

**High Performance Computing**  
**Prof. Matthew Jacob**  
**Department of Computer Science & Automation**  
**Indian Institute of Science, Bangalore**

**Module No. # 06**

**Lecture No. # 29**

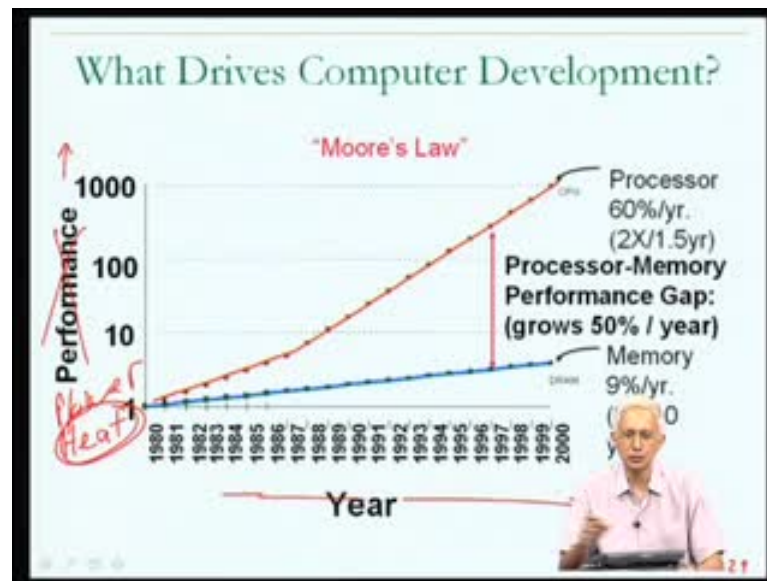
Welcome to lecture twenty-nine, of our course on high performance computing. We are currently looking at the operation of cache memories, trying to understand how they affect the performance of our programs.

Towards the end of the previous lecture, I had giving you a rough idea of one of the determinants of computer development, and I have shown this to you in the form of a graph, which is often described as the Moore's law graph. What this graph actually plotted was over a period of time, where on the x-axis, there was the linear progression of time with years.

How the performance of computer systems or the components of computer systems was improving due to technology developments and the important thing about the Moore's law graph was that, it was showing you on the log scale on the y axis, linear scale on the x-axis, that the speed with which processor circuits were becoming faster was much more than the speed at which memory circuits were becoming faster.

In other words, the speed disparity between processors and memories, was growing faster and faster under the Moore's law prediction, and the consequence of this was that cache memory hierarchies may have ended up becoming a little complicated, but before actually coming to that, I did want to comment - if you look at the Moore's law graph, which was one of the slides in the previous lecture, then you would have noticed that the graph was plotted for a period of time from about 1980 until about 2000, in the graphs that I used. Originally, Gordon Moore had plotted the graph for data from then much earlier points in time, 70s and into the 80s, and subsequently, people had found that this trend had continued. Occasionally, the slope of the performance improvement line may have been more or less, depending on how technology trends had moved.

(Refer Slide Time: 02:16)

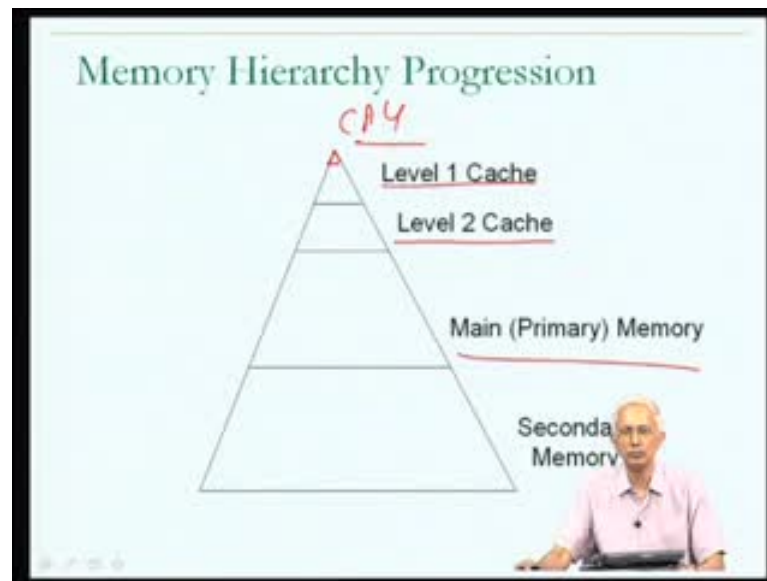


But one additional fact, which should be borne in mind to understand why the consequences of the Moore's law graph may not be that much today, should be by bearing in mind, that when the performance of a computer system goes up, it is possible that the amount of power, that the amount of energy, that is consumed for the execution of a program may also have to go up.

By power, you might view the amount of heat that is generated when a program runs on the computer and it turns out that the power and the heat are alternative labels for the y-axis in the Moore's law graph. In other words, as one looks along time at the amount of heat generated by a processor, one finds the similar exponential or similar behavior to what was plotted for performance.

As a consequence, it soon became over. As time went on, it became very clear that one - computer architects could not continue designing processors, which became more and more complicated, and more and more powerful, because at the same time, the amount of heat, the amount of power would be growing at a very fast rate and would become very difficult to cool down, and to even to provide the energy for these computers to keep on executing, which is why the technology trends did not proceed on in this direction for very long. And some tradeoffs have to be made between improvements in performance and reduction in power and so on.

(Refer Slide Time: 03:32)



But in any event, the bottom line from our quick understanding of the Moore's law graph is that, if we had thought of a memory hierarchy in which there was cache, main memory and secondary memory, then, with time, as the speed disparity between processors and main memories became worse, there may have been the need for improving the nature of the caches, to have small fast caches in addition to larger slower caches, just to close the speed gap between the CPU and the main memory. Remember that the CPU operates at speeds which are comparable to those of the very highest part of the memory hierarchy, such as the registers.

Now, with those additional comments to my quick discussion about the Moore's law from the previous class.

(Refer Slide Time: 04:18)

The slide is titled "Cache and Programming" in green text. In the top right corner, there is a handwritten note in red: "16 KB / 32B". The slide contains a bulleted list of objectives and a diagram of a cache configuration. The diagram shows a box divided into three sections: "Tag : 18b", "Index: 9b", and "Offset: 5b". Above the diagram, there is a handwritten note "Direct mapped 16 KB write back cache with 32B block size" with a red arrow pointing to the "Index: 9b" section. Another handwritten note "address" is written above the diagram with a red arrow pointing to the "Offset: 5b" section. A small image of a man in a pink shirt is visible in the bottom right corner of the slide.

### Cache and Programming

- Objective: Learn how to assess cache related performance issues for important parts of our programs
- Will look at several examples of programs
- Will consider only data cache, assuming separate instruction and data caches
- Data cache configuration:
  - Direct mapped 16 KB write back cache with 32B block size

Diagram components:  
Tag : 18b    Index: 9b    Offset: 5b

Let's move on to an important topic in connection with cache memory, and that is how does our knowledge of cache memory change our perspective on the programs that we write. And I've titled this set of slides- caches and programming, since we are going to look, we understand enough about the cache memories now and the way that they operate to be able to view performance of our programs, if running on particular kinds of cache hardware.

