UNIT V

Hardware and Software for VLIW and EPIC

(Reference: Appendix G Patterson)

Brief Introduction

The chapter discusses about the various compiler technology that is used to increase parallelism in VLIW architecture as well as hardware support needed for them.

The core concepts cover the statically based techniques such as finding parallelism, reducing control and data dependences, and using speculation—are the same techniques we saw exploited in Chapter 2 using dynamic techniques. But the major difference is that the techniques in this appendix are applied at compile time by the compiler, rather than at run time by the hardware.

VLIW Compiler Responsibilities

Following are the compiler responsibilities of a VLIW processor:

1. Schedules to maximize parallelism – code scheduling technique

2. Schedules to avoid data hazards – Register renaming technique.

3. Guarantees intra instruction parallelism- Multiple functional unit capable of doing parallel processing

Detecting and scheduling loop level parallelism

Analysis of Loop level parallelism:

Loop-level parallelism is normally analyzed at the source level when compiler generates the instructions. Analysis of loop level parallelism guarantees if a loop can be executed in parallel or not.

Basically focuses on determining whether data access in later iterations are data dependent on the data values produced in the earlier iterations. the process of investigating this is called as **Loop carried dependence.** This is nothing but dependence between different iterations of loop.

In general a loop which is having loop carried dependence cannot be executed in parallel where as a loop without loop carried dependence can always be executed in parallel.

Consider the example shown below:

```
for (i=1000; i>0; i=i–1)

x[i] = x[i] + s;
```

In this loop, there is dependence between the two uses of x[i], but this dependence is within a single iteration and is not loop carried. Hence it can executed in parallel.

Elimination of loop carried dependence

A loop carried dependence can be eliminated if and only if there is no cycle in dependencies. If there is an existence of non cycle loop carried dependence, they can be eliminated by converting them to **dependence within the loop iteration.**

To know about this in detail. We will go through the examples:

Example 1: This example gives the clear idea of cycle of loop carried dependence for the statements present within the loop which cannot be eliminated.

Consider a loop like this one:

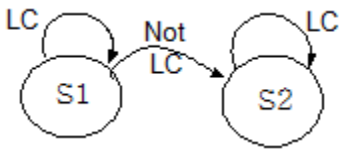
```
for(i=1; i<=100; i=i+1) {

A[i+1] = A[i] + C[i]; /* S1 */

B[i+1] = B[i] + A[i+1]; /* S2 */

}
```

The dependencies for the statement s1 and s2 in above cases are :

1. S2 uses the value, A[i+1], computed by S1 in the same iteration. This is not a loop-carried dependence. ie dependence within the same iteration.

2. S1 uses a value computed by S1 in an earlier iteration, since iteration *i* computes A[i+1] which is read in iteration *i*+1. The same is true of S2 for B[i] and B[i+1]. This is a "loop-carried dependence": between iterations

Dependencies between the statements S1 and S2 can be plotted using the state transition diagram to know if it is a cycle of dependency or not as shown:



The plotting of state transition diagram above gives a very clear picture that there is a cycle in the loop carried dependence and whenever there is a cycle of loop carried dependence, it cannot be eliminated and hence cannot carry out parallel execution.

Example 2: This example deals with the method of converting loop carried dependence into a non loop carried dependence when the loop carried dependence is not a cycle.

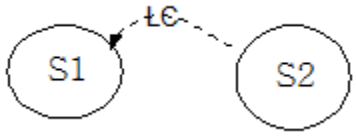
Consider a loop like this one:

for(i=1; i<=100; i=i+1) {

A[i] = A[i] + B[i]; /* S1 */

B[i+1] = C[i] + D[i]; /* S2 */}

State transition diagram for the dependency of the statements S1 and S2 of loop is shown below:



State transition diagram above shows that there is a loop carried dependence between the statements S1 and S2. But the loop carried dependence is not forming a cycle. Which means that the execution of the above loop can be carried out in parallel provided it has to be converted into **not loop carried dependence.**

Eliminating of loop carried dependence uses a simple technique were in a loop carried dependence is converted to **Not loop carried dependence** by the method of code conversion were the dependencies between the iterations of loop are converted to dependencies within the iteration of a loop.

