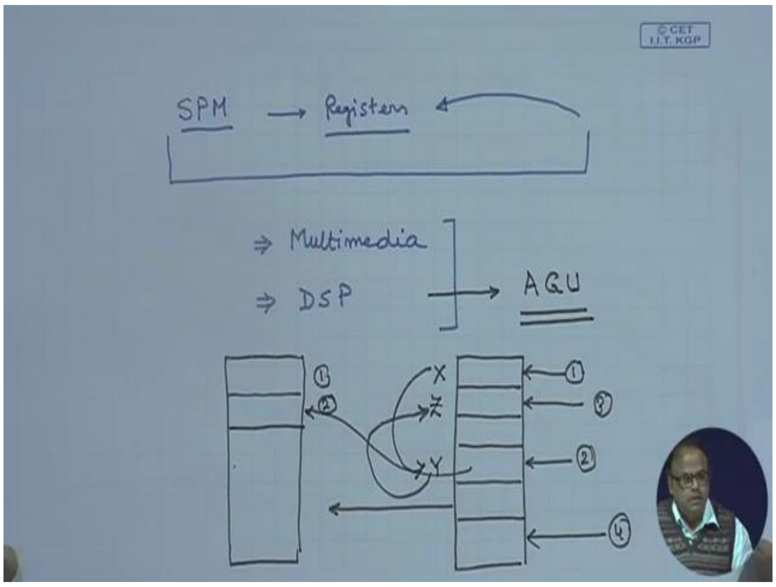
Embedded Systems Design

Prof. Anupum Basu.
Department of Computer Science and Engineering
Indian Institute Technology, Kharagpur

Lecture – 47
Optimization – II

So, we had discussed about different ways by which a compiler can help in optimizing the code. And optimizing, generating an efficient code, which will save they on the execution time. Today, we will see in this module. We will see some more measures which are taken. There are many measures, but will not be able to discuss all of them.

(Refer Slide Time: 00:51)

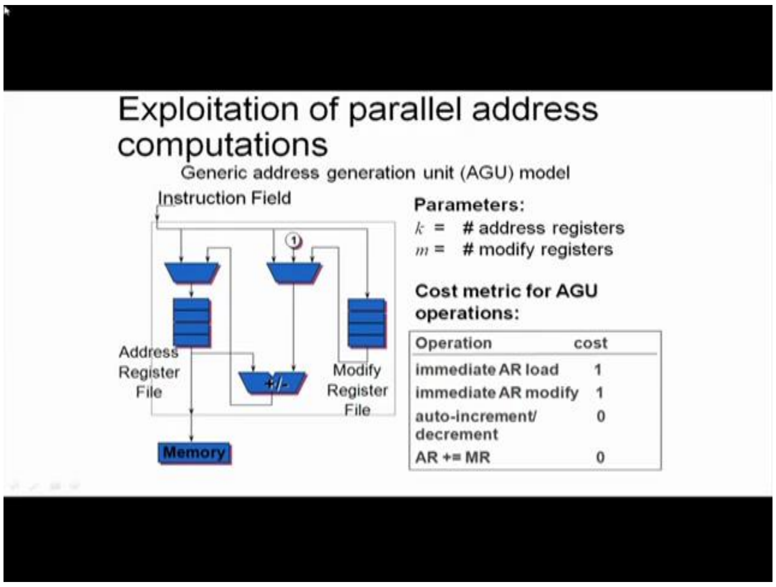


For example, there are measures on optimizing on the scratchpad memory; SPM means scratchpad memory; that means the number of the registers. There are optimization techniques by which we try to map in more number of variables onto the scratchpad. So, that the memory references are reduced, but that we are not discussing in detail we will be discussing two approaches which are specially useful one is for multimedia type of applications, where embedded systems are having an increasing role. Another is our good old digital signal processing.

Now, let us first start with the digital signal processing scenario. If you recall you remember that in a typical digital signal processing architecture, we have got address generation unit which generates the address for accessing the arrays. So, typically in a

DSP application there are, a lot of array handling involved. So, there are arrays, and I may like to access different parts of the array. Maybe, I first access this one, then I access this one, and then I access this one, and then I access this one, like that. So, continually, I need to compute the addresses of these indices. Now, for that address generation unit is a part of the DSP digital signal processor because, it allows us to do the address update along with the other updates.