Now, in order to do this; Our objective in doing this, rather, is to for the programs that are, that we write or parts of programs that are important to us, to learn techniques using which we can assess the cache related performance issues. In other words, get some kind of an understanding of how a particular part of a program of ours would execute or would benefit from a particular kind of a cache. That is our objective, so, to put on knowledge of cache together with our understanding of programs, to get some benefit from the programming side. The way I am going to do this is by basically looking at a number of examples of simple programs.

Now, to keep the discussion somewhat simple, but yet realistic, we will not take into account, the behavior of the instructions of our programs. So, when I talk about a program or an important part of a program, we will concentrate primarily on the interaction between the program and memory, as far as it is data accesses are concerned. We could assume for example, that we are running the program on a computer system

which has separate instruction and data caches, and that we will assume that the instruction cache behavior is not of interest to us right now.

We are concentrating only on the data cache. That is essentially, what we will start by doing most of the examples that we will look at. So, we will consider only the data cache in our examples and typical kind of a data cache configuration, I will use in these examples, may be something like this. I will use some variations on this kind of a configuration, but I just wanted to remind you about what we have learned about the 4 q's of cache organization.

So, in this particular quick description of a cache, I mention that the size of the cache is 16 kilobytes. I mention that the size of each block in the cache is 32 bytes. I also mention that the cache is direct mapped, from which you understand that direct mapped block placement is used, which means that for any main memory block there is a unique cache block into which, that main memory block might be copied.

In addition, I am informing you that the cache uses the write-back policy which means that on a write-hit on a write access to the cache, which is a hit in the cache, the main memory is not updated at that point in time, the update is done only for the cache copy. And subsequently, if that particular cache block is replaced, the main memory copy would be updated.

So, these were some of the key words from our understanding of cache organization in the frame work of the 4 q's of cache organization and I will give you similar brief descriptions of the cache, in order to assess the behavior of a particular program. Okay, Now, for this particular cache organization, remember that we were able to use the information in this description to understand how an address generated by the processor would be viewed by the cache hardware, and just to quickly remind you, we had to decide how many bits were going to be used for the block offset, how many bits were going to be used for the cache index and we knew that the remaining bits would be used for the tag.

For this particular example, given that the size of a block is 32 bytes, we immediately know that the offset is going to require 5 bits, because  $\log_2(32)$  is equal to 5 from the fact that, its direct mapped with the 16 kilobyte cache. You were able to calculate

that 16 kilobytes is the size of the cache and the size of each cache block is 32 bytes from which, we know that the number of cache blocks is 512, 16 k divided by 32 and log base 2 of 512 is the 9 bits that are needed for index.

(Refer Slide Time: 08:44)

**Example 1: Vector Sum Reduction**

```
88 double A[2048], sum=0.0;
for (i=0; i<2048; i++) sum = sum + A[i];
```

- To do analysis, must view program close to machine code form (to see loads/stores)
- Recall from static instruction scheduling examples how loop index  $i$  was implemented in a register and not load/stored inside loop
- Will assume that both loop index  $i$  and variable  $sum$  are implemented in registers
- Will consider only accesses to array elements

So, we were able to calculate the number of bits for index, the number of bits for offset and assume that the remaining bits are used for the tag and therefore, this is the form in which this information was used in our analysis of, in our description about how the cache organization went about. Now, we are going to start with a very simple example of a code fragment. Now, the particular example I will start with, I describe as “vector sum reduction” and from the description, you will be able to understand that this seems to be an operation or a program which is doing something on a vector of data, and it seems to be doing a sum reduction on the vector of data.

The word reduction suggests that there are some reductions or lessening in the amount of data, and the word sum suggests that, the way that the reduction is done is by adding the values of all the elements within the vector and that is exactly what this simple program is going to be doing. So, it is as shown by this c implementation.

So, in this example I have a vector which is implemented as a floating point array  $A$ .

The size of the vector is 2048 which means that the array elements will be referred to as  $A$  of 0 up to  $A$  of 2047 and in addition, I have indicated that each of the array elements is

of type double. Remember that a double float is a float which is not 32 bits in size, but which is 64 bits in size which means that the size of each array element is 8 bytes and this particular code segment, what is being done is, for  $i$  equal to 0  $i$  less than 2048  $i$  plus plus. The elements of the array of the vector  $A$  are being added to a running sum which has been initialized to 0. So that, at the end of the execution of this for loop, the variable sum will contain the sum of all the vector elements, which is why it is called a sum reduction.

At the end of this, a small piece of code, the variable sum contains the sum of all the vector elements  $A$  of 0 through  $A$  of 2048. So, that is a nice simple code example. It is of course not a complete program, but if we can understand this particular program from the perspective of our cache behavior, and if this is a very important part of a program of ours, then we probably have a good understanding of the overall behavior of the program itself, a larger program.

Now, recall that I indicated that, we are not going to concern ourselves with the instructions corresponding to this program. We are going to concentrate on the data accesses generated by this program, but in order to do the analysis, it will probably be easier for us if we can view the program not in this C form, but in a machine form, in a machine code form. So that, we can actually see the individual loads and stores.

In other words, we can identify what the different memory accesses are. Remember, we are going to have to view this program in terms of the loads and stores that it sends to the cache memory, and exactly what address is associated with each of the loads and the stores. Only then will we be able to talk about how the cache memory will perform for a particular program.

(Refer Slide Time: 08:44)

The slide is titled "Example 1: Vector Sum Reduction" in a red-bordered box. Below the title, the C code is shown: `double A[2048], sum=0.0;` and `for (i=0; i<2048; i++) sum = sum + A[i];`. Handwritten red annotations include "88" with an arrow pointing to the loop, "A[A] ... A[2048]" above the loop, and a circle around the `sum + A[i]` expression. Below the code, there are four bullet points: "To do analysis, must view program close to machine code form (to see loads/stores)", "Recall from static instruction scheduling examples how loop index i was implemented in a register and not load/stored inside loop", "Will assume that both loop index i and variable sum are implemented in registers", and "Will consider only accesses to array elements". A small inset image of a man in a pink shirt is visible in the bottom right corner of the slide.

So, we need to move down from understanding vector sum reduction in C to understanding the vector sum reduction in terms of the load and store instructions, how many load and store instructions are executed and what are the addresses associated with the operands of the load and store instructions.

Okay, Now, if you remember from our discussion of the simple code examples for loops, that was somewhat similar, we had a loop index and in the examples that I used, the loop index was typically implemented inside a register and therefore, the loop index  $i$  may be not a variable that we have to worry about as far as the cache behavior is concerned. So, if we were to assume that the compiler generates machine language code for this particular loop in such a way, that the loop index  $i$  is handled inside a register, rather than being accessed out of memory each time the variable  $i$  is referred to. The variable  $i$  is referred to several times in each loop iteration. So, if I make the assumption that the loop index is accessed out of a register, then I can ignore the loop variable, the loop index  $i$  completely in my analysis of this program.

In addition, I could assume that the sum variable also is handled outside through a register, and it should be fairly easy to see that a compiler could quite easily do that or if the compiler did not do it, I could modify the program so that, it did that in order to reduce the number of memory accesses that this program makes. So, with these two assumptions, our perspective on vector sum reduction will actually get simplified



substantially. We would no longer worry about the accesses to “sum” or to “i” in this program and what we are left with is, just the references to the array A, to the vector A and in short, that is how we will view this particular program.

We will view this program not in terms of its instructions, because we have already decided that we will be worried only about the data cache behavior of the program. Further, we will completely ignore the loop indices, we will completely ignore many of these variables, in this case, it is only one variable, “sum”, and will be primarily worried about the accesses to the vector A.

