Compiler Design

Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 17

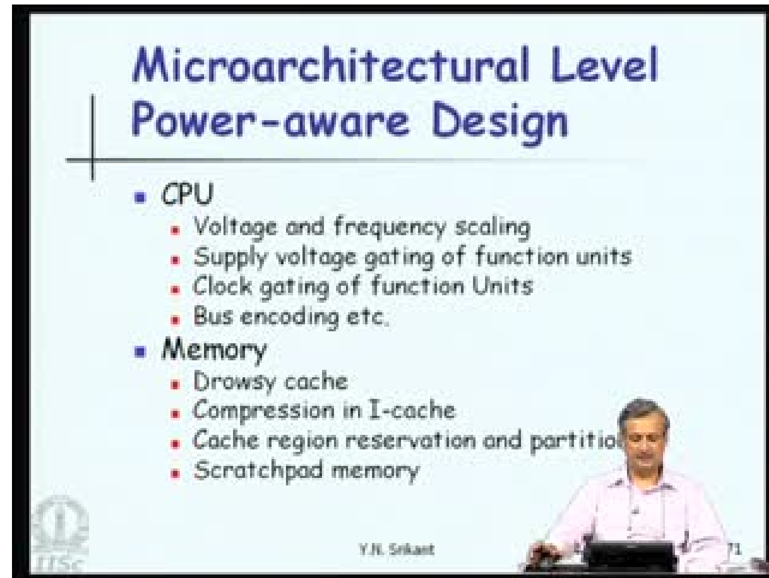
Lecture No. # 34

Energy-Aware Software Systems-Part 3

(Refer Slide Time: 00:21)

Welcome to the lecture part 3 of energy aware software systems. So, today we are going to look at Microarchitectural Techniques to save energy and move on to how compliers are going to be useful in saving the energy using the Microarchitectural Techniques.

(Refer Slide Time: 00:36)



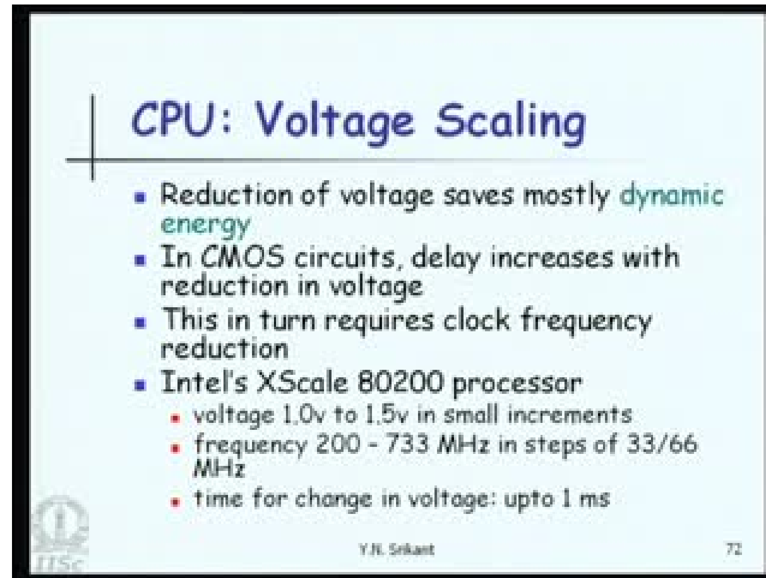
For example, the CPU and memory are the energy guzzlers in a computer system. So, on the CPU front, we can use voltage and frequency scaling, change the voltage and frequency of the CPU. We can use supply voltage getting of function units inside the CPU. We are going to look at each of these a little more in detail.

Then, we can gate the clock of function units and we can do some bus encoding for the buses inside the CPU. On the memory front, a new type of memory called the drowsy cache can be utilized. It has a facility to switch off power supply to several cache lines which are not active.

Compression in the instruction cache is possible to save energy because if we compress instructions, then the space occupied will be much lesser. So, the access will be faster and less energy consuming.

Cache region reservation and partitioning, scratchpad memory - these are some of the techniques which are possible at the Microarchitectural Level.

(Refer Slide Time: 02:00)



What is CPU voltage scaling? It basically reduces the voltage of the CPU and this saves mostly dynamic energy of the CPU. It cannot change the static energy consumption of the CPU. In CMOS circuits, the delay inside the CPU will increase with reduction in voltage.

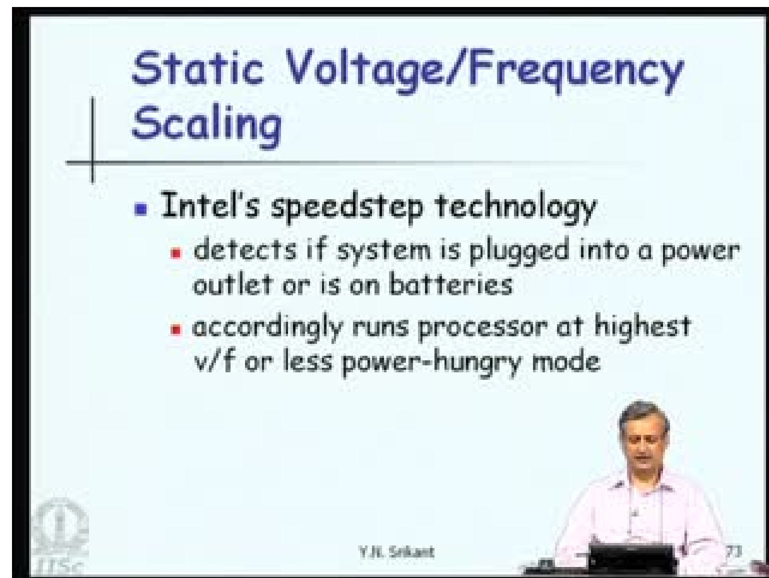
So, we definitely have will be forced to reduce the clock frequency, if we reduce the voltage. This in turn reads to performance degradation; so, the challenges in voltage scaling would be to retain the same level of performance, but with reduced energy consumption.

For example, inside the Intel XScale 80200 processor, voltage can be change from 1.0 to 1.5 volts in small increments. Automatically, the frequency will also change from between 200 to 733 Megahertz. In other words, the highest frequency may be 733 and the lowest frequency would be 200 in steps of 33 or 66 Megahertz. So, if we increase the voltage, frequency increases; if we reduce the voltage, frequency also reduces.

Another important factor is the time for change in voltage. There is hardware inside the CPU in the voltage regulator, which has to stabilize its output when we want the voltage to be change from 1 step to another step; this requires a finite amount of time. This is quite large, for example, up to 1 millisecond.

The reason is there are capacitances inside the voltage regulator; these require time to charge and discharge. That is the reason why this change in voltage requires some extra time.

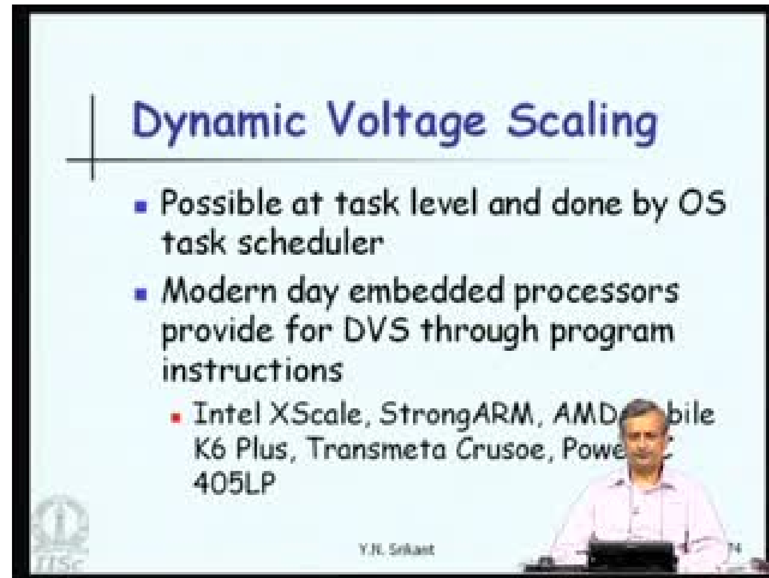
(Refer Slide Time: 04:10)



We will look at static voltage scaling which implies we change the voltage of the CPU right in the beginning of running the program and we do not change it during the running of the program itself.

