Parallel Computer Architecture

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

Week - 03
Lecture - 16

Lec 16: Six basic cache optimisations (2)

Hello everyone. We are doing module 2 on memory hierarchy. Continuing with the 6 basic cache optimizations. The fourth optimization is targeting to reduce miss penalty. Okay. So to reduce miss penalty, we have to understand what are the different components of the miss penalty. From the cache, we have to go to the next level, either to the next level cache or to the main memory to bring. And eventually at some level of the cache, we will have to go to the DRAM.

So DRAMs are known to be slower than the processors. So there is a technology gap, which is making DRAM slower and compared to the processor. So overall time to fetch data from the DRAM increases, which means miss penalty is increasing over time. But at the same time, we want the caches to be faster.

We want a very fast cache, which matches the processor speed. At the same time, we want a larger cache so that we can reduce the widening gap between the processor and the main memory. Main memory is slower. So when we go there, we might as well bring more and more data and keep storing it here so that our misses are lesser. So less misses, capture more data, go less time to the DRAM because DRAM is slow.

So our objective is we want a large cache because the DRAM is slow. At the same time, I want a fast cache because the processor is fast. And if you think these two are conflicting requirements, because a large cache cannot be fast and a fast cache cannot be a larger cache. So how do I solve this problem? Okay. So we want to do both of them, both being conflictive. Is it possible that we can do both of them? So we want a fast cache and a large cache.

So I can do both if I associate one property to a different level of a cache. I'll use two levels of cache. The first level can be fast and the second level can be large. Okay. So I use the first level, which is small and fast, and the second level, which is large. So this larger cache will help us in reducing the miss penalty.

So this concept of having two levels or multiple levels is called the multiple level cache optimization, which helps in reducing the miss penalty. Okay. So when these two caches come into picture, my AMAT formula also changes. Initially AMAT was hit time plus miss rate into miss penalty, simple three terms. Now we have two caches or maybe three caches. So if we take just two caches, then the AMAT formula changes to something like this.

You need to take the hit time. So hit time is as it is plus miss rate into miss penalty. Now miss rate of the first level. So it is the miss rate of the first level multiplied by miss penalty of the first level. So what is the miss penalty of the first level? Initially it was simply the main memory access time, but now the miss penalty of the first level cache is the AMAT of the second level.

So if you see this, the miss penalty of the first level is the hit time of the second level followed by the miss rate of the second and the miss penalty of the second level cache. So the miss rate of the second level is now measured from the leftovers of the L1. So if you see this. Okay. So AMAT is first we look at the hit at L1 and then the miss penalty at L1. This miss penalty at L1 is hit at L2 plus of course miss rate is here, miss rate of L1, miss rate of L2 into miss penalty of L2. Okay. So this term now expands to something more. Right.

So what is the miss rate of L2? It is the leftover accesses from L1, the ones which L1 cannot satisfy will come to L2. Hence we define two types of miss rates, one a global miss rate and second a local miss rate. Local miss rate is very intuitive, the number of misses divided by the number of accesses to this cache. That is if 100 accesses came out from the processor, right? 100 accesses came to the processor, they all went to L1. L1 was able to have say 50% of them as hits. Okay.

Then the remaining 50% would go to L2. So out of this 50 went there and 50 were serviced. Now among these 50 you will get further hits or misses. So probably out of those 50 you got 10 hits, but then you missed on 40 accesses. Right. So this is the scenario.

Now if I talk of L1, what is the local miss rate of L1? L1 has missed 50 accesses out of 100 accesses. So it will be the local miss rate of L1 is 50 over 100. If I do the same thing for local of L2, what is the local miss rate of L2? L2 has missed 40 accesses out of 50 accesses. Remember L2 only got 50 accesses, it did not get 100. L2 gets the leftover accesses from L1.

So L2 has missed 40 out of 50, whereas L1 has missed 50 out of 100. So who is doing better? Definitely L1 is doing better, the local miss rate of L1 is much better than the local miss rate of L2. But you would say hey it is not fair because L2 itself got very less accesses, it was still able to cater to some of them. Because the local miss rate of L2 depends on how good or bad L1 is performing. So do we have another measure? So the other measure is called the global miss rate, which says that not the accesses which reach to this cache, but we will consider all the accesses generated by the processor.

