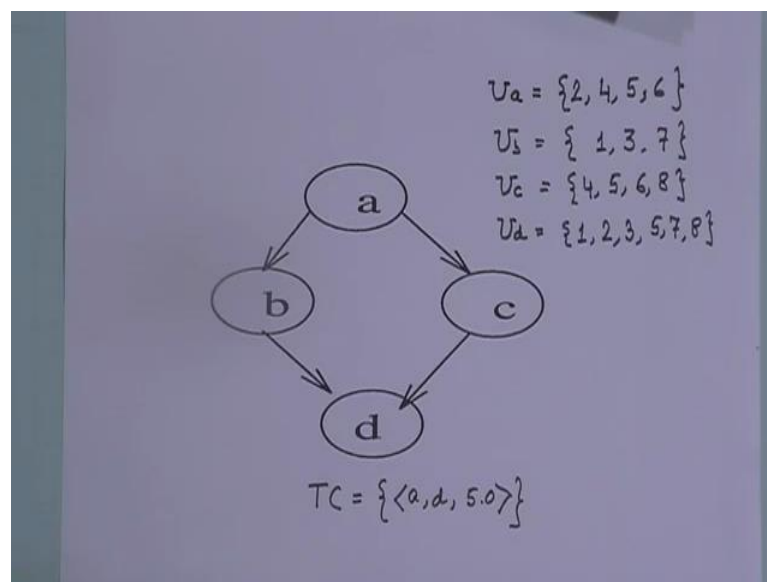


Embedded Systems Design
Prof. Anupam Basu
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 46
Optimization – 1

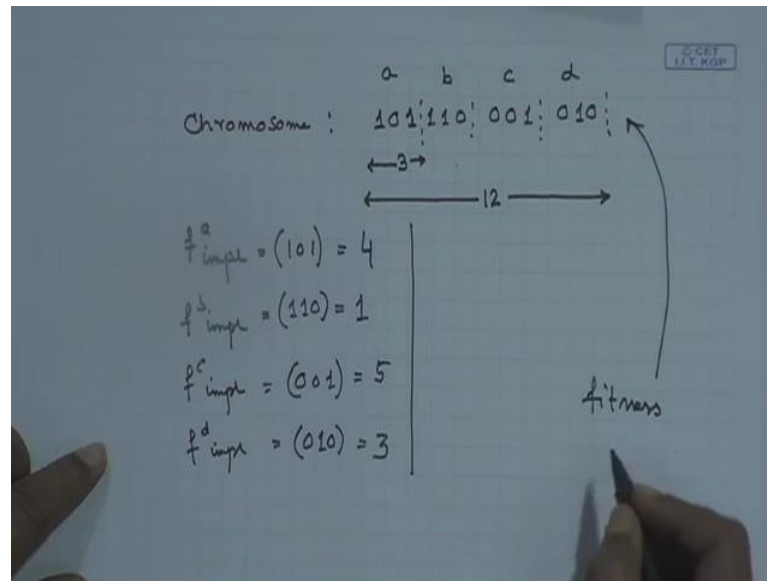
Before moving to the discussions on optimizations, I would like to complete the discussion on genetic algorithm that was initiated in the last class, that harder software partitioning problem is also very internal of an optimization and can be solved using genetic algorithm also. I will quickly give an example without going into all the details because, it is expected that you know about the genetic algorithms separately.

(Refer Slide Time: 01:02)



So, here we start with. See here task graph, as can be seen here tasks a b c d and here are the dependences, you can see and there is time constraint that has been specified which is specified in this way the time constant from a to d; a to d is 5 some 5 units of time. I could have specified from a to c and c to d also, but that is not specified here also. So, these are the functions and for each of these functions a b c d let us assume that we have got implementations like U_a implementation is I am just showing some integers 2, 4, 5, 6 what does it mean? 2 4 or 5 or 6 can implement a this function a; similarly U_b is 1 3 7 is a toy problem absolutely toy problem that I am showing U_c is 4, 5, 6, 8 and d is most flexible can be done by 1, 2, 3, 5, 7, 8. So, for each of them, these are the candidates.

(Refer Slide Time: 02:59)



Now, suppose I take a chromosome say I take a chromosome 101:110:001:010 as a solution of for a, solution for b, solution for c and solution for d. Now, so if for each solution in this chromosome I have taken the code length the length to be 3 that means each solution for solution for each task a b or c or d is given by a gene of length 3. Therefore, my total chromosome for 4 tasks is of length 12.

So, now suppose that means, now this one is a solution. This solution may be say if I say that 101 is actually I can write it in this way, the implementation of a is 101 and suppose that is 4 it is not a binary 101 is not the binary of 4 101 is the code for device 4 because, it could be implemented with 4. So, similarly say if b implementation b's implementation is 110 and suppose 110 is 1 because for b 1 was one of the possibilities of implementation. Similarly, if c implementation is 001, which will be 5 and if d implementation will be 010 and that is 3.

So, 4 1 5 3 I can see that for 1 4 was the candidate, for this 1 1 was a candidate 5 was a candidate and 3 was a candidate. So, this is one possible solution this chromosome is depicting one possible solution in that way, for all the possible solutions, if I have a cross product of all these. So, I can have 2 1 6 2 all those things could be possible solutions now each of these solutions one chromosome depicts a solution and for all of them. I carry out a fitness function for each of these chromosomes. I will compute a fitness function.

(Refer Slide Time: 06:42)

Handwritten notes on a blue background. At the top right, there is a small box containing the text "SECRET" and "T.T. KOP". The notes define a fitness function f in two cases:

$$f = \frac{FIT - \text{cost-penalty}}{FIT}$$

if time-penalty = FALSE
concurrency-penalty = FALSE

$$f = \frac{FIT - \text{cost-penalty}}{FIT \cdot \text{TIME.PEN.WT}}$$

if time-penalty = TRUE
conc. - pen = FALSE

Below the formulas is a diagram labeled "Gantt". It shows two horizontal bars representing tasks. The first bar is shaded with diagonal lines. The second bar is unshaded. They are positioned such that they overlap in time, with an 'X' marked below the overlapping region. Below these two bars is a third, longer bar, also shaded with diagonal lines, which spans the duration of both tasks above it.