Now, if we are primarily worried about the accesses to the vector a, then viewing this program in terms of the machine instructions is very simple. I do not even have to worry think about how this program gets compiled. I very clearly understand that, the first time through the “for” loop, there will be a need to load from A of 0, into a register, because, subsequently the value of the contents of that register will be added to the variable sum, which is also in a register.

(Refer Slide Time: 14:45)

The slide, titled "Example 1: Reference Sequence", contains the following text:

- <sup>0xA000</sup> load A[0] load A[1] load A[2] ... load A[2047]
- Assume base address of A (i.e., address of A[0]) is 0xA000, 1010 0000 0000 0000
- Cache index bits: 100000000 (value = 256)

The slide also features a small inset image of a man in a pink shirt sitting at a desk in the bottom right corner.

Therefore, each time through this loop, the loop corresponding to vector sum reduction, there will be a load instruction which loads an element of the vector into a register. So, in short, in thinking of the vector sum reduction, I can think about the vector sum reduction as being a series of load instructions. The first time through the loop, it is a load of A of

0 second times through the loop. It is the load of A of 1 and so on. So, I view the vector sum reduction as being a sequence of 2048 loads, load of 0 through load of 2047. Remember that these were the, that the size of my vector was 2048 hence the sequence from load of A of 0 to load of 2000 A of 2047.

Now, in order to proceed to trying to think about, what the impact of this program and remember- this is now the program that I am worrying about, this is what I have distilled out of the vector sum reduction loop, this is the essence that we are going to concentrate on. In order to proceed to understand the impact on the cache, I have to be able to calculate the address of each of these vector elements. It is not enough to talk about A of 0, I have to talk about a specific address. Therefore, I will make an assumption about what the address of A of 0 is. So, I will assume that A of 0 has some address in this particular example, I am assuming that the address of A of 0 is hexadecimal A000.

Let me remind you that, if whenever you see a number which has 0 x in front of it, is a hexadecimal number. We have not use hexadecimal for some time now, but this is telling me that the address which is of concern to us is being given to us in hex, which means that base 16 and A is 1010. These are the 3 zeros, if I wanted to view this as a 32 bit address, then I would put additional zeros to make it into the required 32 bit address.

Now, if I am assuming that the address of A of 0 is hex A000, what can I assume about the address of A of 1? What I will assume is that, the way that the compiler lays out the array inside the virtual address space and remember, it is the compiler that decides the order in which different elements are placed in the virtual address space, corresponding to the process that will come in to existence when this program executes, the compiler makes a decision. I will assume that the compiler, if it assumes that the address of A of 0 is hex A000 will put A of 1 into the next memory location, and it will put A of 2 into the memory location following that and so on.

(Refer Slide Time: 17:35)

**Example 1: Reference Sequence**

- load <sup>0xA000</sup> A[0] load A[1] load A[2] ... load A[2047]
- Assume base address of A (i.e., address of A[0]) is 0xA000, 1010 0000 0000 0000
  - Cache index bits: 100000000 (value = 256)

The slide features a presenter in the bottom right corner. Handwritten annotations include a red circle around 'A[1]' and a red arrow pointing to '0xA000'.

(Refer Slide Time: 17:37)

**Cache and Programming** 16KB/32B

- Objective: Learn how to assess cache related performance issues for important parts of our programs
- Will look at several examples of programs
- Will consider only data cache, assuming separate instruction and data caches
- Data cache configuration:
  - Direct mapped 16 KB write back cache with 32B block size

The slide features a presenter in the bottom right corner. A diagram shows the address field divided into three parts: Tag (18b), Index (9b), and Offset (5b). Handwritten annotations include a red circle around '32B', a red arrow pointing to the 'Index' field, and a red line under 'address'.

Tag : 18b	Index: 9b	Offset: 5b
-----------	-----------	------------

In other words, it will put the elements of the vector into contiguous neighboring memory locations. That seems like a reasonable assumption, then I can now view each of these memory addresses from the perspective of how the cache will look at that particular memory address. And let me just go back to our understanding of the cache.

In this particular example, we are going to assume that the cache is direct mapped 16 kilobyte, write through with 32 byte block size, which means that in looking at the address, we have to view the address in this way. The 5 least significant bits of the

address are offset, this is followed by 9 bits of index and then 18 bits of tag, if we are talking about a 32 bit address.

(Refer Slide Time: 18:00)

**Example 1: Vector Sum Reduction**

*88* *A[0] ... A[2047]*

```
double A[2048], sum=0.0;
for (i=0; i<2048; i++) sum = sum + A[i];
```

- To do analysis, must view program close to machine code form (to see loads/stores)
- Recall from static instruction scheduling examples how loop index *i* was implemented in a register and not load/stored inside loop
- Will assume that both loop index *i* and variable *sum* are implemented in registers
- Will consider only accesses to array elements

*(Note: In the original image, 'A[i]' in the code and 'A[1]' in the next slide are circled in red.)*

(Refer Slide Time: 18:02)

**Example 1: Reference Sequence**

- *0xA000* load A[0] load A[1] load A[2] ... load A[2047]
- Assume base address of A (i.e., address of A[0]) is 0xA000, 1010 0000 0000 0000
- Cache index bits: 100000000 (value = 256)

Now, when I think about the address hex A000, I can view it in this light, I can look at the least significant 5 bits, I can look at the next 9 bits and so on.

What happens when I do that?

The least significant 5 bits, remember the least significant 5 bits for the offset. The next 9 bits were the index, and the remaining 18 bits were the tag. So, if I will remove the least significant 5 bits as being the offset, and then I count the next 9 bits 1 2 3 4 5 6 7 8 9, what I come up with is this value, the bit value 1 followed by 8 zeros, and that instead of referring to it as binary 1 followed by 8 zeros, I will refer to it as decimal 256.

You can evaluate this as  $2^1$ ,  $2^2$ , etc.

This is actually decimal  $2^{56}$ . So, in short, for the particular cache organization that we are using, a 16 cache with 32 byte block size under direct mapping, the array element A of 0 will have its index bits equal to 10000000 which is decimal 256. What about array element A of 1? Now, in order to find out the address of array element A of 1, I have to know how much space A of 0 occupies.

(Refer Slide Time: 19:32)

**Example 1: Vector Sum Reduction**

```
double A[2048], sum=0.0;
for (i=0; i<2048; i++) sum = sum + A[i];
```

- To do analysis, must view program close to machine code form (to see loads/stores)
- Recall from static instruction scheduling examples how loop index  $i$  was implemented in a register and not load/stored inside loop
- Will assume that both loop index  $i$  and variable  $sum$  are implemented in registers
- Will consider only accesses to array elements

The slide includes handwritten annotations: '8B' with an arrow pointing to the array declaration, 'A[i] · A[2048]' above the loop, and a red circle around the '+A[i]' in the loop body. A small inset image of a man in a pink shirt is visible in the bottom right corner of the slide.

(Refer Slide Time: 19:39)

The slide is titled "Example 1: Reference Sequence". It contains the following text:

- load A[0] load A[1] load A[2] ... load A[2047]
- Assume base address of A (i.e., address of A[0]) is 0xA000, 1010 0000 0000 0000
- Cache index bits: 100000000 (value = 256)

Handwritten annotations on the slide include "0xA000" in red above "load A[0]", and a red circle around "A[1]".

