Parallel Computer Architecture

Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

Week - 04
Lecture - 19

Lec 19: Cache Coherence Problem

Hello everyone. We are starting the new module number 3 on cache coherence. Lecture number 1 is going to discuss about the cache coherence problem. A quick recap related to shared memory because this whole aspects which we are going to discuss is based on the shared memory multiprocessor. So in two minutes we will quickly take a recap about shared memories. Okay. So when we discussed in the first module about introduction to parallel architectures, we saw that the shared memory paradigm has been a prevalent architecture.

Why because it consists of small computers or processors connected to an interconnect. So the interconnect could be bus or bigger global interconnect and to this interconnect we have connected the different memory modules and the disk etc. Okay. So overall we found a symmetric multiprocessor if the accesses to the memory are equal for every processor. And then we also have other types of architectures where we have non-uniform access and so on. Okay.

So in this setup where I have a processor, each processor has a cache of its own and then once a processor and a cache forms a module it is connected to the interconnect and then through that interconnect it accesses the memory. So all these processors are connected over a common interconnect and they form building blocks of a larger system. Okay. So how am I going to use such a system? I am going to use it as a throughput engine. It is very good for parallel programming and also for the operating systems. Okay. So let us see how it is acting as a good throughput engine.

We have several computer systems connected. So I can have different programs running on them, maybe not a big parallel program but small tiny programs can run on multiple such processors and we can complete them very fast in parallel. So essentially I am increasing the throughput of the system, hence such a symmetric multiprocessor can be used as a throughput engine. It is also good for parallel programming. Why? Because a parallel program normally needs to share data amongst the threads or the different processes and a shared memory paradigm helps me to do this very easily using normal memory load and store instructions. Right.

So if I have a shared memory then I can do load and stores among these shared threads and improve my parallel program across multiple processors. Right. So a parallel program can run on a single processor or across multiple processors. Then such a system is also useful for the operating system because OS itself is a multi-threaded a big program and if it has this facility of sharing the memory, it can run across multiple processors and give a good throughput. Right. So the shared memory is not only useful for parallel programs, it is also useful for message passing because if you recollect we had discussed about convergence of the shared memory and the message passing that at the end of the day we have load stores when we want to send a message also we want to do read and write across memory. So message passing can also be implemented using the shared memory.

So that is the usefulness of a shared memory. Okay. So that is why we need a shared memory system. A quick recap on the existing designs which we have discussed. The first one as you can see uses shared cache. So here there are different processors but every processor is using the same first level cache.

There is a single cache used by all the processors and after the cache is the main memory. So all of them go through the same cache before they reach to the main memory. The second is a bus based design where every processor has its own cache but this cache is then connected over the interconnect to the memory and the I/O devices. The third one if you recollect this is the uniform memory access UMA dancehall architecture which gives similar latency of access to any memory module. So if you are accessing this memory module or this memory module it is going to take the same amount of time that is uniform memory access.

And the third one is a distributed memory which is the NUMA architecture. So this architecture is non-uniform memory access because if the data item is found here, this is fast and if we need to go over the interconnect to a distant memory block then this will be a slow access. Okay. So these are different varieties of shared memory architectures. In all of this if you notice that every processor is using a cache. The first one has a common cache but the others have their own private caches.

And why are we using a cache here and not directly with the memory? Because the cache helps us to bring the data closer because of the property of locality of reference and due to this every processor is able to access the data very quickly increasing the access time sorry improving the access time and it also needs to go very less amount of time to the main memory. So if more caches are found then the processor module does not go on the interconnect to access the memory and hence the bandwidth requirement of every processor goes down. Okay. So that is the importance for a cache that it improves the data access time and reduces the bandwidth requirement. So it reduces the data access time and also reduces the bandwidth requirement because most of the accesses are satisfied by the local cache. Right. And the infrastructure is such that it helps me to seamlessly do load and store.

So a processor does a load or a store. So this load store actually only goes to the cache from the processor's perspective but from the processor it goes over the interconnect. Then it goes to some memory and maybe something more before the responses are generated and given to the processor. Right. So this whole thing is happening using simple load and store instructions and which is transparent to the processor. So we have discussed why the caches are critical to the performance.

Now we will understand the property of the memory abstraction. So when I say we have a memory what is the model of memory I am assuming. Right. So what is the intuitive model of memory? Memory is nothing but a storage device. It helps us to store information in a set of locations. Okay. So memory is nothing but a set of locations in which I am going to store data.

And when I store data inside this I am also guaranteed by the memory model that I am able to read the latest value of the written data. So if I have written x equal to 5 inside this and there is no other change to this x then I am going to get x equal to 5. In case somebody else changes x and I go to read x after that so I should get the latest value of x. Right? So the intuitive model says that I have a set of locations which help me to get the latest value of the given location. And at the same time it also gives me a shared address space for inter-process communication.

