Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 08
Lecture - 43

Lec 43: Directory Overhead Optimisations

Hello everyone. We are doing module 5 on scalable shared memory systems. This is lecture number 4 on directory overhead optimizations. So in the previous lectures, we have seen the directory organization, that it was centralized, distributed and distributed had again a few varieties. The flat and the cache based were the two organizations of our interest. So in these two organizations, are there any storage overheads? If you want to recollect, we said that the flat memory based had a bit vector and the cache based system had a doubly linked list, just as a recap.

So now in this lecture, we are going to look at the overheads and can we optimize the storage overhead. So this was the directory organization centralized, distributed and here memory based and cache based are the ones we are going to focus on. Right. So for memory based, what is the storage overhead? So here I have shown a picture of the directory. So there is an associated memory storage, memory data is kept here and with each of these blocks, this is the bit vector. Okay.

So this is the bit vector for a memory, which has got m entries and this is a p bit vector. So horizontal, this is size is p bits. So how many total memory blocks do I have in the system? I have m memory blocks in the system because I have p processors and each processor has got small m number of blocks. So small m number of blocks per processor, p is the total number of processors in the system and hence the total memory throughout the system is capital M, which is number of processors into small m that is blocks per node. So the directory storage overhead for the global M, I mean, this directory is divided into small pieces across all those distributed memory banks, but now I am just combining it together and calculating the overhead. Okay.

So we have this M1-D1, M2D-2 such combinations throughout the network. So if you have p nodes, you have m, p number of memory banks and with every memory bank, you have a directory. So I have combined all this into this table and just to see that how much is the storage overhead. And for this, we will just do some calculations to get an idea. So we will do that on the next slide. Yeah, here.

To calculate the percentage overhead I incur for storing this bit vector. All right. So assume that one block is 64 bytes and with these 64 bytes of information, I want to store information of 64 nodes. That is my capital P is 64 and my memory block size is 64 bytes. So with 64 bytes of data, I am keeping information of 64 nodes. So 64 nodes when I say, how much information I need, I need to keep 64 bits because one bit for every node, 64 bits becomes 8 bytes and so overall I am keeping 8 bytes of information for every 64 bytes of data.

So you can calculate the percentage for this. See how much that comes out to be. So you can pause the video, calculate the percentage and also solve for the other two. Okay. So the overhead is I am keeping 8 bytes of information for every 64 bytes of data. The same exercise we will do for the next one.

Suppose I have 256 nodes, then my bit vector is going to be 256 bits, that is for every row. Here, this is 256, this is 256, this is 256 bits. So got it. So this is the overhead I am talking about. 256 bits translates to how many bytes? and so I will do 256 by 8 which is, so this comes to 32 bytes and we have 32 bytes with respect to 64 bytes of data. Okay.

So for storing 64 bytes of data, I am using 32 bytes. So what is the overhead? 50 percent. Okay. So for every memory block, I am using 50 percent more storage to keep the bit vector. Do it for 1024. So 1024 bits.

So 1024 bits becomes 128 bytes and I am using 128 bytes of bit vector for 64 byte data which translates to 200 percent overhead. Okay. So as you increase the number of nodes, you can observe that the number of bits required by the directory storage goes very high. If you have solved it earlier, you can cross check for 64 byte data block and 64 nodes, you have a 12.5 percent overhead. We did this 50 percent and it goes up to 200 percent for 1024 nodes.

So percentage overhead in terms of the P term. So how do I optimize on this will be my next question. Right. So this full bit vector scheme, I can optimize either by addressing the P term or by addressing the M term because M is, that is the number of blocks which is P into m. So if I reduce any of these, either the number of rows or the number of columns in my directory structure, I can have lesser storage overhead. So we can address the width term, that is the P term or we can address the height term which is the M term.

So reduce the height of the directory or reduce the width of the directory. Okay. So let us start with reducing the width of the directory, that is the P term. So what is this P term? It is storing the total number of bits per block and this bit vector has those bits

equal to 1 which are the sharers, remaining other bits are 0. And suppose I have 1024 processors for example, is a cache block going to be cached by all 1024 at a given point of time? Very unlikely. So normally we see that a block would be cached by 5 to 6 nodes or maybe 10 to 12 nodes and not 1000 nodes, not even 100 nodes.

So all these out of 1024 bits, I am only going to use 10 to 20 bits and remaining are always going to be 0. So why to keep them at all. I do not need to maintain an exhaustive bit vector for doing this. So what is an alternative? You would say that, if I have 5 to 6 or say 20 nodes which are sharing, can I explicitly keep information about them instead of a one to one mapping bit vector? Okay. So if you want to do this, you have to list the IDs or the addresses of the nodes which are caching this. And the easiest way to do is using pointers. Okay. To address the p-term, instead of storing the bit vectors, we think that can we keep pointers to the nodes ,that is the addresses of the sharers in this instead of the bit vector.

