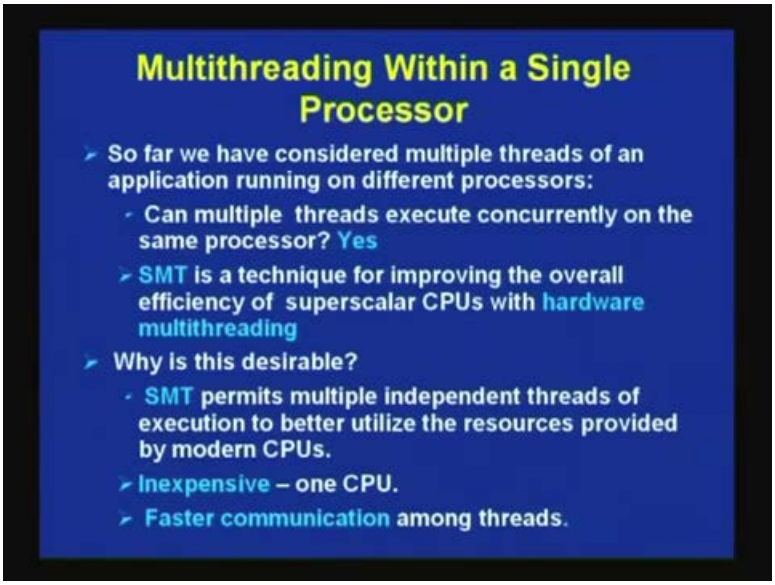
**High Performance Computer Architecture**
**Prof. Ajit Pal**
**Department of Computer Science and Engineering**
**Indian institute of Technology, Kharagpur**

**Lecture - 38**
**Simultaneous Multithreading**

Hello and welcome to today's lecture on Simultaneous Multithreading, this is also known as hardware multithreading.

(Refer Slide Time: 00:45)



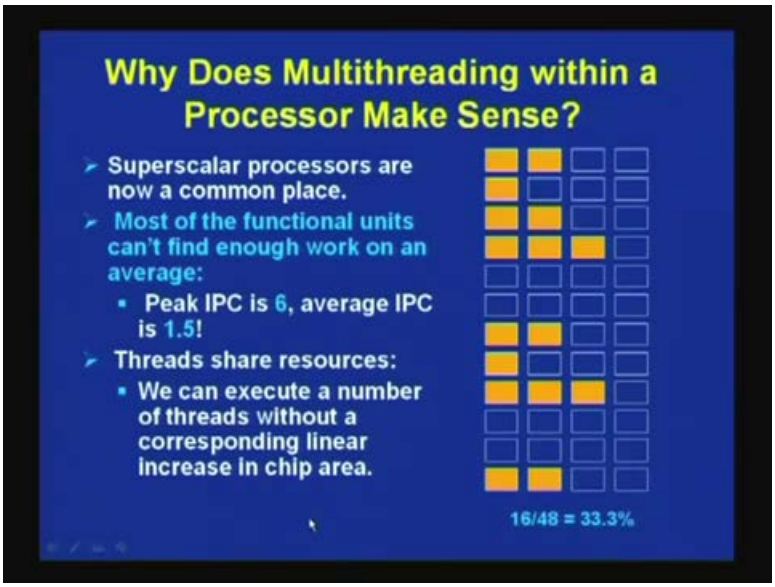

And earlier, so far we have considered multiple threads of an application running on different processors, we have in the last lecture I have discussed at length about this multithreading. And particularly in a context of multiple processors or multi cores, multi core architecture, and we have seen how it can be done, what are the advantages, so various aspects we have discussed in detail in the last lecture.

So, today we shall focus on something else, the question is can multiple threads execute concurrently on the same processor, so we have seen for multiple processors or multi core architectures, multithreading comes naturally. And with the help of that, the utilization reprocesses or cores is improved, but here we are posing a different question, we have a single processor is multithreading relevant in single processor architecture. Or does it help in any way in improving the performance, so all these questions will be answered in this lecture.

So, to tell I mean in a sentence SMT is a technique, for improving the overall efficiency of superscalar CPU's with hardware multithreading. So, this is particularly useful in super scalar architecture, and we shall see how the various resources, the functional units available, and other resources available in superscalar architecture can be used in a efficient manner. Question is why is this desirable, why this multithreading in super scalar architecture is desirable, we shall see that these simultaneous multithreading permits, multiple independent threads of execution of to better utilize the resources provided by modern CPU's.

So, modern CPU's are essentially super scalar having many resources, and that to be truthfully utilized. And particularly this simultaneous multithreading is inexpensive, because you do not require multiple processor, just one processor can be used to implement this multithreading. And this is also desirable from the view point of communication, we have seen that whenever we use multi processing or multithreading there is some communication among the processes or threads. That communication will be faster, whenever all the process, all the threads ran on a single processor. So, as a consequence these simultaneous multithreading has become very useful in the context of super scalar single processors architecture.

(Refer Slide Time: 03:57)




So, here just we can have a relook at the super scalar architecture, and as I mentioned superscalar processors are now a common place, most of the functional units cannot find

enough work of an average, this is one serious limitation of super scalar processors. We have seen that, we have multiple functional units in a single processor, so from the instruction window, that is available with the issue unit, issues different operations for different functional units.

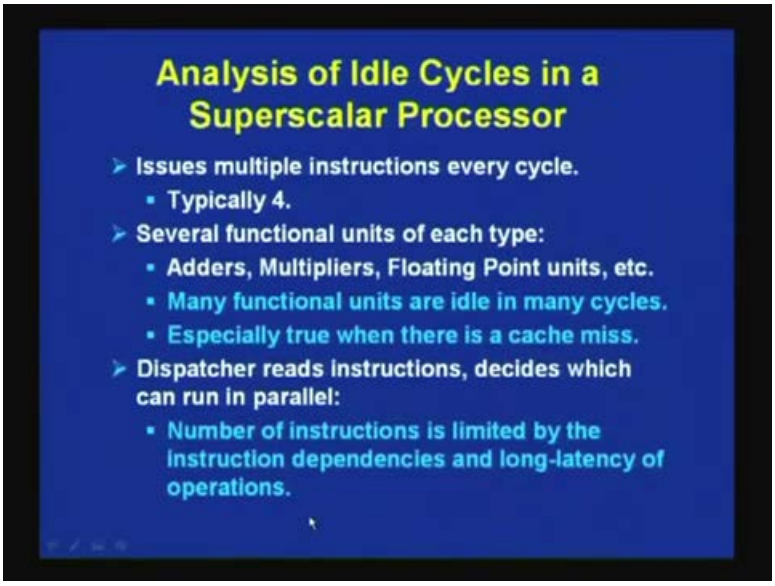
And in most of the cases, the utilization of the functional units is very poor, for example if we have a peak instruction for cycle is 6. So, whenever the peak instruction for cycle is 6 that means, you can have 6 instructions in a single cycles, because of the availability of 6 functional units. But, in reality the average instruction per cycle is only 1.5, so roughly 1 4th, so you see that the incase being just by having super scalar architecture, your utilization is quite poor.

For example, in ((Refer Time: 05:23)) this diagram, we have a super scalar architecture with 4 slots that mean, four functional units, 4 slots may have more functional unit. But, 4 maximum four instructions can be issued, and in this context you can see those yellow rectangles signify that functional units are used, or the other the, with the instructions are issued for those functional units. On the other hand those other rectangles which are empty that means, without this yellow color they are remaining unutilized.

So, you find that not only in a single cycle, several slots are remaining unutilized, some slots are altogether remaining unutilized, so in this particular case we have got altogether 48 different slots. That means, indifferent cycles, considering 4 instructions can be issued 48 instructions can be issued, but unfortunately only 16 instructions are being issued, so utilization is only 33.3 percent, so this shows the limitations of super scalar architecture. And one more point you must notice threads share resources that means, having multiple threads has some advantage, because threads will be sharing resources.

What are the different resources that will be shared, we shall discuss shortly, but since they are sharing resources with small incremental increase in the hardware resources, you can go for multiple threads. That means, the we can execute a number of threads without a corresponding linear increase in chip area that means, the increase in chip area will not linearly increase as you keep on increase the multiple threads. So, with small additional hardware, you can increase in improve the, I mean you can have more number of threads, so that is the reason why multithreading within a single processor make sense.

(Refer Slide Time: 07:50)

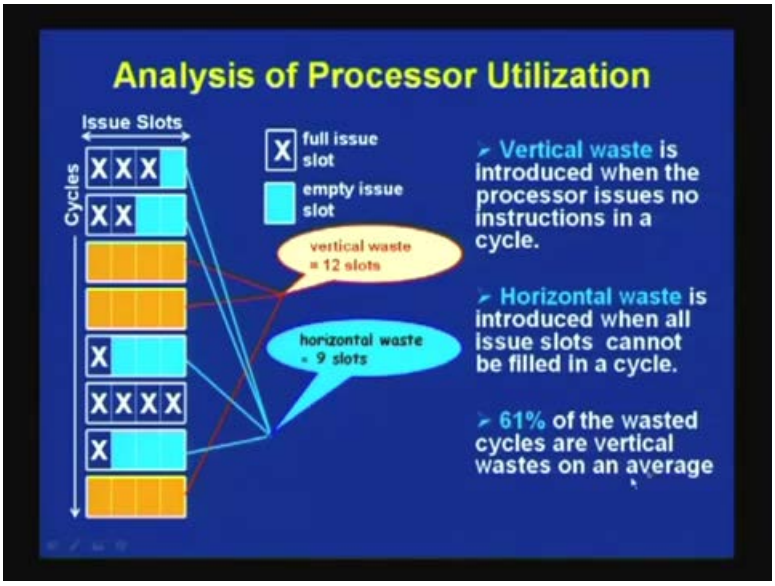



So, let us now have some analysis of ideal cycles in a super scalar processor, so let us assume the typically four instructions are issued, and as the super scalar processor can have different types of functional units, like adders, multipliers, floating point units, then the branch units and so on. And many functional units are ideal in many cycles, that we have already seen and particularly this is true when there is a cache miss. So, whenever there is a cache miss, then until the pipe line is refilled with new set of instructions, then the functional units remains unutilized.