But we know that the vector sum reduction loop is dealing with double float values from which we understand that, the size of each array element is 8 bytes. So, the size of an array element under double is 8 bytes, which tells me that in any particular cache block, given that the block size is 32 bytes, 4 consecutive array elements will be present.

So, in any one of the cache blocks, let us suppose this is one of the cache blocks of size 32 bytes, we actually have enough space to store four 8 byte doubles. Therefore, if I have A of 0, I could also have A of 1, A of 2, and A of 3, all within the same cache block.

That is what this simple observation allows me to learn. In other words, if A of 0 has address A000 and A of 1 has address A008, which is 8 bytes after the address of A of 0 and so on, and I do the same calculation using its least significant bits, I will find out that A of 0, A of 1, A of 2, and A of 3 are all going to be in the same cache block and will all have that index 256.

In other words, once A of 0 is in the cache, I will know for sure that A of 1, A of 2, and a of 3 are also in the cache because they all occupy, they all come from the same cache block. They just differ in their index bits, if you look at the addresses of A of 0 through A of 3. In fact, this what I have over here. Here, I am showing you the addresses of A of 0, A of 1, A of 2, and A of 3.

Notice that A of 0 is 1 followed by all zeros. We are looking only at the index bits and bits following. A of 1 is going to be 8 bytes later, which means that it will have the address A008. A of 2 is going to have 8 bytes after that, which is going to be 100000 at the last bits and so on and you will notice that they all have the same index bits.

So, with this observation we can project, look forward to the array elements A of 4 through A of 7 and do the same calculation. And we would find out that, given the base address of A of 0, we could calculate the base address of A of 4 and find out that the base address of A of 4 is such that, it will have an index of 257 within the cache, that we were talking about and A of 4 through A of 7 would all have that same property.

In other words the next 4 array elements, the first 4 array elements, all have index of 256, cache index of 256, then the array of the next 4 array elements after that, all have cache index of 257 and so on, and this is entirely because the 4 array elements A of 0 through A of 3, all occupy one cache block, given that the size of a cache block is 32 bytes. Similarly, array elements A of 4 through A of 7, all occupy one cache block and this could proceed, this reasoning could be used to argue all your way through the complete program, in other words from loading of A of 0 all the way through loading of A of 2047.

(Refer Slide Time: 22:53)

**Example 1: Cache Misses and Hits**


A[0]	0xA000	256	Miss	Cold start
A[1]	0xA008	256	Hit	
A[2]	0xA010	256	Hit	
A[3]	0xA018	256	Hit	
A[4]	0xA020	257	Miss	Cold start
A[5]	0xA028	257	Hit	
A[6]	0xA030	257	Hit	
A[7]	0xA038	257	Hit	
..	..	..	..	
..	..	..	..	
A[2044]	0xDFE0	255	Miss	Cold start
A[2045]	0xDFE8	255	Hit	
A[2046]	0xDFF0	255	Hit	
A[2047]	0xDFF8	255	Hit	

Cold start miss: we assume that the cache is initially empty. Also called a Compulsory Miss

Hit ratio of our loop is 75% -- there are 1536 hits out of 2048 memory accesses

This is entirely due to spatial locality of reference.

If the loop was preceded by a loop that accessed all array elements, the hit ratio of our loop could be 100%, 25% temporary due to



But to do this systematically, we will actually come draw a table in which we will try to calculate the number of cache misses and the number of cache hits that would result for that vector sum reduction program,. So, I am going to a draw table of this kind. So, what I have drawn in this table, in this table in the left most column what I am showing you is the program itself.

The program first loads A of 0 then it loads A of 1. This is a program which, the only interesting instructions in this program to us right now are the load instructions. In the program, it first loads A of 0, then it loads A of ,1 then it loads A of 2 and keeps on going until it loads A of 2047.

Now, we know that the base address of the array, which is the address of A of 0, we assumed was hex A000 and given that the size of each array element is 8 bytes, we calculated the base, the address of A of 1, the address of A of 2 and so on. So, we were able to calculate the addresses of each of the array elements using the base address, assumption and our knowledge about the size of each array element.

The size of each array element is decided by the type declaration used in the program. In our particular program, each array element is of type double which means that occupies 8 bytes, which is why there is an 8 byte difference between each of these addresses and then, knowing what the cache hardware does with an address for the particular cache that we are concerned about, I can understand what the index bits for each of these addresses is and as we had said, the first 4 array elements have index of 256, the next 4 array elements have index of 257, and we can keep on doing this and in doing so, we will find out that the last 4 array elements have index bits of 255.

Now, the next question is, how well is the program going to do on hardware which has a cache of the kind that we had described? In other words, how many of these accesses are going to be hits and how many of these accesses are going to be misses within the cache? Now, the way to think about this is, we could start by assuming that, when the program starts executing, none of the array elements will be present inside the cache and therefore, the when the program executes its first load instruction, load A of 0, it is going to encounter a miss. Nothing would have come into the cache since nothing had been requested by the program before, as far as the address space of this process is concerned and therefore, we would guess that the first access is going to be a miss.



However, as a result of that miss on A of 0, there is going to be a fetching of a block from the main memory into the cache and that block is a 32 byte block which contains not only A of 0, but also A of 1, A of 2, and A of 3. Hence, when the second load instruction is executed to load A of 1, we have guessed that the load instruction of A of 1 is going to benefit a cache hit, as will the load of A of 2, as will the load of A of 3 and we can continue this kind of thinking for all the addresses in this table and what we will end up with is something like this.

Let me just get read of this unnecessary notation here. So, what we see here something like this. So, once again, this is the same table we had before. We had ,let me just go back and make sure, yeah it is exactly the same table that we had before. So, this is the column with the array elements, this is the column with, they are addresses based on the assumption, the array of 0 is that address A000. These were the index, or the cache indices which we calculated from that and this was the reasoning that we came up with.

The first access to array element A of 0 would be a miss because the program, initially, when it started executing, would not have had anything fetched into the cache by the hardware.

(Refer Slide Time: 22:53)

**Example 1: Cache Misses and Hits**

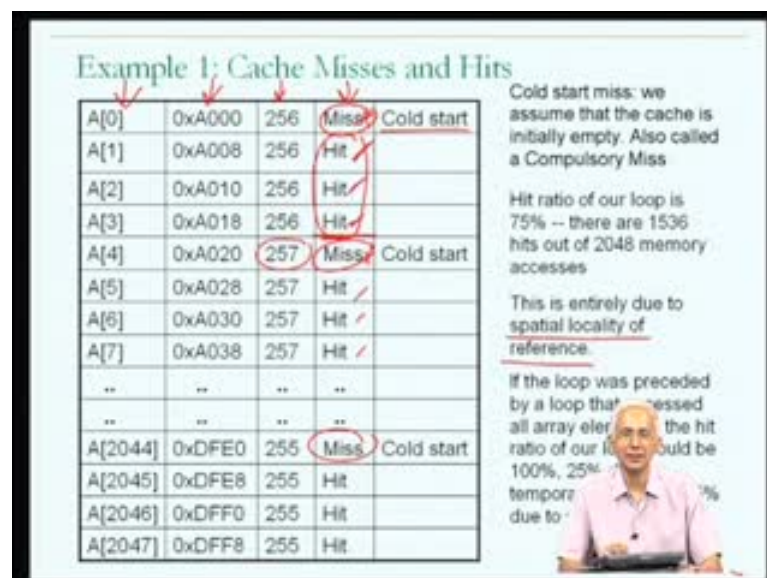
A[0]	0xA000	256	Miss	Cold start
A[1]	0xA008	256	Hit	
A[2]	0xA010	256	Hit	
A[3]	0xA018	256	Hit	
A[4]	0xA020	257	Miss	Cold start
A[5]	0xA028	257	Hit	
A[6]	0xA030	257	Hit	
A[7]	0xA038	257	Hit	
..	..	..	..	
..	..	..	..	
A[2044]	0xDFE0	255	Miss	Cold start
A[2045]	0xDFE8	255	Hit	
A[2046]	0xDFF0	255	Hit	
A[2047]	0xDFF8	255	Hit	

Cold start miss: we assume that the cache is initially empty. Also called a Compulsory Miss

Hit ratio of our loop is 75% -- there are 1536 hits out of 2048 memory accesses

This is entirely due to spatial locality of reference.