This Intel's speed step technology which detects if the system is plugged into a power outlet or is a battery, and accordingly runs the processor at highest voltage frequency or less power hungry mode, thereby using less voltage frequency it actually saves battery power.

(Refer Slide Time: 04:54)



Then, there is dynamic voltage scaling. So, this is possible at task level and is done by the operating system task scheduler. So, of course, it is also possible to do it inside the program; we will see that a little later, but that requires compiler help.

Modern day, embedded processors provide for dynamic voltage scaling through program instructions. For example, Intel XScale, StrongARM, AMD mobile K6 Plus, Transmeta Crusoe and Power PC 405 LP - these are some of the processors which have instructions to change voltage.

So, when the program is running, it is possible to change the voltage of the CPU. It is also possible for the operating system to execute these instructions and change the voltage of the CPU.

(Refer Slide Time: 05:51)

CPU Clock gating

- Clock gating of Function Units
  - Make FU clock zero during idle period
  - Reduces dynamic energy usage, but static leakage remains
  - Both circuit-level techniques and compiler techniques are possible

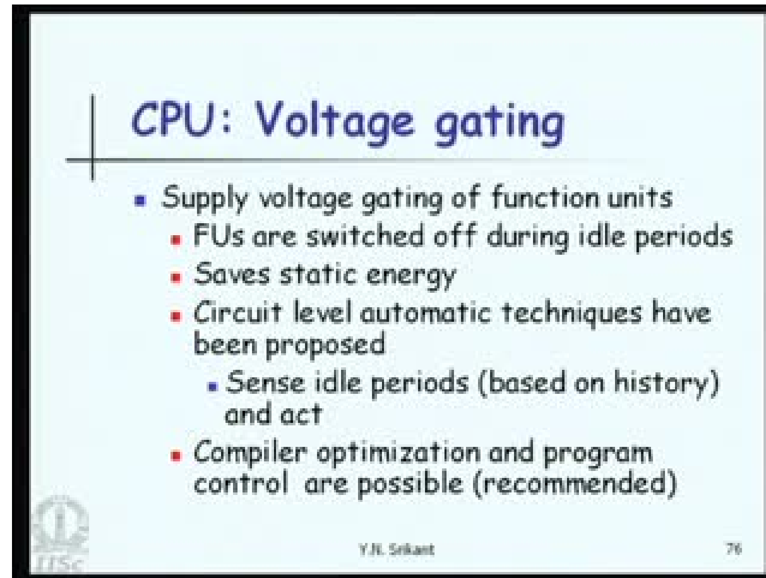
Y.N. Srikant 75

Then I mentioned CPU Clock gating. Basically, if you gating implies stopping, so the clock to the function units is stopped implies that the function unit cannot run at all. These are all synchronous systems; they require a clock in order to run.

Make the function unit clock 0 during the idle period. So, if the clock is 0 during the idle period, then the clock energy would be reduced. It also reduces the dynamic energy of the function unit, because the function unit cannot run when the clock is zero.

It reduces dynamic energy usage, but the static leakage remains. Not much can be done about it unless we cut the voltage of the function unit itself. The change in the energy due to clock gating is not really great, but it definitely helps; we need to save energy in every possible way so that the total sum of energy saved due to several techniques is reasonable.

(Refer Slide Time: 07:24)



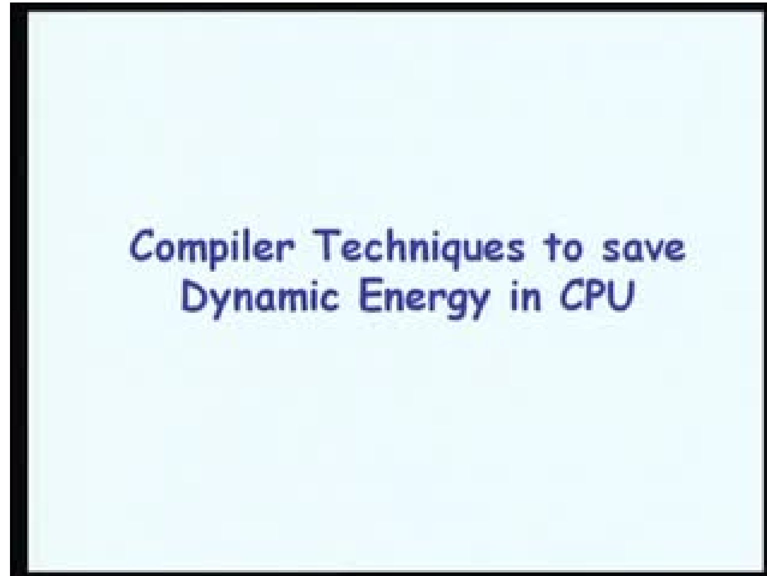
Both circuit-level and compiler level techniques are possible in this case for gating the CPU Clock; we will see that a little later. So, supply voltage gating is another possibility for function units; what we saw before was supply voltage gating for the whole CPU.

Now, are rather supply voltages scaling for the whole CPU? We are looking at supply voltage gating of function units. So, function units are switched off during idle periods. So, this implies that even the static energy is going to be saved and circuit level automatic techniques have been proposed for this purpose.

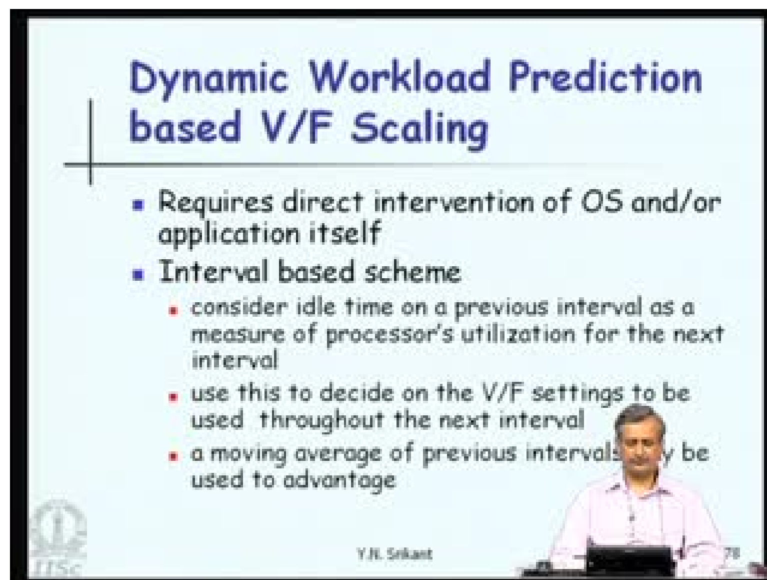
So, they have a mechanism to sense idle periods. They see how many cycles the function unit has been idle and based on that the algorithm switches off the supply voltage to the function unit. Let us say two cycles of inactivity, they trigger switching off of the supply voltage to the function unit and then it remains in that state until the function unit is required again.

So, once it is required, the supply voltage is reactivated and the function it becomes functional. But both switching off and switching on requires some finite amount of time. Now, that has to be factored in when we want to actually use this technique in the architecture or through the compiler mechanism.