(Refer Slide Time: 03:10)



Exploitation of parallel address computations

Generic address generation unit (AGU) model

Parameters:

k = # address registers

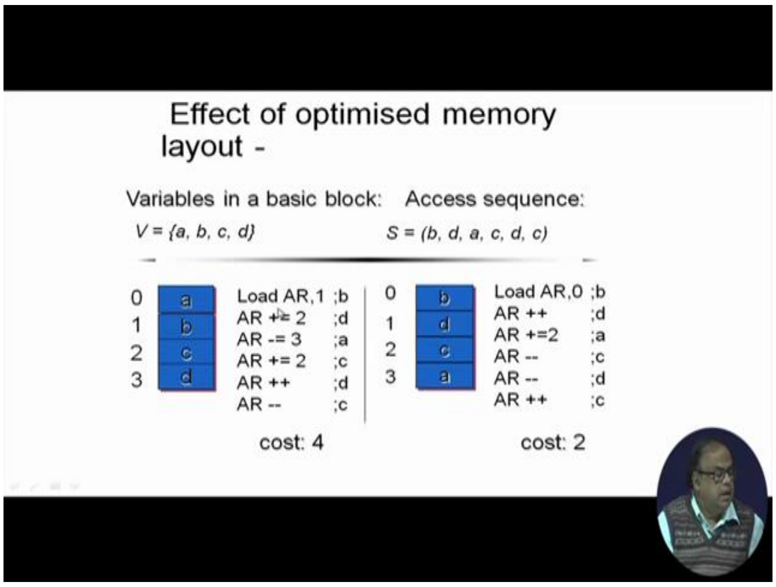
m = # modify registers

Cost metric for AGU operations:

Operation	cost
immediate AR load	1
immediate AR modify	1
auto-increment/ decrement	0
AR += MR	0

So, if we look at this slide once again, this was a typical DSP type of architecture, where there are address register files and these are the modified register files. Modified register files means this normal scratch pad and here I generate the address, and then from there I access the memory. Now, the key point is here, I have got say k address registers an m modify registers or scratch pads. Now, if I have immediate address register load; that means, say I add a particular register value, increment it with 1 or I load it with a particular address, the cost of that is1, if I modify the cost is 1. On the other hand, if we are given the auto increment and auto decrement facility by which parallely I can increment the register, I can auto increment Y 1 and auto decrement by1. If I can do this increment and decrement parallely, by hardware then that is 0 cost.

(Refer Slide Time: 04:35)



Effect of optimised memory layout -

Variables in a basic block: $V = \{a, b, c, d\}$

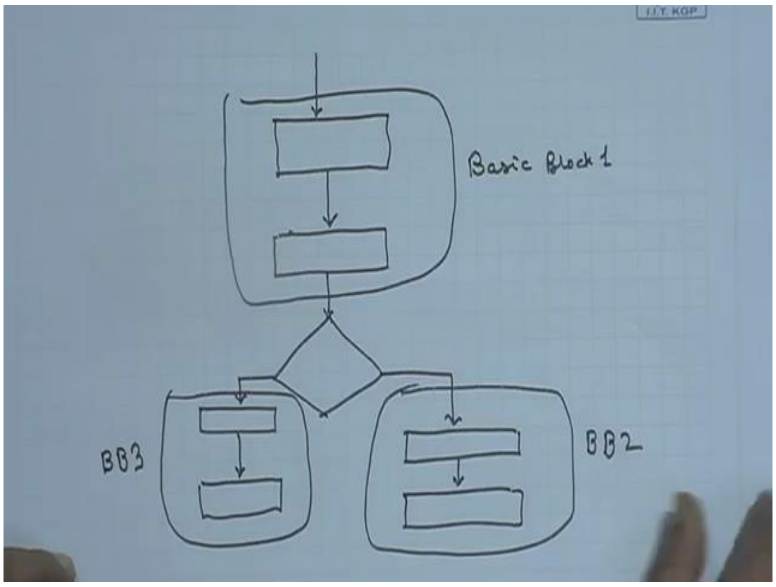
Access sequence: $S = (b, d, a, c, d, c)$