And the fitness function that can quickly be decided on, say; fitness function F is fit is say the maximum of the cost penalties in a population the maximum penalty that I can pay, with all these will have a population, possible choices. If I take the cross product of all these, I will have a huge set of chromosomes and that is all possible solutions together and the worst maximum cause I take as fate and for every solution I take a find it is cost penalty. So, I can compute fitness as how much the cost penalty reduces how much this fit reduces. So, there is a cost penalty. So, they will value for this and I normalize with this, if the time penalty is false.

Time penalties false, means what? Two functional elements cannot be scheduled at the same time and that has been done already, say; in a solution that has been done. So, that is false and another thing is that if the time penalties validate say for example, I have got a time constraint here. So, if because of this allocation, I violate this. Then the time penalty is violated, then time penalty is true and also if concurrency penalty is false, that is one concurrency penalty means if two any two functional elements are concurrent in nature and both have been assigned.

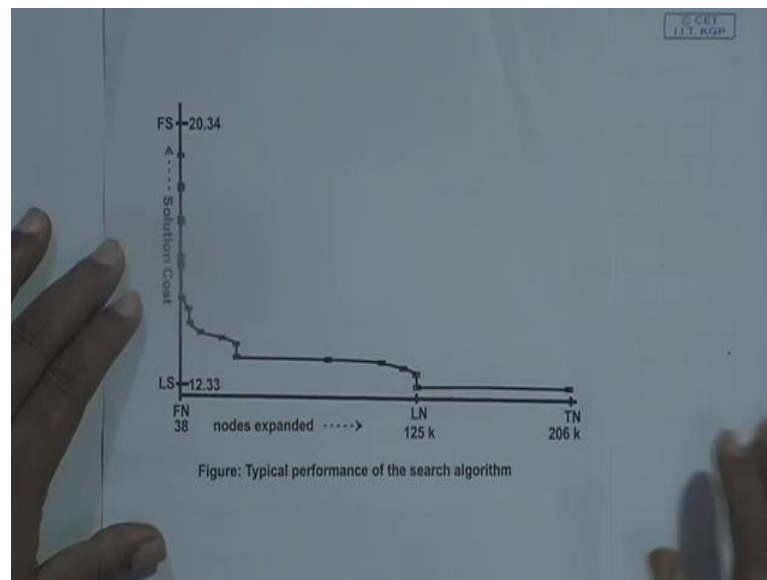
Say for example, software implementations then the concurrency penalty is coming in if in a particular solution they are concurrent but I have put both of them into software then there is a concurrency penalty. So, if none of them are the case, I compute the fitness here. Otherwise, I can say let me fit minus cost penalty divided by fit multiplied by time

penalty. I can put a time penalty wait some time penalty wait. So, if time penalty is true and concurrency penalty is false, then this will be my fitness function. Similarly, if concurrency penalty is true and time penalty is false then it will be multiplied by concurrency penalty. If both of them are true, then it will be fit time penalty with into concurrency penalty.

Accordingly, if there are more both penalties, my fitness is reducing. So, accordingly I have different fitness functions for the different solutions next step. What is done in a genetic algorithm? I do the crossover typically in genetic algorithm, if I have two chromosomes and I can select probabilistically any crossover point, if I do this crossover point then I take this part of this and this part of this to come to a new chromosome. So, a new chromosome will be something like this that is what crossover gives me there is a crossover between these two is it giving me new one and in that way by doing the crossover. I will get new solutions but in this case I cannot probabilistically do anywhere what do I have to do at the code length points. I cannot break a gene here because, this is solution for one functional element, I cannot have mixed solution. So, that is one restriction and based on that I go on iterating and ultimately. I will find that the fitness function is increasing and whenever any of the solutions have got their fitness below our threshold which is experimentally found out that is discarded.

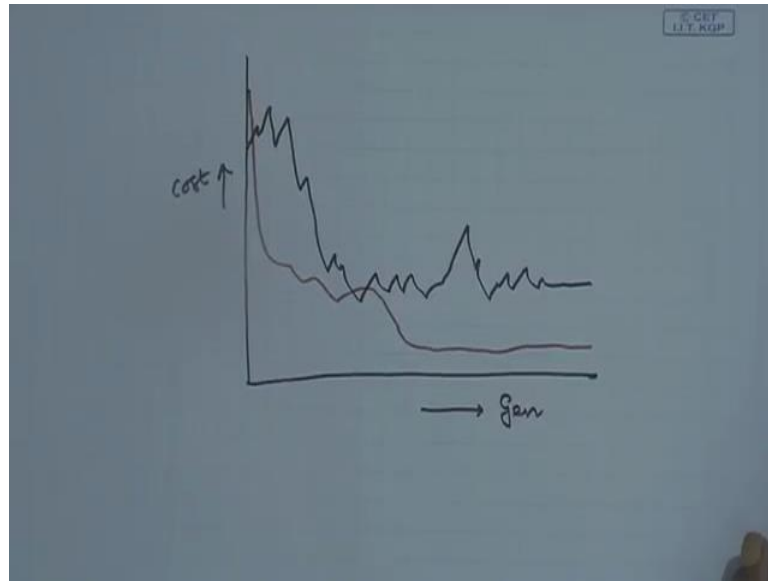
Now, for all these hardware, software partitioning, we have discussed two particular approaches. I mean multiple approaches of course, one is absolutely hardware moving to software, gradually software moving to hardware or a consistent leveling problem or the genetic algorithm. Either of these, the greedy of course do not guarantee the optimal solution the consistent leveling problem, if we try to solve using some domain specific heuristics then also it will give us a local minima and in genetic algorithm, also we cannot always say that is a local minima but whenever we come to a particular solution, we give a mutation and by giving the mutation that means I, one particular gene, I change and by doing that I again move away from the local minima.