(Refer Slide Time: 09:05)



(Refer Slide Time: 09:12)



Now, let us look at compiler techniques to save dynamic energy in CPU. The first one is not strictly a complier technique, but has been mentioned because it is widely referred to in the literature. We accumulate the work load history, what is the load on the CPU, for how many and if during this particular interval the CPU has not been working at its best. For example, its activity factor is probably less, may be 0.5, 0.4, and 0.6.

In such a case, it is possible that by running the CPU at a lower voltage and frequency during the next interval. We are going to save some energy, but at the same time, stretch

the activity factor to 1 and make sure that the CPU itself, rather the program itself does not suffer in performance.

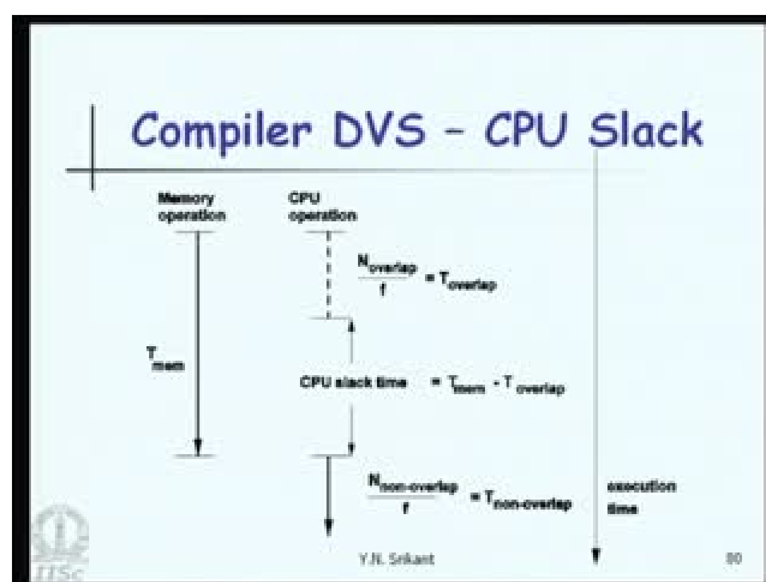
If this requires direct intervention of the operating system and or application, it in interval based techniques. It considers, as I said, the idle time on a previous interval as a measure of processor's utilization for the next interval.

So, the assumption is whatever happened in this interval will happen in the next interval also. Use this to decide on the voltage frequency settings to be used throughout the next interval. It is also possible to use a moving average of the previous intervals and this came possibly give some better advantage.

Inside the compiler, how do we incorporate a dynamic voltage scaling technique? The basic idea is to make different parts of a program, operate at different voltage frequency pairs. During CPU stall, that is when memory operations are happening and the result of the memory operation is required for the next instruction to execute and therefore the CPU is doing nothing.

During such a stall, it is possible to scale down the CPU voltage and frequency, save energy, but performance will not be degraded because CPU is really doing nothing. So, we can reduce its voltage and frequency.

(Refer Slide Time: 12:01)

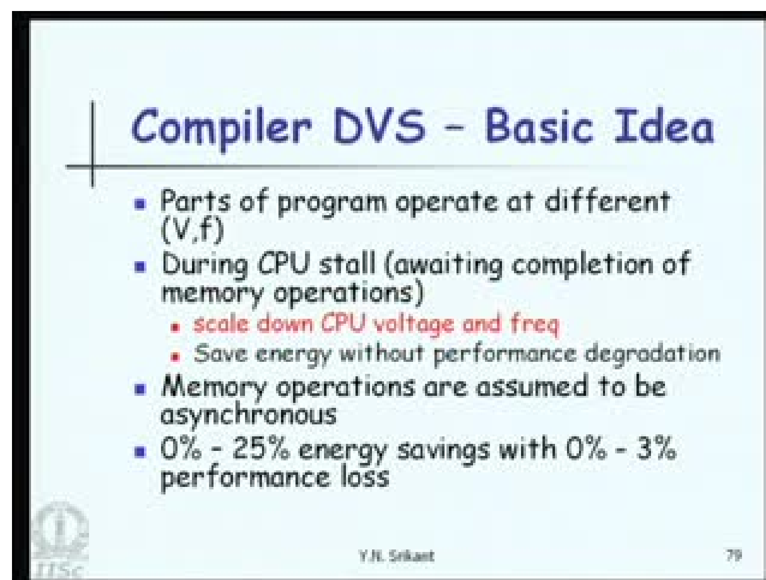


Let me show you a picture to make this very clear. The memory is supposed to operate at a constant voltage and frequency. So, let us say, the memory operation is going this way and it requires T mem cycles to finish.

The CPU operation is happening. So, its starts here; it goes on until this point. Then, until it gets the memory operand, it can do nothing. So, it stalls and waits until the memory operation is completed, receives the memory operand and then continues its operation.

Now, there is nothing wrong, if we stretch this period - which is actually the useful activity region of the CPU - up to this point by reducing the voltage and frequency of the CPU; so, because during this period, the CPU is doing nothing. We can stretch this period up to this period, make the CPU run at a lower speed and thereby we actually during this entire period anyway CPU was active. Now, we have reduced the voltage; the time remains the same; so, we have saved energy. At this point, we increase the CPU speed, the frequency and voltage. So, that the performance is not degraded. This is the basic idea behind the voltage scaling.

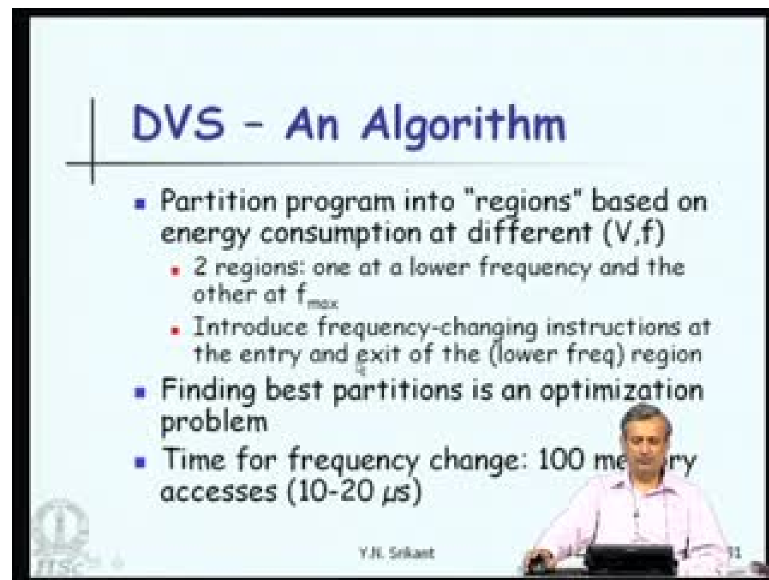
(Refer Slide Time: 13:33)



Memory operations are assumed to be asynchronous. So, they go on independently of the CPU and they always operate at the highest voltage frequency. It is the CPU whose voltage and frequency will be change. Schemes of this kind have been shown to provide 0 to 25 percent energy saving with 0 to 3 percent performance loss.

It is possible that the whole program has to be run at a single frequency - the highest possibly and it may not be possible to divide every program into couple of regions which run at different frequencies. That is the reason why, the savings vary between 0 to 25 percent.