0	a	Load AR,1	;b	0	b	Load AR,0	;b
1	b	AR += 2	;d	1	d	AR ++	;d
2	c	AR -= 3	;a	2	c	AR +=2	;a
3	d	AR += 2	;c	3	a	AR --	;c
		AR ++	;d			AR --	;d
		AR --	;c			AR ++	;c
		cost: 4				cost: 2	

Therefore, let us first come once again here, that my access is here are like this; I am accessing1, then I am accessing 2. Now, this is not being achieved by auto increment, because I have got some address X. Say this address is X and I have to add X plus 1 2 3 in order to get this and then. So, this is Y, Y minus 2 get Z. So, I have to do, I am my traversal is like this and which is not achievable using auto increment and auto decrement. Therefore, I need more number of computational cycles to generate the addresses.

Now, let us see how we can optimize it by finding it, finding out the proper layout. See no one prevented me to come up with a layout, where this one will be put here, and this would be laid out here, and this would be layout next, like that. So that I could have done in our auto increment, I could have maximized the auto increment and auto decrement operation, maximize the number of accesses using auto increment auto decrement method. So, that is our objective of finding an optimized memory layout. So, let us look at this example; I am considering a basic block. What is meant by a basic block?

(Refer Slide Time: 06:33)



A basic block say this is something, we did not meant discuss earlier that say in a control and data flow graph you have got some points, some number of computations are coming up, and then there is a decision point, and based on this decision point you are doing some other computations.

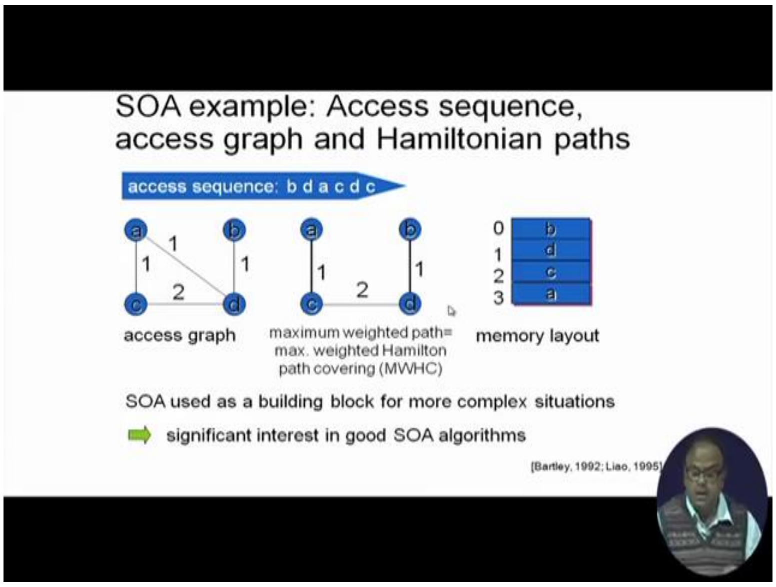
And here you are doing some other computations. So, this is a basic block. This is another basic block; this is another basic block, where we are not taking any branches. So, these are known as basic blocks 1, basic block 2, like that. So, now, we are just considering one particular basic block and say the variables in the basic block are a b c and d 4 variables. And suppose they are laid out in this way a b c d, sequentially. Now, the access sequence, the way the program is trying to access is that first it is trying to access b, then d, then a, then c, then d, and then c.

Consequently, let us see what will happen to this address register. First, the address register will be loaded with 1. So, that it comes to b, then the next access is d therefore, this will be plus 2, then next access is a will be minus 3. Look at this mouse pointer from d I have to do minus 3. Now, none of these are auto increment auto decrement. After that I mean now a, after a access c, therefore, from there I have to do plus 2 again not auto increment auto decrement. After accessing c d, so, after accessing c I can do an auto increment and after that again c, so, I can do another auto decrement. So, these 2 are 0 costs, but I have got a cost of 4 already incurred.