So, in such a situation utilization is very poor, whenever there are cache misses and we have we know that dispatcher reads instructions decides, which can run in parallel. So, we have already discussed the architecture of super scalar processor, we have seen that instructions are fetched by the fetch unit, then they are decoded, then are sent to the dispatch unit. And dispatch unit identifies, which instructions can be paralleling scheduled or paralleling executed, then can paralleling issued to the functional units of level inside the processor.

And it has been found that this number of instructions is limited, by the instruction dependencies and long latency of operations, so some operations make take longer time and so on. So, because of these reasons the number of instructions that can be issued is very much limited.

(Refer Slide Time: 09:40)

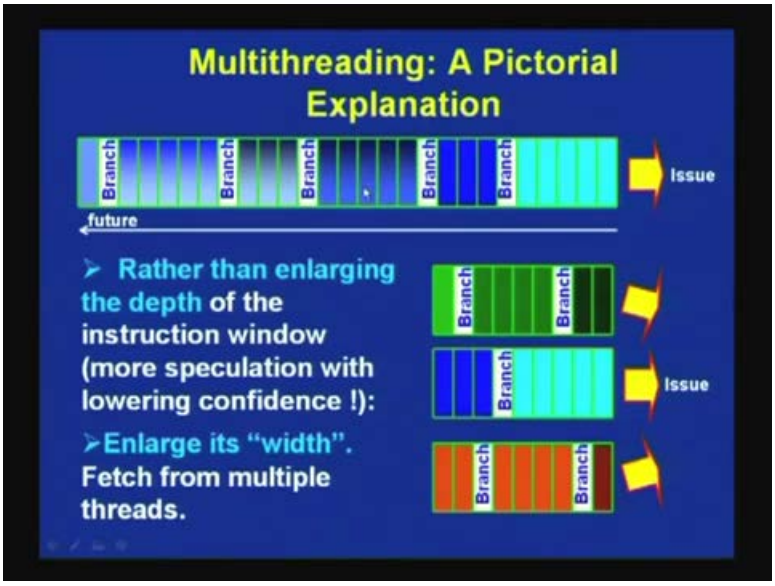



So, this source a diagram where you have got four issue processor and for different cycles it is shown, and you can see that X stands for full issue slot that means, those slots have been utilized. On the other hand those green colored rectangles are showing empty slots, so you find that we have got two kinds of wastes, one is vertical waste that is introduced when the processor issues no instructions in a cycle.

So, in a single cycle not a single instruction issues, so not a single functional unit is busy, so these are essentially vertical wastes, so ((Refer Time: 10:27)) this is a vertical waste, this is a vertical waste, this is another vertical waste. So, you find three vertical wastes and leading to wastes of 12 slots, on the other hand you can have horizontal waste is introduced, when all issue slots cannot be filled by in a cycle. So, for example, in this if you look at the first instruction cycle, you find that 3 slots are getting utilized and 1 slot is wasted.

Similarly, in the second cycle 2 slots are wasted and 2 slots are utilized, so in this way if we consider different types of wastes, the utilization of the decreases. And it has been found that 61 percent of the wasted cycles are vertical wastes on an average, so this is because of the limitation of the, first of all non availability of instruction level parallelism. And the limited size of the instruction window and the capability of the instruction issue hardware, because of all these things 61 percent of the wasted cycles are vertical wastes on an average.

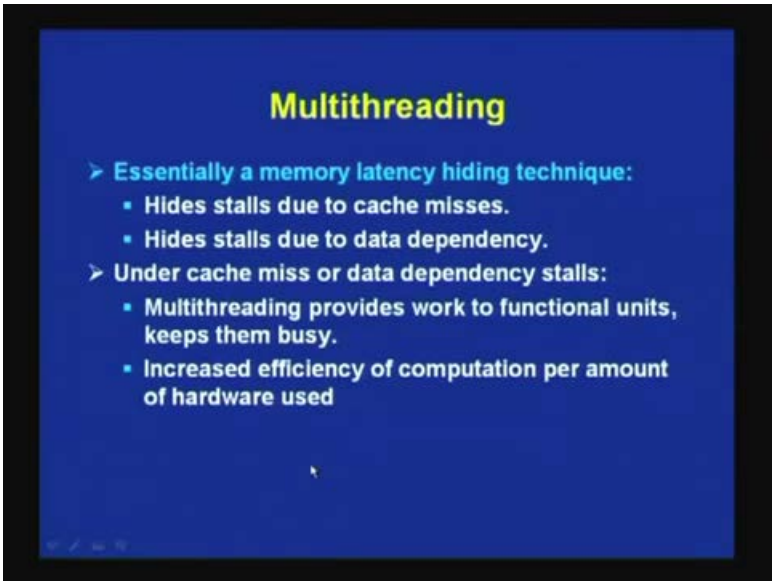
(Refer Slide Time: 11:39)




So, this diagram shows a pictorial explanation, the difference between the multithreading and in this particular case, we can we have seen that the instruction issue window has been increase, rather enlarge and it is having a much larger depth. So, you have a single instruction window with a larger depth, and by using I mean instructions are issued, more speculation with lowering confidence.

Because, as we have larger window size, several ranges, we know that the confidence that speculation with lower confidence take place, as the size of the window is larger and because you have to do branch prediction and so on. On the other hand, whenever you are you go for multithreading, so you can see here you have three different threads. And from three different threads, instructions are being used and we are enlarging the width here to fetch multiple instructions, I mean fetch instructions from different threads.

(Refer Slide Time: 13:00)

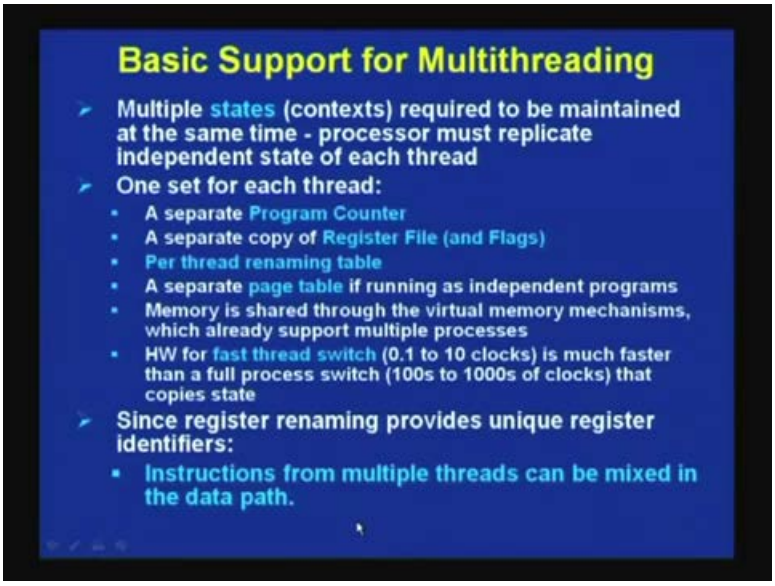



So, multithreading can be considered as a, I mean memory latency hiding technique, so you have seen, whenever we are having multithreading and whenever there is a stall, then several cycles are wasted. But, if you have multithreading, instead of single threading then if one thread is blocked, we can go for another thread issuing instructions for another thread, and that is how you can utilize the functional units.

So, hides stalls due to cache misses, hides stalls due to data dependencies, whenever these type of I mean stalls occur, then we can take help of the multithreading to for utilizing the functional units. And it leads to increase the efficiency of computation per amount of hardware used, so you can look from another angle, we have got certain amount of hardware.

So, what is the utilization of the hardware or what is the computation that you are getting per hardware, per amount of hardware that is available. So, from that point of view multithreading helps you to improve the efficiency.

(Refer Slide Time: 14:23)




Question naturally arises, what kind of support you require for multithreading, multithreading does not come free, in this world nothing is free, so it is not a free launch, you have to pay for supporting multithreading. What you have to pay, what is additional requirement for multithreading, so fortunately it has been found that multiple states required to be maintained at the same time. So, you have to maintain multiple states processor must replicate independent state of each thread.

So, if you it is a single thread, then you have to maintain, you have to store a single thread and when switching occurs, you have to store information of one thread. But, whenever you are having multiple threads, you have to maintain the states of multiple threads. And a one set for each thread requires a separate program counter, a separate copy of register file, a separate renaming table for each thread, a separate page table if running as independent programs.

