Topics:
- Multi-Core Mesh/Torus
- Distributed LLC
- Partial Directory

$$\begin{tikzcd}\text{Memory Consistency} \arrow[r] & \text{Many Cores} \arrow[r] & \text{End!}\end{tikzcd}$$

Many Cores

A discussion of the challenges encountered in implementing a multi-core system.

Many Core Challenges

-As the number of cores increases, coherence traffic increases bc writes to a shared memLoc cause invalidations & misses, increasing coherence traffic on the shared bus.

-As the number of cores increases, the bus traffic increases to the point of bottleneck

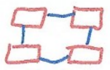
The bus only allows one request at a time, in part bc we rely on the bus to provide that "easy ordering" that we get between writes to maintain coherence, but this causes bottleneck. So, what we need is a scalable on-chip network that allows the traffic to grow as the number of cores grows, & then we also need to use directory coherence so that we don't rely on the bus to serialize everything for us.

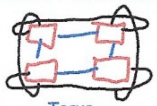
Network on Chip

Instead of a bus, use a mesh connection. This will increase the throughput of the entire network.

With a mesh, each additional core increases the number of connections to the network.

Torus Networks are a wrapped mesh network.


Mesh


Torus

Having a mesh network instead of a shared bus removes the single bottleneck. Cores will only utilize the links they need to talk directly to the cores that they need to talk to. This increases throughput.

Many Core Challenges 2

More cores means increased coherence traffic, requiring:

1. A scalable on-chip network
2. Directory Coherence We want to use Directory Coherence to serialize shared memory accesses (we used to rely on a shared bus for that).

As the # of cores increases, so does the "Off-Chip Traffic". Each added core means an additional # of on-chip Caches. So each core doesn't necessarily generate more misses itself, but it generates the same number of misses PER CORE (regardless of how many cores we have). So more cores/caches means more memory requests (caused by a cache miss).

Off chip traffic increases while the number of pins on the chip increases but not at the same rate. To solve this problem:

The last level cache is shared and distributed among all the cores and its size increases with each core.

The pins are what connect the chip to off-chip resources (like memory), so the # of pins that we get means the more off-chip throughput that we can achieve. However, the number of pins is not proportional to the number of cores, so more cores does not necessarily mean more off-chip throughput. This becomes a bottleneck.
To solve this problem, we want to reduce the # of Memory Requests PER CORE. We do that by making the Last Level Cache (LLC), which is the L3 cache in most modern processors, SHARED (so that all cores go to that cache) and have a SIZE that is proportional to the # of Cores using it. HOWEVER, making one big LLC is just moving the problem (bottleneck) from memory to the LLC. Instead, we need a DISTRIBUTED LCC.

Distributed LLC

A Distributed LLC is logically a single cache. It's still 1 big cache that allows us to prevent the many cores from all trying to access memory, but it removes the single entry point (bottleneck). Rather, each slice has its own entry point.

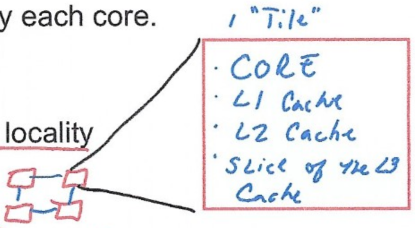
The distributed last level cache (LLC) is sliced up and controlled by each core.

How to slice the cache?

-Round robin - this method is not good for locality

-Round robin with page numbers - this method is better for locality

With a Distributed LLC, more cores means more slices which means a larger distributed LLC, which also means more entry points to the LLC.



All of the Cache Blocks that are part of the same Page go to the same Slice. This provides better locality than just round-robin distributing all of the Cache Blocks to different slices.

Many Core Challenges

The Coherence Directory becomes too large when there are many cores.

We need a Directory entry for each Memory Block, but we could have many GBs of memory, so that would mean we'd need Billions of Directory Entries, which won't be able to fit all on a single chip. How do we deal with this problem?

On-Chip Directory

So the on-chip directory is sliced and distributed to each core.

The directory is sliced the same as the LLC.

This means that for a given block, the "Home Node" where that directory entry for that block COULD be found would be on the SAME NODE that has the LLC Slice that's responsible for that same block.

Partial Directory - a limited number of entries reserved for blocks that are in at least one cache.

We have seen that we only keep the data in the LLC Slice for those blocks that we actually have in our L3 Cache. Originally, a Directory would need to have a Directory Entry for EVERY POSSIBLE Memory Block that might ever come into that slice, which is a big problem because there are a lot of memory blocks. Instead, we realize that the entries that are NOT in the LLC (or any caches on chip) are not shared by any core, so there's NO NEED to maintain that directory information. We have a Partial Directory where our Directories will have a limited # of entries and only allocate entries for blocks that are present in at least ONE of the PRIVATE CACHES (L1 or L2). For a block that we know is NOT in a Private Cache but MAY OR MAY NOT be present in the LLC, WE DON'T NEED THE DIRECTORY ENTRY because that entry would just be all zeroes.

On-Chip Directory 2

Regarding the last sentence, why does it not matter if it MAY OR MAY NOT be present in the LLC? Because remember that the LLC is a SINGLE SHARED CACHE that is used by ALL the cores. This is the same relationship that the cores have with MEMORY - ITS A SINGLE SHARED ENTITY. So the LLC is not like the other L1 and L2 caches, which are private and multiple caches can have their own copies of a given piece of data. Since they're private, they NEED the directory in order to maintain coherence. However, with a SINGLE SHARED cache, there's no need "maintain coherence" because the fact that there's only a single thing means that "coherence" and "ordering" are inherent properties.

Even with a "Partial Directory", we could still run out of entries. What will we do when we run out of space for entries?

Similar to how we'd run out of Cache Lines and we need to use a replacement policy, we'd do the same thing when a Directory runs out of entries.

When a directory becomes full, use an LRU protocol to replace an entry. This type of miss is caused by invalidation due to a replacement.

So using a replacement algo (ex: LRU), we decide that we'll need to replace the Directory Entry "E". This entry "E" might have some Present Bit set, so for each Present Bit that is set in entry "E", we need to send out an Invalidation to the corresponding cache. Since the Directory Entry tells us which caches has a copy of this memory block, we need to tell those caches to invalidate their copies of the block (and write back if necessary) so that we can set all the Presence Bits for this Directory Entry "E" to 0, which will effectively be the same as if we didn't have that entry E in our Partial Directory.

Many Core Challenges 3

The power budget available for a chip must be split between cores, so the frequency and voltage must be reduced.

As the # of cores goes up, the power that we can spend in each core goes down, which means that the Frequency & Voltage that we can use in each core goes down, which ultimately means that with the SAME SINGLE THREADED PROGRAM, it will execute SLOWER on processors with more cores. Ex: A single core processor will get the entire power budget of the entire chip, however a processor with 64 cores will only get 1/64th of the power budget available for the entire chip.

Multi-Core Power and Performance

The more cores, the slower each core can operate. To combat this, the cores that are operating get a boost of power and frequency.

Multi-Core Challenges 4

The final challenge for Multi-cores is Operating system confusion caused by multi-threading and cache sharing between cores.

SMT, Cores, Chips

The operating system needs to know on which core to run the thread to obtain the best performance.

...that requires the operating system to actually know what's going on. So most of the well known operating systems like Windows, and Linux, and so on, will actually be able to figure this out. But that means that as we have this fancy hardware that has parallelism at different levels of granularity, we also have to make our operating systems aware of that. So it's not just expose more thread contexts that the operating system can use. It actually matters how you use this thread context.