Now, with a little intelligent layout, I could have reduced on this, let us see how; say, I come up with a layout b d c a. Here the cost was 4, why 4? Because I had to do the address computation 4 times; by the way here I am not showing the instructions at all. I am just showing the addresses which I want to access, only the addresses which I want to access. So, here again, so, initially as I have laid out be here. So, I have to load it, no doubt I have to load it, and after that b d just by auto increment, 0 cost then for after d a, I have laid out a here. So, I have to do plus 2. So, one more cost, once again I have to do the computation, after I come to a then c auto decrement from a to c I can come just by decrementing, the address register.

Next comes, when I come to c from there I have to go to d and that is again by another auto decrement. No extra cost, after that d, I have to come to c auto increment, no extra cost. So, the total number of load address register update cost is 2. Thereby, I can save on the time. Now, we have to see how we can arrive at this; this we have done as a simple example, will remain within simple example, but say, suppose I have got an access sequence as I was showing here b d a c d c.

(Refer Slide Time: 11:08)



SOA example: Access sequence, access graph and Hamiltonian paths

access sequence: b d a c d c
a
b
c
d
1
1
1
2
access graph
maximum weighted path= max. weighted Hamilton path covering (MWHC)
memory layout
0
1
2
3
b
d
c
a

SOA used as a building block for more complex situations

significant interest in good SOA algorithms

[Bartley, 1992; Liao, 1995]

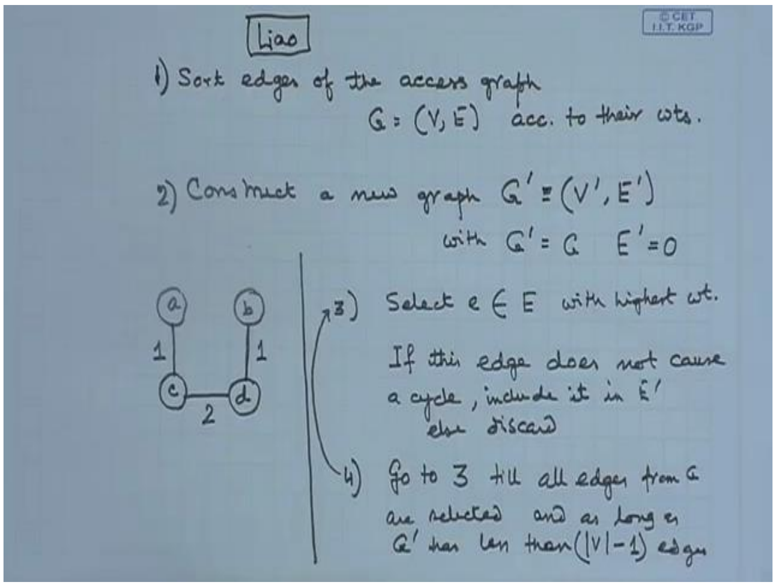
So, first I build an access graph, the first thing that is built is an access graph, where a each of the nodes are different accesses, and there is an edge between two nodes, if their adjacent accesses. So, b is adjacent to d, d is adjacent to a, a is adjacent to c, c is adjacent to d, d is adjacent to c.

So, that is how we form the graph. Now, we put weights on these edges. The weights on these edges are the same as the number of times this edge established, a number of times this adjacent access is made. So, you can see here that after c, I am accessing d. So, c to d is 1 and again d to c. So, this will be required this consecutive access will be required twice. So, we form such a graph. Now, from this graph we want to find out; now what do we want to find out? We want to find, what we want to maximize what is our objective.

Objective is to maximize yes maximize the auto increment auto decrement operation. So, for that this adjacent see graph will be helpful and for that I will find out the maximal graph of this, maximal adjacent graph and that will be this here, maximum weighted path here, maximum weighted path. You can think of a Hamiltonian path covering also. So, we get this and this is exactly what we were doing here the access, therefore, becomes b you can you could have started with a also, but the sequence would be same b d c a and for that what would you my cost; for this access pattern will be 2.

In any case I have to load B. So, that is how we get the memory layout. Now, what is the is it clear; we are finding out the maximum weighted path. So, that all the nodes are covered it will be applying the maximal spanic spanning free algorithm to cover all the nodes. Now, we can make more complex situations. So, there is, now before this let us present an algorithm for this right. So, what do you do with this? So, suppose let us consider this first graph; this one. Let us consider this graph.