So, since memory is shared through virtual memory management technique, which already supports multiple processes, but in such a case you require separate page table to support multiple threads. And not only that, your hardware should provide first thread switching, so that means whenever you are going from one thread to another thread, we already know that, the thread switching is much faster than process level switching. However, the hardware should be designed such that, this is quite fast, so that you can make use of multithreading more effectively. Then it has been found that, for thread

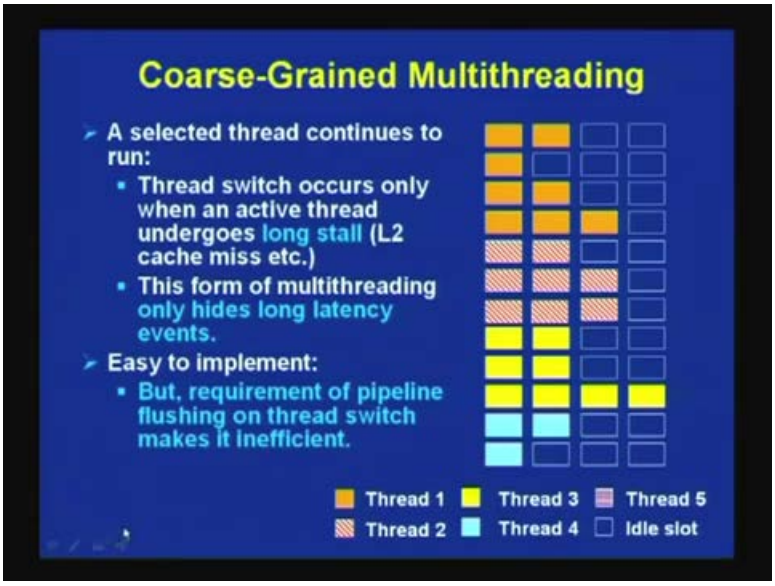
switching you require about 0.1 to 10 clock cycles compared to 100 to 1000 clock cycles that is required in the context of process switching. Because, the more information has to be stored, moreover since register renaming provides unique register identifiers, so you are using register renaming. And that is help you to have instructions from multiple threads, that can be mixed in a in the data path, so register renaming also helps and in supporting multithreading.

(Refer Slide Time: 17:01)


So, multithreading in it is most basic form, requests processors interleaves execution of instructions for different threads. So, in simple terms you can say that, the processor he is executing instruction in a inter leave manner, instructions from different threads; and you can have three different thread scheduling. So, you can categorize the multithreading techniques, on a single processor in three different categories, number 1 is coarse grained multithreading.

Second one is fine grained multithreading, third one is simultaneous multithreading which is of greater importance, because of the higher efficiency, that is provided by simultaneous multithreading. Let us have a look at the differences among these three types of multithreading.

(Refer Slide Time: 17:59)



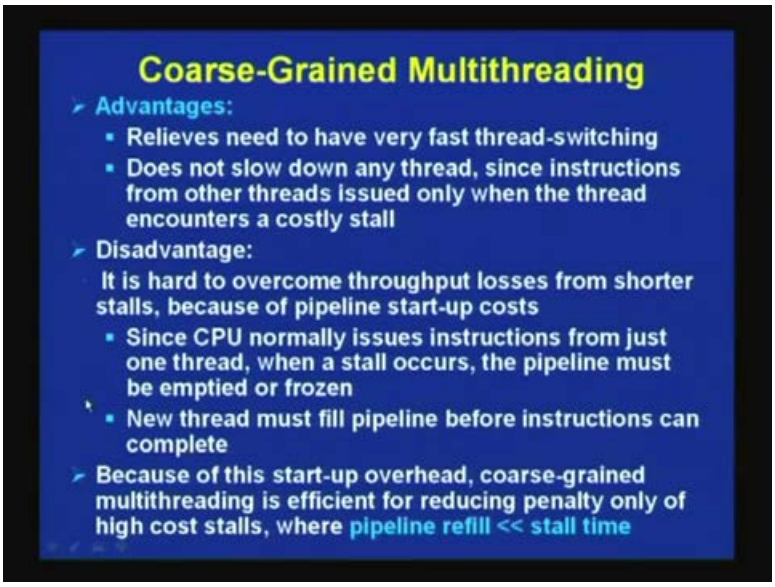

First one is coarse grained multithreading, here as you can see the switching is taking place, I mean one thread is getting executed, so these yellow colored slots correspond to a single thread. Then here after four cycles a switching is occurring to differ another thread, so for the next three cycles another thread is being executed, again thread switching is occurring. Then yellow colored slots correspond to another thread, so thread switch occurs only when an active thread undergoes long stall.

So, that means, whenever there is a L 2 cache miss, so whenever there is a L 2 cache miss obviously, in such a case we have to fetch from main memory and obviously, it will inform long delay. So, that memory cycle latency see can be overcome, so in such a case you can hide this form of multithreading only hides long latency events, long latency events means you have long stalls that means, L 2 cache miss or something like that. So, this is the coarse grained multithreading, and it has been found that it is easy to implement.

So, for example, in this ((Refer Time: 19:27)) here you have got five different threads, and one for several cycles, one thread is executed, then we are going for another thread, then we are going for another thread and so on. However, whenever we implement it, there is a requirement of pipeline flushing on thread switch, I mean this makes it inefficient. So, whenever we are switching from one thread to another thread, you have

to flush the pipe line that means, you have to fill the instruction pipe line with new set of instructions, and leading to inefficiencies, so this is your coarse grained multithreading.

(Refer Slide Time: 20:11)



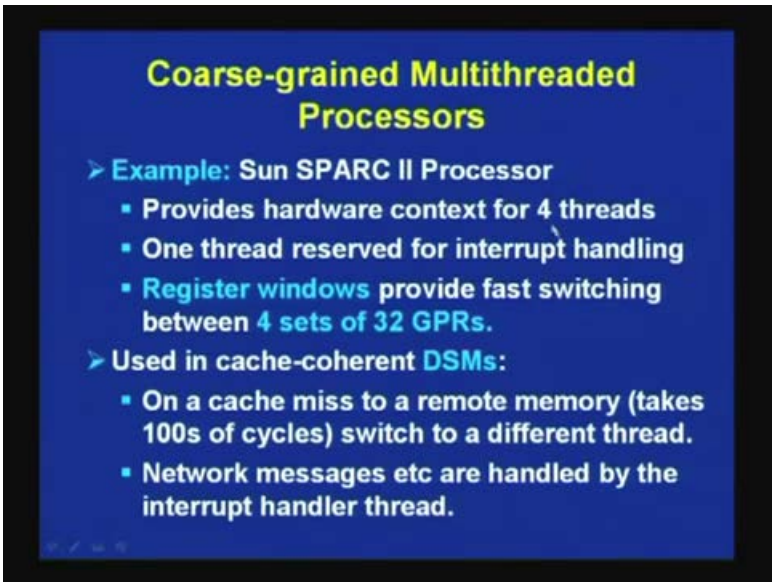

Here are the advantages of coarse grained multithreading, it relieves the need to have fast thread switching, so thread switching need not be very fast, it will request some clock cycles. So, when we use coarse grained multithreading, since it is occurring after several cycles, it need not be thread switching need not be very fast, and does not slow down any thread since instructions from. Other threads issued only when the thread encounters a costly stall, so in this case you are executing different threads, may be of a single application.

So, the execution of different threads are not delayed, why it is not delayed, because a particular thread switches I mean it is stopped or blocked, only when it encounters some costly stall that means, L 2 level of cache miss. Of course, there are several disadvantages, number 1 is it is hard to overcome throughput losses from shorter stalls, because of pipe line start-up costs. So, we have seen that instruction pipe line has to be filled up, there is start up cost and whenever there is a shorter stall. So, instead of several cycles if it happens may be after 1 or 2 cycles, then there it leads to throughput losses.

And since, CPU normally issues instructions from just one thread, when a stall occurs the pipe line must be emptied or frozen, and so as I have already mentioned. And new thread must fill the pipeline before instructions can be complete, because of these pipe

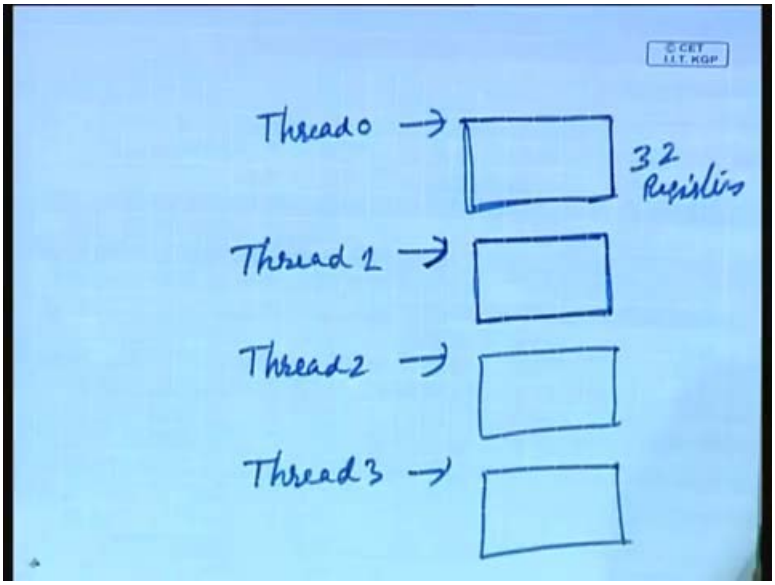
line start-up, first I mean shorter stalls leads to throughput losses. So, this is the limitation of coarse grain multithreading, and what you have to see, because of this start up overhead coarse grained multithreading is efficient for reducing penalty, only of high cost stalls. That means, only when this stall time is very long compared to the pipe line refill time, only then this coarse grained multithreading makes sense.

(Refer Slide Time: 22:45)




And this course grained multithreading has been used in several processors, like sun SPARC II processor, it provides hardware context for four threads. And one thread is reserved for interrupt handling, and it uses the concept of register window, that provides fast switching. You may recall that, that means the sun SPARC II processor has got a large number of original purpose registers. So, which are used to form register window, having 4 sets of 32 general purpose registers that means, each thread can be allocated one register window.

(Refer Slide Time: 23:39)




And whenever thread switching occurs, suppose you have got 4 register windows, this is one register window comprising 32 registers, this is another register window, this is another register window, I mean another register, so these are the 4 register windows, and each are having 32 registers. So, this may be you can say thread 1 or thread 0, this can be thread 1 can use this group of registers another group, then thread 2 is another window and thread 3 is using another window.