(Refer Slide Time: 13:33)



So, the experimentally what we find is the typical solution. Here is a solution of running the consistent leveling problem hardware software partitioning version of the consistent leveling problem you can see here on the Y axis is the cost and the X axis the number of nodes expanded during the search as we go on expanding the nodes ultimately we come to a set I mean goes on decreasing and we find that after 125 k nodes expansion. We have come to a point after which we are not economizing any further now, you may not like to devote. So much time for the hardware software partitioning computation but the beauty of this sort of branch and bound is that if the more time you specify the better solution you will get. So, you can come anywhere here and find a good enough solution because, ultimately we need the good enough solution, satisficing solution the same thing happens for genetic algorithms also.

(Refer Slide Time: 14:45)

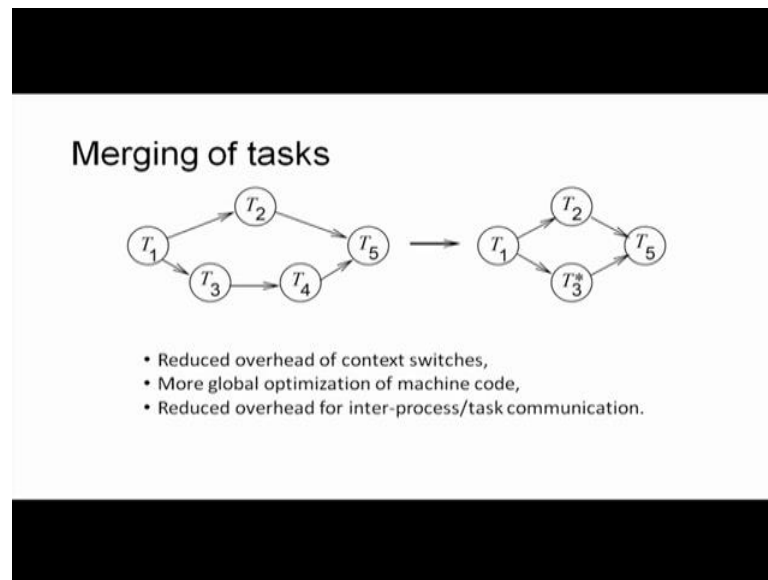


We get an average cost actually the curve comes like this. The average cost is coming down something like this sort of nature and with mutation. The given here, we can find that here is a generations number of generations and here is a cost. We get similar curve and if we look at the best cost that we are getting, the best cost comes to something like this and then remains a while here and then comes down. So, it is a gradually coming to a better solution. So, either of these depending on your requirement you can formulate when you actually delve in doing a hardware software partitioning for a real life situation.

Given that discussion on hardware software partitioning which of course takes into account. Now, all these discussions were based on the availability of a good library, which had the different costs mentioned. Typically or ideally it should also have the power cost of the hardware devices, that are; there should be mentioned but in our experiments or the discussions that we have done we have talked about the area cost and all those power costs can also come in.

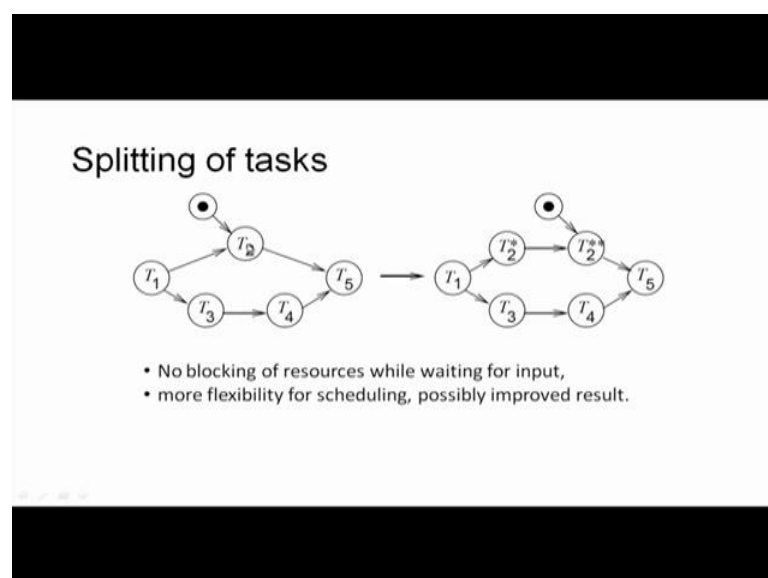
Now, we will be discussing on some optimizations that needs to be carried out at the compiler level till now. We have talked about the architecture level hardware software partitioning and everything. Now, at the compiler level, we can do number of optimizations and we will briefly discuss those.

(Refer Slide Time: 16:52)



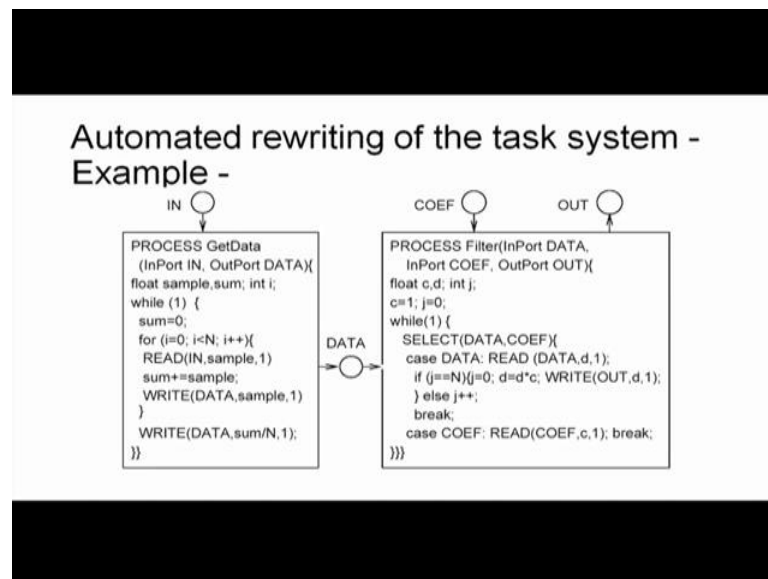
For example; one thing that we can do, we can merge the tasks. For example, here have got a task graph. You can see that T_3 and T_4 I can merge these to be T_3^* start. What is the advantage if you think of the overall computation time? Whenever a task ends another task starts, then there is a context switching over it from the operating system. So, that context switching over it will be reduced wherever I can merge the tasks. We can show that we can have blown more global optimization in the machine code and reduced overhead of inter process communication. For example; and also we can do just like as we can merge task.

