Discussion #7: Performance and Caches

Written by Justin Hsia (7/13/2011)

Performance

The most common performance metrics are *throughput* (*bandwidth*) and *response time* (*latency*), the difference being that throughput is the rate of total work done and response time is time to completion of a task. Faster execution is considered better performance, so we use:

$$\text{Performance} = \frac{1}{\text{Execution Time}}$$

Most performance questions can be solved using the following equation:

$$\underbrace{\frac{\text{Seconds}}{\text{Program}}}_{\text{CPU Time}} = \underbrace{\frac{\text{Instructions}}{\text{Program}}}_{\text{IC}} \times \underbrace{\frac{\text{Clock Cycles}}{\text{Instruction}}}_{\text{CPI}} \times \underbrace{\frac{\text{Seconds}}{\text{Clock Cycle}}}_{\text{CC}}$$

We can clearly see that this works by dimensional analysis. In general, a tricky part of performance questions is making sure your units are correct.

- Clock cycle time (CC) is often given as clock rate (Hz = cycles/sec = 1/CC)
- Average CPI can be calculated from an instruction breakdown of a program. For example, a program of 5 loads taking 4 cycles each and 3 adds taking 1 cycle each has an average CPI of (5*4 + 3*1)/(5 + 3) = 23/8 = 2.875.

Direct-Mapped Caches

What is Caching?

A cache is a small, but fast layer of memory that is used to improve system performance. Fast memory is expensive and cheap memory is slow. To compromise, we use caches in our memory heirarchy to give the illusion of having access speed close to that of the fastest technology and having as much space as the cheapest technology. Caching takes advantage of the following principles:

Temporal Locality – a recently-accessed item will often be accessed again soon.
Spatial Locality – we tend to access nearby items to ones that have been accessed recently.

A cache stores chunks of sequential data known as *blocks* (or lines). Because of its limited space, we can only hold so many blocks at a time. A *direct-mapped cache* is essentially a hash table where each address in memory corresponds to a single location within the caches (called a *row*). In particular, the hash function used is modulus by a power of 2. Convince yourself that `x%(2^n)` returns the lowest `n` bits of `x`. By making our caches use a power of 2 number of rows, we arrive at the following convenient method of doing cache accesses:

TIO Breakdown:

Above is how we break down a request for data at a given address. As shown the address is `A` bits wide and the splitting is always done in the given order, where the `T` leftmost bits form the Tag field, the `I` middle bits form the **Index** field, and the `O` rightmost bits form the Offset field.

- **Offset field** ("column index") indicates the location of the requested address within its block.
- **Index field** ("row index") indicates the row within the cache that the requested address goes in.
- **Tag field** ("block identifier") indicates if the present block matches the one you want.

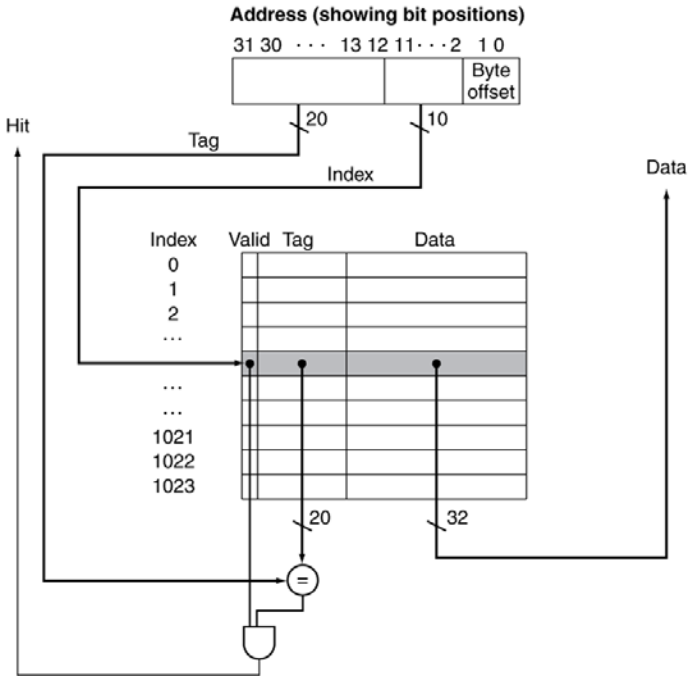
For calculating field width or cache parameters, recall the Power of Two portion from Discussion 1. We can represent up to 2^{n} things using n bits, which means we need $\log_2(2^{\mathrm{n}}) = \mathrm{n}$ bits to represent 2^{n} things.

$$\texttt{O} = \log_2(\text{Block size/B}), \qquad \texttt{I} = \log_2\left(\text{Cache rows} = \frac{\text{Cache Size}}{\text{Block Size}}\right), \qquad \texttt{T} = \texttt{A}-\texttt{I}-\texttt{O}$$

Cache Terminology:

Cache Size – total amount of just data held in the cache (Block Size × Cache rows)
Cache hit – requested address already in the cache (fast return)
Cache miss – cache index "empty" (not valid), so read from memory and write to cache (slow).
Cache miss, block replacement – cache index has wrong block (non-matching tag), so read from memory and override block (slow).
Cache hit rate – For a *given* set of requests, the percentage that result in cache hits.
Cache miss rate – Same as hit rate, but percentage that miss (1 – hit rate).

Using the Cache:



Here is a diagram showing how a direct-mapped cache determines a cache hit.

The cache itself is represented by the table in the middle and the requested address is shown at the top in what is known as its **TIO breakdown** (tag-index-offset). Here the field widths are set to 20, 10, and 2, respectively, but these will change based on the design of our cache.