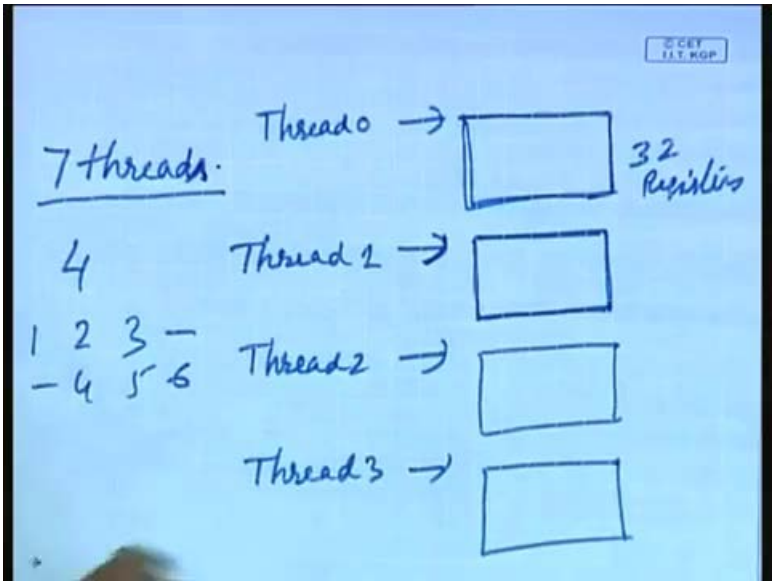
So, whenever a thread switching occurs, only the pointer to this register window has to be modified, so there is no need to store and restore all the registers. Because, all the registers are already present in the CPU, only thing the pointer to this window, has to be provided whenever thread switching occurs.

(Refer Time: 24:47) So, this is also used in cache coherent distributed CR memory architecture, so we have not yet discussed the cache coherent problem, in the next lecture I shall discuss about this. And we shall see that whenever on a cache miss to a remote memory, that is present in a distributed CR memory architecture, that may larger number of cycles. So, whenever this occurs, whenever this occurs we must switch to another different another thread, so similarly network messages etcetera, can be handled by interrupt handler thread. So, there is another thread that can handle the network messages, so we find that the coarse grain multi thread processors are available, and they are used in practical applications.

(Refer Slide Time: 25:51)


Now, coming to fine grain multithreading, here you have got few active threads, so context switching among the active threads on every clock cycles. So, here context switching is occurring among the active threads on every clock cycle, and this is usually done in a round robin fashion, skipping any stalled threads.

(Refer Slide Time: 26:22)



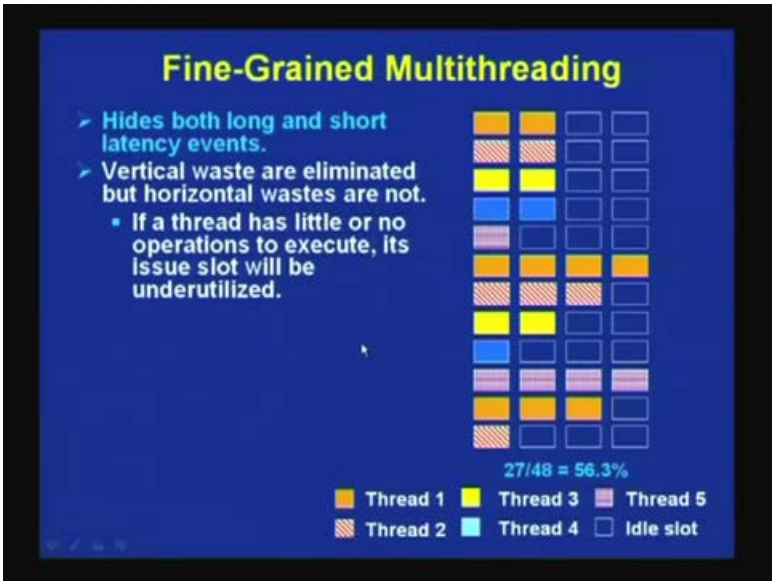

So, you may be having say 7 threads, 7 threads may be active, but your number of instructions that can be issued let it be 4. So, these 7 threads can be showed in a round robin fashion, depending on which are not stalled, may be say 1, 2, 3, then this is empty others are not ready, are not having cannot be issued at this moment. Then may be 4,5 and 6, this slot remains empty, so for different functions you can issue, them for different functional units.

In this way different threads can be issued and in a cycle it can be done, so occupancy of the execution core would be much higher, compared to coarse grain multithreading. Here the occupancy will be much higher, although you are having context switching in every cycle, that is the reason why CPU must be able to switch threads in every clock cycle that, in other words the that switching should be very fast. So, the advantage is it can hide both short and long stalls, since instructions from other threads are executed when one thread gets stalled.

And disadvantage is that it slows down execution of individual threads, since a thread ready to execute without stalls will be delayed, by instructions from other threads. So, this is one disadvantage here, whenever we are having fine grained multithreading, so earlier we have seen when we are using coarse grained multithreading, then a single thread does not get delayed. But, in this case a single thread may get delayed, because in a single slot you are issuing instructions from different threads.

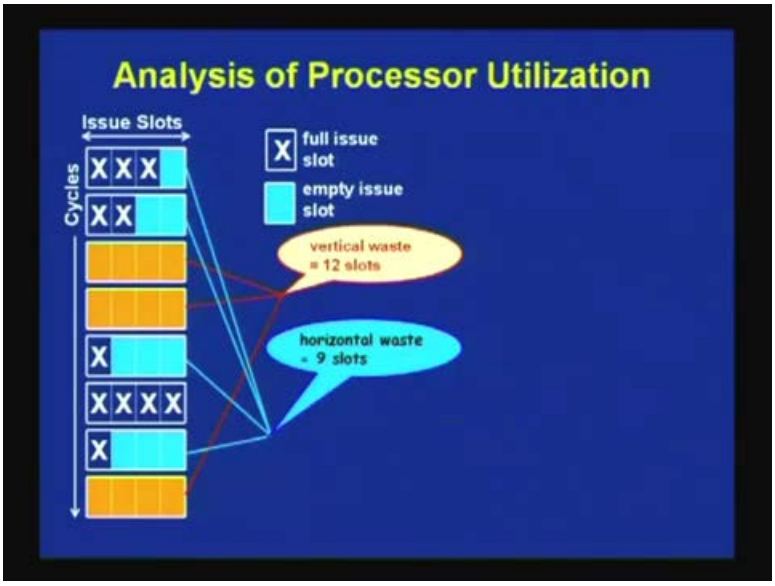
So, although that thread is ready to execute without stalls even then we are issuing instructions from other slots, that is the reason why individual threads may get delayed, but for all throughput we will improve, as we shall see.

(Refer Slide Time: 28:46)




So, this shows the fine grained multithreading, here you can see this yellow colored slots correspond to one thread, then the shaded pink slots correspond to another thread, this corresponds to another thread. So, there is a context switching in each cycle, so thread switching is occurring in each cycle, from one thread to another thread and in different cycles, different threads are getting executed. So, earlier we have seen as long as a single threads do not encounter stall that is executed, but here it is not shown. So, in this particular case, vertical wastes are eliminated, but horizontal wastes are not, we have seen there are three types of wastes earlier we have seen that.

(Refer Slide Time: 29:49)

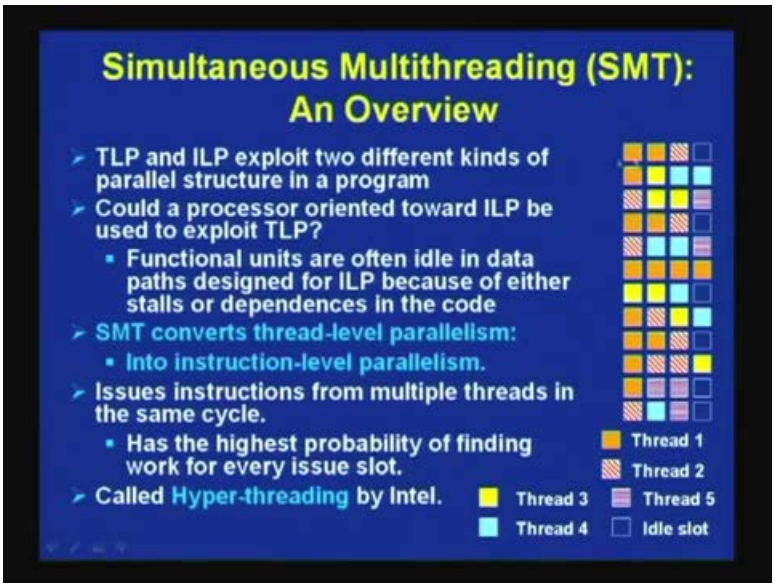



So, those vertical wastes, these vertical wastes are eliminated whenever we shall be having these fine grain multithreading ((Refer Time: 29:58)), so you can see not a single cycle is empty, so vertical wastes are eliminated. And if a thread has little or no operations to execute, it is issue slot will be underutilized, so however if for a single thread if some slots, I mean if some instructions cannot be issued to keep some slots some execution units busy, those slots remain unutilized. That means, there will be a kind of I mean horizontal slots will remain unutilized, whenever we use this fine grain multithreading.

(Refer Slide Time: 30:35)

So, issue instructions only from a single thread in a cycle again may not find enough work every cycle, but cache misses can be tolerated. So, in this case cache misses can be tolerated, and this particular approach has been used in Sun's Niagara chip, which is having 8 cores in s single processor.