(Refer Slide Time: 17:43)



We can split tasks also. Say here again the same graph but, suppose this task T 2 needs a particular data. Now, till that data comes T 2 held up but it is probably possible to break up this T 2 into these two parts, where this part is not requiring that data and I can go ahead and accept the data here. So, the amount of blocking on resources is reduced. So, that is that gives us more flexibility in scheduling and possibly improves result.

(Refer Slide Time: 18:28)

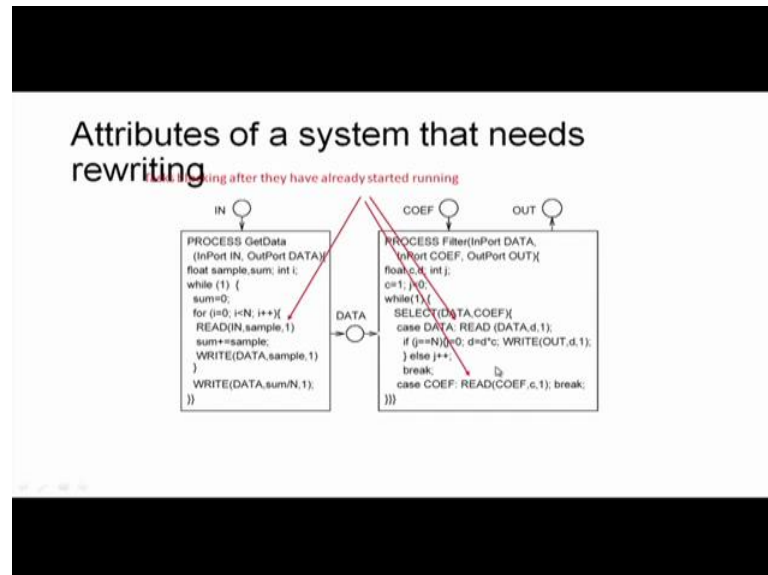


So, here is an example and this can be done automatically. So, you see here, there are two processes one is get data and another is filter. Now, this get data is taking the data from input in and the filter has got two data inputs; one is a coefficient, another is a data that is being sent by the gate data and filter is sending out. Now, what it does it comes here reads a data and through the ins and computes the sum and sends the data to the sum and after a while it sends the average to this data port. Now, this one is selecting either data or coefficient. These are you must have heard about the programming language ADA. So, ADA lite construct actually and this is also ADA lite construct implemented in a programming language called flow.

See now, here select means nothing, that it is looking at these two input points and whichever gets the data it will start on that. So, if the data arrives, what it does? It reads the data and ignores. If N, it waits for N and actually waits for the average after this N, this average will come right here, it is going on sending data; ultimately, after N data it is sending the average. So, it is actually interested in the average and then it writes that out.

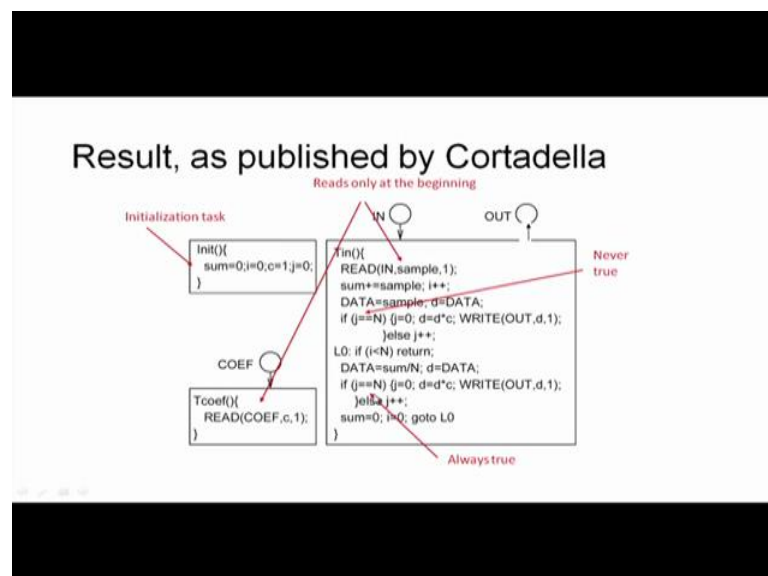
If the coefficient comes, then it reads the coefficient. So, that is now, you see there are three reads here; one is here, one is here, one is here.

(Refer Slide Time: 20:42)



Now, each of them can be blocking instead. See the tasks, they have already started but it may be blocked on any one of those because, the data is not available. Therefore, it is possible to break this out. This particular task you understand that it is reading this. Suppose, this reading is not done, then this is also not coming, all those things are happening.