To eliminate loop carried dependence for the code shown above, modifications for the code are made as shown below:

A[1] = A[1] + B[1];

for(i=1; i<=99; i=i+1) {

B[i+1] = C[i] + D[i];

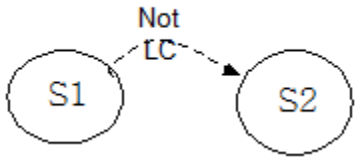
A[i+1] = A[i+1] + B[i+1];

}

B[101] = C[100] + D[100];

For eliminating, it divides loop code into three parts: start-up code(code before loop), loop body and end-up code(code after loop body). We will be discussing on this technique more in detail going forward.

State transition diagram for the above converted code is shown below:



The state transition diagram shown above gives a very clear picture that there is no more loop carried dependence between the statements S1 and S2. Even though there is a dependency between the statements, the dependencies are within the iteration of a loop. Hence the loop can be executed in parallel using some code scheduling techniques.

Loop carried dependence occurring in recurrence

A recurrence is when a variable is defined based on the value of that variable in an earlier iteration, often the one immediately preceding, as in the below fragment.

```
for (i=2;i<=100;i=i+1) {
Y[i] = Y[i-1] + Y[i];
}
```

When you have a recurrence relation of this form, there will be a cycle of loop carried dependence and the loop cannot be executed in parallel. Nevertheless some recurrence relations can be made to execute in parallel when there is good number of data dependence gap between the iterations of the loop as shown below:

```
for(i=6;i<=100;i=i+1) {
Y[i] = Y[i-5] + Y[i];
}
```

In the above code, there is a loop carried dependence, but the successive iteration has a gap of 5 iterations between them for the occurrence of the **true dependency.** By the time an iteration requires the result, the result would be computed by the previous iteration.

Conclusion: Even the recursive loop can be executed in parallel, provided there has to be a good number of gap between the dependent iterations.

Finding Dependences

Finding dependencies is an important part of three tasks of the compiler:

1. Good code scheduling

2. Determining parallelism of the loop.

3. Eliminating name dependencies.

The complexity of dependence analysis arises because of the presence of arrays and pointers in languages and there is a technique to find out the dependence when arrays are used in the loop and the technique is called as **GCD test.**

GCD Test

One of the heuristic tests that are being used to detect the dependence when an **affine** arrays are used in the loop. In simplest terms, a one-dimensional array index is affine if it can be written in the form a×i+b, where a is the displacement and b is the starting address c and I is the loop index variable. Sparse array accesses, which typically have the form x[y[i]], are one of the major examples of non affine accesses to which **GCD** test cannot be applied.

For example, suppose we have stored to an array element with index value $a*j+b$ and loaded from the same array with index value $c*k+d$, where j and k are for-loop index variable that runs from m to n.

Dependence exists if two conditions hold:

1. There are two iteration indices, j and k , both within the limits of the for loop. That is, $m<=j<=n$, $m<= k <= n$.

2. The loop stores into an array element indexed by $a*\cdot j + b$ and later fetches from that *same* array element when it is indexed by $c *k + d$. That is, $a * j + b = c \cdot k + d$.

In this situation, according to GCD test, if a loop carried dependence exists, then GCD(c,a) must divide d-b. I.e. (d-b) % GCD(c,a) = 0.

Examples:

1. Use the GCD test to determine whether dependences exist in the following loop:

for (i=1; i<=100; i=i+1) {

X[2*i+3] = X[2*i] * 5.0;

}

Answer

Given the values $a = 2$, $b = 3$, $c = 2$, and $d = 0$

GCD(a,c) = 2 and $d - b = -3$.