(Refer Slide Time: 14:22)

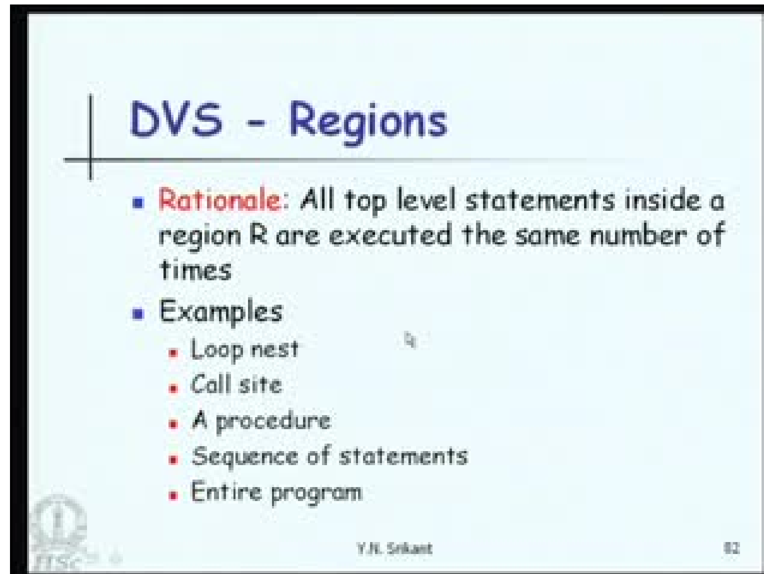


Now, how does this perform? So, we try to partition a program into regions based on the consumption of energy at different voltage and frequencies. It has been shown, that two regions are usually sufficient: one of the region operates at a lower frequency and the other operates at the maximum frequency.

Introduce the frequency-changing instructions at the entry and exit of the lower frequency region. In other words, when the program control enters the lower frequency region, the frequency is changed to a lower one and when it exits the lower frequency region, the frequency is changed again to the higher one.

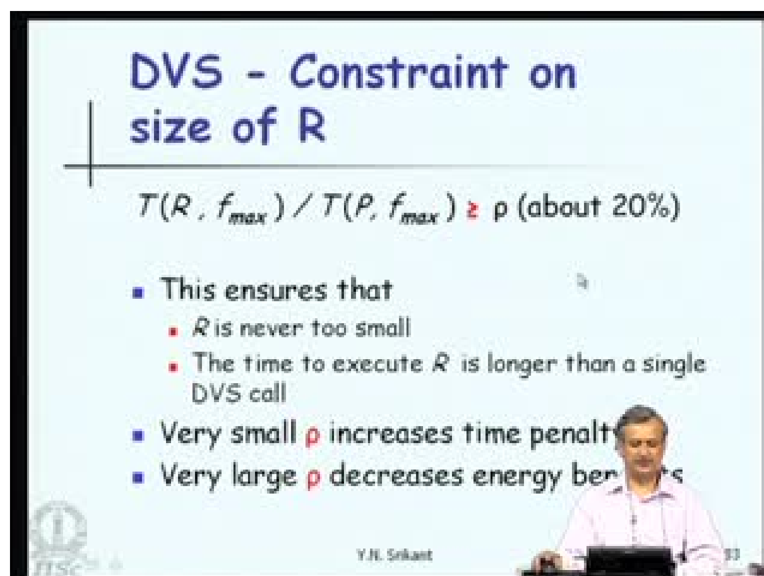
Finding best partitions in this fashion is an optimization problem. We need to factor in the frequency change timing: 100 memory accesses may be needed 10 to 20 microseconds depending on the processors frequency. In order to perform this change, I already mentioned that, this is due to the voltage regulator and its capacitances.

(Refer Slide Time: 15:39)



What is a region? All top level statements inside a region R are executed the same number of times. This is the rationale behind dividing a program into regions. So for example, a loop nest is a region, call site, a procedure sequence of statements or entire program; they are all regions. So, this is an informal description of a region, but how do we use this.

(Refer Slide Time: 16:07)



So, what we really do is, let us assume, that we have built or rather partitioned the program into 2 regions. Now, given that there are 2 regions: there is a region R and there is a whole program P.

There is a constraint on the size of the region R. In other words, T (R, f max) is the time required to run the region R at the highest frequency f max and T (P, f max) is the time required to run the whole program at f max.

So, T (R, f max) divided by T (P, f max), the ratio of these is required to be greater than or equal to rho, which is usually about 20 percent. Why do we want this constraint? The point is, if we have a very small region, then the number instructions executed in that region will be very small. So, the time that the CPU spends inside the region R will be very small.

If the region is very small, then the overheads of the switching; that is we have to switch to a lower frequency and then again to a higher frequency at the borders of the region R. This overhead may become quite substantial. In order to make sure that this overhead is not very high, we make the region of a reasonable size.

If the region is very large, then the energy benefits are going to be very low. We have to make the region of reasonable size somewhere in between. So, in the optimization problem, there is a constraint on the region which is set by experimentation for each processor is could be different. So, we set it to approximately 20 to 25 percent.

(Refer Slide Time: 18:24)

DVS - The Minimization Problem

$$\min_{R,f}\{ P_f . T(R,f) + P_{fmax} . T(P\text{-}R, f_{max}) + P_{trans} . 2 . N(R) \}$$

subject to

$$\{T(R,f) + T(P\text{-}R, f_{max}) + T_{trans} . 2 . N(R)\} \le \{(1 + r) . T(P, f_{max})\}$$

r : performance degradation toler

Y.N. Srikant

What is the minimization problem? So, it says minimize over all regions R and all frequencies f. We have to find the region R, let us see later how to find it. For a given R, we need to experiment with all possible frequencies f. What is it, which we need to minimize? The energy: total energy of the CPU.

The energy of the CPU or the program is in 3 parts: the first part is powers spent by the region R, at frequency f multiplied by the time spent by region R, at frequency f. So, power into time is energy. This is as far as region R which is executed at lower frequency f.

Then, we have the region P minus R, which actually operates at the highest frequency. So, the power consumption, at this level is P f max and T of P minus R comma f max, is the time spent in the rest of the region in the program; that is P minus R.

The last part is the energy spent for switching. So, P of trans into 2 into N of R. N of R is the number of times the region R is executed. There is a factor 2 here, because we need to switch twice. We need to switch to the lower voltage, when we enter the region and we need to switch to the higher voltage, when we exit the region. P trans is the power consumed during switching.

This is the total energy consumption for the program. This must be minimized over all possible regions and frequencies subject to a constraint. The constraint is the total time

required to execute the program should be below a certain threshold. So, r is the performance degradation tolerated, let us say, 5 percent. T (R, f) is the time required to execute region R, at frequency f. T P minus R comma f max is the time required to execute the rest of the program - P minus R, at the frequency f max.

Then, we have the switching time. T trans into 2 into N R, the time required for switches. This entire thing is the time for the program with a lower voltage and frequency selected for a region. This time should be less than or equal to 1 plus R into T of P comma f max. So, assume that we did not do any reduction and voltage for any region in the program.

Then, the time required would be T of P comma f max at the highest frequency. So, we are willing to spend a little more time. So, that is why the 1 plus R factor. So, 1 plus R into T of P comma f max, is a little more than T of P comma f max and we do not want the degradation to be more than this particular specification of r, say, 4 to 5 percent.

(Refer Slide Time: 21:59)

DVS - Implementation

- Code instrumentation for basic regions (call sites, loops, if-else)
  - to measure $T(R, f)$ accurately (using a high precision timer) and $N(R)$ at different frequencies
- P_{fmax} is either measured directly or using a simulator (*Wattch*)
- P_f is obtained from P_{fmax} using interpolation
- Combining basic regions using simple composition rules
- All the region combinations are enumerated and the optimal one is chosen

Y.N. Srikant

How does one implement dynamic voltage scaling? So, there is going to be code instrumentation for very basic regions, such as: call sites, loops, if then else and so on. The instrumentation measures, the time R comma f. So, assuming that the region is small, we require a very high precision timer. So, T(R, f) is the time required to execute region R, at frequency f and N R is the number of times region R is executed. This can be measured using a counter.