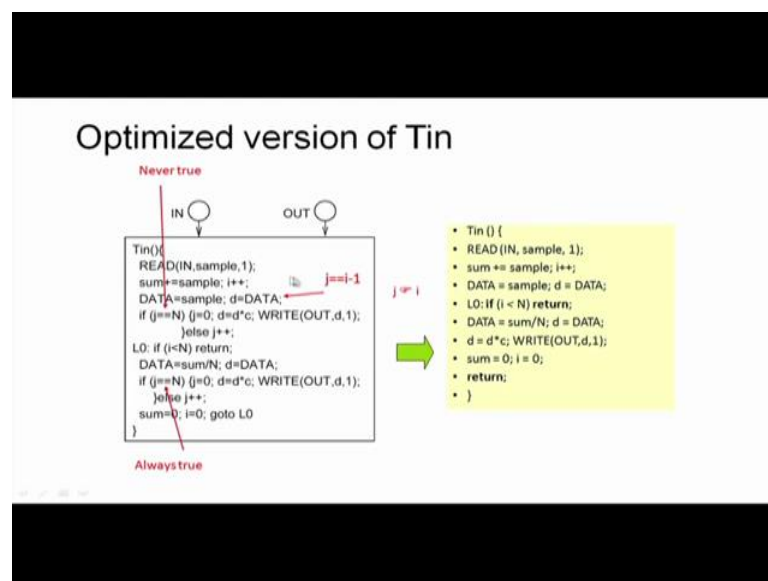
(Refer Slide Time: 21:09)



So, if I can break it down as was done by cortadella a researcher who put this example in there is an initialization task. There is a coefficient computation task, which is separately reading the coefficient, not integrated with this task. So, the non arrival of coefficient will not block this part, here this entire process is not blocked. So, here you see and this one is reading only at the beginning. So, nowhere I need to wait for wait further. So, it is reading the sample. So, I rewrite the code simply.

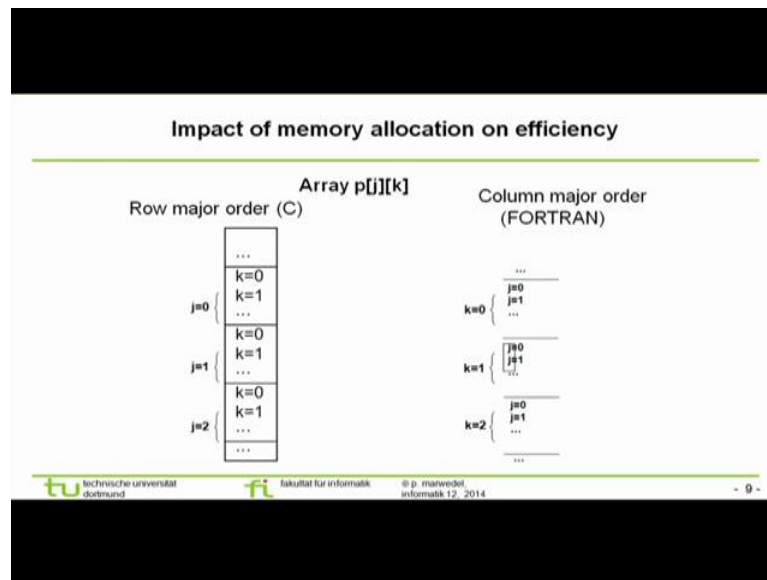
So, just to show that by rewriting the code I can have that task breaking and this J equal to N is never true and J here this part will never be true and this part will always be true. You just look at this part will always be true because, data is coming after here after N samples have been read here the N 's of, this is till when the first data has been raised. So, that will never be true this one will be true therefore, I need not bother about this. So, look at the scenario as it was here, where I had put this. Now when I split it I have put it merge these two together and rewrote the code I have merge these two and rewrote the code as coming in here where this J equal to N is not actually required here.

(Refer Slide Time: 23:06)



So, it can be very well optimized, to be this, was there never true. The entire code can be optimized in this way and present compilers can actually do that and by that we are saving on a number of things. We are saving on the size of the code, saving on the number of input, outputs, all those.

(Refer Slide Time: 23:33)



Another very important and very useful optimization besides task splitting task merging and rewriting the code is loop transformation. Now, you know that FORTRAN and loop is a very common occurrence in DSP programs and most of the many of the embedded system applications.

Column major and row major, in FORTRAN an array is stored in a column major form that means it may be a two dimensional matrix but ultimately it is stored in a linear array. So, in a linear array the k 0 then k 1, for the first column corresponding to the first column all the rows are stored, then second column all the rows are stored, third column all the rows are stored, that is how is done in Fortran, but in C it is row major. So, first I stored the first row, then the second row, then the third row that is a row major order. So, that we have to take into account, when we do optimization. Now suppose, I am trying to deal with an array p j k, p j k this is the array.

(Refer Slide Time: 24:55)

Best performance of innermost loop corresponds to rightmost array index

Two loops, assuming row major order (C):

<pre>for (k=0; k<=m; k++) for (j=0; j<=n; j++) p[j][k] = ...</pre>	<pre>for (j=0; j<=n; j++) for (k=0; k<=m; k++) p[j][k] = ...</pre>
--	--


Same behavior for homogeneous memory access, but:


For row major order

...	k=0	k=1	k=2	k=3	...
...	j=0	j=1	j=2	j=3	...

↑ Poor cache behaviorGood cache behavior ↑

memory architecture dependent optimization

 technische universität dortmund

 fakultät für informatik

© p. marheide, informatik 12, 2014

- 10 -

Now, see there are 2 loops; one is k 0 to m, k 0 to m means, I am moving column wise, another is j 0 to n row wise. So, first I move column wise and then move row wise. Inner loop is in row wise form, which one will result in what since C compiler will store it in a row major form if I do this this is a row major form right. So, every time for as J is changing here I have to go to this then again come back here but what is actually done in my cache memory the contiguous blocks are taken, therefore there will be cache misses also.

So, lead that leads to poor cache behavior and poor cache behavior means what more time and more number of cycles wasted and each of the cycles lead to switching and each of the switching consumes power but if I had done it in this way the same thing. I could have done in this way right then I would have obtained good cache behavior. So, I can do the loop transformation in this way best performance for innermost loop is this so it gives me poor cache behavior, this gives good cache behavior it is obviously memory architecture dependent optimization.