Since 2 does not divide –3, no dependence is possible.

2.Find Loop carried dependence using GCD test

ANSWER

for (i=0; i<300; i++)

{a[301*i+5] = b[i]*c[i];

d[i] = a[301*i]+e; }

GCD(c,a)=GCD(301,301)=301

d-b=0-5=-5

GCD(c,a) does not divide d-b

5%301!= 0

The GCD test breaks data dependence between the statements. Thus, there is no dependence between the statements in this loop.

Situations where Dependence Analysis Fails

1. When objects are referenced via pointers rather than array indices;

2. When array indexing is indirect through another array (Sparse arrays).

3. When a dependence may exist for some value of the inputs, but does not exist in actuality

Eliminating Dependent Computations

Compilers can reduce the impact of dependent computations so as to achieve more ILP. A key technique that is used to reduce or eliminate dependent computations is called as **Back substitution method.**

The back substitution method is again classified into two types:

1. Copy propagation

2. Tree height reduction method.

1. Copy propagation method:

One of the back substitution method that eliminates the instructions (operations) that copy values into another instruction.

The technique is similar to elimination data dependency by code elimination technique were in the modification for the dependent instruction is required once after eliminating the instruction that copies the value.

Consider the following instructions shown:

DADDUI R1,R2,#4

DADDUI R1,R1,#8

to

DADDUI R1,R2,#8

2. Tree height reduction method:

This method of back substitution technique depends on the rile of associativity, were in the dependence is eliminated by making the operation wider by reducing the height

The method can be used when there is a chain of dependence within the instructions and if the end instruction in the chain is used to commit the result. Method mainly aims on breaking the chain of dependence so that instructions can execute in parallel in same clock cycle.

Consider the following code sequence:

ADD R1,R2,R3

ADD R4,R1,R6

ADD R8,R4,R7

Notice that this sequence requires at least three execution cycles, since all the instructions depend on the immediate predecessor. By taking advantage of associativity, we can transform the code and rewrite it as

ADD R1,R2,R3

ADD R4,R6,R7

ADD R8,R1,R4

This sequence can be computed in two execution cycles. When loop unrolling is used, opportunities for these types of optimizations occur frequently.

Software Pipelining: Symbolic Loop Unrolling

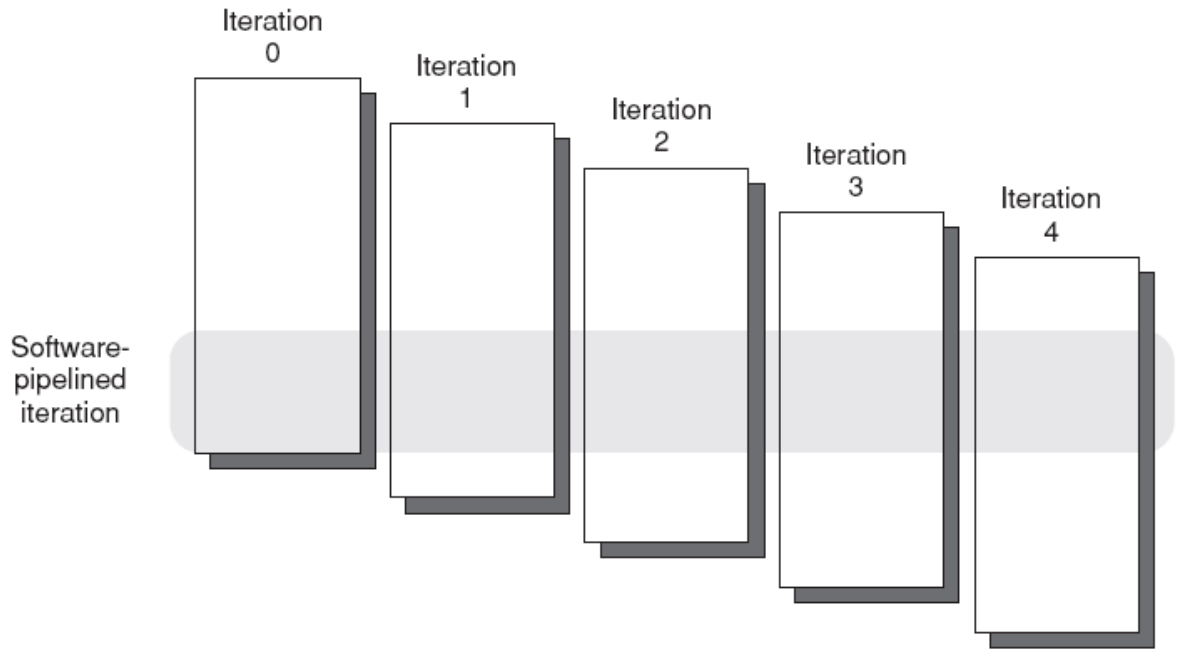
Software pipelining is a technique for reorganizing loops such that each iteration in the software-pipelined code is made from instructions chosen from different iterations of the original loop.