If the loop was preceded by a loop that accessed all array elements, the hit ratio of our loop would be 100%, 25% temporal locality due to



But once A of 0 was a miss, the cache block which contains A of 0 will also contain A of 1, A of 2, and A of 3 and hence the accesses to load A of 1, to load A of 2, and to load A

of 3 would all be hits. After this, when the time comes to fetch A of 4, once again, it is a different cache index, the cache hardware would find out that there is a miss, but the subsequent four accesses to A of 5, A of 6, and A of 7 would be hits and so on.

Now, in terms of terminology, people often talk about that first miss, which happens because your program has not previously accessed the particular cache block. It is often referred to as a cold start miss. Since when your program starts executing, we assume that the cache is initially empty and it is compulsory, there is no way to avoid this miss and its starting because the cache started off empty.

So, this particular kind of a miss might be described as a cold start miss, its coming because the program initially started executing in a situation where the cache did not contain any of its data. Subsequently we have the hits. Now, we can go ahead and calculate the hit ratio for this particular program for the particular cache that we are talking about, and we find out that the hit ratio is in fact, 75 percent.

How do we come up with this? We know that there are total of 2047 accesses of which, every fourth access is a miss and the remaining three fourths of the accesses are hits, which tells us that there will be a hit ratio of 75 percent. So, 75 percent of all the memory references which this program, a 75 percent of the accesses which this program makes to memory will actually be satisfied directly by the cache, which we had talked about. More specifically, out of 2048 memory accesses that this program makes, 1536 will be hits, and if we think about it a little bit, all of these 1536 hits are happening because of spatial locality of reference. Why do I say it is happening because a spatial locality of reference? It is happening because; just think about the 3 hits that we had over here.

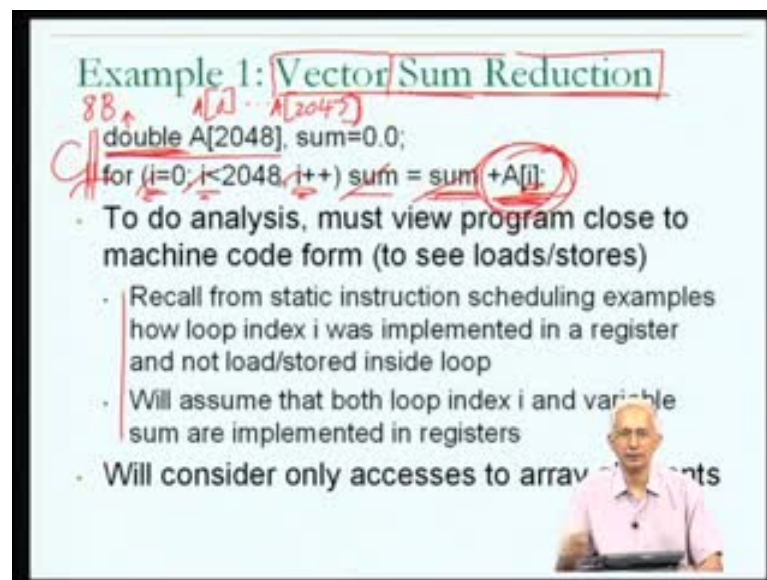
Why was the reference to A of 1 a hit?

It was a hit because of the fact that, the cache hardware is designed in terms of blocks, in which any one of the blocks contain 4 neighboring elements of the vector A, spatial locality is automatically being exploited, I get a miss on A of 0 and I have to- the hardware has to get the block from memory into the cache, but the benefit that one gets is the neighbors of A of A of 0 are immediately present out of the cache, and that was what we talked of a spatial locality of reference.

This particular program is showing very good spatial locality of reference and that, after A of 0 was referenced, its three neighbors A of 1, A of 2, and A of 3, were in fact, referenced in the immediate future and if we had a larger block size, we would have got more and more benefit out of the spatial locality of reference.

This particular cache has a block size of only 32 bytes, which is why we got the spatial locality of reference benefit for only the next 3 elements. Now, other observations which we could make, this is an interesting observation- let me just read it out to you. Let us imagine that we had written our program in the following ways: If the loop that we had just seen was preceded by a loop that accessed all the array elements, then the hit ratio of our loop could be 100 percent.

(Refer Slide Time: 30:46)



The slide is titled "Example 1: Vector Sum Reduction" and contains the following content:

```
88, A[1], A[2043]
double A[2048], sum=0.0;
for (i=0; i<2048; i++) sum = sum + A[i];
```

- To do analysis, must view program close to machine code form (to see loads/stores)
- Recall from static instruction scheduling examples how loop index i was implemented in a register and not load/stored inside loop
- Will assume that both loop index i and variable sum are implemented in registers
- Will consider only accesses to array elements

A small inset image of a man in a pink shirt is visible in the bottom right corner of the slide.

Let us just think about this a little bit, so, the idea that I am working with here is, we had this vector sum reduction and if we are now considering this loop, and this loop- we now understand, for the cache that we are concerned about, we know that for this particular loop, the hit ratio was 75 percent because A of 0 suffered a miss, but A of 1, 2 and 3 suffered hits giving a 75 percent hit ratio.

The current suggestion is that, knowing this, I could actually have included another loop over here, in which all that I do is, for each of the array elements, I just access A of i. So, this could be something like s equals A of i, just an access to A of i.

Now, what is the purpose for doing this?

What I am doing is, I am preceding the loop, this is the loop that I am interested in. I am preceding the loop that I am interested in by another loop and all that this loop is doing is, this accessing each of the array elements. What is the purpose of that loop? The purpose of that loop is actually to cause the contents of the entire array to be present in the cache. So, when the loop that I have just written executes, it is going to cause A of 0 and A of 1 and A of 2 and A of 3 to come into the cache.

As a consequence, when the loop that I am interested in starts executing, the entire vector A will be present in the cache and therefore, rather than executing at a hit ratio of 75 percent, the loop that I am interested in will execute at a hit ratio of 100 percent. It will get 100 percent cache hits. Every single access to the array during this loop will be a hit.

What is the price that I am paying?

The price that I am paying is, I am executing additional instructions, I am executing an additional 2048 load instructions, and of those 2048 load instructions, how many of them will be hits and how many of them will be misses?

(Refer Slide Time: 33:03)

**Example 1: Cache Misses and Hits**


A[0]	0xA000	256	Miss	Cold start
A[1]	0xA008	256	Hit	
A[2]	0xA010	256	Hit	
A[3]	0xA018	256	Hit	
A[4]	0xA020	257	Miss	Cold start
A[5]	0xA028	257	Hit	
A[6]	0xA030	257	Hit	
A[7]	0xA038	257	Hit	
..	..	..	..	
..	..	..	..	
A[2044]	0xDFE0	255	Miss	Cold start
A[2045]	0xDFE8	255	Hit	
A[2046]	0xDFF0	255	Hit	
A[2047]	0xDFF8	255	Hit	

Cold start miss: we assume that the cache is initially empty. Also called a Compulsory Miss

Hit ratio of our loop is 75% -- there are 1536 hits out of 2048 memory accesses

This is entirely due to spatial locality of reference.