So that is called the global miss rate. So global miss rate is number of misses divided by total accesses generated by the processor. So if I say global miss rate of L1, what would that be? The total number of misses, so it missed on 50 out of 100. So it is same as the local miss rate. And global miss rate of L2, what would be that? It had 40 misses out of 100. Okay.

So that would be with this example, this is the value we have arrived at. And if you now see that the global miss rate is more fair comparison or fair value for L2 rather than the local miss rate. Okay. Because the local miss rate of L2 becomes a function of L1, how good or bad L1 is and if you change L1, the local miss rate of L2 will change. Hence global miss rate is a more correct measure to use. Okay. So we are going to use global miss rate when we evaluate L2 caches.

So as we all now understood the behavior of L1, L2 interaction, if you have a larger L2, L1 cache would rather, it like skims the cream of the memory accesses it and only sends some of the accesses to L2 and hence the local miss rate of L2 is mostly not so good. So local miss rate of L2 is not a good measure, global miss rate is and therefore we are going to use this. So you see the formula, global of L1 is same as local of L1, but global of L2 is a product of miss rates of both L1 and L2. Okay. And why global? Because this is going to tell, indicate what fraction of memory accesses go all the way to the memory. How many go all the way to the main memory is more important.

So that was having multi level caches to reduce miss penalty. The other option to reduce miss penalty is to give priority to reads over writes. So when a cache encounters a read access and if the read misses into the cache, that is the block is not present in the cache. So for bringing this block, we go to the next level, bring the block and when you bring the block it has, it is replacing an existing block which is dirty. So to replace a block, you have to first write back the existing block and only then you can load this new block. Okay.

So that is the scenario I am discussing about. So when will you finish doing the read? Only when you have written this block back to the next level and loaded the required block. Okay. But this is going to take time because writing back the block is going to take long latency. So to prioritize this, giving priority to reads over writes, can I do the write in the background? Okay, so see the scenario, a read for a Y has come and it misses in the cache because X is sitting there. Whereas if you see the value of X in the main memory is equal to 2, so this implies that X is a dirty block, this has to be written back.

So we have to first write back this block here. Once that is written back, only then the Y comes in here. So first we write back the block, then we bring the block Y into this cache and then further complete the read. So this is going to take long time. So what can I do? Let me do the write in the background by using a write buffer. Right.

So we use a write buffer here and say that instead of 1 followed by 2, I will put this X equal to 5, I will call this step as 1' because this is a different step. So I will put X equal to 5 inside this and once I have put X equal to 5 in this orange box, I can bring Y. So step number 2 can be completed and in the background, we can go and update this in parallel to step 2 or 3 or later. So that is the use of a write buffer. This write buffer sits in parallel to the cache and carries data to the main memory.

So what should it contain? It contains the address and the value to be stored into the main memory. Okay. The advantage of this is it eliminates all the stalls on the misses because the processor need not wait. As soon as there is a miss, the data is transferred here and the cache and the processor are free to continue their work. But once we have additional storage or maybe a different copy of a data sitting in a different buffer, it comes with some problems which we need to cater to. Okay. So here we will see what problem and what is the solution.

It depends on the type of a cache. If you have a write back cache, write back cache, then the block is dirty. The dirty block is kept into the write buffer. We have, for example, I wrote that X equal to 5 inside this write buffer. Okay. And then we started doing the normal operations. Later, suppose there comes a read for X.

Then what will happen? In that example, X was supposed to be going to the DRAM. It has not yet reached DRAM and a new read for X comes. And it misses in the cache because X is not there in the cache. So what should the cache controller do? Should it read X from the main memory or should it read it from the write buffer? Because if it goes to the main memory, it is going to get the wrong value. And hence, in presence of a write buffer, we first need to check the write buffer and only then go to the memory.

So in case of a read miss, check the write buffer. If it is satisfying the value, then good. Otherwise, you go to the next level and read the data. The other option is that if you do not want to check the write buffer, then every time wait till the write buffer is empty. So, wait or stall the processor, flush out all the contents of the write buffer, and then start servicing the read misses.