This technique separates the instruction from different loop iterations without unrolling the loop.

Software pipeline consists of three things: loop body, start-up code(prolog) that is needed before the loop begins and the clean up code(epilog), a code to finish up after the loop is completed.

The technique is also called as symbolic loop unrolling because initially it symbolically unrolls the loop based on the instructions in the iteration and then selects different instruction fron each iteration choosing nth instruction from the very first iteartion

Typical structure of the software pipeline is shown below:



A software-pipelined loop chooses instructions from different loop iterations, thus separating the dependent instructions within one iteration of the original loop. The start-up and finish-up code will correspond to the portions above and below the software-pipelined iteration.

To get a better understanding of the technique consider the following example:

Show a software-pipelined version of this loop, which increments all the elements of an array whose starting address is in R1 by the contents of F2:

```
Loop:  L.D F0,0(R1)
       ADD.D F4,F0,F2
       S.D F4,0(R1)
       DADDUI R1,R1,#-8
       BNE R1,Loop
```

First it carries out the symbolic loop unrolling for three different iterations as show:

Iteration i: L.D F0,0(R1)

ADD.D F4,F0,F2

S.D F4,0(R1)

DADDI R1,R1,#8

Iteration i+1: L.D F0,0(R1)

ADD.D F4,F0,F2

S.D F4,0(R1)

DADDI R1,R1,#8

Iteration i+2: **L.D F0,0(R1)**

ADD.D F4,F0,F2

S.D F4,0(R1)

DADDI R1,R1,#8

Then it starts selecting different instructions from different iterations which is marked with bolded letters as shown in the above code fragment. While selecting it starts from the selection of very last instruction from the first iteration.

Then before implementing the pipeline, it starts with the start up code that is required for the instructions of the loop body for the execution. Once after writing the start up code, then the loop body is written and the remaining code, which is the left out operation is written under clean up code section to complete the operation.

The code for this software pipeline is as shown:

```
        LD      F0,0(R1)
        ADDD    F4,F0,F2
        DADDI   R1,R1,#8
        LD      F0, 0(R1)
        DADDI   R1,R1,#8
LOOP:   S.D     F4,16(R1)       ;stores into M[i]
```

ADD.D	**F4,F0,F2**	**;adds to M[i-1]**
L.D	**F0,0(R1)**	**;loads M[i-2]**
DADDUI	**R1,R1,#-8**	
BNE	**R1,R1,Loop**	
ADDD	F4,F0,F2	
SD	F4, 8(R1)	

The major advantage of this technique is that, it consumes less code space compared to traditional loop unrolling method.

Hardware Support for Exposing Parallelism:

Predicated Instructions

Techniques such as loop unrolling and software pipelining can be used to increase the parallelism if the behavior of the branches can be well predicted at the **compile time.**

There are situations were the behavior of the branches cannot be predictable at the compile time, then control dependency severely limits the amount of parallelism. To overcome such problems, at the architectural level VLIW extends its support to include **conditional** or **predicated instructions** which has the capacity of eliminating branches converting control dependence to data dependence.