(Refer Slide Time: 31:02)




Now, coming to the simultaneous multithreading, we are familiar with two types of parallelism, one is your instruction level parallelism, another is your thread level parallelism. So, here we shall see how the concept of instruction level, and thread level parallelism can be combined to achieve simultaneous multithreading. So, here question is could a processor be oriented towards instruction level parallelism be used, for used to exploit thread level parallelism.

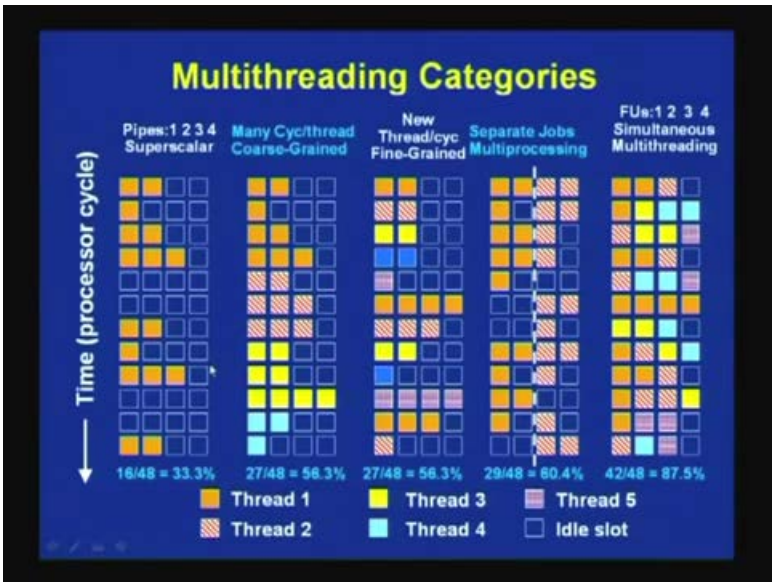
That means, when we are using thread level parallelism, which is the basis for multithreading, whether it can be oriented towards instruction level parallelism. That means, functional units are often idle in data paths designed for designed for ILP, because of either stalls or dependencies in the code. So, here what we shall try to do a single, I mean in a single cycle you can see, we are filling up slots from different threads, so in earlier in single cycles the instructions from only one thread were issued.

But, in this case we can see that, these two in the first cycle first two instructions correspond to one thread, and third instruction correspond to another thread, unfortunately 1 slot, 4 slot is remaining unutilized. Because, there is no other, I mean

instruction which is ready for issue, so SMT converts thread level parallelism into instruction level parallelism. So, that is how the combination of both is taking place, the issues instructions from multiple threads in the same cycle, has the highest probability of finding work for every issue slot.

So, whenever we do this, we can see the majority of the slots are getting filled up, so very few slots are remaining unutilized, whenever we go for this simultaneous multithreading. So, this technique, this simultaneous multithreading is also known as hyper threading, which this particular turn has been coined by Intel, hyper threading. So, hyper threading and simultaneous multithreading are same, where instructions for different threads are issued in a single cycle.

(Refer Slide Time: 33:44)




So, this diagram shows different types of multithreading techniques in a single diagram, so you can compare their performances, and you can see how the utilization of the processor resources taking place, in this different categories of multithreading. So, first one correspond to simple superscalar architecture, where you are executing a single thread. So, in a normally we execute a single thread, and you can see the many threads are remaining, many slots are remaining unutilized.

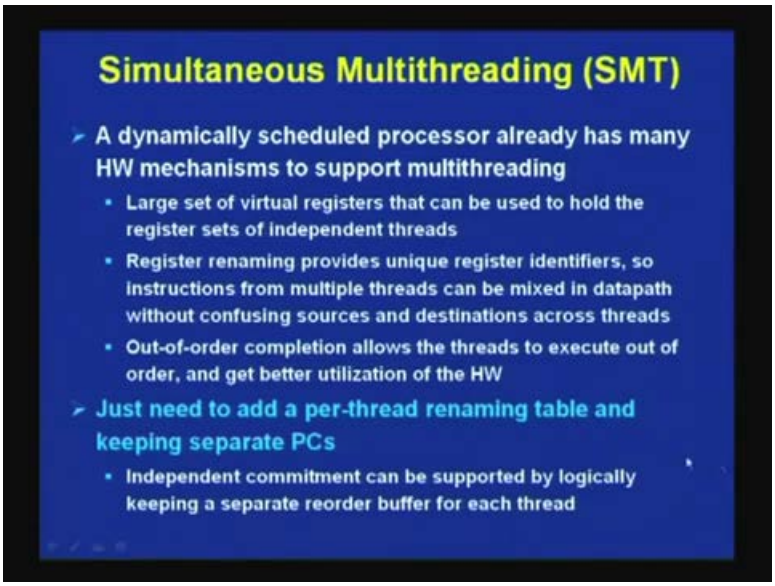
And so 16 out of 48 slots are utilized, so 33 percent is the utilization of the processor resources, on the other hand in case of coarse grain multithreading, we can see when one thread get stalled, another thread is being started. And this way one thread followed by

another utilization thread and so on, so in this case utilization of the processor increases, and so efficiency is 56.3 percent. So, only 27 out of 48 slots gets utilized, we giving you 56.3 percent of utilization of the resources, similarly in fine grain multithreading, you can have better utilization compared to simple super scalar architecture.

However, in this case we find that number of, I mean efficiency same as coarse grain multithreading, but in general in fine grain multithreading we get better efficiency. So, but for this particular example, we find efficiency that that utilization of the processor resources is same 56.3, then here of course it is not multithreading, but multi processing. So, in this particular case we have got two processors, separate jobs on separate processors, two different processors, so each processor is performing I mean one thread.

So, one thread is operating on one processor, another thread is operating on one processor, here also the utilization is good 60.4 percent. But, whenever we go for simultaneous multithreading in a single processor, we find that utilization is pretty high 42 out of 48 slots is getting utilized leading to 87.5 percent of efficiency of the processor utilization of efficiency, so this is the multithreading categories.

(Refer Slide Time: 36:37)




Now, we have question naturally arises, what is the additional thing that is required to support simultaneous multithreading, I have already told it will not be free, but for what is the additional hardware resources, that you require. We have already discussed about the dynamically scheduled processor, and we have seen they require, they have many

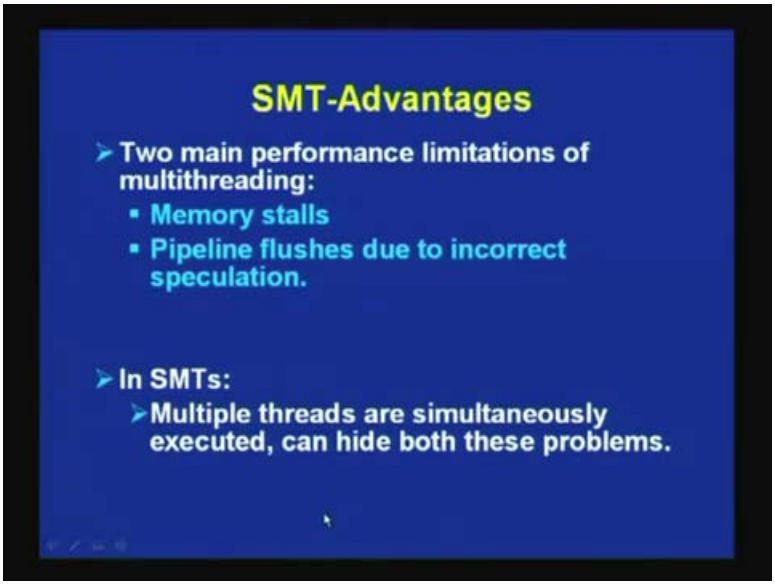
hardware mechanisms, and they support multithreading. Because, in a dynamic dynamically scheduled processors, we support multi multiprocessing, then they are many hardware resources are already available.

Those hardware mechanisms can be utilized in the context of simultaneous multithreading for example, large set of virtual registers, that can be used to hold the register sets of independent threads. Then register renaming, which is used in dynamically scheduled processors, can be used in simultaneous multithreading, register renaming provides unique register identifiers. So, instructions from multiple threads can be mixed in the data path, without confusing resources and destinations across threads.

So, that register renaming can be fruitfully utilized in simultaneous multithreading, then out of order completion allows the threads to execute out of order, and it gets better utilization of the resources, we have seen out of order execution is allowed in dynamic scheduling of instructions. However, you require commit unit, where final riding into the registers is performed with the help of commit unit. So, here in simultaneous multithreading, we can use that and we shall see we will require multiple commit unit for that purpose.

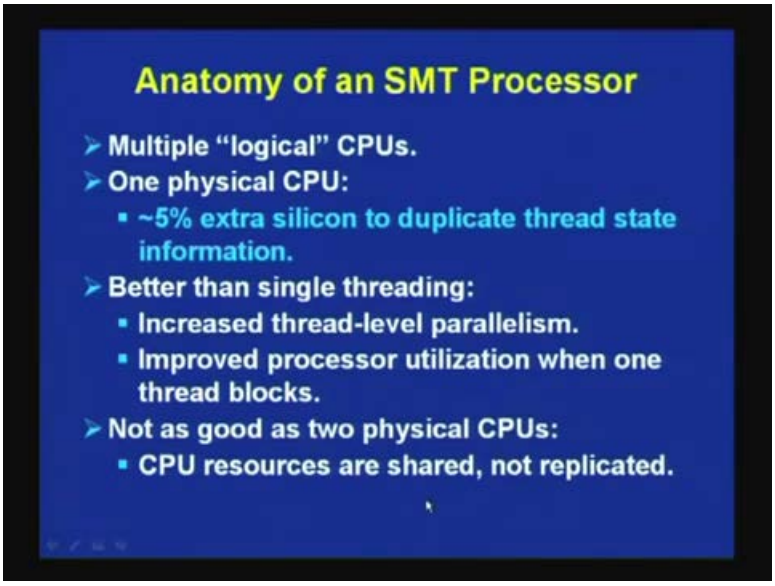
So, just need to add a per thread renaming table. so we will be require multiple renaming tables, for each of the threads we require a separate renaming table, and separate program counters, that you will be required. Because, multiple threads will be executing separately, this is the additional requirement compared to dynamically reschedule processor. So, independent commitment can be supported by logically keeping separate reorder buffer for each thread, so you can have separate reorder buffer for each thread; and that is how you can support this simultaneous multithreading.

