Parallel Computer Architecture

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

Week - 03
Lecture - 14

Lec 14: Memory hierarchy questions (2)

Hello everyone. We are doing module 2 memory hierarchy. In this lecture, we are going to address the next two questions in the memory hierarchy, question 3 and 4. Question 3 is, which block should be replaced on a cache miss? And why does this question arise? Because our cache is small and the memory is very big. So we just have say four blocks in this and there are more than a 1000 blocks in the memory. So if block A, B, C, D is sitting here and if block F wants to come into the cache, one of them has to move out.

So who should go out is the question. So which block to be replaced? Okay, so this question is interlinked with the previous questions on the choices allowed for a block. In a direct map cache, every block has got just one choice. So if this F is coming, it comes with its address and it says I am going to sit where B is sitting.

So B has to move out from there and F will start sitting in that position. So we do not have much choice, B has to move out. Whereas in a fully associative or a set associative cache, you have multiple choices. Right? There are four chairs in your group and if a new group member joins there, then one of the group members has to move out. Now which of these four group members should go out will be the question to be answered. Okay.

So this is what we are going to address in this question number three. So direct map cache is very simple. You have a single choice and this also makes the hardware simple because there is nothing to decide. Okay, everything is pre decided. So we have a simple hardware because there is a single choice.

Whereas in a fully associative cache, there are many options. Why many options? Because how do you decide who goes out? One is easiest. Use a random choice. The second one is the one who has been sitting here for a long long time. Let us move that one out.

So that is first in first out. One is the one who has not been doing a work recently, can be moved out, hoping that because this block was not being used for a long enough time may not be used in future. So that is the second option. And the fourth one is an optimal algorithm. Okay, so we elaborate on these four choices.

The first choice is random. So we will use a pseudo random generator to decide which block goes out and definitely the seed will be fixed so that we can reproduce the thing for debugging. Least recently used LRU will replace the block that has been unused for longest time. And why? Because we are relying on the principle of locality because this block was not used right now, it is likely that it will not be used in future. So with this information, because of locality of reference, I decide that the oldest block will be removed. Okay.

But with this, the hardware complexity increases. Why? Because I need to keep track about which block was used when and so on. So the recency of users, everything has to be recorded, maintained and decide to decide which block goes out. So the hardware complexity of this decision is very high. The third one is first in first out.

So this is not LRU, but this is deciding the oldest block in time, the one which was accessed in time much earlier will be removed. The fourth one is the optimal algorithm. Now what do we want? We actually want to be able to see in future and decide that among these four blocks, which block I will not need in future and then this will be the best choice. The best choice is the optimal algorithm, which says I will not require such and such block for the longest time. So we should be able to look into the future, which is definitely not possible, but we can design predictors or something to decide this.

So this is the best choice, but we could use any one of these or even random. Okay. So these are a few methods to decide block replacement. The next question is what happens on a write? Now why are we interested in writes particularly? Because most of the time reads happen in a processor. Okay. If you see any program execution, maximum of the accesses are read accesses. So I will give you an example.

Here, there is a program which has LD is the load, ST means store, loads are 26% and stores are 10%. Now I will ask you a question, what is the memory traffic on a write? And what is the traffic to the data cache? So we have instruction cache, data cache and then the main memory behind it. So you can pause the video and calculate this. Okay. So I will do it here. So what is the memory traffic on a write? How many accesses go to the memory which are of the type write? That is the question.

So 10% are stores. So this 10 %. So this 10% of how many total accesses? So I am going to do divide this 10% by what all accesses go to the memory, this 10% itself goes to the main memory plus the 26% of loads. This is about data and the instructions, all the instructions go to the memory. So 100% instruction accesses. Okay.

So if I do this, this comes out to be 7% approximately. Okay. So if you have calculated it, this comes out to be 7%. Then what is the traffic going to the data cache? Data cache, not the instruction cache, data cache handles only loads and stores, it does not handle the instructions. So out of this value, 10% are the writes and 26% are the reads. So we just have to say what percentage is 10 percent of the total accesses.

