Okay, let's review our plan.

The processor starts an access by sending an address to the cache.

If data for the requested address is held in the cache, it's quickly returned to the CPU.

If the data we request is not in the cache, we have a cache miss, so the cache has to make a request to main memory to get the data, which it then returns to processor.

Typically the cache will remember the newly fetched data, possibly replacing some older data in the cache.

Suppose a cache access takes 4 ns and a main memory access takes 40 ns.

Then an access that hits in the cache has a latency of 4 ns, but an access that misses in the cache has a latency of 44 ns.

The processor has to deal with the variable memory access time, perhaps by simply waiting for the access to complete, or, in modern hyper-threaded processors, it might execute an instruction or two from another programming thread.

The hit and miss ratios tell us the fraction of accesses which are cache hits and the fraction of accesses which are cache misses.

Of course, the ratios will sum to 1.

Using these metrics we can compute the average memory access time (AMAT).

Since we always check in the cache first, every access includes the cache access time (called the hit time).

If we miss in the cache, we have to take the additional time needed to access main memory (called the miss penalty).

But the main memory access only happens on some fraction of the accesses: the miss ratio tells us how often that occurs.

So the AMAT can be computed using the formula shown here.

The lower the miss ratio (or, equivalently, the higher the hit ratio), the smaller the average access time.

Our design goal for the cache is to achieve a high hit ratio.

If we have multiple levels of cache, we can apply the formula recursively to calculate the AMAT at each level of the memory.

Each successive level of the cache is slower, i.e., has a longer hit time, which is offset by lower miss ratio because of its increased size.

Let's try out some numbers.

Suppose the cache takes 4 processor cycles to respond, and main memory takes 100 cycles.

Without the cache, each memory access would take 100 cycles.

With the cache, a cache hit takes 4 cycles, and a cache miss takes 104 cycles.

What hit ratio is needed to so that the AMAT with the cache is 100 cycles, the break-even point?

Using the AMAT formula from the previously slide, we see that we only need a hit ratio of 4% in order for memory system of the Cache + Main Memory to perform as well as Main Memory alone.

The idea, of course, is that we'll be able to do much better than that.

Suppose we wanted an AMAT of 5 cycles.

Clearly most of the accesses would have to be cache hits.

We can use the AMAT formula to compute the necessary hit ratio.

Working through the arithmetic we see that 99% of the accesses must be cache hits in order to achieve an average access time of 5 cycles.

Could we expect to do that well when running actual programs?

Happily, we can come close.

In a simulation of the Spec CPU2000 Benchmark, the hit ratio for a standard-size level 1 cache was measured to be 97.5% over some ~10 trillion accesses.

[See the "All benchmarks" arithmetic-mean table at http://research.cs.wisc.edu/multifacet/misc/spec2000cache-data/] Here's a start at building a cache.

The cache will hold many different blocks of data.

For now let's assume each block is an individual memory location.

Each data block is "tagged" with its address.

A combination of a data block and its associated address tag is called a cache line.

When an address is received from the CPU, we'll search the cache looking for a block with a matching address tag.

If we find a matching address tag, we have a cache hit.

On a read access, we'll return the data from the matching cache line.

On a write access, we'll update the data stored in the cache line and, at some point, update the corresponding location in main memory.

If no matching tag is found, we have a cache miss.

So we'll have to choose a cache line to use to hold the requested data, which means that some previously cached location will no longer be found in the cache.

For a read operation, we'll fetch the requested data from main memory, add it to the cache (updating the tag and data fields of the cache line) and, of course, return the data to the CPU.

On a write, we'll update the tag and data in the selected cache line and, at some point, update the corresponding location in main memory.

So the contents of the cache is determined by the memory requests made by the CPU.

If the CPU requests a recently-used address, chances are good the data will still be in the cache from the previous access to the same location.

As the working set slowly changes, the cache contents will be updated as needed.

If the entire working set can fit into the cache, most of the requests will be hits and the AMAT will be close to the cache access time.

So far, so good!

Of course, we'll need to figure how to quickly search the cache, i.e., we'll a need fast way to answer the question of whether a particular address tag can be found in some cache line.

That's our next topic.