(Refer Slide Time: 14:48)



Liao

1) Sort edges of the access graph
$G = (V, E)$ acc. to their wts.

2) Construct a new graph $G' = (V', E')$
with $G' = G$ $E' = 0$

3) Select $e \in E$ with highest wt.

If this edge does not cause a cycle, include it in E'
else discard

4) Go to 3 till all edges from G are selected and as long as G' has less than $(|V| - 1)$ edges

a
b
c
d
1
1
2
© CET I.I.T. KGP

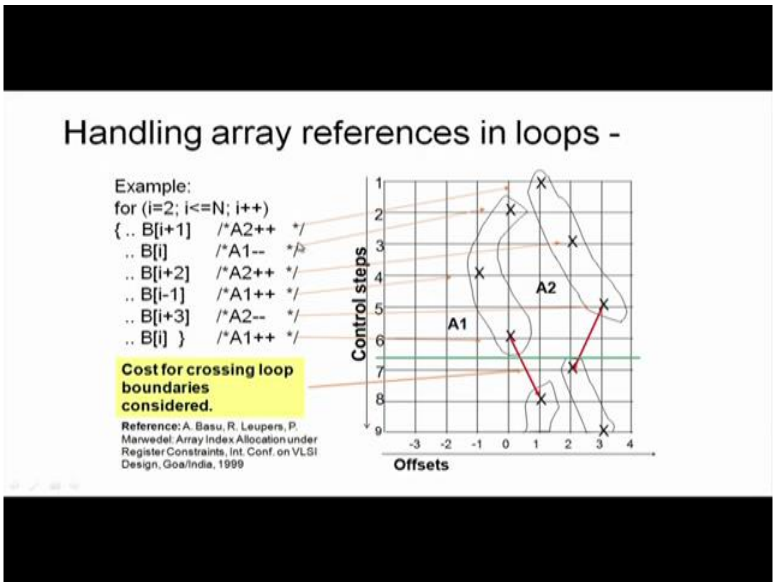
So, we sort, I will put an example also like this of and we will build it up. We first sort the edges of the access graph G, which is V E, according to their weights, the first step. Then second is, we construct a new graph G prime, which will have V prime E prime with say G prime, to be G and E prime to be null. So, we start with this graph, but the edge is not there.

I am just defining it as V prime. So, what will, I am having initially; I am having the 8 said being null. I am having a b c and d and then. So, this is step 1, this is step 2, step 3 will be select an edge; select an edge e belonging to E from the original graph, that is if you look at here in this graph that is Z, which is having the highest weight. So, for us the highest weight is between c and d. So, if this, now the decision is if this edge does not cause a cycle, then include it. So, you include this with this edge 2.

Next, so, once we do that, otherwise will discard it, else discard. Now, we go on doing this. So, we go to this 3 till all the edges. So, what is happening is I add include it in E prime. So, I am adding it any prime. So, go to 3 till all edges from G are selected, and as long as G prime has less than edges, as long as it is not covered. So, what here, as I move here, I have got a choice between all these 3. Now, If I include a, I suppose, I include this a to c. So, all these are; now, next these are not covered, not next edge I select this one that will form a cycle. So, I discard this ultimately, come to this and this is my maximally waiting. So, as I do this, this is only for a very simple case, for more complicated case this algorithm, which is by Liao, to propose this algorithm.

Using which we can find out a nice allocation of this, is the algorithm clear; so, that is a nice way of doing it. Now, this can be extended to more complex cases.

(Refer Slide Time: 20:07)



Handling array references in loops -

Example:

```
for (i=2; i<=N; i++)
{ .. B[i+1]    /*A2++  */
  .. B[i]      /*A1--  */
  .. B[i+2]    /*A2++  */
  .. B[i-1]    /*A1++  */
  .. B[i+3]    /*A2--  */
  .. B[i]  }   /*A1++  */
```