If the loop was preceded by a loop that accessed all array elements, the hit ratio of our loop could be 100%. 25% temporal locality due to



If I actually do 2048, then I will get 75 percent hit ratio on that loop, but the overall hit ratio of the entire program will be more than what I had originally, in other words more

than 75 percent and that is therefore, an interesting idea. Let me just put down what we have over here. The observation is, that if I had preceded the loop that I am concerned about by a loop which essentially just accessed all the array elements, thereby causing them to be present in the cache when my vector sum reduction loop executed, then my vector sum reduction loop would execute with 100 percent hit ratio and in this particular case, I could not say that all of the hits are because of spatial locality of reference. I would be able to say, the 25 percent of the hits are due to temporal locality of reference.

Why temporal locality of reference? Because the previous loop which I had executed had accessed A of 0. Therefore, a little later on in time, when my vector sum reduction loop executed and accessed A of 0, because of temporal locality of reference, it found A of 0 within the cache and therefore, it benefited from a hit. Therefore the 100 percent hit ratio that my vector sum reduction loop benefits from, will be 25 percent due to temporal locality, and only 75 percent due to spatial locality.

Of course, the overall behavior of the program is not improved because I have caused more instructions to be executed and therefore, the program will take more time. In fact, the program will suffer as many misses in other words as many main memory accesses as it did, without the preceding loop. Therefore, this is just for the purpose of understanding that it is possible to exploit both temporal and spatial locality of reference and that your program could be doing one or the other and what is actually important is to understand how your program is behaving. Now, let us understand little bit more about what is happening by looking at a small variant of example one. Now, you will recall that in example one, we did vector sum reduction on a vector, a double vector of size 2048. Now, I am going to modify that.

I am going to consider vector sum reduction with vector of size 4096. So, this is a vector which is double the size of the previous vector. Now, what will happen for this particular, for the same cache that we have talked about in the previous example, what will happen for this particular cache? And the first of all why should it make a difference? Now, the reason that we should understand that it makes a difference is that, think about the previous example that we had where the size of the vector was 2048.

What was the actual size of the vector?

The vector contains 2048 elements, and the size of each element was 8 bytes which means that the total size of the vector A was 2048 multiplied by 8 and if you do this calculation, you will see that it is of size 16 kilobytes, and what is important about 16 kilobytes? 16 kilobytes is the size of the cache that we are currently dealing with. In other words, when I had vector sum reduction on an array of size 2048 elements, the entire array could fit into the cache. On the other hand, if I deal with vector sum reduction on in array of size 4096 double precision floats, then the size of the vector is now going to be twice the size of the cache. In other words, this vector will not fit into the cache completely. The vector is larger than the cache.

Therefore, if I had considered preceding my vector sum reduction loop by loop, which just accessed each of the array elements and the array was of size four 4096, then since there, if entire array no longer fits into the cache, there is no benefit that I will get in the case of the 4096 example and I will still suffer the misses. In other words, just preceding the loop of interest to me, the vector sum reduction loop, by a loop which accesses each of the 4096 elements could not help at all. It will just make the program slower. I will still get only 75 percent hit ratio and this is entirely because this vector itself does not fit into the cache.

(Refer Slide Time: 33:03)

**Example 1: Cache Misses and Hits**


A[0]	0xA000	256	Miss	Cold start
A[1]	0xA008	256	Hit	
A[2]	0xA010	256	Hit	
A[3]	0xA018	256	Hit	
A[4]	0xA020	257	Miss	Cold start
A[5]	0xA028	257	Hit	
A[6]	0xA030	257	Hit	
A[7]	0xA038	257	Hit	
..	..	..	..	
..	..	..	..	
A[2044]	0xDFE0	255	Miss	Cold start
A[2045]	0xDFE8	255	Hit	
A[2046]	0xDFF0	255	Hit	
A[2047]	0xDFF8	255	Hit	

Cold start miss: we assume that the cache is initially empty. Also called a Compulsory Miss

Hit ratio of our loop is 75% -- there are 1536 hits out of 2048 memory accesses

This is entirely due to spatial locality of reference.

If the loop was preceded by a loop that accessed all array elements, the hit ratio of our loop could be 100%. 25% temporal locality due to ..



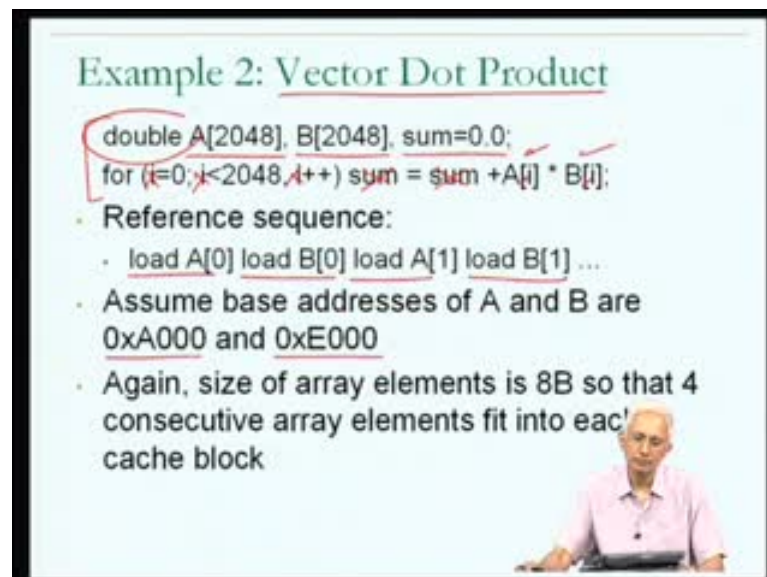
So, after execution of the preceding loop, the second half of the array will be in the cache and our loop sees misses as we just saw. So, there will be no benefit from preceding the

loop by a loop, which accesses each array element as we saw in the case of the smaller vector.

We, in the case of the smaller vector, by having that is the small loop before the vector sum reduction loop, I could artificially give the vector sum reduction loop hit ratio of 100 percent, but that is not going to happen if I have a larger vector in which case, I will get only the hit ratio of 75 percent, even for the vector sum reduction loop.

Now, the kinds of misses that cannot be avoided in this case are what are called capacity misses. We could not avoid the capacity misses for the larger vector because of the capacity problem. The vector is bigger than the capacity of the cache and there is no avoiding those misses whereas, for the smaller vector, since the vector was smaller than the size of the cache, I could avoid some of the capacity misses. Because, that the cache was capacious enough or large enough to accommodate the entire vector and I could therefore, preload in some sense, the vector into the cache by using a preceding loop which accesses to each of the array elements- vector elements.

