A part of a cache.

Symbol: s, denotes $\log_2$ of the number of sets.

EE 4720 Lecture Transparency. Formatted 17:05, 1 December 1999 from lsli13.
13-12

Memory (Effective) Address Relationship to Cache Structure

Address bit positions, *fields*, can be used to locate data in cache.

For

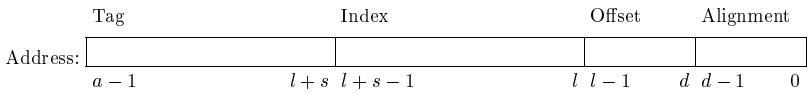
a-bit address space,

2^s-set cache,

2^l-character lines,

and $d = \log_2(w/c)$ character bus width

the fields would be as follows:



Details given on following slides.

Alignment

Bits: $0 : (d-1)$

where $d = \log_2(w/c)$ (characters per bus width).

Assuming aligned fetches, the d least significant bits of an address will either be zero or ignored.

This part of the address is ignored by the cache.

These bits are labeled "offset" in figure 5.8. Three of those five offset bits are actually alignment bits.

Offset

Bits: $d : (l-1)$,

The part of the block to be accessed.

(A block can hold more than one bus-width worth of data.)

This and the index are used to look up the data in each of the data memories.

These bits are labeled "offset" in figure 5.8. Two of those five bits are offset bits as defined here and are shown connected (along with index) to the address inputs of the data memories.

Index

Bits: $l : (l+s-1)$

The set to be accessed.

Used to look up a tag in each of the tag memories.

Along with the offset, used to look up the data in each of the data memories.

These are labeled "index" in figure 5.8 and are shown connecting to the data and tag memories.

Tag

Bits: $(l+s) : (a-1)$

The part of the address stored in the tag memories.

The number of tags stored per set is equal to the associativity.

The tags that are retrieved (see index, above) are compared to the tag of the address being

accessed.

There is a hit if a tag matches and the corresponding valid bit is 1.

Labeled "tag" in figure 5.8. The figure omits the data-in port to the tag memories, which is used for writing a new tag and valid bit on cache replacement.

Tag Store Information

Tag Store

Memory with one entry for each block.

Each entry holds ...
... tag (high-order part of block address) ...
... valid bit (indicating that block is in use) ...
... dirty bit (in write-back caches, indicating higher levels not updated).

Write Options

Write Through

Update memory (or higher levels) immediately.

Write Back

Update memory (or higher levels) on eviction.

Write Allocate

Cache data after a write miss.

Write Around or *No-write allocate*

Don't cache data on a write miss.

Cache Miss Rate v. Block Size

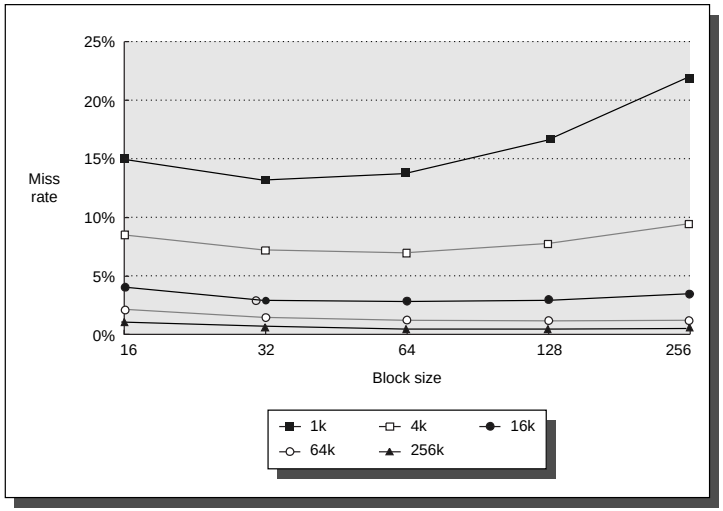


FIGURE 5.11 Miss rate versus block size for five different-sized caches.

Set-Associative Cache

Motivation

A direct-mapped cache can store no more than one block with a particular index.

