Okay, one more cache design decision to make, then we're done!

How should we handle memory writes in the cache?

Ultimately we'll need update main memory with the new data, but when should that happen?

The most obvious choice is to perform the write immediately.

In other words, whenever the CPU sends a write request to the cache, the cache then performs the same write to main memory.

This is called "write-through".

That way main memory always has the most up-to-date value for all locations.

But this can be slow if the CPU has to wait for a DRAM write access - writes could become a real bottleneck!

And what if the program is constantly writing a particular memory location, e.g., updating the value of a local variable in the current stack frame?

In the end we only need to write the last value to main memory.

Writing all the earlier values is waste of memory bandwidth.

Suppose we let the CPU continue execution while the cache waits for the write to main memory to complete - this is called "write-behind".

This will overlap execution of the program with the slow writes to main memory.

Of course, if there's another cache miss while the write is still pending, everything will have to wait at that point until both the write and subsequent refill read finish, since the CPU can't proceed until the cache miss is resolved.

The best strategy is called "write-back" where the contents of the cache are updated and the CPU continues execution immediately.

The updated cache value is only written to main memory when the cache line is chosen as the replacement line for a cache miss.

This strategy minimizes the number of accesses to main memory, preserving the memory bandwidth for other operations.

This is the strategy used by most modern processors.

Write-back is easy to implement.

Returning to our original cache recipe, we simply eliminate the start of the write to main memory when there's a write request to the cache.

We just update the cache contents and leave it at that.

However, replacing a cache line becomes a more complex operation, since we can't reuse the cache line without first writing its contents back to main memory in case they had been modified by an earlier write access.

Hmm.

Seems like this does a write-back of all replaced cache lines whether or not they've been written to.

We can avoid unnecessary write-backs by adding another state bit to each cache line: the "dirty" bit.

The dirty bit is set to 0 when a cache line is filled during a cache miss.

If a subsequent write operation changes the data in a cache line, the dirty bit is set to 1, indicating that value in the cache now differs from the value in main memory.

When a cache line is selected for replacement, we only need to write its data back to main memory if its dirty bit is 1.

So a write-back strategy with a dirty bit gives an elegant solution that minimizes the number of writes to main memory and only delays the CPU on a cache miss if a dirty cache line needs to be written back to memory.

That concludes our discussion of caches, which was motivated by our desire to minimize the average memory access time by building a hierarchical memory system that had both low latency and high capacity.

There were a number of strategies we employed to achieve our goal.

Increasing the number of cache lines decreases AMAT by decreasing the miss ratio.

Increasing the block size of the cache let us take advantage of the fast column accesses in a DRAM to efficiently load a whole block of data on a cache miss.

The expectation was that this would improve AMAT by increasing the number of hits in the future as accesses were made to nearby locations.

Increasing the number of ways in the cache reduced the possibility of cache line conflicts, lowering the miss ratio.

Choosing the least-recently used cache line for replacement minimized the impact of replacement on the hit ratio.

And, finally, we chose to handle writes using a write-back strategy with dirty bits.

How do we make the tradeoffs among all these architectural choices?

As usual, we'll simulate different cache organizations and chose the architectural mix that provides the best performance on our benchmark programs.