A *cache hit* is determined by checking accessing the row of the cache specified by the **index** field of the address and then comparing the **tag** fields. If they match and the data is **Valid**, then return the requested part of the data block specified by the **offset** field.

Handling Writes:

So far we've only mentioned caching in terms of access to data. Caches are used for all memory accesses, including data writes. Here we define a **write hit** as a request to write to an address that already exists in the cache and a **write miss** as a request to write to an address that is not present in the cache.

On a write hit, the address is in the cache already and the next request for that address will come straight from the cache. So we can either update the data in both the cache and in memory (**write-through** policy) or we can just write to the cache (**write-back** policy). Write-through ensures that the data in memory matches that in the cache, which is known as *consistency*, but writing to memory every time is slow. Write-back only writes to memory when a changed block is being replaced in the cache. We keep track of which blocks need to be written back by adding a **"Dirty" bit** to each block.

On a write miss, our choices are to either fetch the requested address from memory into the cache and then write to both memory and cache (**write allocate**) or bypass the cache and write only to memory (**no-write allocate**). Write allocate obviously takes longer, but keeps the data consistent between memory and the cache.

Set Associative Caches

So What's the Difference?

In a direct-mapped cache, each block can only go in a single location in the cache. This can lead to a high volume of collisions and block replacements in poorly-optimized code. A set associative cache allows each block to fit into a specified *set* of locations, which should reduce the number of replacements. An ***n*-way set associative cache** uses sets of size n, meaning each block can fit into n different locations in the cache. Of special note, a **fully associative cache** is a cache using a single set that is the size of the entire cache, meaning that every block can be stored anywhere in the cache.

Geometrically, this is equivalent to placing n blocks (and their associated tag fields) in each row. In this sense, "rows" and "sets" are the same, but for the sake of consistency we will only use "row" for direct-mapped. An associative cache will still use a similar hashing function of modulus a power of 2, but the number of cache sets is different. To access a block, you must check EVERY tag field in the set! Here we gain some performance by reducing the miss rate, but lose some performance by increasing the data access time. When a replacement does need to happen, usually you replace the **least recently used** (LRU) block within the set.

TIO Revisited:

The Index field specifies which *set* of the cache to look in. So whereas for a direct-mapped cache we had $\#\ \text{rows}_{\text{d-m}} = \frac{\text{Cache Size}}{\text{Block Size}}$, now $\#\ \text{sets}_{\text{assoc}} = \frac{\text{Cache Size}}{\text{Block Size} \times n}$ and $\texttt{I} = \log_2(\#\ \text{sets})$. To convert from a direct-mapped cache to an n-way associative cache, this is equivalent to moving $\log_2 n$ bits from the Index field to the Tag field.

Multilevel Caches

Not only can you have multiple caches, but we can layer them such that misses from the first cache (which we will call the L1$) then pass the request on to the second cache (L2$). Misses from L2 then go back to main memory (MM). The reason for doing this will become more apparent as we discuss cache performance next.

Cache Performance

Cache performance is measured in average time (in clock cycles or seconds) it takes to perform instructions. Memory hierarchy accesses are slow and cause the processor to *stall* while it waits for the data to be retrieved. Here we define the additional terminology for performance:

Hit Time – The time it takes a specified memory layer to return a piece of data that it is currently storing.
Miss penalty – The time required to fetch a block into a specified memory layer.
Average Memory Access Time (AMAT) – Averaged time to access data for both hits and misses.

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$

For a single cache, this is quite straightforward. Miss Penalty will be access time of main memory (MM). For multilevel caches, this equation can be expanded using the relation:

$$\text{Miss Penalty}_i = \text{Hit Time}_{i+1} + \text{Miss Rate}_{i+1} \times \text{Miss Penalty}_{i+1}$$

You keep expanding until you hit the Miss Penalty for MM. Multiplying everything through will get you your AMAT. Here the Miss Rate is a **local miss rate**, the fraction of requests to a specified cache that miss. The **global miss rate**, the fraction of requests that have to go all the way to MM, is not a property of a specific level of caching, but a property of the overall memory hierarchy. The global miss rate can be calculated as the product of all local miss rates for each level of the memory hierarchy.

$$\begin{aligned}\text{CPI}_{\text{stall}} &= \text{CPI}_{\text{base}} + \text{Average Memory-stall cycles} \\ &= \text{CPI}_{\text{base}} + \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss Rate} \times \text{Miss Penalty}\end{aligned}$$

$\text{CPI}_{\text{stall}}$ (also called "Total CPI" in P&H) is just our normal CPI, but now taking into account processor stalls for memory accesses. CPI_{base} is the CPI if we never had to access memory. Here we typically specify that there are separate caches for instructions (I$), which must be fetched from the code section, and for data (D$), which fetches requested data from the stack, heap, or static data. I$ will be fetched from all the time, which D$ is only fetched from on load and store instructions. So $\frac{\text{Memory accesses}}{\text{Instruction}}$ will be 1 for instruction fetches and < 1 for data accesses (30% ld/st means $\frac{\text{Memory accesses}}{\text{Instruction}} = 0.3$). $\text{CPI}_{\text{stall}}$ calculations will typically include accesses to both I$ and D$.

Notice any similarities? It turns out that we can also use the following relationship:

$$\text{CPI}_{\text{stall}} = \text{CPI}_{\text{base}} + \frac{\text{Memory accesses}}{\text{Instruction}} \times (\text{AMAT} - \text{Hit Time})$$