So these are the roles satisfied by a memory. So sequential programs definitely rely on this as well as parallel programs. So in parallel programs that you shared memory to communicate if these programs are running on the same processor then they will go through the same cache hierarchy that is the shared item is there in the memory but all the parallel programs are running on the same processor so they have the same cache hierarchy to reach to the main memory and hence there is a single path so there is not much problem. But when a parallel program runs across different processors then it will go through different caches so the cache hierarchy will be independent of these two threads and that might create problems. So our objective is that whether they run on the same processor or whether they run on different processors the end results have to be the same.

I want the final result to be the same no matter whether the program runs on single processor or across multiple processors. Okay. So when I expect the same result the problems will come when I am accessing the memory through different caches. Now let us see how. So these private caches which are there with the processor might have copies of variables, the shared variables and multiple such caches will have these copies and any write by one processor to its local cache variable will not be visible to the others because then in this case the others will continue to access the stale data. Okay. So all. So to repeat it again processors have cache if they change variables in their local caches these changes will not be seen by other processors and overall this is called the cache coherence problem.

Now what to do about this? This problem is occurring because I am allowing a processor to change a variable in its local cache which is shared by others. So one easy solution is don't allow caching of shared values but then the caches are good for performance, so I cannot take that solution. So disallowing caching is not a good solution. The other thing is maybe ask OS's help to manage the shared data. However disallowing is not good taking help of OS is time consuming and hence we want to handle this whole thing inside the hardware. So the cache coherence problem is required to be handled in the hardware.

So we will quickly see how this problem arises. Right. So this is memory which has got some variable x and if this processor wants to do a read of x so if it does a read x the x comes here. Later this processor might do a write x. Similarly this processor also might do a read and a write of x. So these two writes the write number 1 and write number 2 by the two different processors can happen independently and any values of P1 are not seen by P2. So P1 and P2 cannot see each other's values.

Similarly when P3 goes to the memory to read it is going to get some old value and not the one which is changed by P1 and P2. Okay. Because of the private caches this is going to happen. So I will take similar example and elaborate it further. Here I have taken not variable x but a variable u. Now this variable u is done a read by P1 and then P3 also reads it.

So this is the sequence. After reading u equal to 5 P3 modifies it to 7. So u becomes equal to 7 and later P1 issues a read of u. So when P1 issues a read of u, if you will see this is going to result into a cache hit and the value of u which will be read will be equal to 5. Whereas if you see in real life the value of u is equal to 7. Actual value is 7 but I am reading a 5. What if P2 tries to read u? So here P2 will result into a cache miss and it will go to the memory bring the value of u.

So very likely it is going to read the 5 and not the 7. Okay. So whether P2 reads a 5 or a 7 will depend on the type of cache P3 is having. Okay. So if P3's cache is a write back cache, P3's cache is a write back, so write back caches will evict the data item only when the block is removed. Right? So when this data item is evicted only then it will update the memory. So if it is a write back cache, u equal to 7 will remain with P3 and it will not reach the main memory.

So here let us list the inconsistencies. So in this case P1 will get the value of u equal to 5 and similarly P2 will also get the value of u equal to 5. So these are wrong values or incorrect values. So P1 and P2 both are going to get incorrect values in this scenario and definitely the memory is also inconsistent. So even the memory will not have the correct value at the end of event 3, Right? Because event 3 is only updating the local variable u inside P3.

So that is about write back. Now we will take the second scenario when P3's cache is write through. If P3's cache is write through the u equal to 7 will have reached here and this will become 7 at the end of event 3. So because of this write through behavior which modules will have the wrong data? P1, yes because P1 has a local cache hit. What about P2? In this case P2 will get the correct data because P3 is a write through cache and when P2 comes to the memory it will get the new updated value. This is updated P1 gets the old value and P2 may get a new value.

So I am not saying whether it gets old value or new value because when P2 comes to read the memory it depends whether the write through has finished or not. So after the event 3 the write through takes place. But if the P2 goes to the memory before the write through has finished, then it might get the old value. If it goes after the write through has finished, then it might get the new value. Hence we are saying that P2 may or may not get the new value. Okay. So we have seen the effect of write through as well as a write back cache. Okay.

Then another scenario is when two processors perform write back the same variable back to back. So in this suppose P1 and P3 both change the values to the memory. So both evict the blocks and P1 will write some its own value of 5 or 6 something and P3 will write its own value of 7. So P1 sends u equal to for example 5 and P3 sends u equal to 7. So both of them evict these blocks and both these blocks go to the memory.

So both these write backs will happen in some order and so what is the final value which the memory will have for u? So inside the memory what is the value of u? Is it equal to 5? Is it equal to 7? Right? So it depends on what order the write backs reach the memory and although anything is okay but from the program's correctness point of view, we have to be sure what is the final value which we are expecting to be written into the memory. So the ordering of writes is also equally important. Right. So therefore maintaining the shared data items coherent across all these caches, right, all these private caches are these shared data items and they need to be maintained coherently and this challenge of maintaining coherence arises because of the private caches and that this problem is called the cache coherence problem and we are going to learn how to solve this. Right. So this lecture was to introduce what the cache coherence problem is about. Thank you.