So $10\%/(10\% + 26\%)$. So this comes out to be approximately 28%. Okay. So the same calculation is given here. What do we conclude from this example? We conclude that first of all the writes are less to a data cache and if we translate them to the total memory traffic, they only translate to 7% of the total memory traffic. So in this example or in general, the writes which go all the way to the main memory are very less and hence the reads are frequent and we learn from Amdahl's law that we have to make the common case fast.

So make the reads fast. So from this we conclude that we have to make the reads fast. So how do we make the reads fast? In a set associative cache, you imagine those four ways are there and you parallely compare the tags of them. So while you are comparing the tags, can we start the read in an associative cache? Not possible, but in a direct map cache it is possible that you can start reading the block. So in a direct map cache, we can do a parallel tag compare as well as block read. So you read the block and you compare the tag together.

If it is a hit, then you send the block. If it is a miss, then you simply ignore the block, ignore your read. So this is how I can make reads fast. So start reading the block in parallel to tag compare. But this parallel approach is not possible in a write.

Why? Because write is going to modify the data and if your access happens to be a miss, you would have modified a wrong location or a wrong address. So you are not permitted to start writing until the hit or miss decision has been established. So you need to wait for the hit or miss to be decided. That is the first hurdle. Then writes are going to take some more time.

It is not like just go read, you have to first identify the location, go there and probably only write one or eight bytes. You are not going to modify the complete block. You are just going to modify a few words in that block. So even this is going to take time. So writes are small and they do take time to do this.

So how do we improve upon our writing abilities or what are the different options we can exercise so that the writes become faster and so on. Okay. So we have two write options, write through and write back. Now what do they mean? Write through means I have to update the data all the way to the next level. This is the cache, small cache backed up by the next level.

I will just take two levels. If it is a multi-level hierarchy, then your write through traffic only goes up to the next level. So a small cache backed up by a big memory. In this, if I change a variable x, if I do x++ which is sitting here, so the change in x, x is 4, it becomes 5. So this new value of 5 should get written here and it should also get updated to the main memory where x is sitting. The idea here is the block which is there in the cache will get the latest value but the value in the main memory is stale.

So we want to keep the main memory consistent with the cache. If it is a multi-level cache, we want to keep the next level of cache consistent with this level of the cache. So any change in the first level will be percolated to the second level. So that is why it is called a write through. So you have to do a through and through write from the small cache to the next level.

So both these updates have to take place. The other option is don't go all the way to the memory. Just keep on accumulating changes in the small cache and only send them to the main memory when you evict the block. So x sits here and x keeps on getting updated. Right?

You keep on updating x. Several changes to x occur and eventually when x gets evicted, why will it get evicted? Because some other blocks wants to come and sit in this position so x will be removed. So when x gets removed, at that time you will go and update it in the main memory. So this is called the write back approach. In the write through approach, we will update the cache and also update the main memory.

So both of them have to be updated. In a write back cache, update only the cache and only on the block replacement, we will send it to the lower level. Now how do I know that when I replace the block I should write the new value of x? Maybe the values are same. So whenever a block gets changed in the cache, we have to keep track that yes this particular block when it gets removed has to be written back. So we maintain that using a new bit called the dirty bit. There was a valid bit which we inserted when we designed the cache.

Now we are introducing a new bit called the dirty bit. It says that if this bit is 1, the block has become dirty that is we have updated the block. If it is 0, the block is clean.

Okay. So write back when you evict the block, you check the dirty bit. If it is set, replace, write the block contents to the main memory otherwise simply ignore the blocks.

Advantages of a write back cache, all the writes occur at the speed of the cache. Why at the speed of the cache? Once we update the value inside the cache, the processor can continue doing other activities. Compare this with a write through because in the write through, the data has to go all the way to the next level and once we have updated both these levels only then the processor can continue. So write throughs will be slower. Okay. So writes in a write back cache occur at the speed of the cache.

Then we can accumulate multiple writes and make a final single write to the lower level. So multiple writes can be done together. Another advantage is on the memory bandwidth. We need less memory bandwidth. Why because we are not going to the main memory now and again. Right?

We only go when the block is removed. The advantage with this is that I can use the main memory for multiple processors. So there can be multiple processors accessing the same memory because every processor is not having a heavy access or heavy use of the RAM. So RAM sharing is possible in a multi processor environment. It also saves power because we are accessing RAM less number of times. We are going less amount of times on the interconnect.