(Refer Slide Time: 38:31)



**Example 2: Vector Dot Product**

```
double A[2048], B[2048], sum=0.0;
for (i=0; i<2048; i++) sum = sum + A[i] * B[i];
```

- Reference sequence:
  - load A[0] load B[0] load A[1] load B[1] ...
- Assume base addresses of A and B are 0xA000 and 0xE000
- Again, size of array elements is 8B so that 4 consecutive array elements fit into each cache block

*(Note: In the original image, red annotations highlight the array declarations and the loop index increment in the code, and underline the reference sequence and base addresses in the list.)*

So, we can now move to another example. I am going to move to an example, which is a little bit different from the previous example that we just saw- when we are going to proceed along very similar lines. Once again, we are going to look at a c version of an operation. We are going to understand what the operation is. You will ignore all the

instructions. We will ignore all the variables that we can like loop indices and so on and concentrate on the critical elements.

The critical data elements accessed by the piece of code. Now the second example we are going to look at is something called vector dot product. So, this is something when if you had a word of it, is the multiplication of two vectors resulting in a value. So, the scale of value. So, in this particular example then, I need to have in the c code. I have a dot product of a vector A of size 2048 with a vector B also sized 2048 and the dot product is going to end up in a variable called sum.

So, what the dot product does is, it multiplies A of 0 by B of 0, multiplies A of 1 by B of 1, multiplies A of 2 by B of 2 and so on and adds all of these products together into the running sum constituted by the variable sum. We are going to use the same size of cache. The same cache organization that we had used in the vector sum reduction example. So, once again, we have to reduce this c program into a version that we can understand the cache from.

In other words, we have to reduce into a sequence of loads and stores and then we have to make some assumption about the address of the different, the base address of A, the base address of B and then we can actually calculate the hit ratio for this particular program. So, instead of actually running through all the reasoning from the previous example, once again we can immediately shortcut to a description of what the references that will result from this program are. Once again, we will ignore any reference to the variable i because, the variable i can be is stored in a register and therefore, need not show itself to the cache memory, need not result in a load or store instruction.

So, any reference to i can be ignored right then. i need not be loaded. Similarly, I will assume that the sum is maintained inside a register and therefore, all that we are left with is the loading A of 0, loading B of 0, then second iteration- loading a of one, loading b of one and so on. So, therefore, we have reduced the vector dot product program to the following reference sequence. So, these are the memory references that the program would make, under the assumption that we have had to make initially. The first reference is to load A of 0, then there is an instruction to load B of 0.



Then the next iteration of the loop- load A of 1, load B of one and so on. Therefore, it will end with load B, load A of 2047, load B of 2047. Now just, as in the case of my vector sum reduction, I need to make some assumption about the base the addresses of the vectors A and B and therefore, I will just assume that the starting address of the vector A is A000 in hex and I will assume that the starting address of vector B is E000, once again in hex. Once you will observe that, we are still dealing with double precision array elements and therefore, just as in the case of A vector sum reduction, each cache block will be large, is large enough to hold for consecutive array elements.

In other words, A of 0, A of 1, A of 2 and A of 3 will all be stored in one array element and one cache block. Similarly, for B, B of 0, B of 1, B of 2, B of 3 will all fit into one cache block. Subsequently B of 4, B of 5, B of 6, B of 7 will be in the other cache block.

(Refer Slide Time: 42:23)

**Example 2: Cache Hits and Misses**

A[0]	0xA000	256	Miss	Cold start
B[0]	0xE000	256	Miss	Cold start
A[1]	0xA008	256	Miss	Conflict
B[1]	0xE008	256	Miss	Conflict
A[2]	0xA010	256	Miss	Conflict
B[2]	0xE010	256	Miss	Conflict
A[3]	0xA018	256	Miss	Conflict
B[3]	0xE018	256	Miss	Conflict
..	..	..	..	..
..	..	..	..	..
B[2023]	0xFFFF	511	Miss	Conflict

Conflict miss: a miss due to conflicts in cache block requirements from memory accesses of the same program

Hit ratio for our program: 0%

Source of the problem: the elements of arrays A and B accessed in order have the same cache index

Hit ratio would be better if the base address of B is such that these cache indices differ

*Index*

*(204)*

118

So, we could run through the same mechanisms that we used in the vector sum reduction and we would end up with a table that looks something like this- let me just run through each of the columns and rows of this table. So, we understand what is happening. So, in the first column- we have the program itself. Remember the program loads A of 0, then it loads B of 0, then it loads A of 1, then it loads B of 1 and so on.

It ends with loading. This should go up to 2040 odd. So, it ends up loading until the last array element. So, there is a mistake over here. Now, in the next column, we have the

address is corresponding to, I will just bring your attention to the fact that the arrays of size 2048. In this particular table, I have gone only up to be A 1023. We should have gone up to B of 2047 and the address will be different, but regardless of that. So, the base address of A of 0, we are assuming is A000. The base address of B of 0 is E000. We know that A of 1 is going to have an address, which is 8 bytes after the address of A of 0.

Similarly, B of 1 and so on. So, we can understand column, the second column. Based on our understanding of the second column of vector sum reduction, once again using the same principle as far as the interpretation of the bits is concerned. We calculate that, the base, the index, this is the index, cache index associated with A of 0 is to for 256. If you do the same calculation for B of 0, you will notice that the index is exactly the same and as we had seen A of 1, A of 2 and A of 3 are going to have the same index as A of 0. Since they come from the same cache block. So, this is the situation that we will end up with, in terms of the assumption that the base address of A of 0 is A000, and the base address of B of 0 is E000.

So, what we have now is the situation where, if we assume that the program starts in the cold state, then as before, we have argue that the access to load A of 0 is going be a miss because it is the cold start. What about the next reference which is to B of 0. Now B of 0 is going to be a miss, because we know that, A of 0 and B of 0 have the same cache index, which means that in this direct mapped cache, only one of them can be in cache at a time. Since my first reference would have brought A of 0 and its block into the cache, we can be sure that the second reference to B of 0 will be a miss and we could think of this as a cold start miss in the sense that, we have not referenced B of 0 before and therefore, even if the program had been on the other hand, if the program had been running a little bit as running instructions prior to this, B of 0 could have been in the cache

But this is happening because the cache starts of empty. Now, when we move on to A of 1, let me just back of little bit. The reference to B of 0 is a miss, and we are arguing that we could describe it as a cold start miss. But as a consequence of B of 0 being a miss, the block containing B of 0 is going to be fetched from main memory into the cache. Therefore, the block containing A of 0 would have been removed from the cache, and the block containing B of 0 occupies that particular direct mapped cache block. Therefore, when the reference, the next reference to A of one happens, it will find out that it is a

miss because, the block containing B of 0 is currently in the particular cache block, which should have been occupied by the block containing A of 1 and we refer to this as a conflict miss.

The conflict miss is a situation where, because of the accesses made by my program, a miss has resulted. If my program had not referenced B of 0, between the reference to A of 0 and A of 1, then this would not have been a miss. It would, in fact, have been a hit. Because of spatial locality of reference, as we saw in the case of the vector sum reduction loop. Therefore, this is the miss that we encounter as a result of A of 1, is a different kind of a miss, from the cold start situation. It is a miss which is, because of the way that the program has been written and it is therefore, called the conflict miss. It is a miss due to conflicts in the cache block requirements from the memory accesses of my program of the same program. As we run through the sequence of events, unfortunately we find out that all of the references made by this program will be misses.