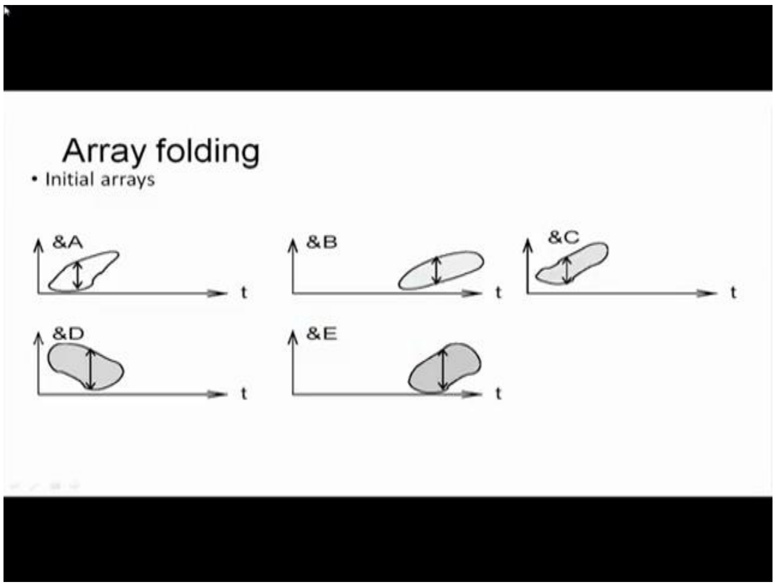
Cost for crossing loop boundaries considered.

Reference: A. Basu, R. Leupers, P. Marwedel: Array Index Allocation under Register Constraints, Int. Conf. on VLSI Design, Goa/India, 1999

Say for example, here is the case where it has been, there are a little complicated access pattern and. So, here not only the auto increment the auto decrement minimization, we also try to utilize multiple I mean auto increment auto decrement registers because they are giving multiple registers and therefore, what has been done here is that you can see the access pattern is such, if you look at these are the control steps, you can see the first one is here, the next one is very close there is a jump here. So, that will be a load here, and then again a load here, again of huge load here, I like this it is going on now. So, here what has been tried is that, how you can utilize minimum number of address registers not one, but more than one minimize on the number of registers and minimize the number of loads.

Thus, you can see that we can use two address registers A 1 and A 2, A 1 can very well devote itself to these addresses, these accesses, and A 2 can devote on these accesses; however, there are some cases where the loads. Here, there is no auto increment; yes you can do that. Here also you can do, it is still auto increment of the distance between clusters. So, you can see that here we are playing with to address registers A 2 and A 1, where we are playing the same game and we are trying to come to a layout, so, that we can minimize. Here, we are having a loading cost of 2 in the earlier case, but if we had used to address registers you could have minimized. So, that is another way this can be achieved.

(Refer Slide Time: 22:24)

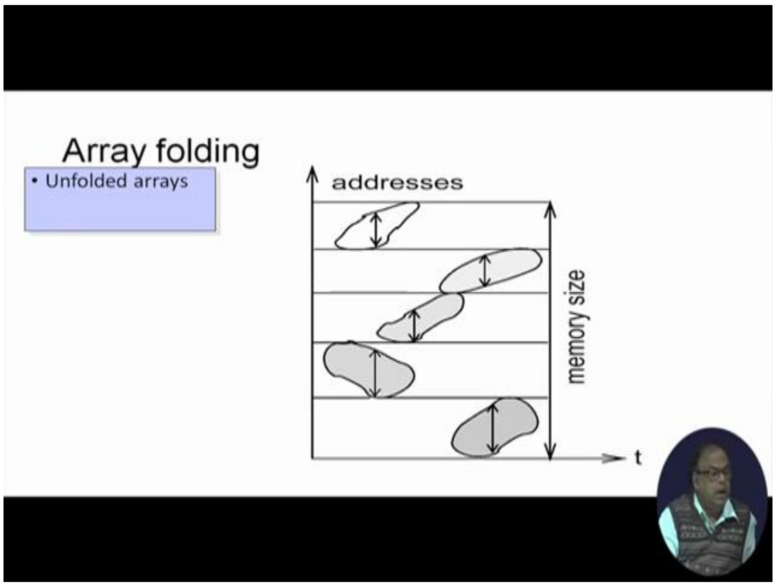

Array folding

- Initial arrays

Next, I will talk about another technique, very simple technique, very intuitive technique, that is very much applicable for multimedia applications. Typically, in multimedia applications, we have got large arrays which we handle and the problem here is how to accommodate all these arrays into the memory, because in an embedded system we will have limited restricted memory. So, here you can see initially we have got arrays A B C D E, 5 different arrays and on this side is the addresses which are accessed, on the Y access, on the X access we are showing time.