Code instrumentation is needed to start and stop the timer and also increment the counter at the various basic region levels. P of f max is the maximum power consumption - this is steady power consumption; let us, assume that. So, if once we know that the power consumption is steady, we can either measure it directly using a wattmeter connected to the power supply or we could use a simulator such as, Wattch to tell us what is the power consumption of the processor?

Power consumed at various frequencies to simplify the matter is obtained through interpolation using P of f max. So, change in frequency is assumed to be proportional to change in power. So, we are going to use that equation in order to interpolate and find the power consumption at various frequencies.

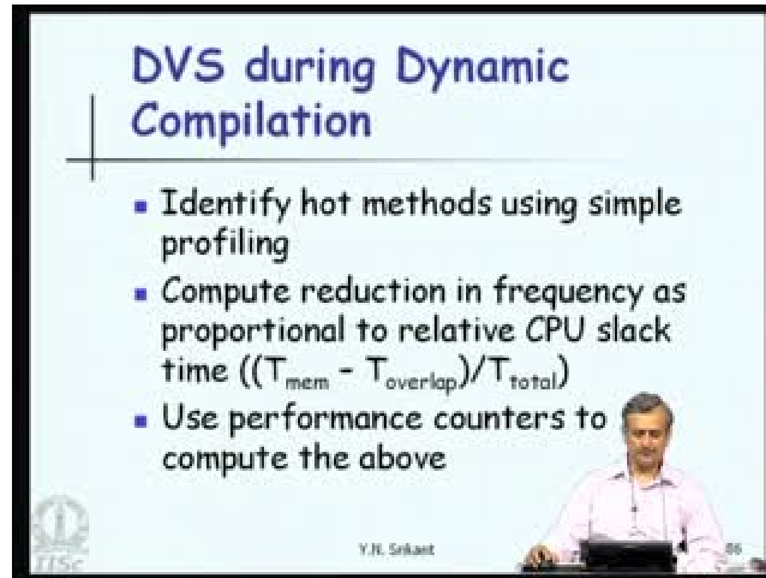
If this is not going to be linear, then it is possible to change the frequency, run the program at that particular frequency and measure the power. We could do this for a couple of frequencies and then fit a curve and calculate the power consumption at other frequencies using this fit at curve.

We have basic regions, but we do not want such very small regions - regions have to be of reasonable size. So, we must provide a simple composition mechanism for combining basic regions.

So for example, we could combine different segments of the program into one using program composition rules. So, the then part and the else part could be combine into 1; 2 loops could be combine into one and so on.

These are the usual programs syntax composition rules that we are going to use for producing bigger and bigger units. So, in the same way as a parser, starts from the smallest expression and builds up parse tree for the whole program. We are also going to use the same rules; combine smaller basic regions into bigger regions. Then compute the power and time requirements of the composed regions. All the region combinations are enumerated and the optimal one is chosen. This is not going to be too large. So, we can afford to do this; otherwise an integer linear program or some other formulation can be utilized as well.

(Refer Slide Time: 25:43)



So, that was about changing voltage during run of the program, rather using run of the program, but determined by the compiler. So, what happens if we have a dynamic compiler, in other words, for example, a just in time compiler for java, is also a form of dynamic compiler. So, what happens in such a compiler, there are java byte code instructions, which are actually translated into machine code during the interpretation of the java byte code.

Therefore, there is no separate time available for us to analyze the program like we proposed in the previous scheme. Then, insert instructions to change frequency voltage etcetera. It is also possibly superior to change the frequency and voltage during by accumulating information during the run of the program, rather than by the compiler.

So, just in time compilers and dynamic compilers try to do this. They try to accumulate information about, which methods are very hot or much executed very frequently using simple profiling methods. Then, they compute reduction in frequency as proportional to the relative CPU slack time. The time required for accessing memory minus the overlap time, which is a useful activity of the CPU - that is, the overlap. So, T mem minus T overlap is proportional to the CPU slack. So, divided by T total is really the ratio, which is used as a measure of the slack of the CPU.

This relative CPU slack time is used to compute the reduction in frequency. Let us see, how T mem and T overlap are computed, we cannot be using very heavy instrumentation

to compute this. We are going to use the performance counters available inside the CPU, which actually have no overheads at all to compute the CPU slack time.

(Refer Slide Time: 28:25)

DVS during Dynamic Compilation

- $T_{mem} / T_{total} \approx$ k_1(#mem_bus_transactions)/ #μops_retired)
- $T_{overlap} / T_{total} \approx$ k_2(#FP_INT_instructions)/ #μops_retired)

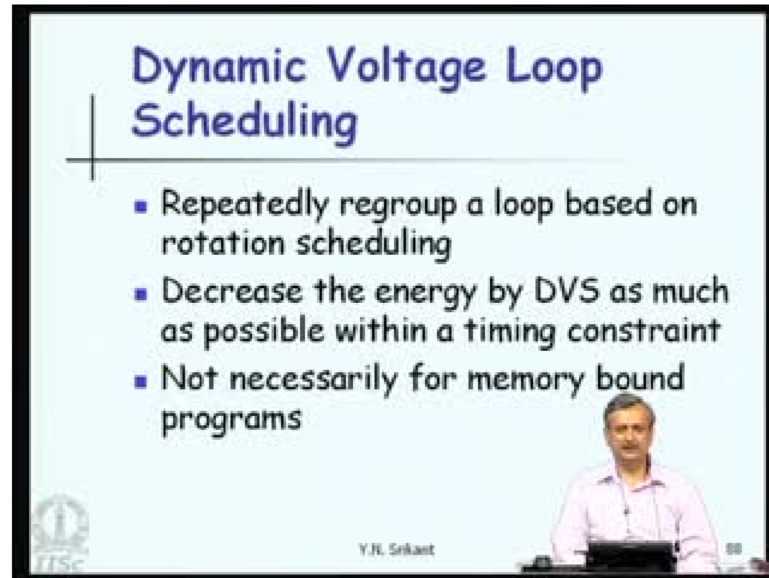
Y.N. Srikant

For example, T mem by T total can be computed as k 1 into number of memory bus transactions divided by number of micro operations retired. So, both these number of memory bus transactions and number of micro operations retired are given by two different performance counters.

So, what is involved is to reading those performance counters. k 1 is a constant which is set by some experimentation. Similarly, T overlap by T total is proportional to k 2 into floating point and integer instructions divided by number of micro operations retired. So, again there is a performance counter which gives us this information and also this information. We could compute a quantity which is proportional to T overlap by T total.

So, computing the slack or relative CPU slack is a fairly straight forward process, once we compute these 2 quantities. So, remember these are obtaining from the performance counters so, this is a fairly straight forward operation.

(Refer Slide Time: 29:37)



So, that was about using dynamic voltage scaling inside a dynamic compiler. Let us see, how dynamic voltage loops scheduling is done. Why should we look at a different technique for loops? The reason is twofold: one is, suppose, we have program which are not necessarily memory bound. So, the previous technique assumed that the programs are memory bound, that is, the CPU is waiting for a memory operation and therefore, it has very little to do; during this time we reduce the frequency and voltage of the CPU; suppose that was not so.

The program was not necessarily memory bound. The other reason is, we could have multicore processors and we still have not seen a technique which can be used for multicore. This particular technique is can be used both on multicore and single core processor. The strategy is to repeatedly regroup a loop based on what is known as a rotation scheduling.

I will give an example to show, what rotation scheduling is very soon. Decrease the energy by dynamic voltage scaling as much as possible within a timing constraint and this is not necessarily for a memory bound program. So, what exactly is rotation of a loop?