(Refer Slide Time: 39:26)




Now, we have already discussed about the simple super scalar architecture, and two performance limitations are overcome, number 1 is memory stalls which I have already seen that means, whenever there is a memory stall, we can initiate another thread, that is how the memory stall latency problem is overcome. And then pipeline flushes due to incorrect speculation that also is overcome by using this simultaneous multithreading; so instead of speculations you are using in multiple threads, so in SMT's multiple threads are simultaneously executed. And one can hide both these problems that means, memory stalls and pipeline flushes due to incorrect speculation, these problems are overcome in simultaneous multithreading.

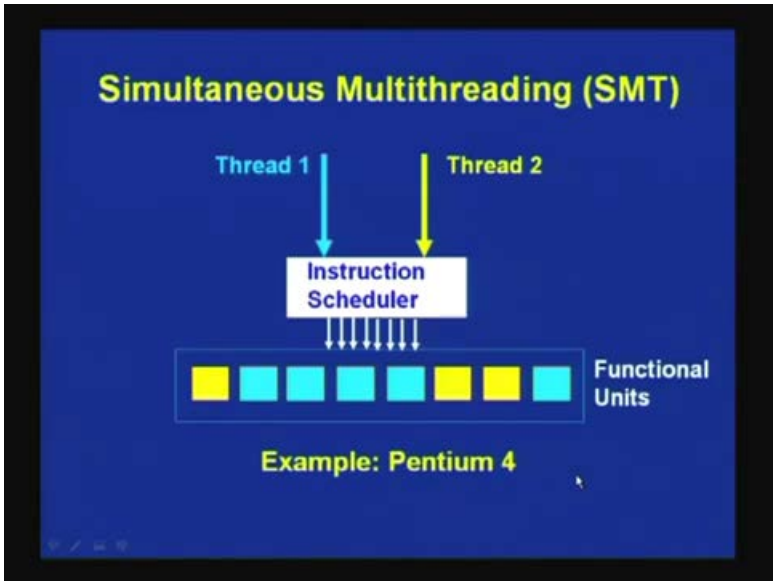
(Refer Slide Time: 40:31)




Now, what is the anatomy of a simultaneous multithreading processors, we have already mentioned that you do not require multiple processor, but you are really having multiple logical CPU's, physically you will be having only one CPU. But, some additional resources are required, that is only 5 percent extra silicon to duplicate thread state information. So, you have for individual threads, the state information have to be stored, for that purpose you will require additional resources, that increase in silicon is only 5 percent.

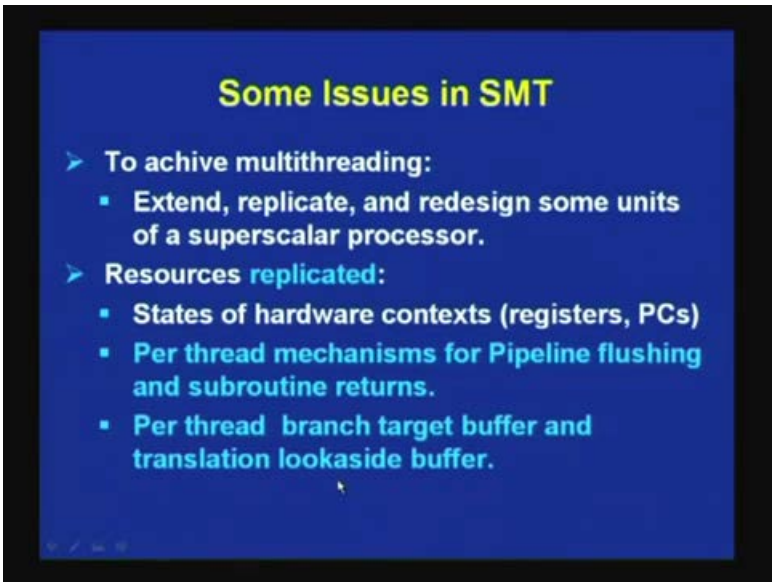
So, this is much better than single threading, where because it leads to increase thread level parallelism and improved processor utilization, when one thread blocks. So, instead of single threading, simultaneous multithreading is much better however, this cannot be compared with two physical CPU's. That means, so whenever we say two different threads, what will be the efficiency or what is the performance that we will get, compared to two different threads. Two different processors will definitely give you better performance, that is being mentioned here. This is true, because CPU resources are shared, and not replicated and as a consequence two different CPU's will definitely give you better performance.

(Refer Slide Time: 42:24)




So, this gives you a kind of visual representation of simultaneous multithreading, here you are having two different threads, then you have got a instruction scheduler. That means, instructions from different threads are scheduled, and different functional units are used for example, this yellow colored correspond to one thread, thread 2. And green colored or blue colored slots correspond to our functional units, correspond to thread 1, so instruction scheduler is scheduling different threads to different functional units.

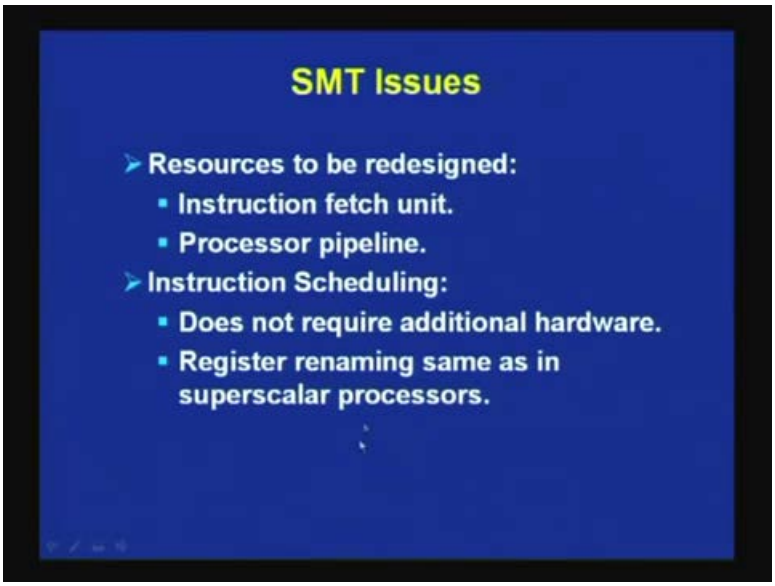
(Refer Slide Time: 43:08)



Now, here are some important issues related to simultaneous multithreading, to achieve multithreading you have to extend replicate and redesign, some units of a super scalar processor. So, here we are high lighting what modification, what change in a super scalar processor is needed, to make it suitable for simultaneous multithreading. You have to replicate some resources, and you have to extend some techniques and you have to redesign some units of super scalar processor.

So, which I shall discuss very briefly, so the resources which are to be replicated is mentioned here, so that means states of hardware contexts, you have to have multiple resources multiple hardware. That means, were the registers program counters can be, coming from different threads, then path per thread mechanism for pipe line flushing and sub routine returns. So, we have to think about the per thread mechanism not for a single thread pipeline flushing and sub routine returns, in other words you will require different areas memory will be required, where you can store this thread information. And per thread branch target buffer, and translation look at buffer that will be required, these are to be replicated whenever we go for simultaneous multithreading.

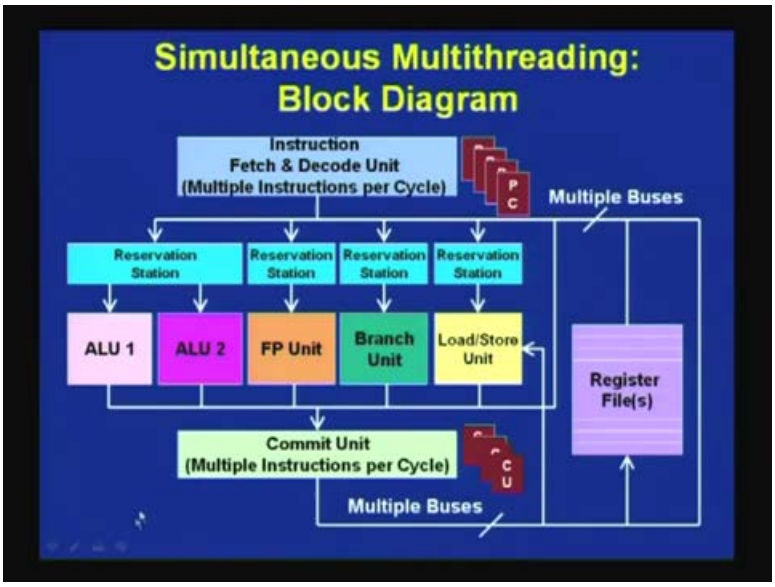
(Refer Slide Time: 45:00)




Then resources to be redesigned are instruction fetch unit, so instruction fetch unit needs to be redesigned in the context of simultaneous multithreading, then processor pipe line are to be redesigned. For example, so far as instruction scheduling is concerned, it does

not require any additional hardware, and register renaming is same as in super scalar processor that means, instruction scheduling part need is not much different.