So, the typical array elements which are being accessed along the time is being shown by this graph. So, at this point of time, I can see that I am accessing this element, as well as this element, at the same time and doing some addition, whatever. So, we are doing here. So, this double ended arrows are showing the maximal range of array access. So, and over time you can see the array access of A, array access of B, array access of C, the addresses are all not the same. This is the address access. Now, typically or conventionally what we do?

(Refer Slide Time: 23:58)

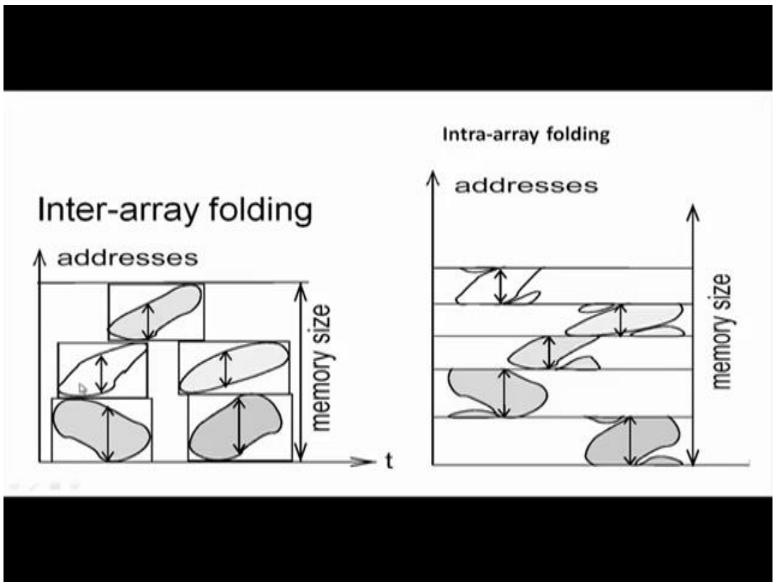

Array folding

- Unfolded arrays

We do the array in an unfolded manner; I put array A here, array B here, array C here, array D here, array E here. And so much memory I need for all these arrays. But if you look at this diagram, one thing will strike you that, this one is accessing addresses which are overlapping with these addresses over time. Now, if I overlay that arrays on the same memory I can overlay them. I have got 2 arrays I can put them in the same memory location when? When they are not being accessed at the same time; so, you can see that the addresses, if I put these two fold them that will not be accessed at the same time; so, can put them in the same place.

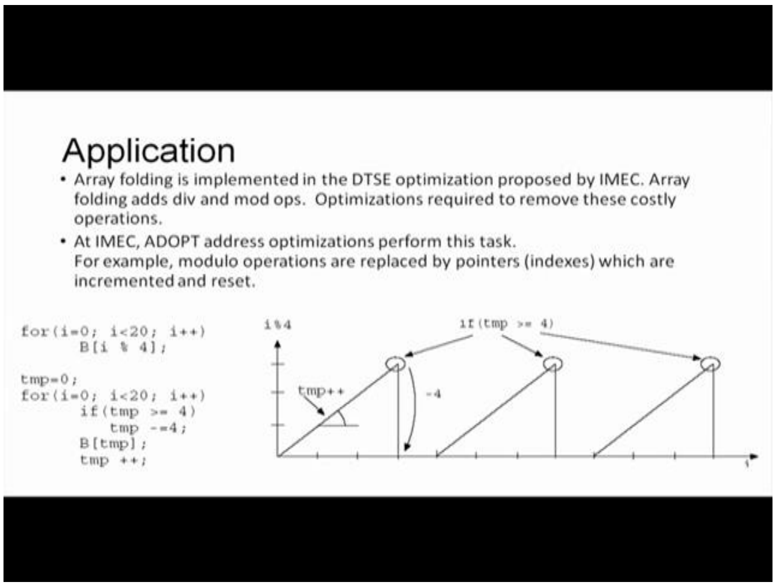
Because, this range will be accessed now and the same range of another array will be accessed at A, later point of time. So, I can swap this; obviously, by that I have to have the swapping over it, but I am now considering the memory optimization. You see all through these optimizations there are contesting parameters. So, if we have got this and if I can fold this, I can fold it in this way that address, these addresses are being accessed now.