(Refer Slide Time: 31:17)

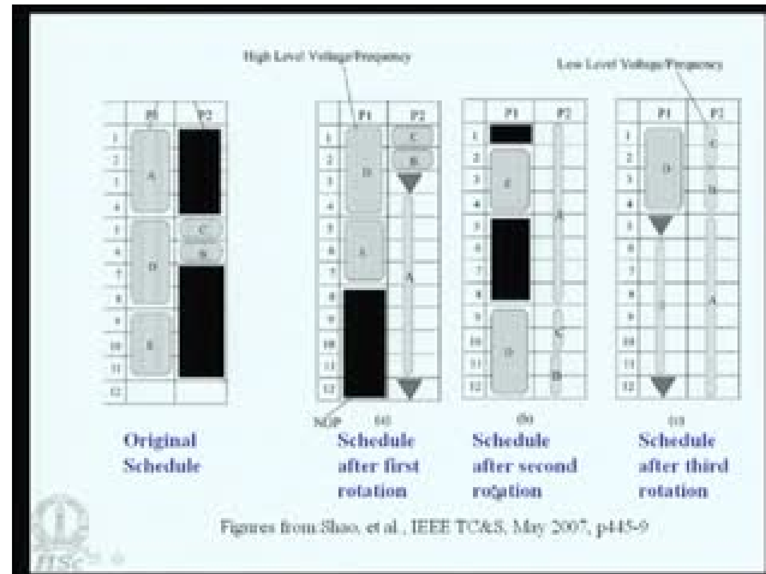
Let's assume that we have a loop of this kind. There is some initialization here and then, we have a for loop with 5 instructions. Remember, that we have the order of these instructions A B C D E. So, loop rotation implies that we change the order of the instructions of the loop subject to the dependents requirements of the various instructions.

So for example, this instruction one of them was taken upwards; so it is here. The loop started with the second instruction onwards and the A i instruction was introduced at the bottom. So, when the loop starts execution, this is executed for the first iteration and then, these are executed and this will be the instruction for the second iteration.

We have changed the instruction subscripts also. It was A i equal to E i minus 3 star E i minus 3 whereas, now it is A i plus 1 equal to E i minus to the star E i minus 2; others have not changed. So, there is a prolog which of some size and then there is an epilog for the rest of the instructions in the last iteration.

Because we have already executed one instruction in the last iteration, the rest of them have to be executed outside the loop. So, this is rotation of the loop by 1, because this instruction has come to the bottom like this. Suppose, we rotate the loop twice, then this instruction will also go up. We start with this instruction and B would actually be placed after A and there would be appropriate epilog here. So, this is the rotation of a loop.

(Refer Slide Time: 33:28)



How does loop rotation help? Let us consider a 2 core processor: there are 2 processors; 2 CPU's and the hatched area indicate that the processor is running at the highest frequency in voltage. The black area indicates that the processor is doing nothing.

So, to begin with for some program, not necessarily the program that I showed you just now; processor 1 is executing the 3 regions: A D and E, and then the processor number 2 is doing nothing for certain duration of time. Then, it is executing C and B; then again it is doing nothing.

The number of time slots is really 12; this cannot be changed, this is a time requirement - time constraint. Suppose, we rotate the loop once, the question is, is it possible to actually do some dynamic voltage scaling and get some benefit out of it.

Now, the processor 1 executes D and E. A is actually scheduled on processor 2. So, processor 2 executes C and B, and then, it reduces the voltage. That is why, there is a linear bar here, executes A in lean mode - that is, lower frequency mode and then, again changes the voltage back to its normal.

Now, during this time, the program is really doing nothing on processor 1. During this time, there is some saving because we have actually reduced the voltage of the processor number 2. Let us do, loops rotation once more.

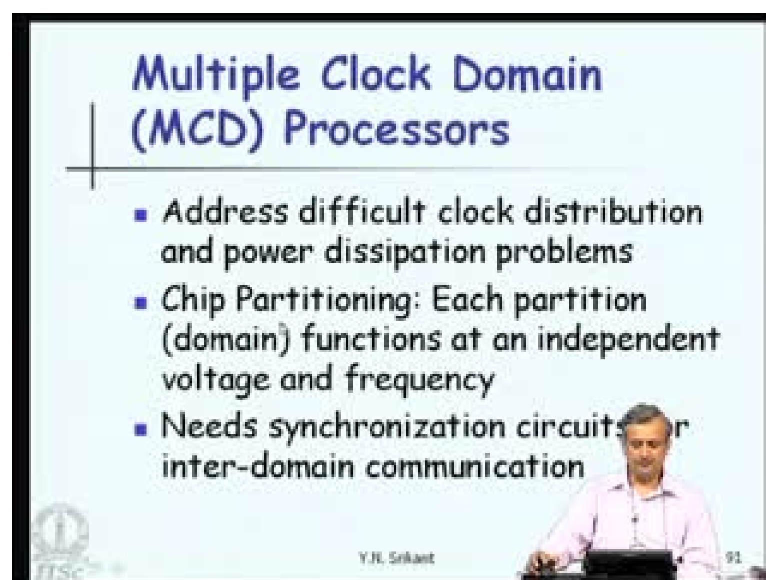
Now, processor 1 does very little here; then it executes E; then it executes D and in between it does nothing. Processor number 2 actually runs the entire duration of 12 cycles. It executes in lean mode, at a lower frequency so, there is a substantial amount of reduction in energy consumption.

If we do, let us say loop rotation once more. In this particular example, we are able to reduce run D on processor 1, then reduce the voltage and run E on processor 1. Then, the processor number 2 runs the other C B and A in lean mode. So, there is saving here and also here. So, in this fashion, we are able to reduce a substantial amount of energy. This only an example, which shows that the third rotation blimps out the maximum benefit; it is not necessarily show in all programs.

It is possible that, only one rotation is allowed in certain loop; 2 may be possible in another loop; 3 may be possible in one more. The number of rotations is dependent on the dependencies in the program and is not necessarily always fixed.

But this example shows, that loop rotation enables slowing down processors without reducing the amount of time. So the time is the same; power consumption is less, so, energy is saved. So, that is loop rotation and scheduling of the parts of the program, rather parts of the loop in order to save energy. Again, we must stress that we are saving dynamic energy and we are not really saving any static energy in this case.

(Refer Slide Time: 37:52)



Let us look at a third type of processor. We saw a single core, multi core and now, multiple clock domain processors. What is an MCD processor? The basic idea is different parts of a CPU will run at different clock frequencies. Why is this needed and how does it solve any problems at all if a t.

In general, this type of MCD processor addresses difficult clock distribution and power dissipation problems. So, it is well know that in a large CPU or large chip, in general, supplying the same clock to all parts of the CPU is very difficult; it is not easy. It is simpler if separate clocks are used in different parts of the CPU, but then clock synchronization between these is a very difficult problem. So, it is not easy to use such processors. Now, why should we actually have large chips? Obviously, the larger the chip, more chip area and more electronics that can be put on it. So, power dissipation becomes very large if we have a very large chip.

We are going to have smaller regions of the chip, which have different clocks. So, the clock distribution electronics and the CPU area needed for clock distribution itself, increases power dissipation. So, if we use local clock in different parts of the CPU; this clock power dissipation also comes down.

So, we are going to partition the chip into domains. So, each partition functions at an independent voltage and frequency, so that is the assumption in an MCD processor. It needs synchronization circuits between for inter-domain communication such as buffers, queues and so on.