But this would hamper the objective of doing the reads fast. Hence, it is always good to check the write buffer on every access. So that's the write back cache. Now what happens in the write through cache? We discussed that write through caches go very well with write no-allocate. So we have not allocated the data, but we cater to writes.

So here if X equal to 5 occurs in the write buffer, even if it is a write through cache, we are using an optimization which says that I am not writing it all the way to the main memory. From the cache, whenever X changes, the write through will put X equal to 5, the new value here, and then later that value will reach the main memory. So it's going to write it later even if it is a write through cache. And it is a write no-allocate, no-allocate meaning X is not brought into the cache at all. Okay. So in future, if there is a miss onto this block or a read miss into this block, we should always again check the write buffer before checking the main memory. Alright.

So either stall until the write buffer is empty, that is one solution which is slower or make provisions to check the contents of the write buffer before going to the main memory. So this care has to be taken when we use the write buffer. And with this facility, we can make reads faster because the writes can be happening in the background. Okay. So this reduces the overall miss penalty. So the sixth optimization is now going to target the hit time.

Of course, it was the first term in the AMAT, but we are targeting it at the end. So hit time consists of going into the cache, indexing into the cache, then comparing the type. Now indexing into the cache requires the address of the memory location. And if you think of a process, every process always deals with virtual addresses and not physical addresses. Whenever you compile a program, you will understand that you always talk in terms of virtual and not physical, logical addresses more and then not physical addresses.

So the processor always generates virtual address or a logical address for every process. It never generates the physical address. This virtual address is first translated to a physical address and then we access the main memory. Now that we have brought a cache sitting in the middle, should I use the virtual address or the physical address to index into the cache? So in theory, both are fine. But in practice, if we think that let me first translate the virtual address to a physical address and then we will go and index into the cache, this is going to take a lot of time and effectively increase the hit time.

My objective is to reduce the hit time, but if I translate the address, it is going to add up to the latency because we will have to go through the translation look aside buffer and then first move from virtual to physical and then to the indexing. Alright. So virtual addresses are better. Again from Amdahl's law, we learn that make a common case fast. So what are common cases? Hits are more than misses. So if the cache hits are more common, let me not translate the address every time because that same address is going to be there.

So you've recently translated it, so you know that it's there. So can I go without translation? Can I use virtual addresses and worry about translation later? So we are going to target that can I index the cache using the virtual address? So such caches which index as well as tag using virtual address are called virtual caches and the traditional caches which use physical addresses called, are called physical cache. Okay. So to make common case fast, hits are fast. So I am going to use a virtual cache which uses index as well as tag as a virtual address and not the physical address.

From the processor, we get the virtual address. This address can be used in the cache or it can be translated through the TLB to give out the physical address. So which one should I use this or that? Okay. A part of the address is used in the index, a part of the address is used for tag comparison. So index and tag compare are the two tasks I want to do. Should I use a virtual address for indexing or should I use a physical address for indexing? That is one question and the same thing, should I use a virtual address for tag compare or should I use a physical address for tag compare? So we want to check all these options and if we want to avoid address translation, then we should go for a virtual cache, that is both of them have to be virtual. But if you have a cache which indexes on virtual and also does a tag comparison on a virtual address, it comes with major challenges.

The first challenge is protection or security because every process when an address is accessed, the system or the operating system guarantees a protection across different segments of a of different processes. They have their own address segment and one process cannot read content of another process. But if the address is virtual, you cannot find out whether this address is allowed by this process or not. So until translation, we cannot guarantee the security and protection across the processes. Okay.

So the page level protection cannot be implemented. So the solution for this is we can include additional TLB related information inside the cache. So keep the cache virtual, but along with it also keep some TLB information for doing the security checks. The second problem with the virtual caches is that when we have a context switch, a new process starts running, the new process will again start issuing virtual addresses and very likely these virtual addresses will be same with the previous process. So should I say that

I am getting cache hits? Definitely not, because the virtual address 50 of the previous process is different than the virtual address 50 of the new process after the context switch. So this what, how do I solve this problem? So for this, we need to completely remove the cache contents that is flush the cache because the address 50 is always going to hit there because the previous process had brought virtual address 50. Okay.