Poor performance when program frequently accesses two blocks with same index.

Example, 4KB cache, block size $L = 2^l = 2^8$, number of sets $2^s = 2^4 = 16$.

```
extern char *a;  // Big array, allocated elsewhere.
for(i=0; i<1024; i++) {
  int b = a[ i ];           // At most one hit.
  int c = a[ i + 0x1000 ];  // Always misses.
  d += b + c;
}
```

Note, hit rate much better if cache size doubled ...
... or using a *set-associative* cache of same or smaller size.

Set-Associative Cache

Idea

Duplicate a direct mapped cache x times ...
... each duplicate sometimes called a *way*.

In typical set-associative caches associativity from 2 to 8.

Simultaneously look up block in each duplicate.

Can hold x blocks with same index.

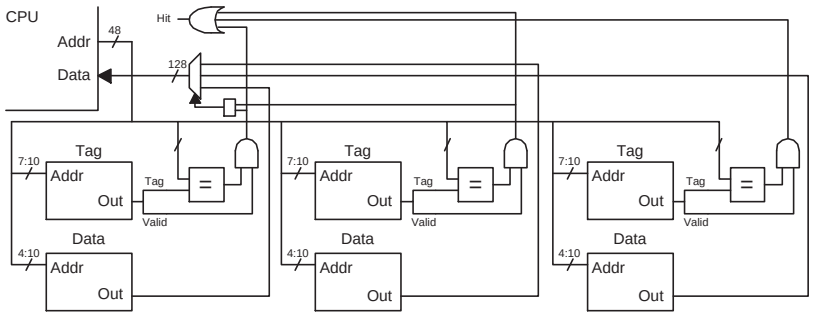
Set ...
... Storage for blocks with a particular index.

Associativity ...
... Number of blocks in a set.

x-Way Set-Associative Cache...
... A cache with associativity x.

Replacement Strategy...
... method used to choose block to evict (when space needed).

Set-Associative Cache Example



Memory System: $a = 48$, $c = 8$ bits, and $w = 128$ bits.

Cache: Three-way associativity (set holds 3 blocks), 128-byte blocks (each block holds 8 bus-widths of data), 16 sets.

Capacity: $3 \times 16\,\mathrm{B} \times 2^{10-4+1} = 6144\,\mathrm{B}$.

Set-Associative Cache Replacement Strategies

Problem not faced in direct mapped caches: which block to evict.

LRU.

Replace block in set that was least recently used.

Is effective because address references usually exhibit temporal locality.

Easy to implement if associativity small.

Too time consuming if associativity is large.

If associativity is large (> 16) LRU can be approximated.

Random

Usually approximated ...
... for example, using least significant bits of a cycle counter.

Less effective than LRU ...
... but effective enough and easy to implement.

Miss Types

Misses Categories

Compulsory or *Cold* Miss ...
... a miss to a block that was never cached.

Conflict Miss ...
... a miss on a system that would not occur ...
... on one using a fully-associative cache of the same size.

Capacity Miss ...
... a miss on a system that would also occur ...
... on one using a fully-associative cache of the same size.

Set-Associative Cache Special Cases

Direct Mapped Cache ...
... a 1-way set-associative cache (not really set associative).

Fully Associative Cache ...
... a set-associative cache with one set ($s = 0$).