(Refer Slide Time: 40:07)

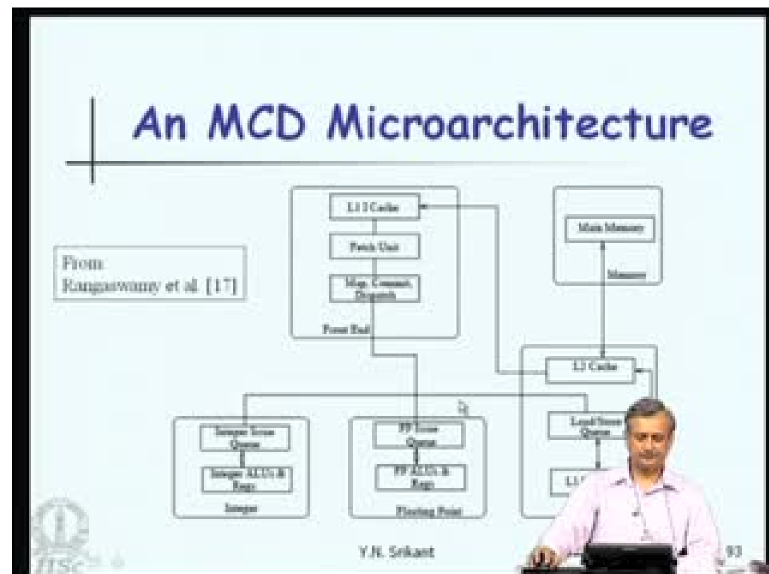
Multiple Clock Domain (MCD) Processors (contd.)

- Domains whose performance is non-critical, can operate at lower voltage and frequency: potentially less power dissipation
- Performance costs for synchronization: not very significant for out-of-order processors

Y.N. Srikant 92

Domains, whose performance is non-critical, can operate at lower frequencies and therefore, they will consume potentially less power. Performance costs for synchronization are not very significant for out-of-order processors. So, if somebody builds MCD processors; it may still be useful.

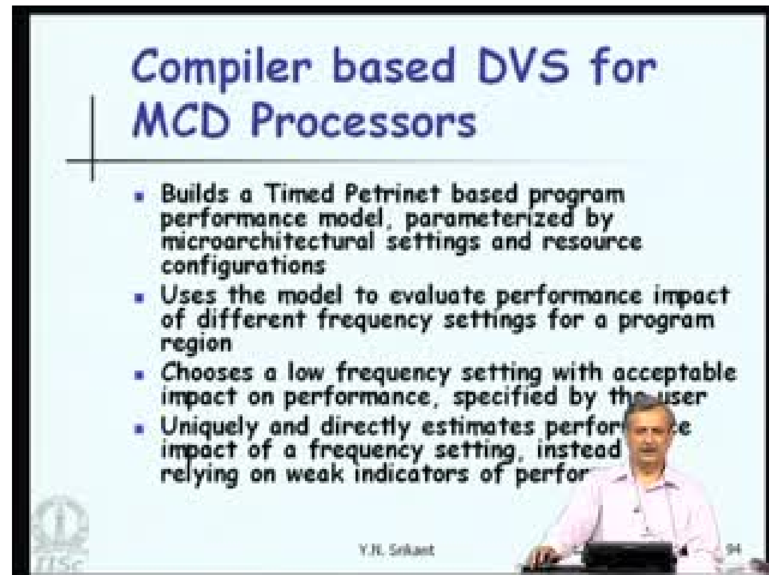
(Refer Slide Time: 40:31)



Here is an example, so there is an MCD Microarchitecture. So, there is a front end which does fetch and decode; so, this is one domain; memory itself main memory is another domain. The L1 data cache, load queue and L2 cache form one more load domain, then,

there is a floating point domain and integer and register domain. So, this is a 5 domain processor.

(Refer Slide Time: 41:07)



How does one use the compiler technique for doing the DVS on such processors? Basically, the technique builds a Timed Petrinet based program performance model; this Petrinet model is parameterized by microarchitectural settings and resource configurations.

I cannot provide a tutorial on Petrinet for lack of time, but it surprises that a Petrinet can model concurrency; it can model nondetermine some and so on. This model is used to evaluate the performance impact of various frequency settings for a program region.

During this evaluation, we really do not have to run the whole program, but it is necessary to just run the program. It rather run the simulation of the Petrinet and watches the performance of that processor using the Petrinet model for certain duration.

So, using this duration based simulation; it is possible to evaluate the performance of the CPU at various frequencies. So, we choose a low frequency setting with acceptable impact on performance this impact can be specified by the user.

The advantage of this particular scheme is that, it uniquely and directly estimates performance impact of a lower frequency setting instead of relying on weak indicators of

performance. So, directly we try to simulate for certain duration and measure the performance.

(Refer Slide Time: 42:59)

Results

- SPEC FP: Many L2 misses; ED improvement - CDVS saves 60.39%, while meeting performance constraints; Profile-based DVS saves 33.91%
- Media Benchmarks: almost no L2 miss; ED^2 improvement of CDVS (PDVS) : 22.11 (18.34%)
- Hardware-based DVS saves less energy; relatively better in media benchmarks where queue occupancies of FP and LS domain are low

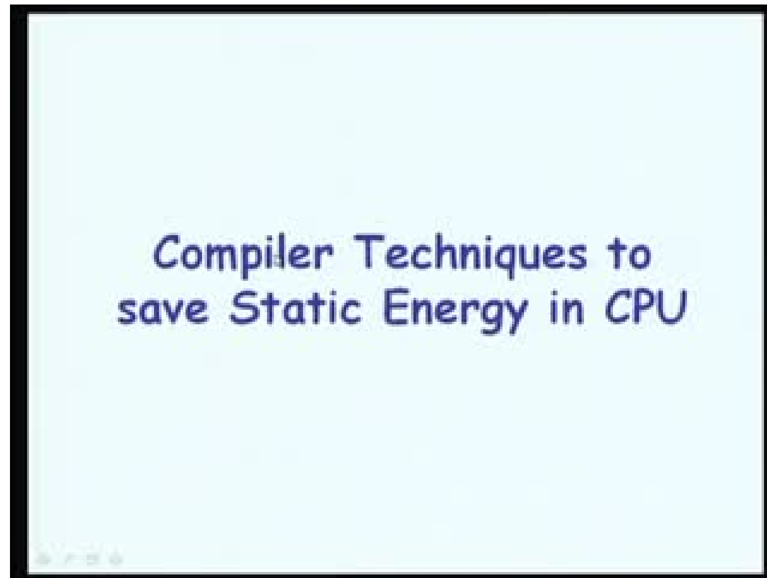
Y.N. Srikant 95

The results available for MCD processors using this Petrinet model based technique. So, for the SPEC FP benchmarks: there are many L2 misses in this bench mark. So, energy delay product improvement is very large. Our technique uses saves 60 percent energy, while meeting performance constraints; whereas, existing hardware profile-based dynamic voltage scaling techniques save an only about 34 percent.

Media Benchmarks have almost no L2 misses; so here using ED has a metric does not help much; so, we use the ED square a measurement metric and improvement in our case, is 22 percent whereas, in the case of profile based hardware technique is about 18 percent.

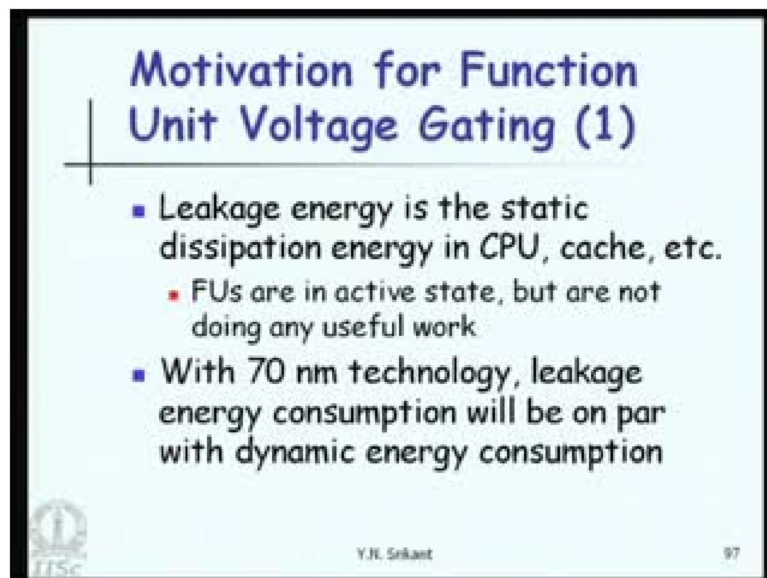
In general, hardware based DVS saves less energy; relatively and it is better in media benchmarks and whereas, the queue occupancies of floating point and the load store domain are very low in media benchmark. That is why; it gives slightly the profile based hardware technique gives much better results. So, our technique is found to be very good if there are many misses L2 misses such as in the SPEC FP bench mark.

