

Cache Coherency

Russell Hitchcock

Introduction

I've written a couple of articles on multi-core technology recently; one was on multi-core CPUs and one was on multi-core programming. Seeing as how I have got multi, multi-core thoughts floating around in my head I thought I would write another article on the topic. This article will discuss cache coherency as it relates to multi-core processors.

What is cache coherency? In the context of multi-core processors, cache coherency refers to the integrity of data stored in each core's cache. But why is cache coherency necessary? To answer this question I will refer to the multi-core processor shown in figure 1. Imagine that there are two threads running through the processor; one in core 1 and one in core 2. Now imagine that each core accesses, from the main memory, variable 'x' and places that variable in its cache. Now, if core 1 modifies the value of variable 'x', then, the value that core 2 has in its cache for variable 'x' is out of sync with the value core 1 has in its cache. This is an important issue with multi-core processors. Actually, this problem is not very different from multi processor (multiple chips) cache coherency problems.

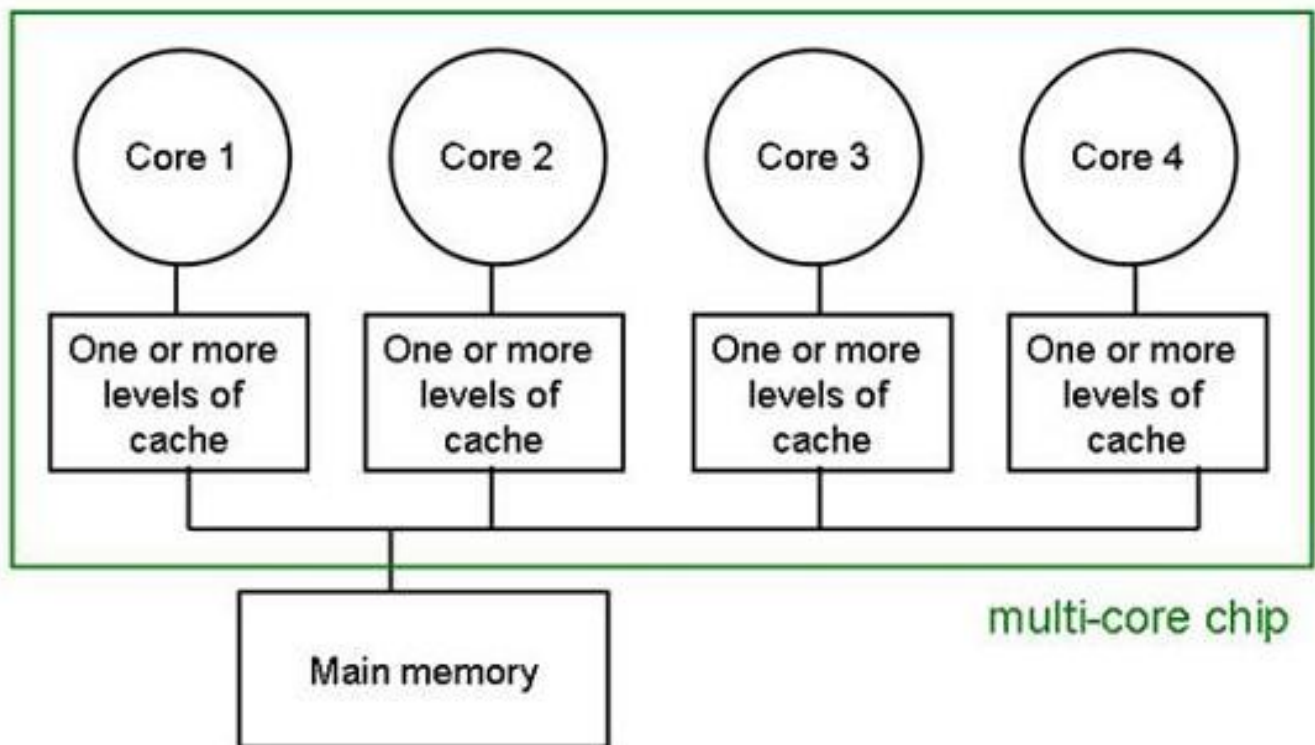


Figure 1: Diagram Of A Multi-Core Processor

In multi-core processors, cache coherency is solved by utilizing an inter-core bus. This inter-core bus is a bus which is connected to the cache of each core; see the cache bus in figure 2.

