Computer memory systems

Void pointer (void \*)

* Void represents the absence of a type
* This allows void ptrs to point to any data type
* The dta pointed to by void ptrs cannot be directly dereferenced. We have to use explicit typecasting before dereferencing it
* Useful for generic interface functions
* Ex: the quick sort function for sorting an array of any type
* Ex: malloc. When mallocing, we just need a chunk of memory from the heap of size n. malloc is not concerned with what should be stored at that memory as that is not mallocs job, so it gives a void ptr.

RAM

* Comes in 2 forms:
  + Static (sram): faster and more computationally efficient but more expensive to buy. Used for ache memories
  + Dynamic (dram) used for the main memory and graphics systems

The CPU memory gap

* The gap widens between dram, disk, and cpu speeds
* Check slide 8 for chart
* The ley to bridging this gap is a fundamental property of computer programs known as locality.

Locality

* Principle: programs tend to use data and instructions with addresses near or equal to those the have used recently
* Temporal locality: recently referenced items are likely to be referenced again in the near future
* Spatial locality: items with nearby addresses tend to be referenced close together in time
* Example: iterating through indexes in an array (spatial locality)
* Ex: referencing the same variable in each iteration of a loop (temporal)
* Being able to look at code and get a good sense of its locality is a key skill for writing good programs
* **Ex on slide 11**: yes it has good locality because it is traversing the array in the same pattern with respect to the rows and columns as the contents show up in linear memory. If the loop order was flipped with N on the outer loop and M on the nested loop, then we would be jumping around in the memory vs going through it in order.

Memory hierarchies

* **Check slide 14 - 15**.

Caches

* A small, faster storage device that acts as a staging area for a subset of the data in a larger, slower device
* Check slide 16.
* Cache memories are small, fast sram memories managed automatically in hardware
* The CPU first looks for data in the cache
* When we need data, we check the cache first. If we find it we get a cache hit. If we don’t see it in the cache we get a cache miss, then we have to check main memory. If the cache is empty and we get a miss, this is called a cold miss.
* Placement policy: determines what position to put a recently accessed memory in the cache
* Replacement policy: determines what memory block to get rid of from the cache when putting in something new.
* Ch**eck slide 22** for exact vocab definitions.
* Replacement policies:
  + Random replacement: pick randomly
  + Least recently used (LRU) policy: remove the block that hasn’t been used in the longest
* Cache organization (S.E.B): s = the sets (rows), e = entries (each block within a row), b = bytes in each block (the data).
* Cache read: the address of a word has t bits for tag, s bits for set index, b bits for block offset.

Check **slide 28** and lecture recording for direct-mapped cache simulation. The MSB is the tag, the 2 middle bits is the set, the lsb is the block offset (since there are 2 spaces for data in each block. The last one (0) is a miss because even though it was inserted originally into set 0, it was replaced with 8.

**Read CSAPP 6.4.2 conflict misses indirect mapped caches.**

1/30/2024

E-way Set Associative Cache (here E= 2 meaning 2 lines per set)

* Check set, now you have more than one line per set, so do a linear search. In this case we are looking for a short int so go to the offset and the data is at the next 2 slots.

2-way set associative cache

* Each set has 2 blocks.

Writes

* Multiple copies of data exist: L1 (level 1 cache), L2, L3, …, Main Memory , Disk. Data has to be consistent along this memory
* What to do on a write- hit? (which is like in c you set an int x to 1 and the program sees x already was in the cache, so if we modify x we need to modify it in main memory as well)
  + Write-through (write immediately to memory)
  + Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
* What to do on a write miss?
  + Write-allocate (load the data on the cache, update that content on the cache)
    - Good if more writes to the location follow
  + No-write-allocate (writes straight to memory, does not load into cache)
    - Bad if you are writing something and want to read it from the cache later
* Typical combos:
  + Write-through + no-write-allocate
  + Write-back + write-allocate

Cache performance metrics:

* Miss rate
  + Fraction of memory references not found in cache (misses/accesses)
  + Typical amount is 3-10% for level 1, less than 1% for level 2 (because this cache is typically bigger)
* Hit time
  + Time to deliver a line in the cache to the processor, includes time to determine whether the line is in the cache
  + Typically 4 clock cycles for L1 and 10 for L2
* Miss penalty
  + Additional time required because of a miss
  + Typically 50-200 cycles for main memory
* Check **slide 16** for math on how to calculate percentage better

Writing cache-friendly code

* Theres more to performance than complexity.
* In the example, were copying one 2D array to another, first doing it row-wise then doing it column-wise. The first one performs better because of locality like we talked about before.
* To write cache-friendly code:
  + Make the common case go fast (focus on the inner lops of the core functions)
  + Minimize the misses in the inner loops
  + Check slide 18

C arrays review

* C arrays are allocated in row-major order. Esch row is in contiguous memory locations
* Stepping through columns in one row:
  + Accesses successive elements
  + In his example, we can fit 4 integers on the cache, so each time we get a miss, we insert the next 4 elements from the array onto the cache. The next 3 will be cache hits. This makes the miss rate ¼.
* Stepping through rows in one column:
  + Accesses distant elements
  + No spatial locality so miss rate = 100%