(Refer Slide Time: 45:30)



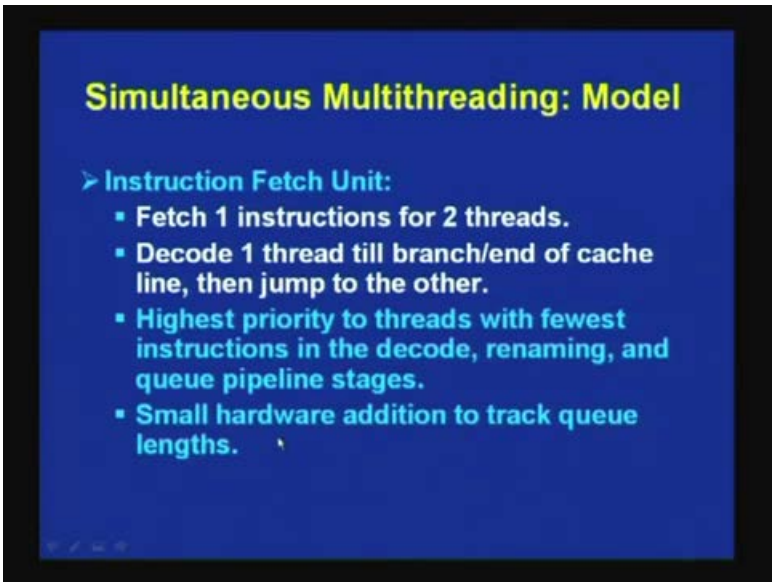

And here we shall compare the architecture of super scalar and then when we go for super simultaneous multithreading, what are the additional thing required that is mentioned. So, here you have got instruction fetch and decode unit, there is single program counter and then you have got reservation stations where there are registers, where the operands are hold operands are stored, and which are provided to different functional units. And then different functional units they will perform, they are execution and there is a commit unit, so you can have out of order execution with the help of the commit unit, we wrote them in the registers in the proper order.

And also we can update the various registers, which are present in the reservation units, I have already discussed in detail, the super scalar architecture. Now, let us have a look at what are the additional things that you required, whenever you go for simultaneous multithreading. So, as I mentioned you will require separate program counters, for each thread, so the program counters are to be replicated which are shown here, and he will require different commit units for each thread.

So, separate commit unit for each thread is required which are shown here, and in this register file we will require register renaming, so which is not mentioned here ((Refer Time: 47:10)), but you will require register renaming. If you have large number of

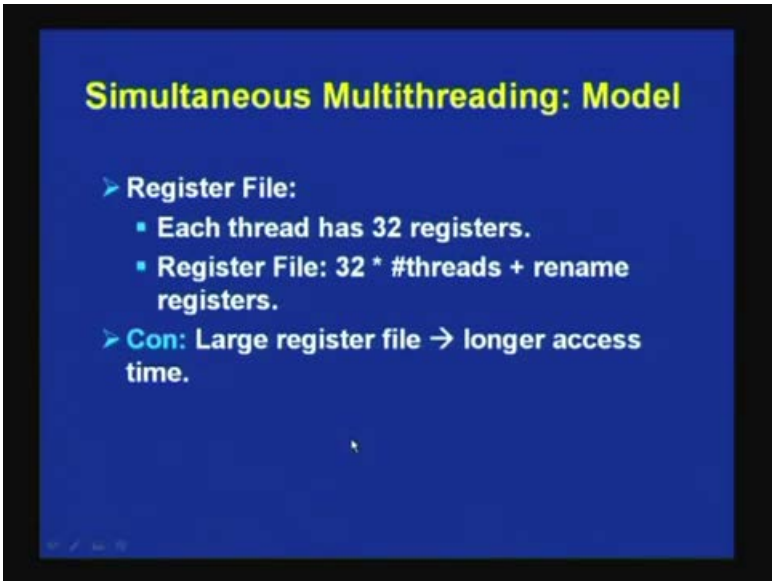
registers, you can assign separate registers to different threads and you can use renaming techniques for that purpose. So, so you can see how the architecture needs to be modified to support simultaneous multithreading, so you can see it can be very easily upgraded, from superscalar architecture can be very easily upgraded to support simultaneous multithreading.

(Refer Slide Time: 47:46)




So, so far as the instruction fetch unit it is concerned you have to fetch, one instruction for 2 threads, then decode 1 thread till branch or end of cache line, then jump to another. So, this is required for multithreading and highest priority threads with fewest instructions in the decode, so you have to use kind of priority, that higher priority threads with higher priority will be executed first. And highest priority to threads with fewest instructions in decode renaming an queue pipeline stages, so this is how priority of the threads can be used, for the purpose of issuing instructions, and small hardware addition to track queue lengths.

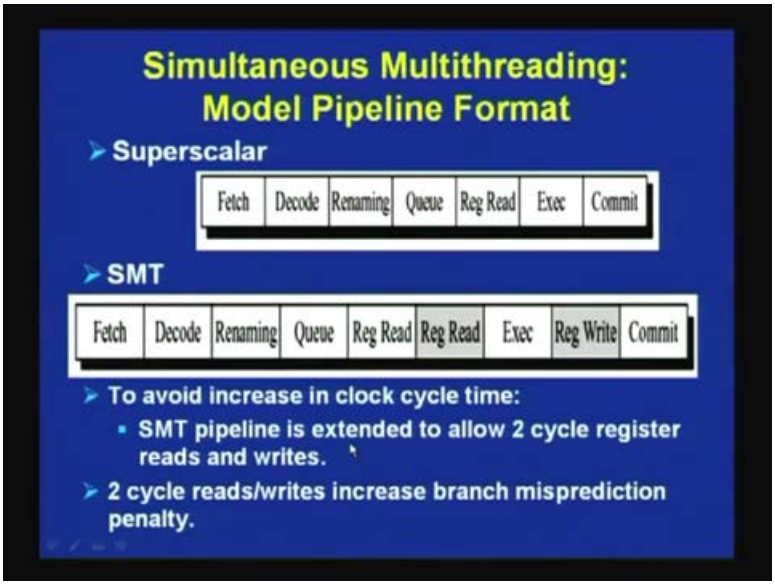
(Refer Slide Time: 48:40)




Then, so far as register file is concerned, as I mentioned each thread can have 32 registers, so register file is 32 into number of threads, plus some more registers for the purpose of renaming. So, the consequences you will require a very large register file, whenever to support this simultaneous multithreading. And when you have got large register file, you know that the access time will be longer, if the number of registers in a register file is small, that decode unit is simpler, so access time will be smaller.

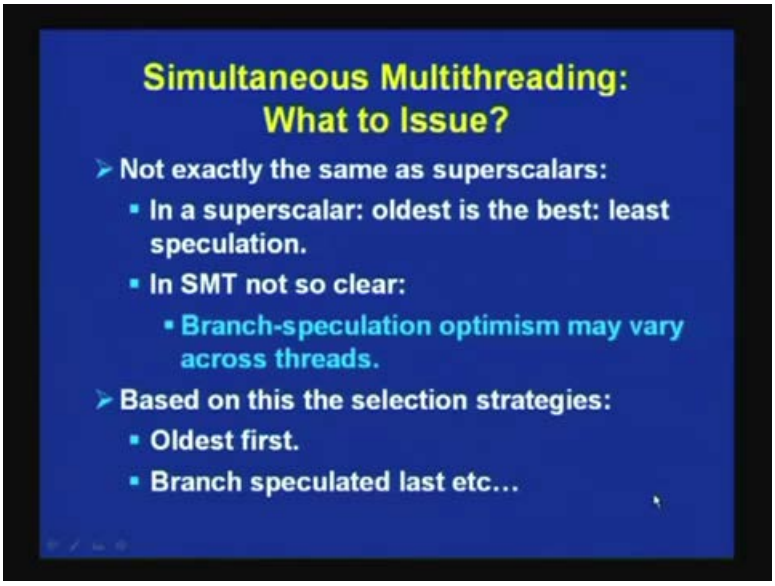
But, whenever you have got a large number of register in a register file, then the decode unit is complex and the access time will be longer, so that is one disadvantage. And that disadvantage you have to accept, if have interested in simultaneous multithreading, because simultaneous multithreading gives you much more benefit, that you have already seen.

(Refer Slide Time: 49:46)



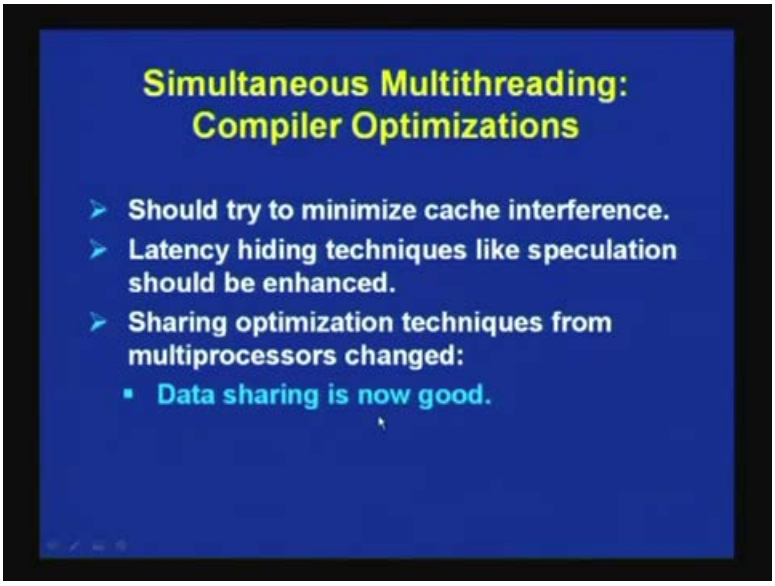

And this is how, that pipe line I have already mentioned that, pipe line architecture needs to be changed modified compared to super scalar, this is the typical super scalar pipeline unit, where you have got fetch, decode, renaming, queue, read register and so on. And here for simultaneous multithreading to increase in clock cycle time, the simultaneous multithreading pipe line is extended to allow 2 cycle register reads and writes. So, you can see here you have got one additional register read and one register write. So, this is required however, whenever 2 cycle reads writes, this will increase the branch miss prediction penalty, so branch miss prediction penalty is increased, because of the change in the pipe line to support simultaneous multithreading.