Cache Coherency

Russell Hitchcock

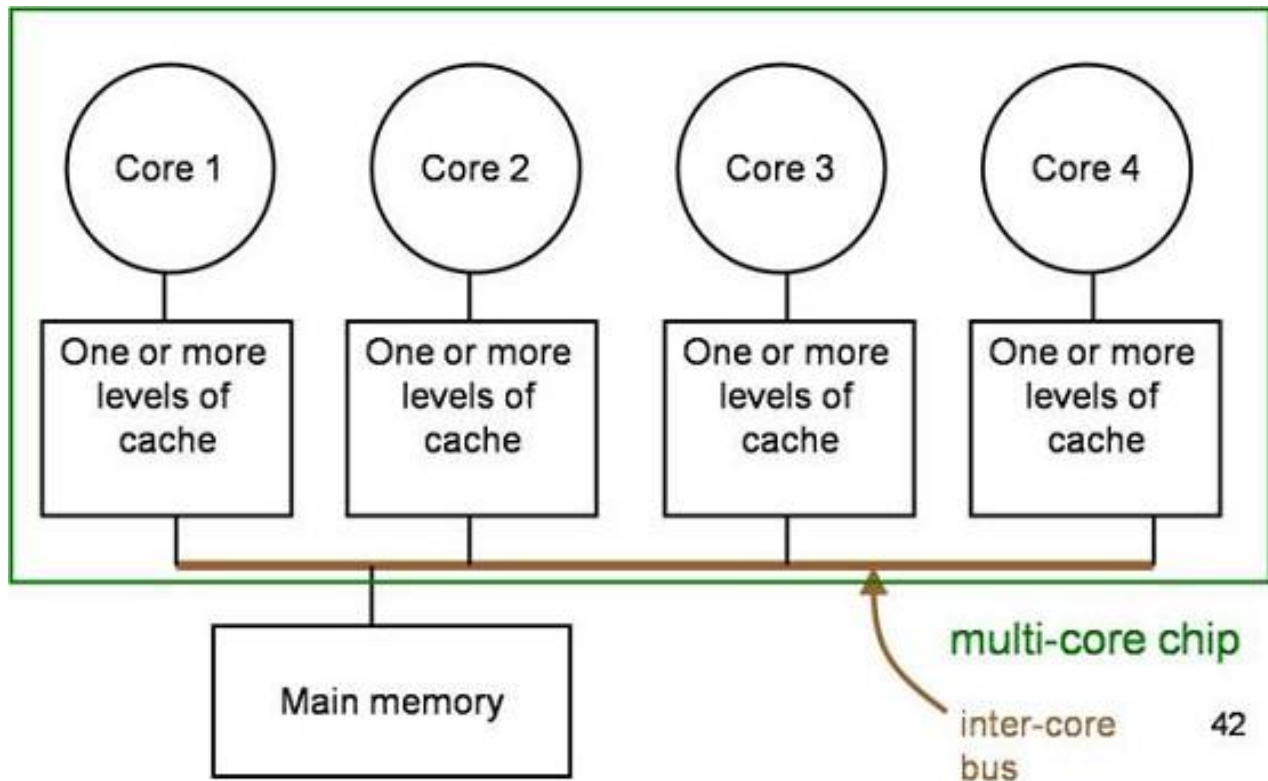


Figure 2: Multi-Core Chip With Inter-Core Bus

Cache Coherency Protocols

There are two basic methods to utilize the inter-core bus to notify other cores when a core changes something in its cache. One method is referred to as “update”. In the update method, if core 1 modifies variable 'x' it sends the updated value of 'x' onto the inter-core bus. Each cache is always listening to the inter-core bus; if a cache sees a variable on the bus which it has a copy of, it will read the updated value. This ensures that all caches have the most up-to-date value of the variable.

