Caching: Replacement Policies

- Learning Objectives
  - Simulate different cache replacement polices
  - Given a workload, a cache size, and a replacement policy, calculate hit/miss rates
  - Given hit/miss rates and the performance of different levels of the memory hierarchy, compute average access times.

What is a replacement policy?

- When the cache is full (e.g., every slot is occupied by data thought to be useful), determines which block you kick out (evict) to make room for a new request.
- In HW: these are very simple – quite often random (because you have very few choices about where a cache block can be placed).
- In SW: typically fully associate, so you have a lot of choices about which cache block to replace.

Goals of a good replacement policy

- Evicts a block you won't need for a long time.
- Best case: evicts exactly that block that will next be accessed farther in the future than any other block.
  - This is called Belady's algorithm.
  - Unfortunately, it requires that you predict the future.
- All other algorithms are attempts at approximating Belady's algorithm.
- Some assumptions we make:
  - Workloads exhibit locality.
  - Spatial locality: you are likely to access data near data you just accessed.
  - Temporal locality: you are likely to access data you accessed recently.

Moving data into a cache i units larger than the application requested.

›ve a large chunk of data and you frequently access data close to other data, this should be a big win.

Saving data on the stack.

:ality and in fact, moving data from one place and putting it on the stack might reduce spatial locality.

Using malloc.

Malloc makes absolutely no guarantees about spatial locality between consecutive requests.

Debugging.

Admittedly, very useful, but again, completely unrelated to locality.

Preview

Exploiting temporal locality

- LRU (least-recently-used):
  - Assumes that data accessed recently is likely to be accessed again.
  - Assumes that the least valuable data is that data that has remained unaccessed for the longest period of time.
  - Is probably the most commonly used replacement policy.

LRU Example

- Example:
  - Assume we have 2 slots in our cache.
  - We access cache blocks in the following order:
    - A, A, B, A, B, C, C, C, A, B,

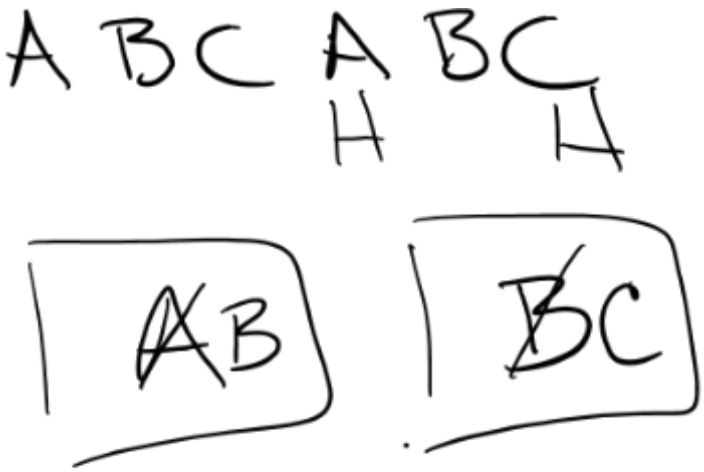
M H M H H M H H M M

$$\frac{\#\text{Hits}}{(\#\text{Hits} + \#\text{Misses})}$$

$$\frac{5}{10} = 50\%$$

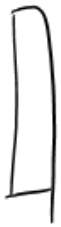
There is no perfect policy

- Although LRU is frequently a good policy, as we saw in the last question there are workloads that make it behave terribly.



Strided Access

- Assume we have a really big array with thousands of elements.
- Consider the access pattern:
  - 0, 16, 32, 48, 64, 1 17, 33, 49, 65, 2, 18, 34, 50, 66, …



Handling Stride

- If you're not careful you could end up having a 100% miss rate.
- If you are allowed to break up the access pattern, you could do much better.
- For example, could we do:
  - 1, 16, 32, 2, 17, 33, 3, 18, 34, 4, 19, 35, 5, 20, 36
  - And then do
  - 48, 64, 80, 49, 65, 81, 50, 66, 82, 51, 67, 83, 52, 68, 84

Wrapping Up

- In hardware, replacement policies are by necessity simple.
- In software, we can (sometimes) be a bit more clever.
- Although LRU is, perhaps, the most widely used replacement algorithm, it's not perfect in all cases.
- High performance, general-purpose caches frequently have a couple of different policies, adapting to the workload.