So even if I have 1024 nodes and I keep 10 pointers, I would still need lesser number of bits than the full bit vector. But if I have even say 20 sharers, if you have 20 sharers, then you would only need to keep 20 pointers and each pointer is 10 bit wide. So 20 sharers, every pointer is 10 bit because we have 1024 nodes in the system. So overall, I am going to need only 200 bits instead of 1024 bits. Okay. So if I have an estimate of the number of sharers, we can reduce this P-term. Okay.

So suppose I have only 5 to 6 pointers kept or 20 pointers kept, that suffices up to a point but in this example with 6 pointers, if the 7th request for the block comes, what will we do? Because what I have done is the P-term is storing 6 pointers. Right. It is pointing to the sharers. Now, this space is full and a new sharer comes, I cannot increase the space in hardware because this is what I have allocated. How do I solve this problem? Because already 6 pointers were kept. So to keep the 7th pointer, I need to do something and that is called handling the overflow. Because now we have an overflow, the storage is falling short and I need to devise overflow handling methods. For this, we are going to use the terminology, Dir_i.

So directory and I will put i here, subscript which says that a directory is keeping i pointers. If I say Dir_5, for easy writing I can also write the 5 here, not necessarily a subscript. So Dir_5 says a directory entry with 5 pointers. Now if the sharers go beyond 5, we need to do something, that is handle the overflow. Overflow can be handled using these 5 methods.

The first one is called the broadcast, second is no broadcast, third is the coarse vector, the software overflow and dynamic pointers. We are going to see them one by one and

the overflow method is abbreviated at the end. So I will say B for broadcast and NB for no broadcast and so on. So these are the abbreviations for the overflow schemes. So we will start with the first one, Dir_i B.

So i pointers and B is the broadcasting method for overflow. Okay. As the word broadcast says that when you exhaust these i pointers, you start broadcasting the invalidations because you don't know how many more sharers are there. So I have 5 entries here, I will have 5 pointers and if there are new sharers getting added to the system, I simply keep an overflow bit and make this bit equal to 1. And once I see that this bit is equal to 1, whenever a write request comes, I am going to send invalidations to everybody in the system. That is, I am going to broadcast the invalidations because I don't know that beyond these 5 sharers, how many more sharers have been added to the system.

So if the sharers are more than i, then broadcast the invalidations to all the nodes. Because this is semantically correct, even if the node does not have the data block, it will simply ignore the invalidation message which reaches. Negative point, it is going to waste your bandwidth. So broadcast is safe method but it is going to take lot of bandwidth. The other option is using no broadcast.

No broadcast if I employ, then how am I going to invalidate or how do I handle the overflow. Here there were 5 entries, 1, 2, 3, 4, 5 and the 6th node comes. No broadcast. So I cannot keep information about extra sharers anywhere. These are the only 5 positions to keep information.

So what I am going to do, I am going to delete one of them and keep the 6th sharer in its place. Because I don't have the broadcast facility available. So the overflow says that the new sharer will take the place of one of the old sharers. And when you delete the old sharer, what does deletion of a sharer mean? That you have to send an invalidation message to the node number 4 and then use its pointer position to store the new sharer number 6. This is good if you have writing type data with limited number of sharers. But if imagine an application which has several readers, that is there is one data block which is read by many processors and sequentially. Right. So they will keep reading on and off again and again.

So the reading order is say 1, 2, 3, 4, 5 and 6. Then again processor 1 wants to read. So what are we doing? We are invalidating each other's data and if there are more readers and more sharers, this method is going to lead to drawbacks. So widely read data is going to suffer in this particular method. Okay. So the third one is a coarse vector where I am going to use i number of pointers and beyond this when the number of sharers

increase, I don't use a broadcast or no broadcast method but I use the same storage of say 5 pointers.

I had 5 pointers and suppose every pointer was 10 bit wide, I had 50 bits available with me. Now these 50 bits I am going to give a new meaning to them and say that now these bit vector is not a pointer but a bit vector pointing to something. Now what could that something be? I have only 50 bits but possibly 1000 processors. So now every bit is going to represent the sharing status of these 1000 divided by 50, those many. So I am going to cluster the number of processors and say that this 1 bit in my 50 bit vector is pointing to this cluster of processors.

That is why the name coarse vector. So the vector is not fine, that is every bit doesn't correspond to a single processor but it corresponds to a cluster of processors. And I can have a cluster of size r. So that is denoted by Dir_i with CV_r. So we have r size cluster and when I turn the policy from a pointer to a coarse vector, we have to turn the overflow bit on. The overflow bit has to be used here. So we will take an example to understand this.