So interconnect and RAM are useless hence power is saved and when you can operate with lesser power such ideas can be used in embedded applications. Okay. So write back caches are good for embedded applications. They are also good for multi processors. Now what is good about a write through cache? Write through cache, on a read miss it is the most straightforward thing because when a miss occurs you simply have to replace the block because you do not have to worry of going to the next level and writing the block and coming back. Right?

The data in the next level is already consistent. So you simply replace the block and how do you replace the block? Overwrite the block with the new content and or simply change the V bit to 0. So whenever a block is deleted from a write through cache we can simply delete it by changing the valid bit to 0 because the blocks are always clean. Right? In a write through cache the blocks are always clean. The next lower level always has the current copy.

Even this is good for us. The next level always has the most up to date data and why is this good? Because it is helpful in maintaining coherence. If you are operating a parallel program or a multi threaded application they are all using shared data and suppose this shared data is cached by this processor it is making changes other threads would not know these updates. Okay. So if the cache is write through the local changes will be reflected in the main memory and hence all the threads will be able to see the current status or current value of the data item. So coherence is easy to maintain if you have a write through cache and hence it is good for multi processors.

So each type of a cache has its own advantages. However write through cache of course has challenge that the processor is stalled until we finish the write to the next level. So you have to wait until the next level is updated. For this there is a small optimization where I can use a write buffer. So there is a processor, the cache and then the memory.

So updating across these two is going to take time. So instead of sending the block to the DRAM we update it here and we also put the update information in a write buffer and then assume that this write buffer will eventually send the information to the DRAM and hence once the processor finishes the write in the cache it can immediately go and start working. Because this write is also copied in the write buffer. So processor can continue processing after putting the data in the write buffer. So memory update and further processing can be overlapped. Write through caches have the challenge of stalling the processor and this particular optimization can solve it.

So that was write back versus write through. A one small tiny question left. Now what happens in a write miss? Cache has a cache hit or a cache miss. Now what is this write miss? When we come to the cache for any activity either a read or write, it results into a hit or miss. So if we have come to read a block , it is a hit it is called a read hit. If we have come for reading a block and the block is not there it is a read miss.

The same thing here you have come to write and the block is not there so this is called a write miss. Now how is this different from the read miss? Write miss is saying I do not want to read the data I have come to update a value. It comes and says write x equal to 6 in this location. It does not say what is the value of x.

It says write x equal to 6. So in that case I am least concerned of what the original value of x is. So here I can think of variety of options. One is this block having x is not there with us. We have a write miss. So should we bring the block from the main memory into the cache and then make that x to 6 or simply go to the main memory and write that x equal to 6 in the main memory.

Okay. These are the two options. The first option when I bring the block first then update it is called write allocate and the second option where I do not bring the block but simply go to the next level and update the block in the next level is called write no-allocate. No-allocate means you do not allocate space for that block and simply go to the next level to change it because we do not need the data when we have a write miss. Okay. So write allocate the block is allocated on the miss during allocation. So bring the block, first allocate that is read the block into the cache, update the data and therefore it is similar to a read miss. So read miss also incurs bringing the block and then reading the data write allocate policy says bring the block and then change the data.

The write allocate write miss has no effect on the cache. The cache contents don't change here because what we are going, we directly go to the next level and update the next level. So we update the lower level of memory here in a write no-allocate policy. So we have seen write back write through write allocate write no-allocate. Now how do we put these four things together? If we have a write back cache then with a write back cache what would you choose? A write allocate policy or a write no-allocate policy. Okay. Write back caches are going to accumulate writes and eventually write it to the next level.

So it is desirable that if you have a write back cache you load the block and then perform the write. So this is the write allocate scheme because the idea behind the write back cache is that we may use the block in future. When you have write through caches any way the cache is write through even if you bring the block the policy is going to change this block as well as the next level. So why to bring the block here at all? So don't bring the block here. So use the no-allocate policy when you have a write through cache because any way we are going to update both the levels so don't bring the block.

So pictorially if you have a write through cache, pair it with the write no allocate policy. If you have a write back cache, then pair it with a write allocate policy. Okay. So with this we have answered those four questions and we finish this lecture. Thank you. .