(Refer Slide Time: 26:50)

Program transformation "Loop interchange"

Example:

```
...#define iter 400000
int a[20][20][20];
void computeijk() {int i,j,k;
  for (i = 0; i < 20; i++) {
    for (j = 0; j < 20; j++) {
      for (k = 0; k < 20; k++) {
        a[i][j][k] += a[i][j][k];
      }
    }
  }
}
void computeikj() {int i,j,k;
  for (i = 0; i < 20; i++) {
    for (j = 0; j < 20; j++) {
      for (k = 0; k < 20; k++) {
        a[i][k][j] += a[i][k][j];
      }
    }
  }
}
start=time(&start);for(z=0;z<iter;z++)computeijk();
end=time(&end);
printf("ijk=%16.9f\n",1.0*difftime(end,start));
(SUIF interchanges array indexes instead of loops)
```

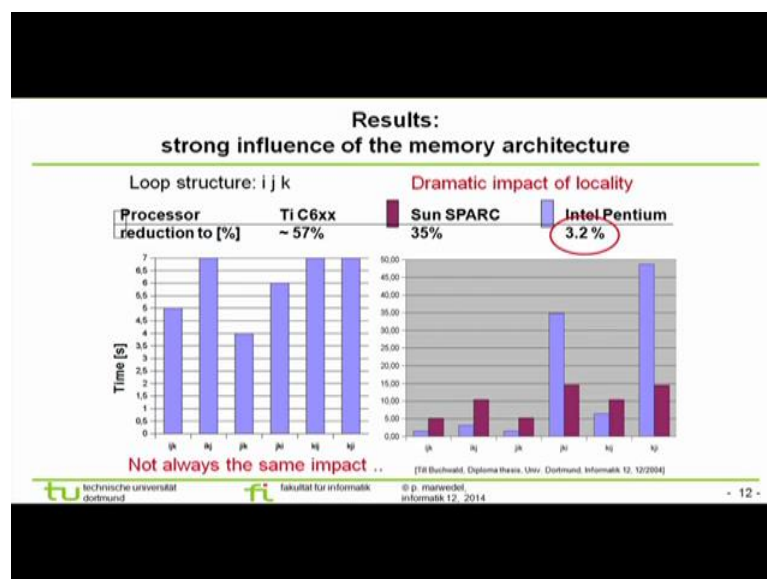
Improved locality

tu technische universität dortmund
fi fakultät für informatik
© p. marwedel, informatik 12, 2014

- 11 -

So, what can be done here you need not look at the entire code but what has been done; here is a compute i j k for outer loop, I middle look j then k and i j k is being added to i j k. Some funny things being done here, but if I go on doing it in a loop then and if I do it in this way i j k, i j k in that case my timing the number of I mean the total time will also be reduced because of the cash means will be avoided. Therefore, by such loop transformation, I can save and this can be done at the compiler level.

(Refer Slide Time: 27:37)




So, here is a typical example; you can see it is very much architecture dependent, also how it stores and all those you can see that the loop structure i j k. If you do this one, improvement will vary i j k, i k j, j i k, j k i, you can see j i k is giving best here and in this case also j i k is giving best among some sparc and intel pentium. So, that is some result that you can get.

(Refer Slide Time: 28:18)

Transformations

"Loop fusion" (merging), "loop fission"

<pre>for(j=0; j<=n; j++) p[j]= ... ; for (j=0; j<=n; j++) , p[j]= p[j] + ...</pre> <p>Loops small enough to allow zero overhead Loops</p>		<pre>for (j=0; j<=n; j++) {p[j]= ... ; p[j]= p[j] + ...}</pre> <p>Better locality for access to p. Better chances for parallel execution.</p>
---	---	--

Which of the two versions is best?
Architecture-aware compiler should select best version.

tu technische universität
dortmund
fi fakultät für informatik
© p. marwedel
informatik 12, 2014
- 13 -

Now, another thing is loop merging. Let us look at this for j to n j++ and p j something and then j to n p j some equal to p j plus something. Now, if the loops are small enough you see there is a loop overhead. What do you mean by loop overhead? Loop overhead means that when a loop is being executed. So, the linear part is computed and then you go back and see whether the loop you have to do the implementation and we have to see if the loop termination condition has been met. And this is a continuous overhead that we will have to incur all the time. You are doing this, if a priori we could compute how many times the loop will go on, then we could have done a by instruction machine level instruction that loop I could have done something like repeat n times.

So, automatically I know I have tried it will do n times just as for loop. I know it will do n times or n by 2 times. Whatever that is one way the other way is that you can also for the loop overhead. We can have some hardware like; if you recall in DSPs we have done automatic address generation and index update. So, if there be some such hardware then that can also be used. Now, if the loops be small enough then if we adopt 0 overhead

loops these two things that we mentioned are also known as 0 overhead for looping, then is fine.

Otherwise, if we will get better locality. If I do this, if I merge these two loops j 0 to n we can find that the loop size is same, loop number of vibrations is same. I could have clubbed them and by clubbing them not only I have better locality but also I can do parallel execution in this loop. I do these two things together and again go up. So, my objective is to show you some of the techniques there are many and people are working on many such, by which we can optimize on the code.

(Refer Slide Time: 30:56)

Example: simple loops

```
#define size 30
#define iter 40000
int a[size][size];
float b[size][size];

void ss1() {int i,j;
for (i=0;i<size;i++){
for (j=0;j<size;j++){
a[i][j]+= 17;}}
for (i=0;i<size;i++){
for (j=0;j<size;j++){
b[i][j]-=13;}}}

void msl() {int i,j;
for (i=0;i< size;i++){
for (j=0;j<size;j++){
a[i][j]+=17;
b[i][j]-=13; }}}

void mml() {int i,j;
for (i=0;i<size;i++){
for (j=0;j<size;j++){
a[i][j] += 17;
b[i][j] -= 13;}}}
```