(Refer Slide Time: 42:23)

**Example 2: Cache Hits and Misses**

A[0]	0xA000	256	Miss	Cold start
B[0]	0xE000	256	Miss	Cold start
A[1]	0xA008	256	Miss	Conflict
B[1]	0xE008	256	Miss	Conflict
A[2]	0xA010	256	Miss	Conflict
B[2]	0xE010	256	Miss	Conflict
A[3]	0xA018	256	Miss	Conflict
B[3]	0xE018	256	Miss	Conflict
..	..	..	..	
..	..	..	..	
B[1023]	0xFFFF	511	Miss	Conflict

Conflict miss: a miss due to conflicts in cache block requirements from memory accesses of the same program

Hit ratio for our program: 0%

Source of the problem: the elements of arrays A and B accessed in order have the same cache index

Hit ratio would be better if the base address of B is such that these cache indices differ

118

There is not a single access made by this program in the sequence among the references that we are concerned about. The references to the array A and the array B, vector A and the vector B, that is going to be a hit with this particular cache. In short, we are going to see a hit ratio of 0 percent. This program is going to suffer from a hit ratio of 0 percent. Every single access to array A or array B is going to have to be satisfied out of memory. It will involve fetching the block from memory into the cache and reading the array

element out of the cache. But that is going to happen for every single access made by this program. The program suffers a hit ratio of 0 percent or miss ratio of 100 percent. What is the source of the problem over here? The source of the problem here is exactly the same problem that we had identified with direct mapped caches. In general, this is a situation where I am accessing A of 0 and then B of 0 and A of B of 0 conflicts with A of 0.

It actually wants to occupy the same cache block because, A of 0 and B of 0 have the same index. The 2 different entities which have the same cache index, both wanting to occupy the same cache block under the direct mapping scheme. So, the problem that we had identified with direct mapping is hitting us here. We have a situation where the elements of the array A and the array B, even though they are accessed in order, are being accessed with the same cache index. They are conflicting with each other and therefore, our direct mapped cache is suffering from it, is the worst possible situation.

Now, one simple observation which we could make here is that, the hit ratio could have been made better or the hit ratio would have been better if the base address of B had been selected such that, it differs enough from the base address of A. So, that they have different index bits. We will note that, we had made the assumption that the base address of A was A000 and that the base address of B was E000 and that is where we came up with. This conflict from the fact that both A of 0 and B of 0 had in index the cache index of 256.

Both of these happen because of the assumptions about the base addresses. Remember the base address decides the index, the cache index because, it is bits from the base address which would be used by the cache hardware to decide where to look into the cache under the direct mapped scheme of things. Therefore, this is the sort of bases for trying to improve the quality of our program. If we can somehow come up with the mechanism by which we can cause the base address of B, in other words the address of B of 0 to be different enough from the base address of A,

(Refer Slide Time: 50:00)

**Example 2 with Packing**

*A: 0xA000 / 256*  
*B: 0xE000 / 257*

- Assume that compiler assigns addresses as variables are encountered in declarations
- To shift base address of B enough to make cache index of B[0] different from that of A[0]

`double A(2052), B(2048);`

- Base address of B is now 0xE020
  - 0xE020 is 1110 0000 0010 0000
  - Cache index of B[0] is 257; B[0] and A[0] do not conflict for the same cache block
- Hit ratio of our loop would then be 75%

So that, they do not have the same index bits, then we can improve the hit ratio for this program to, the question becomes how can we do that? I am going to describe one idea which I will refer to as- packing by which, a programmer can get some kind of control over the base address of something like B of 0. Now, the base assumption. The assumption that we are going to make in coming up with this scheme is, we are going to assume we know that it is the compiler that assigns virtual addresses to the different variables in our program, like the vector A, the vector B and so on. I am going to assume that the compiler assigns addresses, virtual addresses to the different variables as an when it encounters the declaration of that variable. Therefore, it assigns the address hex A000 when it comes across double 2048. Next it encounters double B 2048 and therefore, it assign the address hex E00.

Now, if I work with this information, and I want to cause the base address of B to be different then, I could actually just cause the base address of B to be shifted, may be by doing something artificial like this. What have I done? What I have actually done is, I have instead of declaring A to be of size 2048, I had declared A to be of size 2052, B once again is of size 2048. What is the impact of declaring A to be of size 2052? The impact is going to be that, if the compiler was going to assign the base address of A is hex A000, and under the old scheme of things, it was going to assign the base address of B to be E000. With this new declaration of A, I have made the size of the vector A bigger by 1 cache block and therefore, if the compiler just goes ahead and uses the next

address for the base address of B, then by default the base address of B would be hex A00.

But it will be such that, the index will be one more than the index would have been with E000. In other words, if the index associated with hex A000 is 256, then by shifting the base address of B by 1, cache block by artificially making A 1 cache block bigger. In other words, 4 array elements bigger. I will be shifting the index address of B to 257 rather than the 256 that it was before. Therefore by artificially increasing the size of A, I have apparently caused the compiler to give an index for B, which is not going to conflict with the index of A. The base address of B will now be, yeah this is the number hex E020. In other words, it is going to have an index of 257. If you do the calculation of the index, take the hex address E020 and break it up into offset index and tag. You will find out that, instead of having the value 256, it has the value 257 in its intermediate bits. The bits from the not least significant 5 bits for the next 9 bits.

So, that was the, that is apparently going to be the impact of changing the declaration of A in this small way. Now, I am not saying that the vector, the vector A is being made into a larger vector. All that we are trying to do here is, to artificially cause the compiler to give us a more favorable address for the vector B and I could have achieved the same effect using a slightly different mechanism. For example, I could have kept the size of the vector A at 2048. But between the declaration for A and B, I could have included, let us say 4 other double variables- x y z and w. All double variables, the effect could have been that, each of them would have been assigned in address, which was after the address of A.

So, x would have an address of E000. Why would have an address of E008 and so on. And the net effect would have been that, B would have benefited it the same way to have an index value different from that of a right.

(Refer Slide Time: 50:00)

**Example 2 with Packing** *A: 0xA000 / 256*  
*B: 0xE000 / 257*

- Assume that compiler assigns addresses as variables are encountered in declarations
- To shift base address of B enough to make cache index of B[0] different from that of A[0]  
*double A(2052), B(2048);* *257*
- Base address of B is now **0xE020**
  - 0xE020 is 1110 0000 0010 0000
  - Cache index of B[0] is 257; B[0] and A[0] do not conflict for the same cache block
- Hit ratio of our loop would then be **75%**

So, the net effect now is going to be that, if I run the same program, but just making this one change, by causing the base address of vector B to be shifted by one cache block, then the performance of the program is going to be different and you could actually calculate by filling out the table. Once again you will find out that, I will get a hit a miss on A of 0 a miss on B of 0.

But I will get a hit on A of 1, a hit on B of 1, a hit on B of 2 because of spatial locality of reference and once again I will be back at a 75 percent hit ratio, which is what I could get in the case of the vector sum reduction. Therefore, this very small change in the program changes the program from one which had a 0 percent hit ratio to program which had a 75 percent hit ratio and this is something under programmer control and so, with this example.

We have seen 2 examples of how some understanding of cache and some understanding of how the compiler works can be used to understand the way that our program behaves in connection with the cache. In the lectures that follow, we will look at a few more examples. I have stop here today. Just reminding you that in this lecture, we have seen 2 examples of analysis of the behavior of a simple program. The programs have been very simple. Just doing operations on vectors, simple loops doing operations on vectors. This allows to do a somewhat thorough analysis of the impact of the cache on the behavior of the program and in the second program we found that, by understanding what was

happening we could quite easily find mechanisms through which we could substantially improve the behavior of the program from a 0 percent hit ratio to 75 percent hit ratio and we proceed to look at four or five more examples in the next lecture.

Thank you