(Refer Slide Time: 25:36)



These addresses, this range of address are being accessed at disjoint times. So, I can simply put them in the same, decide to put them in the same locations because, they are being accessed at different points of time and similarly, these two, their addresses are over lab, but the address ranges are here is the range, but they are being accessed at disjoint points of time. This is known as inter array folding. We can further look at this any particular array and analyze that, and see which other elements sub arrays, which are being accessed simultaneously and in the same way you can do intra array folding also.

(Refer Slide Time: 26:31)



Application

- Array folding is implemented in the DTSE optimization proposed by IMEC. Array folding adds div and mod ops. Optimizations required to remove these costly operations.
- At IMEC, ADOPT address optimizations perform this task.
  For example, modulo operations are replaced by pointers (indexes) which are incremented and reset.

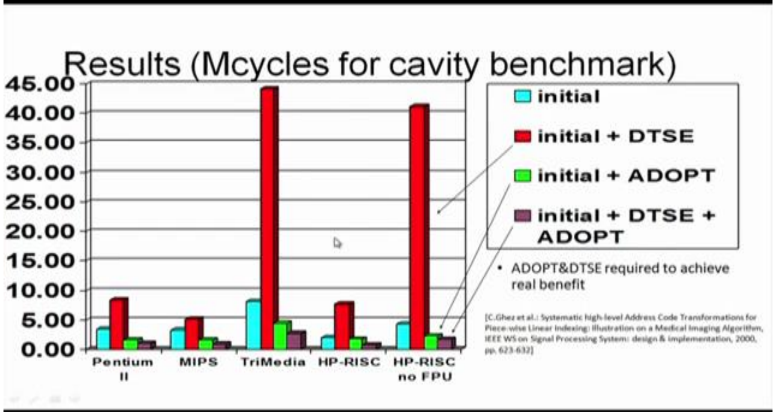
```
for(i=0; i<20; i++)
       B[i % 4];

tmp=0;
for(i=0; i<20; i++)
       if(tmp >= 4)
           tmp -=4;
       B[tmp];
       tmp ++;
```

Now, this has got some magical impact, you know. The idea is so simple, but here is an application of DTAC optimization, where modulo operations and the addition of and the division operations. Division and modulo operations were replaced by unfolding and then, by array folding. For example, here there was a mod operation, in a loop.

Here, you can see that in a loop you are doing a mod operation and mode operation is costly. So, that has been broken down as since it was mod four, this is an equivalent code, and you can see that if the temp variable exceeds for you go over here. So, consequently what happened? So, you are implementing the mod in this way. So, there is a looping here. So, you can see the temp value is going on and then coming down here, as soon as, temp is four and then the next one is starting. So, in that way we could have played, here what we have done is we are playing with the indices.

(Refer Slide Time: 27:52)



- ADOPT&DTSE required to achieve real benefit

[C.Ghez et al.: Systematic high-level Address Code Transformations for Piece-wise Linear Indexing: Illustration on a Medical Imaging Algorithm, IEEE WS on Signal Processing System: design & implementation, 2000, pp. 623-632]

And by that you can see that is a typical result that we have been able to using this technique of these two optimizations of layout and this we could have reduced the time to a large extent, the number of cycles of a particular benchmark. So, that is just an example. The main point that I would like to emphasize are these two compiler optimization techniques.

So, since this is clear right inter array folding is clear to all of you. Now, if I look at this array separately, here as looking at all the arrays together if I look at this array here, I will also find that in temporal behavior that some parts of the array, this part, this part of the array and this part of the array are not being accessed at the same time, so even within that array, that the particular memory location; that of memory locations that I was allocating for this array can be further reduced like this. This much has been reduced to a smaller size because; it can share the same memory locations depending on disjoint access over time. So, that is what we are doing further iteratively on intra array. So, next will move to, let us not have any break.