(Refer Slide Time: 50:45)



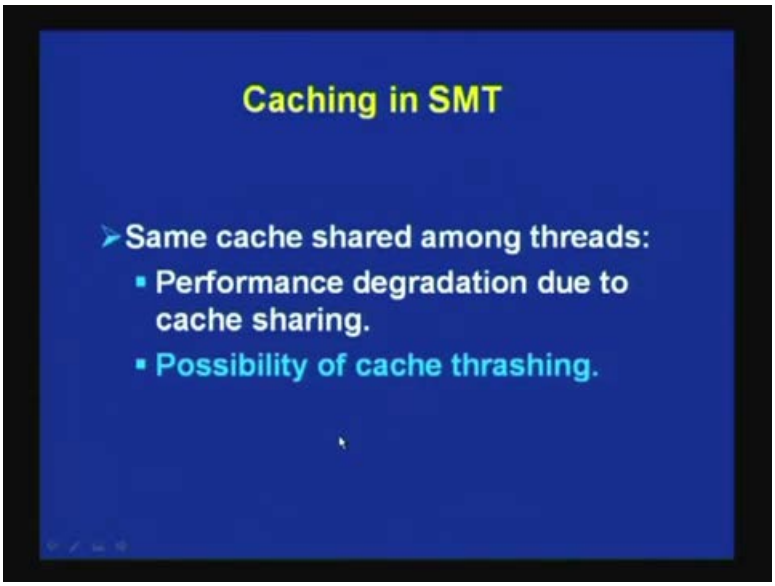

Then, what to issue not exactly the same as super scalars, we have seen in super scalar architecture instructions are issued, and oldest is the best and least speculation is used. But, in the context of simultaneous multithreading, it is not very clear what policy to be adopted for issue of instructions, so branch speculation optimism may vary across threads. So, if you are having different threads, the branch speculation optimism will be different, may be different for different threads, this leads to complication. And based on this, the selected selection strategies can be oldest first, branch speculated last and so on, so this policies are little different and complicated in the context of simultaneous multithreading.

(Refer Slide Time: 51:41)



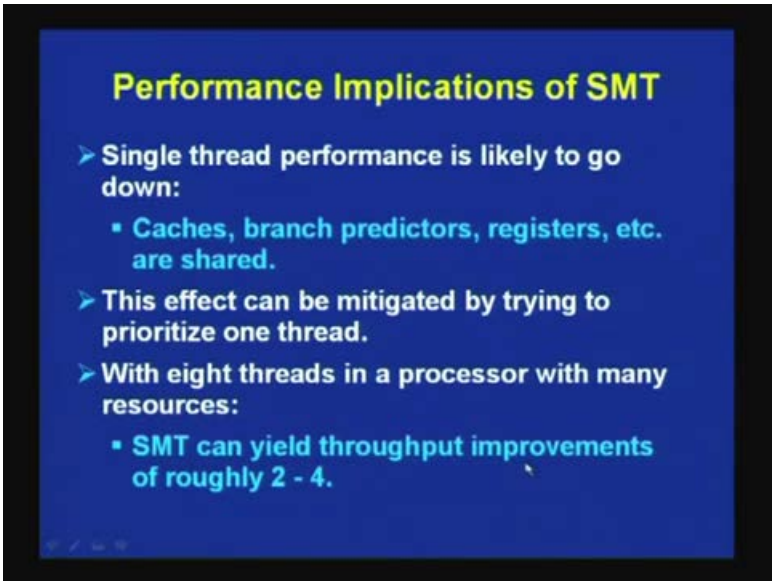

So, far as the compiler optimization is concerned, should try to minifies cache interference, and latency hiding techniques like speculation should be enhanced. Because, one of the primary benefit that we get from simultaneous multithreading is latency hiding, memory latency hiding. So, this speculation helps you to hide this latency, and that should be enhanced in the context of simultaneous multithreading. Ad sharing optimization technique from multi processor has to be changed, and data sharing is now good, so it is little different from multi processor sharing optimization. Here, the optimization has to be done in a different way compared to multi processor architecture.

(Refer Slide Time: 52:31)



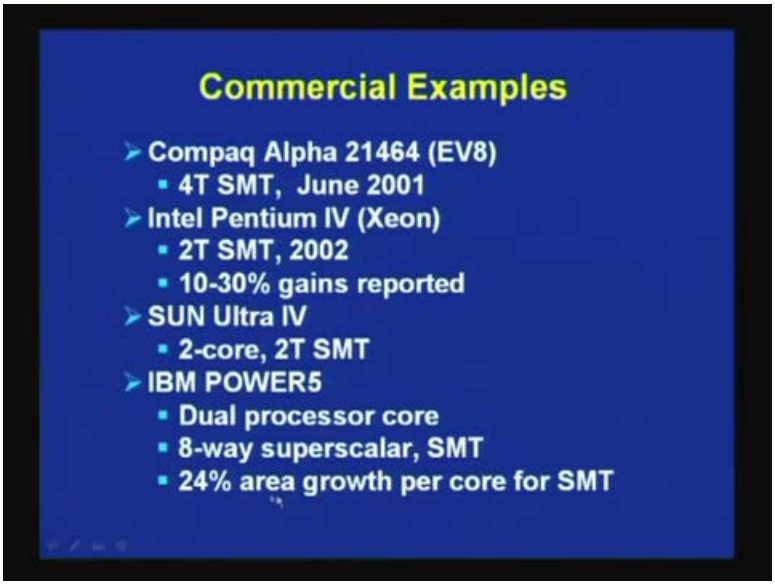
Coming to caching, same cache shared among all the threads, we have a single cache memory and performance degradation due to cache memory occurs. And as a consequence this leads to the possibility of cache thrashing, cache thrashing is essentially all the time the data transfer between cache, and main memory is taking place, without doing giving any real work, that kind of situation can occur in simultaneous multithreading, because same cache is being shared by multiple threads.

(Refer Slide Time: 53:08)



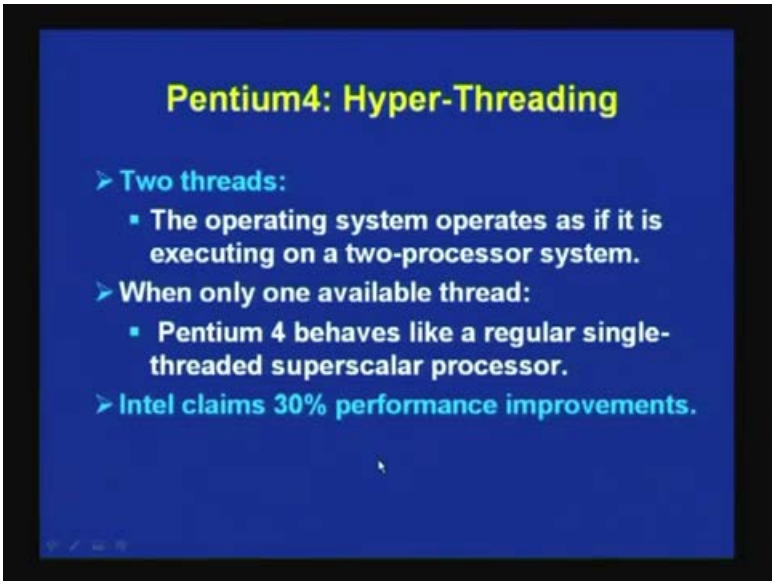

So, this is the performance implication of simultaneous multithreading, single thread performance is likely to go down, as I have already mentioned, because of sharing caches, branch predictors, registers etcetera are shared. And this effect can be mitigated by trying to prioritized 1 thread, so I have already mentioned about prioritizing different threads, and execute it that way. And with 8 threads in a processor with many resources is possible, and single simultaneous multithreading can yield through put improvements in the range of 2 to 4.

(Refer Slide Time: 53:45)




And because of the many benefits of the simultaneous multithreading, they have been used in many commercial processors like Compaq alpha 21464, take alpha Compaq alpha processor, which used 4 threaded multi simultaneous multithreading. Then Intel Pentium 4, 2 threaded simultaneous multithreading, and it has been reported by Intel that it leads to 10 to 30 percent gains in performance. Then SUN ultra 4 which is also 2 core, 2 threaded simultaneous multithreading, so in addition to 2 core, it uses 2 thread for core simultaneous multithreading. Then IBM power, this is also used simultaneous multithreading, dual processor core 8 way super scalar simultaneous multithreading, and 24 percent area growth per code for simultaneous multithreading, so this is some commercial example.

(Refer Slide Time: 54:49)




And in Pentium 4 as I mentioned, that simultaneous multithreading is called in the name of hyper threading, in Intel jargon. And they use two threads the operating system provides operates as if it is executing on two processor system, it can be a single processor systems, but it can be it can use two threads. So, when only one is available one available thread Pentium 4 behaves like a regular single threaded super scalar processor, so as I mentioned 30 percent performance improvement. So, we this we have come to the end of today's lecture on simultaneous multithreading.

Thank you.