tu technische universität
dortmund

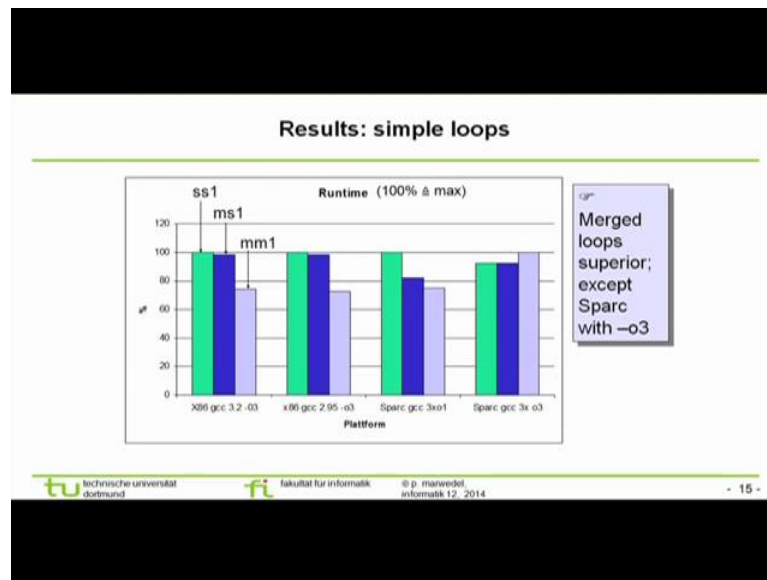
ft fakultät für informatik

© p. marwedel,
informatik 12, 2014

- 14 -

So, here you see simple loop, first let us say; ss1 for I equals 0 to size j i j is i j + 17 for i equal to 0 to size j equal to 0 to size b i j is b i j - 13. I do it explicitly, here what I do is the same thing for I equal to I just take i equal to 0 to this thing and inner loop I merge. What has been done? What is the difference between these two? Here, the j part has been merged the 1 1 I and the J part I do for a, I do for b keeping the row fix. I do for a and I do for b. I should save time in that and here I do, I even merge these two. Why do I keep them the range is same? So, I can further merge these two. So, I come to this and 1 j to both.

(Refer Slide Time: 32:13)



And then move j comparatively, what is the performance you see, except for some architectural peculiarity or peculiarity of sparc leave aside this one. Look at these three ss1 compared to that mm1 is better if, as I merge. I could get substantial improvement in 86 X 86 and sparc that is what we can gain by intelligently generating the code.

(Refer Slide Time: 32:43)

Loop unrolling

```
for (j=0; j<=n; j++)  
  p[j] = ... ;
```

```
for (j=0; j<=n; j+=2)  
  {p[j] = ... ; p[j+1] = ... ;}
```

factor = 2

Better locality for access to p.
Less branches per execution of the loop. More opportunities for optimizations.
Tradeoff between code size and improvement.
Extreme case: completely unrolled loop (no branch).

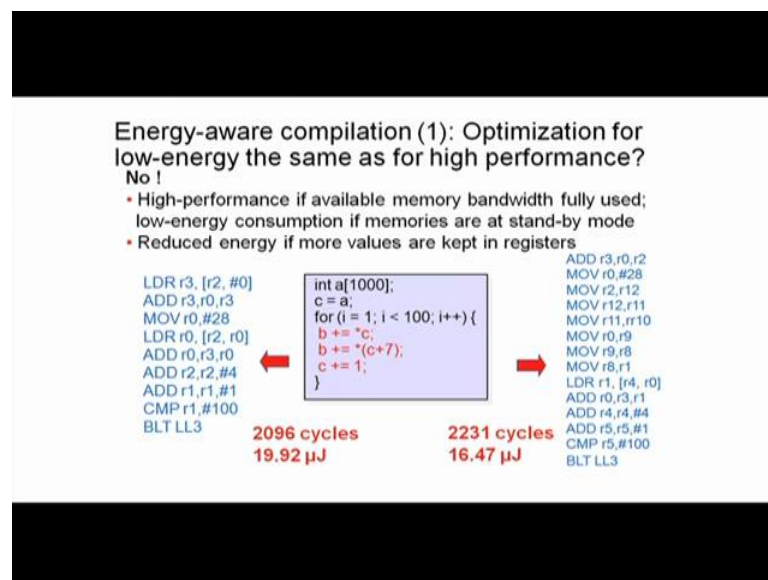
tu technische universität dortmund fi fakultät für informatik © p. marwedel, informatik 12, 2014 - 16 -

Another technique is loop unrolling. This is a very common thing in compiler optimization and code generation, as well as in embedded systems. Here I can say for example; here, j to n , j is 0 to n , I am doing something in this loop, I can very well do $p[j]$,

$p_j + 1$, $p_j + 2$ all those things could be done together because, I know that I am explicitly unrolling the loop now and here it varies, I mean what is the factor by which how many you unroll say; you can do p_j and $p_j + 1$ then you go on doing the loop. Now, what will be the loop index incrementation will change.

So, first time in one loop I was doing only one update here and I was doing N number of looping. Here, what I do; two operations in a loop and do n by two looping there by, also I reduce this is loop unrolling up to a level of two factor of two. It can be also flattened wherever possible but, that will if you flatten then what will the problem, the code size will increase. Code size will be increased and as the code size increased your memory requirement also increases. So, all those things have to be looked into.

(Refer Slide Time: 34:16)



Here is another technique that you can think of energy aware compilation. So, if I optimize for low energy, what would you do in order to read if you want to reduce energy? I want to reduce energy. Energy being the power integral time, I would like to reduce the time as well as the power spent. So, I will also try to reduce the mist memory cycles or even the memory accesses. If I can confine my computation within the registers mostly then I save time. So, here you see a particular source code has been divided into two parts, one is this, one is mostly in the registers.

In the registers, this one, the computations are having minimal memory references, only one or two memory references. Here, I am having my three or four memory references.

So, as I go to the memory references I am needed less number of cycles because the code size is less but the power was more energy was more. Here, I have increasing number of cycles but I have reduced the particular the amount of energy. Now, this varies depending on it will depend on how much sketchpad memory you have, how many you can do on the registers itself all those things are not. So, straightforward we have to do some sort of optimization for that.