1. Conditional Instructions:

These are the type of instructions, were in condition and operation is embedded within a single instruction eliminating branch instruction.

If the condition in the instruction is true, instruction will be executed or else it will be truncated as a no operation instruction.

One example for this type of instruction is **conditional move,** which moves value from source to destination if the condition is evaluated to true

Example

Consider the following code:

if (A==0) {S=T;}

Assuming that registers R1, R2, and R3 hold the values of A, S, and T, respectively, show the code for this statement with the branch and with the conditional move.

Answer

The straightforward code using a branch for this statement is (remember that we are assuming normal rather than delayed branches)

```
BNEZ    R1,L
ADDU    R2,R3,R0
L:
```

Using a conditional move that performs the move only if the third operand is equal to zero, we can implement this statement in one instruction:

```
CMOVZ   R2,R3,R1
```

If R1 is equal to zero, then moves the value from R3 to R2.

A very useful type of conditional instruction but has a major limitation were in it cannot be applied for larger block of code since many conditional move needs to be included which leads to an inefficient coding.

2. Generalized predication

The type of predicated instruction is used to remedy the inefficiency of using conditional moves. In this type of predicate instruction execution of all instructions is controlled by a **single predicate.**

The type of instruction maintains two different instruction to evaluate both true and false condition for the block of the code, were in the previous technique was just making use of a conditional instruction for the true block.

When predicate is false instruction becomes **no op**. The type of instruction is basically used in an if else block kind of statements.

Example

Here is a code sequence for a two-issue VLIW that can issue a combination of one memory reference and one ALU operation, or a branch by itself, every cycle:

First instruction slot		Second instruction slot
LW	R1,40(R2)	ADD R3,R4,R5
		ADD R6,R3,R7
BEQZ	R10,L	
LW	R8,0(R10)	
LW	R9,0(R8)	

This sequence wastes a memory operation slot in the second cycle and will incur a data dependence stall if the branch is not taken, since the second LW after the branch depends on the prior load. Show how the code can be improved using a predicated form of LW.

Answer

Call the predicated version load word LWC and assume the load occurs unless the third operand is 0. The LW immediately following the branch can be converted to an LWC and moved up to the second issue slot:

First instruction slot		Second instruction slot
LW	R1,40(R2)	ADD R3,R4,R5
LWC	R8,0(R10),R10	ADD R6,R3,R7
BEQZ	R10,L	
LW	R9,0(R8)	

Hardware Support for Compiler Speculation

Any processor that carries out speculation requires three abilities in order to carry out efficient speculation:

1. The ability of the compiler to find instructions that, with the possible use of register renaming, can be speculatively moved and not affect the program data flow

2. The ability to ignore exceptions in speculated instructions, until we know that such exceptions should really occur

3. The ability to speculatively interchange loads and stores, or stores and stores, which may have address conflicts

In this section, we mainly deal with the first two abilities mentioned above in detail to know exactly how they can be implemented and how they carry out an efficient speculation.

Hardware Support for Preserving Exception Behavior

To carry out the maximum utilization of speculation, we must be able to move any type of instruction and still preserve its exception behavior.

To make this possible, one of the key idea used is, result of the speculated sequence that is mis predicted will not be used in the final computation.

Apart from this basic requirement, it is also very important to make sure that erroneous exceptions are never executed until they become non speculative. Because once if the exceptions of the speculative instructions are executed and if at all if they cause the termination of the program for the wrong prediction, then lot of processor resources would be wasted and also unnecessarily causes the exception for the correct program.

To implement this ability, four methods that support this are:

1. Hardware and OS co operatively ignore exceptions for the speculative instruction.

2.Speculative instructions that never raise exceptions are used and checks are introduced when an exception should occur.

