We can tweak the design of the DM cache a little to take advantage of locality and save some of the overhead of tag fields and valid bits.

We can increase the size of the data field in a cache from 1 word to 2 words, or 4 words, etc.

The number of data words in each cache line is called the "block size" and is always a power of two.

Using a larger block size makes sense.

If there's a high probability of accessing nearby words, why not fetch a larger block of words on a cache miss, trading the increased cost of the miss against the increased probability of future hits.

Compare the 16-word DM cache shown here with a block size of 4 with a different 16-word DM cache with a block size of 1.

In this cache for every 128 bits of data there are 27 bits of tags and valid bit, so ~17% of the SRAM bits are overhead in the sense that they're not being used to store data.

In the cache with block size 1, for every 32 bits of data there are 27 bits of tag and valid bit, so ~46% of the SRAM bits are overhead.

So a larger block size means we'll be using the SRAM more efficiently.

Since there are 16 bytes of data in each cache line, there are now 4 offset bits.

The cache uses the high-order two bits of the offset to select which of the 4 words to return to the CPU on a cache hit.

There are 4 cache lines, so we'll need two cache line index bits from the incoming address.

And, finally, the remaining 26 address bits are used as the tag field.

Note that there's only a single valid bit for each cache line, so either the entire 4-word block is present in the cache or it's not.

Would it be worth the extra complication to support caching partial blocks?

Probably not.

Locality tells us that we'll probably want those other words in the near future, so having them in the cache will likely improve the hit ratio.

What's the tradeoff between block size and performance?

We've argued that increasing the block size from 1 was a good idea.

Is there a limit to how large blocks should be?

Let's look at the costs and benefits of an increased block size.

With a larger block size we have to fetch more words on a cache miss and the miss penalty grows linearly with increasing block size.

Note that since the access time for the first word from DRAM is quite high, the increased miss penalty isn't as painful as it might be.

Increasing the block size past 1 reduces the miss ratio since we're bringing words into the cache that will then be cache hits on subsequent accesses.

Assuming we don't increase the overall cache capacity, increasing the block size means we'll make a corresponding reduction in the number of cache lines.

Reducing the number of lines impacts the number of separate address blocks that can be accommodated in the cache.

As we saw in the discussion on the size of the working set of a running program, there are a certain number of separate regions we need to accommodate to achieve a high hit ratio: program, stack, data, etc.

So we need to ensure there are a sufficient number of blocks to hold the different addresses in the working set.

The bottom line is that there is an optimum block size that minimizes the miss ratio and increasing the block size past that point will be counterproductive.

Combining the information in these two graphs, we can use the formula for AMAT to choose the block size the gives us the best possible AMAT.

In modern processors, a common block size is 64 bytes (16 words).

DM caches do have an Achilles heel.

Consider running the 3-instruction LOOPA code with the instructions located starting at word address 1024 and the data starting at word address 37 where the program is making alternating accesses to instruction and data,

e.g., a loop of LD instructions.

Assuming a 1024-line DM cache with a block size of 1, the steady state hit ratio will be 100% once all six locations have been loaded into the cache since each location is mapped to a different cache line.

Now consider the execution of the same program, but this time the data has been relocated to start at word address 2048.

Now the instructions and data are competing for use of the same cache lines.

For example, the first instruction (at address 1024) and the first data word (at address 2048) both map to cache line 0, so only one them can be in the cache at a time.

So fetching the first instruction fills cache line 0 with the contents of location 1024, but then the first data access misses and then refills cache line 0 with the contents of location 2048.

The data address is said to "conflict" with the instruction address.

The next time through the loop, the first instruction will no longer be in the cache and it's fetch will cause a cache miss, called a "conflict miss".

So in the steady state, the cache will never contain the word requested by the CPU.

This is very unfortunate!

We were hoping to design a memory system that offered the simple abstraction of a flat, uniform address space.

But in this example we see that simply changing a few addresses results in the cache hit ratio dropping from 100% to 0%.

The programmer will certainly notice her program running 10 times slower!

So while we like the simplicity of DM caches, we'll need to make some architectural changes to avoid the performance problems caused by conflict misses.