(Refer Slide Time: 36:07)

Energy-aware compilation (2)

- Operator strength reduction: e.g. replace * by + and <<
- Minimize the bitwidth of loads and stores
- Standard compiler optimizations with energy as a cost function

E.g.: Register pipelining:

```

for i:= 0 to 10 do
  C:= 2 * a[i] + a[i-1];

```

→

```

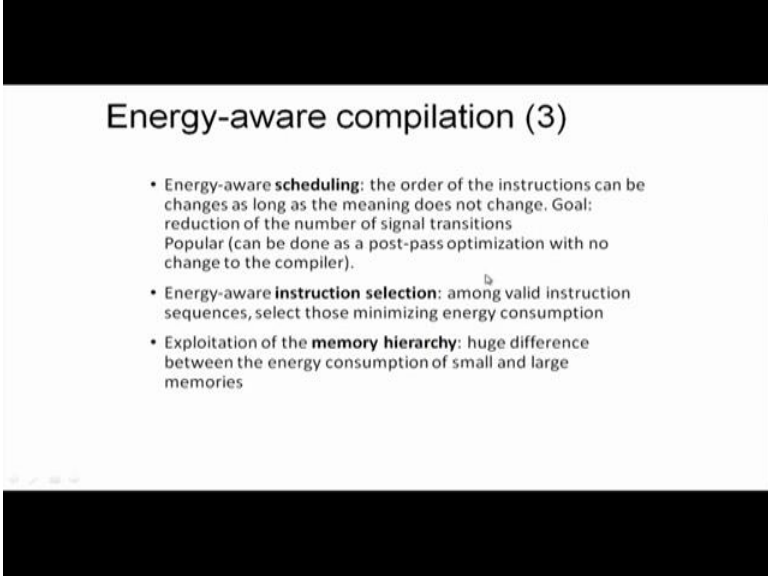
R2:=a[0];
for i:= 1 to 10 do
  begin
    R1:= a[i];
    C:= 2 * R1 + R2;
    R2:= R1;
  end;

```

Exploitation of the memory hierarchy

Here is another example say; i 0 to 10 I am doing C is a i + 1 * 2. Now this one is being done in a loop. Here, what I am doing? I am taking 1 A 0 in a register and then 1 to 10, 0 is already taken. So, from 1 to 10 I am doing a register pipeline. I am just doing it in the register and not looking at the array any further, only just taking one element from the array not to hear. I have to add to access to the memory. Here I take N1 and I go on that one will be the earlier one. So, thereby, I can do register pipeline, I mean register pipelining and save on the number of loops. Also there was the opportunity of doing this multiplication by 2, by simple shift and not multiplication. Thereby, I could have also optimized.

(Refer Slide Time: 37:15)



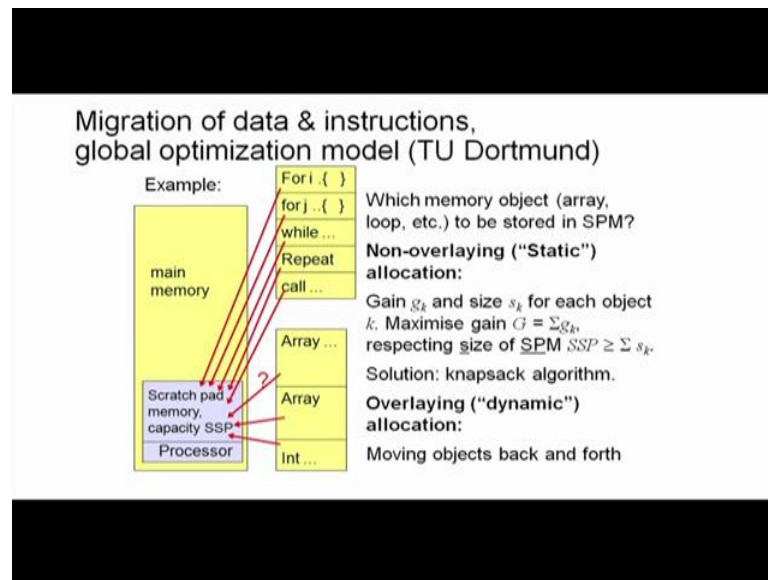
Energy-aware compilation (3)

- Energy-aware **scheduling**: the order of the instructions can be changes as long as the meaning does not change. Goal: reduction of the number of signal transitions
Popular (can be done as a post-pass optimization with no change to the compiler).
- Energy-aware **instruction selection**: among valid instruction sequences, select those minimizing energy consumption
- Exploitation of the **memory hierarchy**: huge difference between the energy consumption of small and large memories

So, energy aware compilation will require me that there are different opportunities; one is energy aware scheduling. Now, what do I do in energy aware scheduling? I schedule the instructions, the ultimate result will not change I will schedule the instructions. So, that the signal transitions are minimized. Think back of your computer organization or architecture courses, whenever I am fetching an instruction from the memory that an opcode with number of bit patterns.

So, if I schedule two instructions, one after another where there is minimal change in the bit pattern that would minimize in the switching. Similarly, what sort of instructions I would select? These are related, suppose, addition and multiplication; I can do the same thing by repeated addition or multiplication or shift and multiplication. Now, I may also prefer not to do with shift but do multiplication by two if I can save on the switching of the consecutive instructions and the other thing that we have already shown is the exploitation of the memory hierarchy, that we have already shown that here also we are exploiting the memory hierarchy between the primary memory and the registered scratchpad memory.

(Refer Slide Time: 38:55)



In that way, there are different ways by which we can put in, what are the elements will put in the scratchpad memory; that is itself. Another issue and for that again it can be modeled as an optimization problem. So, this discussion can go on but my objective was simply to give you an idea that we can save on energy and we can optimize on the performance not only by the hardware decisions that we have taken but also, at the compiler level, when we generate the code. We can look at different optimization steps and make a very efficient system code. So, that the overall system memory energy requirement is minimized and the performance is met.