So we need to do cache flushes to do this. A very expensive operation because if you remove the complete cache contents, you will have to reload the cache when this first process is again scheduled after a while. Okay. So we need not do complete cache flush, but we can associate the process ID with this information that is that virtual address will also be associated with the process ID. So in my example, if the address was 50 for process P1 and if the next time after context switch address 50 comes, we'll first check the process ID, whether it is matching before we declare a hit or a miss into the cache. So PID entry is also included in a virtual cache. The third problem is that different virtual addresses might map to the same physical address and we will end up bringing two copies.

So process has got virtual address 50 and operating system has got a virtual address 100. Both of them are mapping to the same physical address. So if I just say virtual address 50, this is there in the cache and another virtual address 100, it's also there in the cache. Both of them are different, but they are mapping to the same physical address. So both of them point to the same location in the main memory to some address 55 or something. Right.

So two different virtual addresses map to the same physical address. So in this case, 50 and 100 would be declared as two separate ones resulting into a miss, whereas when 100 came, it should have read the contents of 50. So this is very difficult to solve at the VA level. So this can be solved using the technique called anti-aliasing. So the problem of aliases and synonyms are definitely not present in the physical cache, but they are present in virtual cache can be solved using mechanisms called anti-aliasing.

So these are the three problems. One is page level protection, process switching, and OS level address sharing. Right. So these three problems come with their associated solutions if we are using virtual cache. Okay. So can we do something else so that we don't encounter most of these issues? So the best solution for virtual caches is to use a combination of the virtual and the physical cache, which is called the VIPT cache. It says that you index using the virtual address. Why? Because my hit time requires me to index into the cache and that has to be fast.

I have virtual address in my hand. Can I use this to index? After we index, what do we do? We bring out the data, we bring out the tag. So this is going to take some time. So while I bring out the tag, let me do the translation of that and then do the tag comparison using the physical address because we were discussing the index and the tag. So index and tag compare. So here I'm going to index using the virtual address so that this is faster and the tag comparison I will do using the physical address so that all my problems related to virtual cache can be avoided. Okay.

So this is illustrating how we are going to do this. The orange one, this is the address which has come. This is the virtual address. Right. This is the VA which came out from the processor. The virtual address is divided into a block offset, the index and the page number. This virtual page number has to go through the TLB and then the physical address actually or the physical component of that comes out.

So physical address comes out from here, a part of it and then gets appended to the index and the block offset to create the final PA. So PA will be created here but I'm only interested in the first portion of the virtual address because index and block offset I'm using as virtual. So here I'm using the virtual address for indexing the cache. So once we come here, I'm going to read the data, do the mux comparison and while the data is being read, that is the tag is being brought out here, the physical tag is already available. So index the cache, that is read this row using the virtual address, bring the tag out but the tag is stored as part of the physical address.

Hence you have to use the TLB's translated information to do the tag comparison. Okay. So the tag from the cache is physical and this tag is the physical tag after the translation. So what are we doing? We are doing some amount of translation in parallel, that is we are not waiting for the TLB to give the physical address and then go to the cache. We first go to the cache using the virtual index and by the time we bring the tag out, we expect that the TLB has finished the translation and given me the physical tag of that particular virtual address. So we will end up doing the tag comparisons using physical addresses and index the cache using the virtual address.

So this is the concept of a VIPT cache that is virtually indexed, physically tagged. Okay. So this comes with lots of advantages because it is help me, helping me in the hit time, the cache hits are very quickly done but it has a small limitation that the direct map cache size is limited to one page size. So you cannot have a cache which is direct map, which is bigger than the page size in the main memory and this is happening because I am reusing the index. The index is common for the translation and if the index is common, I am limited to the page size for my cache size.

Of course, we can play with associativity. If I say I have one direct map cache, direct map cache, which is 4 KB, which is the page size, I can have just this much but this cache is too less for me. So we might as well can increase the associativity and go for a four-way set associative cache with every parallel cache of 4 KB. Right. What is an associative cache? It is every way of the cache is actually a new cache in parallel. So you have four direct map caches in parallel in a four-way set associative.

So I can overcome this disadvantage by playing with the associativity. Okay. So using all these varieties, we can improve on the hit time of the cache. So these were the six cache optimizations. Thank you so much.