The problem with our simple multicore system is that there is no communication when the value of a shared variable is changed.

The fix is to provide the necessary communications over a shared bus that's watched by all the caches.

A cache can then "snoop" on what's happening in other caches and then update its local state to be consistent.

The required communications protocol is called a "cache coherence protocol".

In designing the protocol, we'd like to incur the communications overhead only when there's actual sharing in progress, i.e., when multiple caches have local copies of a shared variable.

To implement a cache coherence protocol, we'll change the state maintained for each cache line.

The initial state for all cache lines is INVALID indicating that the tag and data fields do not contain up-to-date information.

This corresponds to setting the valid bit to 0 in our original cache implementation.

When the cache line state is EXCLUSIVE, this cache has the only copy of those memory locations and indicates that the local data is the same as that in main memory.

This corresponds to setting the valid bit to 1 in our original cache implementation.

If the cache line state is MODIFIED, that means the cache line data is the sole valid copy of the data.

This corresponds to setting both the dirty and valid bits to 1 in our original cache implementation.

To deal with sharing issues, there's a fourth state called SHARED that indicates when other caches may also have a copy of the same unmodified memory data.

When filling a cache from main memory, other caches can snoop on the read request and participate if fulfilling the read request.

If no other cache has the requested data, the data is fetched from main memory and the requesting cache sets the state of that cache line to EXCLUSIVE.

If some other cache has the requested in line in the EXCLUSIVE or SHARED state, it supplies the data and asserts the SHARED signal on the snoopy bus to indicate that more than one cache now has a copy of the data.

All caches will mark the state of the cache line as SHARED.

If another cache has a MODIFIED copy of the cache line, it supplies the changed data, providing the correct values for the requesting cache as well as updating the values in main memory.

Again the SHARED signal is asserted and both the reading and responding cache will set the state for that cache line to SHARED.

So, at the end of the read request, if there are multiple copies of the cache line, they will all be in the SHARED state.

If there's only one copy of the cache line it will be in the EXCLUSIVE state.

Writing to a cache line is when the sharing magic happens.

If there's a cache miss, the first cache performs a cache line read as described above.

If the cache line is now in the SHARED state, a write will cause the cache to send an INVALIDATE message on the snoopy bus, telling all other caches to invalidate their copy of the cache line, guaranteeing the local cache now has EXCLUSIVE access to the cache line.

If the cache line is in the EXCLUSIVE state when the write happens, no communication is necessary.

Now the cache data can be changed and the cache line state set to MODIFIED, completing the write.

This protocol is called "MESI" after the first initials of the possible states.

Note that the the valid and dirty state bits in our original cache implementation have been repurposed to encode one of the four MESI states.

The key to success is that each cache now knows when a cache line may be shared by another cache, prompting the necessary communication when the value of a shared location is changed.

No attempt is made to update shared values, they're simply invalidated and the other caches will issue read requests if they need the value of the shared variable at some future time.

To support cache coherence, the cache hardware has to be modified to support two request streams: one from the CPU and one from the snoopy bus.

The CPU side includes a queue of store requests that were delayed by cache misses.

This allows the CPU to proceed without having to wait for the cache refill operation to complete.

Note that CPU read requests will need to check the store queue before they check the cache to ensure the most-recent value is supplied to the CPU.

Usually there's a STORE_BARRIER instruction that stalls the CPU until the store queue is empty, guaranteeing that all processors have seen the effect of the writes before execution resumes.

On the snoopy side, the cache has to snoop on the transactions from other caches, invalidating or supplying cache line data as appropriate, and then updating the local cache line state.

If the cache is busy with, say, a refill operation, INVALIDATE requests may be queued until they can be processed.

Usually there's a READ_BARRIER instruction that stalls the CPU until the invalidate queue is empty, guaranteeing that updates from other processors have been applied to the local cache state before execution resumes.

Note that the "read with intent to modify" transaction shown here is just protocol shorthand for a READ immediately followed by an INVALIDATE, indicating that the requester will be changing the contents of the cache line.

How do the CPU and snoopy cache requests affect the cache state?

Here in micro type is a flow chart showing what happens when.

If you're interested, try following the actions required to complete various transactions.

Intel, in its wisdom, adds a fifth "F" state, used to determine which cache will respond to read requests when the requested cache line is shared by multiple caches basically it selects which of the SHARED cache lines gets to be the responder.

But this is a bit abstract.

Let's try the MESI cache coherence protocol on our earlier example!

Here are our two threads and their local cache states indicating that values of locations X and Y are shared by both caches.

Let's see what happens when the operations happen in the order (1 through 4) shown here.

You can check what happens when the transactions are in a different order or happen concurrently.

First, Thread A changes X to 3.

Since this location is marked as SHARED [S] in the local cache, the cache for core 0 ($_0) issues an INVALIDATE transaction for location X to the other caches, giving it exclusive access to location X, which it changes to have the value 3.

At the end of this step, the cache for core 1 ($_1) no longer has a copy of the value for location X.

In step 2, Thread B changes Y to 4.

Since this location is marked as SHARED in the local cache, cache 1 issues an INVALIDATE transaction for location Y to the other caches, giving it exclusive access to location Y, which it changes to have the value 4.

In step 3, execution continues in Thread B, which needs the value of location X.

That's a cache miss, so it issues a read request on the snoopy bus, and cache 0 responds with its updated value, and both caches mark the location X as SHARED.

Main memory, which is also watching the snoopy bus, also updates its copy of the X value.

Finally, in step 4, Thread A needs the value for Y, which results in a similar transaction on the snoopy bus.

Note the outcome corresponds exactly to that produced by the same execution sequence on a timeshared core since the coherence protocol guarantees that no cache has an out-of-date copy of a shared memory location.

And both caches agree on the ending values for the shared variables X and Y.

If you try other execution orders, you'll see that sequential consistency and shared memory semantics are maintained in each case.

The cache coherency protocol has done it's job!

Let's summarize our discussion of parallel processing.

At the moment, it seems that the architecture of a single core has reached a stable point.

At least with the current ISAs, pipeline depths are unlikely to increase and out-of-order, superscalar instruction execution has reached the point of diminishing performance returns.

So it seems unlikely there will be dramatic performance improvements due to architectural changes inside the CPU core.

GPU architectures continue to evolve as they adapt to new uses in specific application areas, but they are unlikely to impact general-purpose computing.

At the system level, the trend is toward increasing the number of cores and figuring out how to best exploit parallelism with new algorithms.

Looking further ahead, notice that the brain is able to accomplish remarkable results using fairly slow mechanisms It takes ~.01 seconds to get a message to the brain and synapses fire somewhere between 0.3 to 1.8 times per second.

Is it massive parallelism that gives the brain its "computational" power?

Or is it that the brain uses a different computation model, e.g., neural nets, to decide upon new actions given new inputs?

At least for applications involving cognition there are new architectural and technology frontiers to explore.

You have some interesting challenges ahead if you get interested in the future of parallel processing!