Another method which utilizes the inter-core bus is called invalidation. This method sends an invalidation message onto the inter-core bus when a variable is changed. The other caches will read this invalidation signal and if its core attempts to access that variable, it will result in a cache miss and the variable will be read from main memory.

At first glance it probably seems like the update method is the preferred method because we all know that cache misses can cause significant performance costs. However, the update method causes a significant amount of traffic on the inter-core bus because the update signal has to be sent onto the bus every time the variable is updated. The invalidation method only requires that an invalidation signal be sent the first time a variable is altered; this is why the invalidation method is the preferred method.

This example of the invalidation method is very basic. Much work has been done over the years to improve cache coherency performance. This has resulted in a number of cache coherency protocols.

MSI

MSI is a basic but well known cache coherency protocol. MSI stands for Modified, Shared, and Invalid. These are the three states that a line of cache can be in. The Modified state means that a variable in the cache has been modified and therefore has a different value than that found in main memory; the cache is responsible for writing the variable back to main memory. The Shared state means that the

Cache Coherency

Russell Hitchcock

variable exists in at least one cache and is not modified; the cache can evict the variable without writing it back to the main memory. The Invalid state means that the value of the variable has been modified by another cache and this value is invalid; the cache must read a new value from main memory (or another cache).

MESI

Another well known cache coherency protocol is the MESI protocol. MESI stands for Modified, Exclusive, Shared, and Invalid. The Modified and Invalid states are the same for this protocol as they are for the MSI protocol. This protocol introduces a new state; the Exclusive state. The Exclusive state means that the variable is in only this cache and the value of it matches the value within the main memory. This now means that the Shared state indicates that the variable is contained in more than one cache.

MOSI

The MOSI protocol is identical to the MSI protocol except that it adds an Owned state. The Owned state means that the processor "Owns" the variable and will provide the current value to other caches when requested (or at least it will decide if it will provide it when asked). This is useful because another cache will not have to read the value from main memory and will receive it from the Owning cache much, much, faster.

MOESI

The MOESI protocol is a combination of the MESI and MOSI protocols.

Directory-Based Cache Coherency

The protocols described above work very well and are commonly seen in both multi-core and multi processor systems. In the example of the multi-core processor I showed above, these protocols would work well. In the future, when processors contain dozens of cores, I am sure that these types of protocols will produce an abundance of traffic on the inter-core bus. One method of dealing with this will be to move towards a directory based approach. In a directory based approach each cache can communicate the state of its variables with a single directory instead of broadcasting the state to all cores.

In figures 1 and 2 the cores of the processor are connected via a bus, this may not be the case when the number of cores starts to increase significantly. I hope that I have shown you that the processor itself is really a network of cores which is quite similar to a network of computers with different interconnected topologies. With this in mind, many processors will be designed with their cores connected in a ring topology. Of course, if we are thinking of the processor as a network of cores there are a number of issues, other than topology, that need to be looked at. For example; application protocols, sessions, and basically any of the topics I discussed in my OSI Reference Model series, will apply to these networks of cores discussed.

Another factor that complicates the cache coherency issue is that some systems contain multiple processors, each with multiple cores. In this case each processor must keep its caches coherent with each other as well as coherent with the caches of the other processors. This requires a cache coherency strategy within each processor and a higher level strategy for keeping the caches coherent on the system level. Keep in mind that the strategies within each processor need not (and should not) be the same strategy used within an adjacent processor. These strategies should be tailored to the capabilities and needs of the individual processor. For example, even today many computers have a separate graphics processor which may have a different number of cores which are used in different ways. This would likely lead to a cache coherency strategy which is different than the strategy optimized for the CPU.