Here Dir_i CV_r means, I have i pointers and r size cluster. In this example I am using 16 processors P0 to P15, P5 and P10 are the current sharers. So how does the directory entry look like? We have the overflow bit which is 0, that is there is no overflow right now and among these 8 bits which I have, these are 8 bit entry, so my 8 bit vector was there, I am using half of it to store the first pointer and half to store the second pointer. So if I want to give a name to this, this will be Dir_2 because I am keeping 2 pointers.

CV we will see later. So this is Dir_2 for 2 pointers. So pointer 1 points to P5, pointer 2 points to P10. Suppose a new sharer comes. Now P15 wants to read or write the block, now it is a new sharer.

We do not have space for the pointer for P15. So in a no broadcast we would delete one of the pointers and add P15 to it. In the broadcast we will broadcast to everybody keeping pointer 1 and 2 intact and setting the overflow bit to 1. In the coarse vector scheme what do we do? We set the overflow bit to 1 and then convert the 2 pointer space to plain 8 bits. Right. So I have just flattened it out and said that, this is my 8 bit vector and now I have to cluster the processors because I cannot address 16 positions using 8 bits. So 8 bits can be used to point to only 2 positions.

So I have clustered them. If you can see this is one cluster and each bit will now point to one of these clusters. Our original sharer was P5. So P5 is pointed by this one which is this bit and P10 was my other sharer. P10 should have come here. Okay, when P15

comes as a new sharer I don't have space to keep it in the pointer array.

So first thing I do is I flatten this into a coarse vector and want to remove the concept of pointers. So all these 8 bits becomes my bit vector. First of all, set the overflow bit. I have 8 bits of information and I have 16 processors. So every bit is now going to point to a cluster of 2 processors.

So the DirCV will become 2 here because I have a cluster of 2 processors. So you can see the processors P0 and P1 are now combined inside this boundary saying that this is my one cluster. I have 8 such cluster. Every bit is pointing to one of those clusters and a bit is set to 1 when any processor in that cluster has shared the block.

So here P5 and P4. So P5 has shared the block but P4 hasn't. However, when you want to send an invalidation, I am going to send invalidation to this cluster which will internally broadcast the invalidation to both P4 and P5. Because we don't maintain information at the level of the processor but we maintain information at the level of the cluster. Same thing here, if you see this bit is equal to 1 which points to this cluster and here only P10 is sharing. However, we are going to send an invalidation to all nodes within this.

So I hope it is clear that we have a set of pointers. When pointers fall short, you convert that bit availability into a coarse vector and accordingly form the clusters. Right. So with this new coarse vector, how do I handle P15? Now P15 is my new sharer and I will turn this bit to 1. This is how I am able to handle almost all the processors in this coarse vector. So even if all 16 become sharers, I can still use the 8 bit coarse vector to handle the information. So we will take an example where I have Dir_8 and I have put a question mark before CV because you can find out the cluster size.

You can pause the video and try to solve this yourself. We have 8 pointers and my system has got 256 nodes. So what is the cluster size? So we have 8 pointers and what is the size of 1 pointer? 1 pointer for a 256 node system is 8 bits. 1 pointer is 8 bits and 8 pointers will become 8 into 8 which is 64 bits and plus the overflow bit. So I have the overflow bit and then a bit vector of 64 bits. In this, I can store either 8 pointers or I can store 64 bits of a coarse vector.

If it is a coarse vector, I have 64 bits and each of this bit is pointing to how many processors, that is cluster of what size? You have 256 nodes and you have 64 bits. You can calculate that to find out what is the cluster size. Okay. So that comes out to be 64 clusters and 256 by 64. So that comes to 4, hence I will say Dir_8CV_4.

This is solved again on the same slide. We have 256 nodes, 8 bits per pointer, 64 bits and that gives us 64 clusters, if every cluster is of size 4. The next overflow scheme is using software. So here I say SW for software. It essentially says you have to keep up to i pointers I can maintain with the directory and any overflow that is new sharers will be handled by software.

For example, I have i pointers, the i plus 1th sharer comes. I don't have space to keep its information. So what I will do, this new sharer plus this old i pointers, all of this information I will take and keep it in the main memory. Okay. And then I know where I have kept in the memory and I free up my bit vector. So my bit vector is now empty to keep fresh new i pointers and I have i plus 1 pointers kept in the memory and in future whenever I need to invalidate or access the sharer information, I am going to use software interrupt routines to do the job for me. Because these software routines will go to the memory, find out the list of pointers or sharers and then send invalidations, etc. So handling of overflow is done by software and definitely we have to also set the overflow bit in this case.

