## 15-213

## Caches March 16, 2000

#### Topics

- Memory Hierarchy
- Locality of Reference
- Cache Design
  - Direct Mapped
  - Associative

## **Computer System**



# Levels in Memory Hierarchy



larger, slower, cheaper

## Alpha 21164 Chip Photo

Microprocessor Report 9/12/94 Caches: L1 data L1 instruction L2 unified

TLB

Branch history



# Alpha 21164 Chip Caches



# Locality of Reference

### **Principle of Locality:**

- Programs tend to reuse data and instructions near those they have used recently.
- <u>Temporal locality:</u> recently referenced items are likely to be referenced in the near future.
- <u>Spatial locality:</u> items with nearby addresses tend to be referenced close together in time.

### Locality in Example:

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
*v = sum;
```

- Data
  - Reference array elements in succession (spatial)
- Instructions
  - Reference instructions in sequence (spatial)

- Cycle through loop repeatedly (temporal) class18.ppt - 6 -

# Caching: The Basic Idea

### Main Memory

Stores words

A-Z in example

### Cache

- Stores subset of the words
  - 4 in example
- Organized in lines
  - Multiple words
  - To exploit spatial locality

#### Access

• Word must be in cache for processor to access



# Basic I dea (Cont.)







Read Z



Cache holds 2 lines Each with 2 words

Load line C+D into cache "Cache miss"

Word already in cache "Cache hit"

Load line Y+Z into cache Evict oldest entry

## Maintaining Cache:

• Each time the processor performs a load or store, bring line containing the word into the cache

- May need to evict existing line

Subsequent loads or stores to any word in line performed within cache

# Accessing Data in Memory Hierarchy

- Between any two levels, memory is divided into lines (aka "blocks")
- Data moves between levels on demand, in line-sized chunks.
- Invisible to application programmer
  - Hardware responsible for cache operation
- Upper-level lines a subset of lower-level lines.



# **Design Issues for Caches**

### Key Questions:

- Where should a line be placed in the cache? (line placement)
- How is a line found in the cache? (line identification)
- Which line should be replaced on a miss? (line replacement)
- What happens on a write? (write strategy)

## Constraints:

- Design must be very simple
  - Hardware realization
  - All decision making within nanosecond time scale
- Want to optimize performance for "typical" programs
  - Do extensive benchmarking and simulations
  - Many subtle engineering tradeoffs

# **Direct-Mapped Caches**

#### Simplest Design

• Each memory line has a unique cache location

#### Parameters

- Line (aka block) size B = 2<sup>b</sup>
  - Number of bytes in each line
  - Typically 2X-8X word size
- Number of Sets S = 2<sup>s</sup>
  - Number of lines cache can hold
- Total Cache Size = B\*S = 2<sup>b+s</sup>

#### **Physical Address**

- Address used to reference main memory
- *n* bits to reference  $N = 2^n$  total bytes
- Partition into fields
  - Offset: Lower b bits indicate which byte within line
  - Set: Next s bits indicate how to locate line within cache
  - Tag: I dentifies this line when in cache



n-bit Physical Address

## Indexing into Direct-Mapped Cache



# **Direct-Mapped Cache Tag Matching**

### Identifying Line



## **Direct Mapped Cache Simulation**



## Why Use Middle Bits as Index?

#### 4-line Cache



### High-Order Bit Indexing

- Adjacent memory lines would map to same cache entry
- Poor use of spatial locality

#### Middle-Order Bit Indexing

- Consecutive memory lines map to different cache lines
- Can hold N-byte region of address space in cache at one time





## Direct Mapped Cache Implementation (DECStation 3100)

31 30 29 ..... 19 18 17 16 15 14 13 ..... 5 4 3 2 1 0



# **Properties of Direct Mapped Caches**

#### Strength

- Minimal control hardware overhead
- Simple design
- (Relatively) easy to make fast

#### Weakness

- Vulnerable to thrashing
- Two heavily used lines have same cache index
- Repeatedly evict one to make room for other



# **Vector Product Example**