3. Poison bits attached to the result registers are set to 1 written by the speculated instructions when instructions cause exceptions.

4. Instruction results are buffered until it is certain that instruction is no longer speculative- **Reorder buffer technique.**

1. Compiler based speculation

This speculation method uses as very simple technique, were in terminating exception raised for the speculated instructions are neglected and execution process is continued as normal instruction applying an undefined value for the exceptional instruction.

The technique works fine when correct programs are executed and is not a best technique when handling the wrong programs.

Example

Consider the following code fragment from an if-then-else statement of the form

if (A==0)

A = B;

else A = A+4;

where A is at 0(R3) and B is at 0(R2):

```
LD      R1,0(R3)        ;load A
BNEZ    R1,L1           ;test A
LD      R1,0(R2)        ;then clause
J       L2              ;skip else
L1:     DADDI R1,R1,#4  ;else clause
L2:     SD R1,0(R3)     ;store A
```

Assume the then clause is *almost always* executed. Compile the code using compiler-based speculation. Assume R14 is unused and available.

Answer

Here is the new code implemented through compiler speculation:

```
LD      R1,0(R3)        ;load A
LD      R14,0(R2)       ;speculative load B
BEQZ    R1,L3           ;other branch of the if
DADDI   R14,R1,#4       ;the else clause
L3:     SD R14,0(R3)    ;nonspeculative store
```

A simple technique were in register renaming is used for speculated instruction to overcome the branch instruction dependency which avoids destruction of register R1 value.

2. Preserving exception behavior using speculative versions of instructions

The method uses the versions of the speculative instructions that never generate the terminating exceptions when raised.

Check instructions are used to check if the exception has really occurred for speculative instructions or not when they become non speculative.

When it is non speculative and if there is a terminating exception, program is terminated.

For example sLD is the speculative version of load instruction that never raise exception and SPECK is the instruction used for checking the exception of speculated instruction

Example

Show how the previous example can be coded using a speculative load (sLD) and a speculation check instruction (SPECCK) to completely preserve exception behavior. Assume R14 is unused and available.

Answer

Here is the code that achieves this:

```
LD      R1,0(R3)        ;load A
sLD     R14,0(R2)       ;speculative, no termination
BNEZ    R1,L1           ;test A
SPECCK  0(R2)           ;perform speculation check
J       L2              ;skip else
L1:     DADDI R14,R1,#4 ;else clause
L2:     SD R14,0(R3)    ;store A
```

In the above code as you can see that LD is speculated and if at the exception is raised it is neglected at that point and then the branch condition is checked, here it will confirm if the predicted instruction is speculative or non speculative. If the predicted instruction is non speculative it will check for the exception occurrence through SPECCK code.

3. Adding of poison bit

A very efficient and reliable technique compared to previous two techniques. In this approach two status bits are added to every register: one of them is the poison bit and the other one is the speculative bit.

poison bit of the destination register is set whenever a speculative instruction results in the terminating exceptions. Speculative bit is set only for the speculative instructions.

If speculative instructions use the register with poison bit on it, then it takes the value and does the execution.

If at all a normal instruction attempts use a register source with poison bit ON, then instruction causes fault and terminates the program.

Here is the code (an s preceding the opcode indicates a speculative instruction):

```
LD      R1,0(R3)        ;load A
sLD     R14,0(R2)       ;speculative load B
BEQZ    R1,L3 ;
DADDI   R14,R1,#4 ;
L3:     SD R14,0(R3)    ;exception for speculative LW
```

If the speculative sLD generates a terminating exception, the poison bit of R14 will be turned on. When the nonspeculative SW instruction occurs, it will raise an exception if the poison bit for R14 is on.

NOTE:

- **All the topics included in this notes are very important.**
- **Please do revert back if you have any difficulties in understanding any of the concepts**
- **Please do study Intel IA as it covers one question every time.**

The time for action is now. It's never too late to do something.

---------ALL THE BEST----------