(Refer Slide Time: 44:27)



Now, let us start looking at Compiler Techniques to save Static Energy in the CPU. So far, we have seen techniques which saved dynamic energy. So, we did not see any techniques to save any static energy.

(Refer Slide Time: 44:43)



Function unit voltage gating is one of the forms, which can save this static energy; that is supply voltage to the CPU for the functionality switched off. Why should we do this? I mentioned this in the introduction: Leakage energy is the static dissipation energy in

CPU cache etcetera and function units are in active state, but are not doing any useful work. That is the time when we actually want to save energy.

With the 70 nanometer technology, leakage energy consumption will be on power with dynamic energy consumption, whereas, above 70, say, 90 around 35, older technologies dynamic energy consumption was the maximum; rather it was actually the major contributor to energy consumption whereas, static energy consumption was very low.

With lower level technologies say 65 and then 32 etcetera. The leakage energy is going to be high. We need techniques to save leakage energy, because whether we do anything useful in the CPU or not; it will still consume as much energy as when the CPU is really doing something useful.

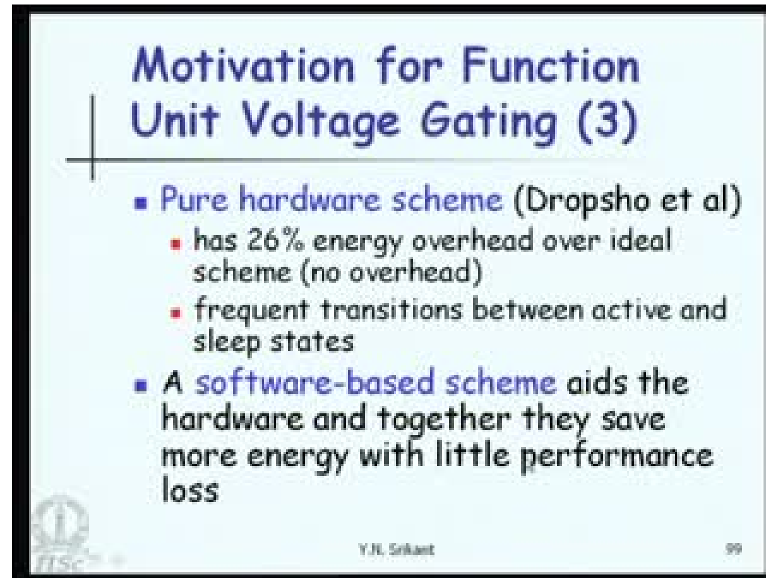
Dynamic energy consumption and static energy consumption will be on power. So, each will contribute 50 percent of the energy consumption. So, we need to save both of them in order to get the maximum benefit.

So, there is something called dual-threshold domino logic with sleep mode. This is an electronic circuit and this can facilitate very fast transitions between active and sleep modes. In other words, when a function unit is in active state; the full voltage is supply to it and when it is in sleep state; the voltage is made 0. So, this is the transition that we are talking about between active and sleep state.

But the nice thing about these dual-threshold domino logic circuit is; they can actually switch between these active and sleep states with very little performance penalty; very moderate energy penalty. So, observe the words, there is not much performance penalty, because we are going to switch within 1 cycle. So, active to low is 1 cycle; low to active is another cycle, but the once the switching time is very low, energy consumption becomes high. There is moderate energy penalty for such switching.

Now, the problem is if the switch requires a fair amount of energy, then is it useful to put the CPU in sleep mode very frequently; obviously not. So, the 1 cycle of idleness is all that we required putting it into low leakage mode, and we also know the factor that motivates us is, integer ALU’s are known to be idle for 60 percent of the time on the average. That is, the integer ALU’s to very little for 60 percent of the time in general.

(Refer Slide Time: 48:23)



There are two schemes: one is a pure hardware scheme proposed by Dropsho. So, they have these dual threshold domino logic circuits. They watch the CPU rather the ALU for 1 cycle and if the ALU is idle for 1 cycle or more; it is put into sleep mode. Automatically, the hardware takes care of switching from low to high and high to low.

The problem is, there is 26 percent energy overhead, over the ideal scheme with no overhead. The switching causes 26 percent overhead; assuming over the scheme which assumes that there is no overhead. There are very frequent transitions between active and sleep states.

So, if software-based scheme such as a compiler based one aids the hardware; then together they can actually save much more energy with very little performance loss.

(Refer Slide Time: 49:38)

Compiler - CPU function unit voltage/clock gating

- Try to bunch instructions which use the same FUs so that "active"and "idle"periods of FUs are increased
- CPU uses supply voltage/clock gating during idle periods
- Leads to better benefits o[illegible] saves transition energy

Y.N. Srikant 100

What is it that we are trying to do here? This technique can be used both for voltage gating and also clock gating of the CPU. So, the CPU is active for some time. Now, we try to bunch instructions, which use the same CPU rather same function unit. So, that the active and idle periods function units are increased; let me elaborate. The function unit is executing some operations. Now, in the program which has not been transformed by any compiler technique, it is possible in the worst case that there are 1 or 2 ALU operations. Then, there is an idle period of 1 or 2 cycles; another ALU operation another idle period and so on. And because the idle period is more than 1 cycle; the hardware switches it to low state and then again to high state, whenever the FU's needed.

Suppose, the ALU instructions which are executed after idle period - two consecutive idle periods are not dependent on each other. In such a case, we can actually push these ALU instructions together, rather the CPU instructions together so, that the function units which was busy in executing 1 or 2 instructions.

Now, continues to execute the next bunch of instructions also, because there was no dependency. The instructions scheduler brought them together. So, the ALU now keeps itself busy as long as possible. That is, as long as there is no dependency on the next instruction and it is force to wait.

So, assuming that there was an idle period between consecutive two instructions, now there will be no idle period between these two instructions. The other advantage is,

because we have made the function unit busy for a longer time. The idle periods actually add up together so, idle periods increase and the busy periods also increase. That means the switching between the active period and idle period has reduced.

So, if there were 4 or 5 switches between; now, it probably reduces to 1 or 2. So, there by, the number of switches reduces and the energy consumed during this switching will also reduce. So, the CPU uses supply voltage clock gating during this idle periods, that is, done by the hardware whereas, the rearrangement of instructions so, that the instructions which use the same function unit are bunch together is done by the compiler.

(Refer Slide Time: 53:06)

Instruction Scheduling

- Reordering instructions
  - To reduce pipeline stalls
  - To exploit instruction level parallelism
- NP-complete (with resource constraints also handled)
- Uses a DAG and is limited to basic blocks
- List scheduling with a ready queue most common approach

Y.N. Srikant 101

So, this leads to better benefits and it saves transition energy in the CPU as well. So, we will stop the lecture at this point and next time we will look at details of the instructions scheduling mechanism which saves energy to a large extent.

So, this is based on the same list scheduling strategy that we have discussed before, with the heuristics for priorities etcetera being different. Thank you.