```
float dot_prod(float x[1024], y[1024])
{
  float sum = 0.0;
  int i;
  for (i = 0; i < 1024; i++)
    sum += x[i]*y[i];
  return sum;
}</pre>
```

## Machine

- DECStation 5000
- MIPS Processor with 64KB direct-mapped cache, 16 B line size

## Performance

- Good case: 24 cycles / element
- Bad case: 66 cycles / element

# **Thrashing Example**



• Access one element from each array per iteration

# Thrashing Example: Good Case





- Read x[0]
  - x[0], x[1], x[2], x[3] loaded

y[0]

y[1]

y[2]

y[3]

- Read y[0]
  - y[0], y[1], y[2], y[3] loaded
- Read x[1]
  - Hit
- Read y[1]
  - Hit
- • •
- 2 misses / 8 reads

#### Analysis

- x[i] and y[i] map to different cache lines
- Miss rate = 25%

Cache

Line

- Two memory accesses / iteration
- On every 4th iteration have two misses

#### Timing

- 10 cycle loop time
- 28 cycles / cache miss
- Average time / iteration =
  - 10 + 0.25 \* 2 \* 28

# Thrashing Example: Bad Case





Cache Line

#### Access Pattern

- Read x[0]
  - x[0], x[1], x[2], x[3] loaded
- Read y[0]
  - y[0], y[1], y[2], y[3] loaded
- Read x[1]
  - x[0], x[1], x[2], x[3] loaded
- Read y[1]
   y[0], y[1], y[2], y[3] loaded
- • •
- 8 misses / 8 reads

#### Analysis

- x[i] and y[i] map to same cache lines
- Miss rate = 100%
  - Two memory accesses / iteration
  - On *every* iteration have two misses

#### Timing

- 10 cycle loop time
- 28 cycles / cache miss
- Average time / iteration =
  - 10 + 1.0 \* 2 \* 28

# Set Associative Cache

### Mapping of Memory Lines

- Each set can hold E lines
  - Typically between 2 and 8
- Given memory line can map to any entry within its given set

## **Eviction Policy**

- Which line gets kicked out when bring new line in
- Commonly either "Least Recently Used" (LRU) or pseudo-random
  - LRU: least-recently accessed (read or written) line gets evicted



# Indexing into 2-Way Associative Cache



# 2-Way Associative Cache Tag Matching

## Identifying Line



## 2-Way Set Associative Simulation

t=2 s=1 b=1 xx x x x			N=16 addresses, B=2 bytes/lin Address trace (reads): 0 [0000] 1 [0001] 13				ne, S=2 sets, E=2 entries/set [1101] 8 [1000] 0 [0000]		
0000		V	tag	data	V	tag	data		
0000	[							O (miss)	
0010	[								
0011 0100		V	tag	data	V	tag	data	_	
0101								13 (miss)	
0110								10 (11133)	
0111 1000		V	tag	data	V	tag	data		
1001								$\beta$ (miss)	
1010									
1011 1100 1101		V	tag	data	V	tag	data	(LRU replacement)	
1110	[							0 (miss)	
1111	[							(IRU replacement)	
class18.ppt					– 25 –			CS 213 S'00	

## Two-Way Set Associative Cache Implementation

- Set index selects a set from the cache
- The two tags in the set are compared in parallel
- Data is selected based on the tag result



# Fully Associative Cache

### Mapping of Memory Lines

- Cache consists of single set holding E lines
- Given memory line can map to any line in set
- Only practical for small caches



#### Entire Cache

# Fully Associative Cache Tag Matching



## **Fully Associative Cache Simulation**



# Write Policy

- What happens when processor writes to the cache?
- Should memory be updated as well?

### Write Through:

- Store by processor updates cache and memory.
- Memory always consistent with cache
- Never need to store from cache to memory
- ~2X more loads than stores



# Write Strategies (Cont.)

## Write Back:

- Store by processor only updates cache line
- Modified line written to memory only when it is evicted
  - Requires "dirty bit" for each line
    - » Set when line in cache is modified
    - » Indicates that line in memory is stale
- Memory not always consistent with cache



class18.ppt

## **Multi-Level Caches**

Options: *separate* data and instruction caches, or a *unified* cache



size:	200 B	8-64 KB	1-4MB SRAM	128 MB DRAM	9 GB				
speed:	2 ns	2 ns	6 ns	60 ns	8 ms				
\$/Mbyte:			\$100/MB	\$1.50/MB	\$0.05/MB				
line size:	8 B	32 B	32 B	8 KB					
larger, slower, cheaper									

larger line size, higher associativity, more likely to write back

## Alpha 21164 Hierarchy



- Improving memory performance was a main design goal
- Earlier Alpha's CPUs starved for data

# **Bandwidth Matching**

## Challenge

- CPU works with short cycle times
- DRAM (relatively) long cycle times
- How can we provide enough bandwidth between processor & memory?

## Effect of Caching

- Caching greatly reduces amount of traffic to main memory
- But, sometimes need to move large amounts of data from memory into cache

## Trends

- Need for high bandwidth much greater for multimedia applications
  - Repeated operations on image data
- Recent generation machines (e.g., Pentium III) greatly improve on predecessors

CPU

cache

bus

Μ

Short

Long

Latency

Latency

# High Bandwidth Memory Systems



# **Cache Performance Metrics**

### Miss Rate

- fraction of memory references not found in cache (misses/references)
- Typical numbers:
  - 3-10% for L1

can be quite small (e.g., < 1%) for L2, depending on size, etc.

### Hit Time

- time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
- Typical numbers:
  - 1 clock cycle for L1
  - 3-8 clock cycles for L2

## Miss Penalty

- additional time required because of a miss
  - Typically 25-100 cycles for main memory