61c NOTECARD OUTLINE/PERSONAL REVIEW

1 Caches

MISC OVERVIEW

Temporal locality: items referenced will tend to be referenced again soon.
Spatial locality: if an item is referenced, items near it will tend to be referenced more often
Large cache sizes reduce miss rate via temporal and spatial locality, but can increase hit time
Large block sizes reduce miss rate via spatial locality but increase miss penalty
Types: Direct mapped: each block in memory maps to exactly one block in the cache. Uses address mapping: (block address) modulo (# of blocks in cache).

CACHE-MEMORY CONSISTENCY

2 policies: Write through and write back.

- write through: Write to cache and write through to the memory. This direct approach is too slow, so include a Write Buffer that writes to memory in parallel to the actions of the processor
- write back: write only to cache and then write to memory **only** when the block is evicted from the cache. Include the dirty bit to see if data was written to the block and only write back if the dirty bit is set.

Sources of cache misses: the 3 C's

- Compulsory (cold start, first reference): the first access to a block will result in a miss - unavoidable. Soln: increase block size to store more blocks, but may increase miss penalty
- Capacity: cache can't hold all the blocks ever! Soln: increase cache size, but this may increase access time.
- Conflict/collision: multiple memory locations mapped to the same cache location. Soln: increase cache size, but this may increase hit time.

Local miss rate: fraction of references to one level of cache that miss: For L2, this would be L2 Misses / L1 Misses.
Global miss rate: fraction of references that miss in all levels of a cache. For a 2 level cache this would be L2 MR * L1 MR = L2 misses / L1 misses * L1 misses / total accesses
Improving cache performance

- Reduce time to hit in cache

  smaller cache. Perhaps use 1 word blocks
- Reduce miss rate

  use a bigger cache and/or bigger blocks
- reduce the miss penalty

  use smaller blocks. Use multiple cache levels. Use a write buffer to hold the dirty (written to) blocks so you don't have to stall until they're written to memory.

EQUATIONS

tag + index + offset = address bits (usually 32)
valid + dirty + tag + block bits = bits per row
[block bits are the block size as bits]
Block size = 2^m // m = offset bits
# of rows = 2^n // n = index bits
Tag field = 32-(n+m+2) // n = index bits, m = offset bits
Cache Size = 2^{n+m} = # of rows · blocksize
Total bits = 2^n· (block size + tag size + dirty size + valid size) [storing the index is not necessary because we need to know it in order to access the block!]
REMEMBER: To select a cache entry, you need a 32-bit address as follows:
[32-(n+m+2) bit tag][n bit index][m+2 bit offset]
Block size is 2^m words, so m bits used for the word w/n the block, and 2 bits are used for the byte part of the address.
Inside the cache, we have:
[valid bit][dirty bit (only if direct mapped)][tag bits][block bits]

MATHS

You can have different combinations of tag, offset and index. Let's say we have a 16kB cache and a block size of 64 bytes. Block size is 64 bytes = 2^6 bytes. Therefore, we have 6 offset bits. How many index bits do we have? Divide 16kB by 64 B: $2^4 \cdot 2^{10}/2^6 = 2^8$. Just read off the exponent: we have 8 index bits. To find the tag bits: do 32 - (6+8+2) = 16. Look at the breakdown:

[1001100011010011][10101010][00000100]

First section is the tag. Second is the index, which changes only every time the offset bits go through a cycle (000000 → 111111). Third is the offset, which distinguish words w/n a cache block. The last 2 bits distinguish between bytes in a word

AMAT

For a 1-level cache:

$$\text{AMAT} = \text{Hit time} + (\text{Miss Rate} \cdot \text{Miss Penalty})$$

For multi-level caches, the miss penalty for level i is the AMAT for level $i + 1$. For the lowest level cache, the AMAT is the access time for the main memory.

SET ASSOCIATIVE CACHES

For a fixed-size cache, increase by a factor of two of associateivity doubles the number of blocks per set and halves the number of sets - decrease index by 1 bit and increase tag by 1 bit

- tag (tag compare) | index (selects the set) | block offset (selects word in block) | byte(offset)
- decrease associativity by decreasing tag (and vice versa). No index: fully associative (only one set). Directmapped: (smaller tags, only a single comparator)
- when we get a miss, replace the least recently used (LRU) block. For a 2 way set-associative cache, one bit per set is set to 1 when a block is used (reset the other block to 0 to indicate it as the LRU)

2 Performance

Latency: time to complete one task (response time or execution time)
Bandwidth/Throughput: tasks completed per unit time
CPU time per program = clock cycles per program * clock cycle time = instructions per program * average clock cycles per instruction * clock cycle time
Average clock cycles per instruction * clock cycle time is the **CPI**, or Clock Cycles per Instruction
Time (seconds per program) =

$$\frac{Instructions}{Program} \cdot \frac{Clockcycles}{Instruction} \cdot \frac{Seconds}{ClockCycle}$$