Cache Miss Rate v. Size and Associativity

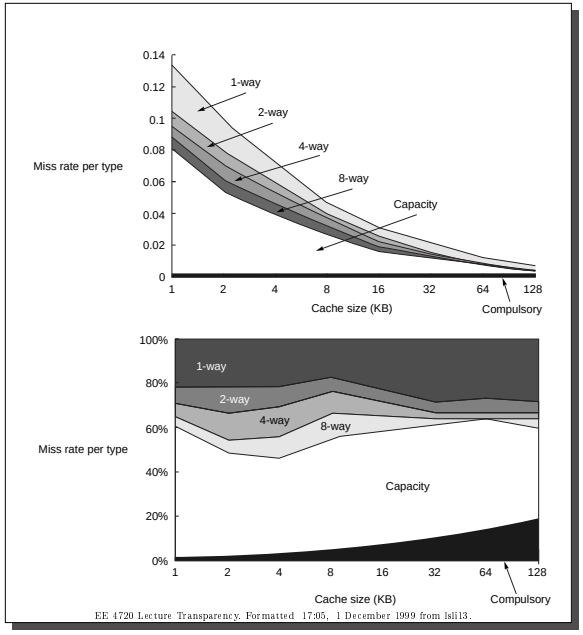


FIGURE 5.10 Total miss rate (top) and distribution of miss rate (bottom) for each size cache according to three C's for the data in Figure 5.9.

Set-Associative Cache Sample Program

What Program Does

First, program exactly fills up cache using minimum number of accesses.

Then, program generates hundreds of misses accessing only five items.

```
#include <stdio.h>

#define LG_LINE_SIZE 4
#define LG_SET_COUNT 10
#define ASSOCIATIVITY 4

#define SET_COUNT (1<<LG_SET_COUNT)
#define CACHE_SIZE ( ASSOCIATIVITY << ( LG_LINE_SIZE + LG_SET_COUNT ) )

char a[ 2 * CACHE_SIZE ];

int main(int argv, char **argc)
{
  int dummy = 0;
```

Fill up cache with minimum number of accesses.

```
{
  int s,w;
  char *addr = &a[0];
  int linesize = 1 << LG_LINE_SIZE;

  for(w=0; w<ASSOCIATIVITY; w++)

    for(s=0; s<SET_COUNT; s++)
      {
        dummy += *addr;
        /* Below, increment index part of address. */
        addr += linesize;
      }
}
```


Generate lots of misses while accessing only five (ASSOCIATIVITY + 1) array elements.

```
{
  int i, w;
  int deadly_stride = 1 << ( LG_LINE_SIZE + LG_SET_COUNT );

  for(i=0; i<100; i++)
    {
      char *addr = &a[0];

      for(w=0; w<=ASSOCIATIVITY; w++)
        {
          dummy += *addr;
          /* Below: Increment tag part of address. */
          addr += deadly_stride;
        }
    }
}
```


Organization of Caches

Single Cache Problems

Need multiple ports, for IF and one or more load/stores ...
... adding ports increases latency.

Larger caches have higher latencies ...
... must balance miss ratio with hit latency.

Solutions

Split cache into multiple *levels*.

Split cache into independent parts (all at same level).

Split Caches

Idea:

Split caches into independent parts.

Each part used by a different part of CPU.

Common Split: Instruction/Data

Instruction Cache ...
... a cache only used for instruction fetches.

Data Cache ...
... a cache only used for data reads and writes.

Unified Cache ...
... a cache that can be used for instructions and data.

Multiple-Level Caches

Due to fabrication technology limitations ...
... cache latency increases as size increases.

Latency and hit ratio balanced using multiple cache levels.

Cache level indicates "distance" from CPU.

Lowest-level (level one) cache checked first ...
... if data not present second level checked ...
... until data found or no more levels.

Lowest Level Criteria

Designed to meet a target hit latency. *E.g.*, one cycle.

For speed, may be direct mapped or low associativity.

Almost always on same chip as CPU.

13-33

Second Level Criteria

Hit latency can be much longer, *e.g.*, 10 cycles.

Latency can be met with a much larger size and associativity.

Sometimes on same chip as CPU ...
... sometimes tags on CPU chip while data off-chip ...
... sometimes entire second-level cache off-chip.

Third Level Criteria

If present, large enough so that a direct mapped organization would have few conflict misses.

13-34

Typical Cache Organization

Cache split into three levels.

Separate level-one instruction and data caches ...
... both on same chip as CPU.

Unified second-level cache.

Second-level cache tags on CPU but data off chip ...
... allowing a level-two hit to be detected early.

Unified third-level cache.

13-35

Miscellaneous Cache Feature

Lock-Free or *Non-Blocking* Cache
A cache that can continue to handle accesses ...
... while processing a miss.

13-36