What is the disadvantage of this? Whenever I have a software scheme, I need a processor to run it because it is not done in hardware. So every overflow will result into an interrupt to the parent processo,r which has to invoke this interrupt service routine, which goes to the memory module, fetches the pointers, maybe i plus 1 or several such because I would be already keeping it inside and then handling the case. So the software overflow schemes your negative point. What is that negative point? I need interrupt service routine so we need to trap to the processor, which involves the processor occupancy. We are unnecessarily disturbing the routine processor job to handle the overflow and it is going to add to latency.

This is implemented in the MIT-Alewife processor, the software based pointers. It uses 5 pointers with 1 overflow bit and these are some statistics of the delay which get added. So overall the software idea is going to add disturbances to the routine processor and going to add to the latency. So can we do the same thing in hardware? So the next idea is DirDP which is for dynamic pointers.

Overall the same theme as SW but here the hardware is doing it. So I will have a dedicated hardware processor. So I will call it a protocol processor implemented in hardware, which is going to manage this list. So in the memory if the directory storage, this is my bit vector, if it falls short, it is going to point to some location where extra information is kept. And all the things which were earlier done by software, that is go here, read this, list out and send the invalidations will now be handled by a separate protocol processor. Okay. So this idea is implemented by this Stanford FLASH

processor. Here, there are no interrupts to the parent processor and hence we have less overhead.

So we have seen several of these overflow schemes and so which to choose will be the next question. We have looked at broadcast and no broadcast. They are not very robust because no broadcast ends up deleting old sharers and broadcast takes lots of latency. Okay. It has its own tradeoff and performance effect so they are not very popular. Then the general consensus is about full bit vectors because I have fixed information, how much invalidations to send and so on.

So full bit vector is more popular. The other scheme is the hardware based overflow management where the hardware protocol processor takes care of storing the extra pointers in the main memory and then handling the further invalidations. Okay. So these two are not very robust. We want full bit vectors which are performing reasonably good. The other candidates is the hardware based overflow. Coarse vector is also good but here you need to handle the accuracy of the overflow.

That is when overflow occurs, we need to do that properly. So hence dynamic pointers is another popular method to handle the overflow. All right, so that was handling the P term. Now we will start handling the M term. Again the same picture, I have shown the M and the P here to set the context.

So we have handled the P by shortening the storage or the width. Now we are managing the height term. How do I reduce M? Because M is the total number of memory blocks and I need to keep information about its sharers. How can I reduce M at all? Well the observation is, that not every point of time all memory blocks have been cached. So the total number of cache entries, even if you look at the cache sizes, it is going to be much smaller than the amount of memory.

So your cache entry sizes is much smaller. On top of it if I am talking of a particular block, the total number of sharers of this block will be even smaller than the total memory size. So why to maintain one entry for every memory line? Can I construct a cache to do this job? Because most of the time, 98 percent of the directory entries are sitting idle. To solve this, I am going to propose that we can use a directory as a cache and when you make it a cache, you can store it in SRAM instead of DRAM, which is faster. So your access becomes faster by making it into a cache. Now how do we do this? Here, not all these entries M and number of entries are in use.

Most of the entries are unused. So only those entries suppose I am using these two. You maintain a directory for only those memory blocks which are getting cached and shared.

Okay. For the other memory blocks, we do not need to keep information. And you will observe that if I have this big enough, I am able to handle all the fresh accesses which are coming. In case the cache becomes full we have to replace one of the entries using some policy. When I replace an entry, suppose this is one entry which I want to remove, then that address or for that address, whatever are the sharers, I need to invalidate them, So that I can remove this entry and make space for the new memory block to sit there.

The directory cache has a single entry and it is size is just one. We do not want special locality because there is no special locality in this. So there is no special locality, cache every entry is of size one. This is the same cache used by all the processors because we have constructed a cache for the directory. If you distribute it, it is possible but you need to handle it separately. But if you imagine that this cache is one central cache storage, you come to this as a point of reference to read the sharers information. This cache is popularly called the sparse directory because it is a directory but keeps information of memory blocks which are sparse, that is lesser blocks information is kept here. If I have one directory with several processors, it is going to create a bottleneck. But you can make it larger, you can also increase the associativity so that you have less conflict misses and manage that sparse directory which is a cache properly, so that we can reduce the overhead of the M term.

So to summarize we have looked at directory organization, centralized, distributed. Distributed was flat memory based and then flat cache based and distributed hierarchical. So, we have seen overview of four organizations. Then for flat organizations we said that there was lot of overhead in terms of the M term and the P term. We reduced the P term by having lesser bit stored and then had several overflow schemes discussed. For the M term, that is the height term, we discussed to store using a sparse directory instead of one entry per memory block. Okay. So with this we finished the optimizations on storage. Thank you so much.