3 code code code (a ship all filled with code)

SIZES

Byte: 8 bits
Word: 4 bytes
Short: 2 bytes
Int: 4 bytes
Char: 1 byte
Unsigned: 4 bytes
Signed: 4 bytes
Array:

POINTERS

- When transferring between C and MIPS, remember that in order to dereference a pointer *p, you must use lw $t0 0($s0) [where $s0 is the *p and $t0 is just a temp variable]
- If you have to increment pointers, increment them by the appropriate size, i.e. if you have an int pointer *p, increment in MIPS with addiu $s0 $s0 4

4 number representations

Least significant bit: rightmost bit
Most significant bit: leftmost bit
Increasing numeric significance from left to right is **little-endian** (most significant bit is 0)
Decreasing numeric significance is **big-endian** (most significant bit is 31)

POWERS OF TWO

1	2	3	4	5	6	7	8
2	4	8	16	32	64	128	256
9	10	11	12	13	14	15	16
512	1024	2048	4096	8192	16384	32768	65536

HEXADECIMAL

0x0	0000	0x1	0001	0x2	0010	0x3	0011
0x4	0100	0x5	0101	0x6	0110	0x7	0111
0x8	1000	0x9	1001	0xA	1010	0xB	1011
0xC	1100	0xD	1101	0xE	1110	0xF	1111

Special cases:

Exponent	Significant	Meaning
0	0	0
0	Non-zero	Denormalized #
1→254	anything	float
255	0	infinity
255	non-zero	NaN

+ and - infinity both possible with the sign bit.
Remember, the exponent bias is 127. EXP - Bias = effective exponent. EXP is what you encode into binary for the float representation. For shortcuts in determining the binary representation, take 10000000 → 1. Then just count up from there (ignore the leading 1 and add 1 to the binary representation of your exponent)

5 pages

- divide memory address space into equal sized blocks (pages). Use a level of indirection to map program addresses into memory addresses. The table of mappings is called a page table
- Processor-generated address (virtual address): page number [20 bits] | offset [12 bits]
- page table contains the physical address of the base of each page (page tables make it possible to store the pages of a program non-contiguously). Each program has it's own page table
- program addresses called virtual addresses (space of all virtual addresses called virtual memory). Memory addresses called physical addresses (correspond to physical memory)
- page table protection via access rights:
  - read: read but not write page
  - read/write: read or write page
  - execute: can fetch instructions from page
- From virtual address: use page number to index into page table: valid [1 bit] | access rights [? bits] | physical page address. Use offset from virtual address with the physical page address to get the physical memory address
- page tables stored in main memory.

translation lookaside buffer (TLB)

- TLB: Valid bit | Read bit | Write bit | execute bit | tag | physical page number
- use virtual page number as the tag (?) to index the TLB
- PAGE FAULT: a page is selected to be replaced. Replaced page is written on the disk - takes forever!
- set dirty bit in TLB when any data in page is written. When TLB entry is replacd, corresponding Page dirty bit is set in page table entry

6 datapaths

1. instruction fetch
2. decode/register read
3. execute
4. memory
5. register write

CONTROL HAZARDS

1. insert special branch comparator in stage 2: as soon as instruction is decoded, make a decision and set the value of the PC. We don't need no-ops for this, but branches are idle in the last three stages
2. predict outcome of a branch. cancel all instructions in pipeline that depended on a wrong guess. (usually predict that all branches are not taken bc it's similar)
3. redefine branches: whether or not we take the branch, the instruction followed by the branch gets executed (branch-delay slot). Delayed Branch means that we always execute inst after branch (MIPS uses this one). Worst-case: put a no-op in the branch delay slot. Better case: place some instruction preceding the branch in the branch-delay slot (as long as this doesn't affect the logic of the program3)

7 dependability | reliability

DEPENDABILITY MEASURES

- Mean time to failure: MTTF
- Mean time to repair MTTR
- Mean time between failures MTBF = MTTF + MTTR
- availability = $\frac{\text{MTTF}}{(MTTF+MTTR)}$

HAMMING STUFF

- hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different. It measures the minimum number of substitutions required to change the one string into the other
- use extra parity bits to allow the position identification of a single error. Mark all positions that are powers of 2 as parity bits (1, 2, 4, 8, ...). All other bit positions are for the data.
  - bit 0001 checks odd positions
  - bit 0010 checks 2,3,6,7,10,11,14,15...
  - bit 0100 checks 4-7, 12-15,20-23...
  - bit 1000 checks 8-15, 24-31,40-47...
- can add extra parity bit covering the entire word for double error detection