A fully-associative (FA) cache has a tag comparator for each cache line.

So the tag field of *every* cache line in a FA cache is compared with the tag field of the incoming address.

Since all cache lines are searched, a particular memory location can be held in any cache line, which eliminates the problems of address conflicts causing conflict misses.

The cache shown here can hold 4 different 4-word blocks, regardless of their address.

The example from the end of the previous segment required a cache that could hold two 3-word blocks, one for the instructions in the loop, and one for the data words.

This FA cache would use two of its cache lines to perform that task and achieve a 100% hit ratio regardless of the addresses of the instruction and data blocks.

FA caches are very flexible and have high hit ratios for most applications.

Their only downside is cost.

The inclusion of a tag comparator for each cache line to implement the parallel search for a tag match adds substantially the amount of circuitry required when there are many cache lines.

Even the use of hybrid storage/comparison circuitry, called a content-addressable memory, doesn't make a big dent in the overall cost of a FA cache.

DM caches searched only a single cache line.

FA caches search all cache lines.

Is there a happy middle ground where some small number of cache lines are searched in parallel?

Yes!

If you look closely at the diagram of the FA cache shown here, you'll see it looks like four 1-line DM caches operating in parallel.

What would happen if we designed a cache with four multi-line DM caches operating in parallel?

The result would be what we call an 4-way set-associative (SA) cache.

An N-way SA cache is really just N DM caches (let's call them sub-caches) operating in parallel.

Each of the N sub-caches compares the tag field of the incoming address with the tag field of the cache line selected by the index bits of the incoming address.

The N cache lines searched on a particular request form a search "set" and the desired location might be held in any member of the set.

The 4-way SA cache shown here has 8 cache lines in each sub-cache, so each set contains 4 cache lines (one from each sub-cache) and there are a total of 8 sets (one for each line of the sub-caches).

An N-way SA cache can accommodate up to N blocks whose addresses map to the same cache index.

So access to up to N blocks with conflicting addresses can still be accommodated in this cache without misses.

This a big improvement over a DM cache where an address conflict will cause the current resident of a cache line to be evicted in favor of the new request.

And an N-way SA cache can have a very large number of cache lines but still only have to pay the cost of N tag comparators.

This is a big improvement over a FA cache where a large number of cache lines would require a large number of comparators.

So N-way SA caches are a good compromise between a conflict-prone DM cache and the flexible but very expensive FA cache.

Here's a slightly more detailed diagram, in this case of a 3-way 8-set cache.

Note that there's no constraint that the number of ways be a power of two since we aren't using any address bits to select a particular way.

This means the cache designer can fine tune the cache capacity to fit her space budget.

Just to review the terminology: the N cache lines that will be searched for a particular cache index are called a set.

And each of N sub-caches is called a way.

The hit logic in each "way" operates in parallel with the logic in other ways.

Is it possible for a particular address to be matched by more than one way?

That possibility isn't ruled out by the hardware, but the SA cache is managed so that doesn't happen.

Assuming we write the data fetched from DRAM during a cache miss into a single sub-cache - we'll talk about how to choose that way in a minute - there's no possibility that more than one sub-cache will ever match an incoming address.

How many ways to do we need?

We'd like enough ways to avoid the cache line conflicts we experienced with the DM cache.

Looking at the graph we saw earlier of memory accesses vs. time, we see that in any time interval there are only so many potential address conflicts that we need to worry about.

The mapping from addresses to cache lines is designed to avoid conflicts between neighboring locations.

So we only need to worry about conflicts between the different regions: code, stack and data.

In the examples shown here there are three such regions, maybe 4 if you need two data regions to support copying from one data region to another.

If the time interval is particularly large, we might need double that number to avoid conflicts between accesses early in the time interval and accesses late in the time interval.

The point is that a small number of ways should be sufficient to avoid most cache line conflicts in the cache.

As with block size, it's possible to have too much of a good thing: there's an optimum number of ways that minimizes the AMAT.

Beyond that point, the additional circuity needed to combine the hit signals from a large number of ways will start have a significant propagation delay of its own, adding directly to the cache hit time and the AMAT.

More to the point, the chart on the left shows that there's little additional impact on the miss ratio beyond 4 to 8 ways.

For most programs, an 8-way set-associative cache with a large number of sets will perform on a par with the much more-expensive FA cache of equivalent capacity.

There's one final issue to resolve with SA and FA caches.

When there's a cache miss, which cache line should be chosen to hold the data that will be fetched from main memory?

That's not an issue with DM caches, since each data block can only be held in one particular cache line, determined by its address.

But in N-way SA caches, there are N possible cache lines to choose from, one in each of the ways.

And in a FA cache, any of the cache lines can be chosen.

So, how to choose?

Our goal is to choose to replace the contents of the cache line which will minimize the impact on the hit ratio in the future.

The optimal choice is to replace the block that is accessed furthest in the future (or perhaps is never accessed again).

But that requires knowing the future… Here's an idea: let's predict future accesses by looking a recent accesses and applying the principle of locality.

d7.36 If a block has not been recently accessed, it's less likely to be accessed in the near future.

That suggests the least-recently-used replacement strategy, usually referred to as LRU: replace the block that was accessed furthest in the past.

LRU works well in practice, but requires us to keep a list ordered by last use for each set of cache lines, which would need to be updated on each cache access.

When we needed to choose which member of a set to replace, we'd choose the last cache line on this list.

For an 8-way SA cache there are 8! possible orderings, so we'd need log2(8!) = 16 state bits to encode the current ordering.

The logic to update these state bits on each access isn't cheap.

Basically you need a lookup table to map the current 16-bit value to the next 16-bit value.

So most caches implement an approximation to LRU where the update function is much simpler to compute.

There are other possible replacement policies: First-in, first-out, where the oldest cache line is replaced regardless of when it was last accessed.

And Random, where some sort of pseudo-random number generator is used to select the replacement.

All replacement strategies except for random can be defeated.

If you know a cache's replacement strategy you can design a program that will have an abysmal hit rate by accessing addresses you know the cache just replaced.

I'm not sure I care about how well a program designed to get bad performance runs on my system, but the point is that most replacement strategies will occasionally cause a particular program to execute much more slowly than expected.

When all is said and done, an LRU replacement strategy or a close approximation is a reasonable choice.