Many coded loops perform only a few instructions, or are very ordered. For example, this loop in C:

```
for(int i=0;i<1000;i++)
     a[i] = b[i] + c[i];
```

might be compiled into this code in DLX

-- assume R1 contains the base address for the 'a' array
and R2 has the base address for the 'b' array and
R3 has the base address for the 'c' array and R4 starts at 1000

```
Loop: LW R5, 0(R2) ; element of b
     LW R6, 0(R3) ; element of c
     ADD R7, R6, R5 ; make next a
     SW 0(R1), R7 ;
     ADDI R1, R1, #4 ;
     ADDI R2, R2, #4 ; increment addresses
     ADDI R3, R3, #4
     SUBI R4, R4, #1 ; decrement loop var
     BNEZ R4, Loop
```

In that DLX fragment, fully half of the instructions in the loop are merely loop overhead -- needed to control the flow of the loop, and access, but adding commands. Also, you are required (in a simple DLX pipelike pipe) to stall for the branch control hazard every ninth instruction. Futhermore, as the load, add, and store form a partial ordering, you will be forced to stall waiting for each command to reach a certain point (with forwarding) or to finish (without) before executing the next instruction.

The goal, then, is to limit the amount of loop overhead as a proportion of the commands, to decrease the number of control hazards in the total run of the loop, and to fill the unavoidable stall spots with independent instructions. The first two can be done somewhat through loop unrolling, and the last through rescheduling.

 

Prev Next

While much of the time, the entity that performs the loop-unrolling will be an optimizing compiler, you the programmer can also do the task in your high-level language. Taking the loop

```
for(int i=0;i<1000;i++)
a[i] = b[i] + c[i];
```

You can quickly and simply halve the number of iterations by changing it to this, instead:

```
for(int i=0;i<1000;i+=2) {
a[i] = b[i] + c[i];
a[i+1] = b[i+1] + c[i+1];
}
```

Note, however, that with a stupid compiler (one that does not optimize well), the second fragment may be worse than the first, because of the extra addtions needed in each loop (i+1). In theory, both loops would be unrolled and rescheduled by the compiler to exactly the same assembly code. Be aware of outsmarting the compiler's optimizing techniques; the folk who write compilers today are very good at what they do.

 

Prev Next

Now before we begin, we make the following assumptions for this example:

- Assume 5 stage DLX Pipelike pipe.
- Assume Forwarding is implemented.
- Assume ignore control hazard.
- Assume ignore contributions to the branch.

First lets look at a simple example. Recall the example from earlier.

```
for(int i=0;i<1000;i++)
     a[i] = b[i] + c[i];
```

might be compiled into this code in DLX

-- assume R1 contains the base address for the 'a' array
and R2 has the base address for the 'b' array and
R3 has the base address for the 'c' array and R4 starts at 1000

```
Loop: LW R5, 0(R2) ; element of b
     LW R6, 0(R3) ; element of c
     ADD R7, R6, R5 ; make next a
     SW 0(R1), R7 ;
     ADDI R1, R1, #4 ;
     ADDI R2, R2, #4 ; increment addresses
     ADDI R3, R3, #4
     SUBI R4, R4, #1 ; decrement loop var
     BNEZ R4, Loop
```

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13
Loop: LW R5, 0(R2)	I	D	X	M	W								
LW R6, 0(R3)		I	D	X	M	W							
ADD R7, R6, R5			I	D	**s**	X	M	W					
SW 0(R1), R7				I	**s**	D	X	M	W				
ADDI R1, R1, #4						I	D	X	M	W			
ADDI R2, R2, #4							I	D	X	M	W		
ADDI R3, R3, #4								I	D	X	M	W	
ADDI R4, R4, #4									I	D	X	M	W

The average CPI for this is 13 clock cycles / 8 instuctions = 1.625. Several stalls happen due to this order. The third instruction "ADD 57, R6, R6" has to stall once for the R6 to finish loading to forward. This also adds a stall to the store for R7. However with a simple rescheduling we can remove these stalls.

Instruction	1	2	3	4	5	6	7	8	9	10	11	12

Loop: LW R5, 0(R2)	I	D	X	M	W							
LW R6, 0(R3)		I	D	X	M	W						
ADDI R2, R2, #4			I	D	X	M	W					
ADD R7, R6, R5				I	D	X	M	W				
ADDI R3, R3, #4					I	D	X	M	W			
SW 0(R1), R7						I	D	X	M	W		
ADDI R3, R3, #4							I	D	X	M	W	
ADDI R4, R4, #4								I	D	X	M	W

The average CPI for the loop is now 12 clock cycles / 8 instructions = 1.5. The CPI has dropped and there are no longer any stalls from the data. Now that we have seen how rescheduling can lower the CPI, lets try unrolling the loop.

 

Prev Next

Okay, so you've seen now how reordering can lower the CPI. However, we still have hit the nasty control hazard when we hit the branch, in this case 1000 times. We can fix this by resheduling and unrolling the loop.

-- assume R1 contains the base address for the 'a' array
and R2 has the base address for the 'b' array and
R3 has the base address for the 'c' array and R4 starts at 1000

```
Loop: LW R5, 0(R2) ; element of b
      LW R6, 0(R3) ; element of c
      ADD R7, R6, R5 ; make next a
      LW R8, 4(R2) ; next element of b
      LW R9, 4(R3) ; next element of c
      ADD R10, R8, R9 ; make next a + 1
      SW 0(R1), R7 ;
      SW 4(R1), R10 ;
      ADDI R1, R1, #8 ;
      ADDI R2, R2, #8 ; increment addresses
      ADDI R3, R3, #8
      SUBI R4, R4, #2 ; decrement loop var
      BNEZ R4, Loop
```

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Loop: LW R5, 0(R2)	I	D	X	M	W											
LW R6, 0(R3)		I	D	X	M	W										
LW R8, 4(R2)			I	D	X	M	W									
ADD R7, R6, R5				I	D	X	M	W								
LW R9, 4(R3)					I	D	X	M	W							
SW 0(R1), R7						I	D	X	M	W						
ADD R10, R8, R9							I	D	X	M	W					
SW 4(R1), R10								I	D	X	M	W				
ADDI R1, R1, #8									I	D	X	M	W			
ADDI R2, R2, #8										I	D	X	M	W		
ADDI R3, R3, #8											I	D	X	M	W	
SUBI R4, R4, #2												I	D	X	M	W

The average CPI for the loop now is 16 clock cycles / 12 instuctions = 1.333. This is a significent improvement from the orginal code, which had a CPI of 1.625. However, this resheduled and unrolled version only hits the BNEZ command half as many times as the orginal, so it has less overall stalls from the control hazard, 50% less. This amount can be lowered even more by unrolling the loop more. Remember that unrolling is limited by the number of registers you have.

Successful rescheduling of instructions depends on enough instructions being independant of each other. When this is not the case, a partial ordering of instructions. The simplest example of a partial ordering of instructions would be two ALU instructions in a row, where one depends on the result of the other. For example:

ADD R1, R2, R3
ADD R4, R1, R5

The second instruction must stall at least until the first has finished the execution of the addition. If you have an instruction indepentent of both, you could possibly move it between the two. Loads and stores not referencing the same registers as ALU commands, and loop iteration variables can many times be used for this. Like so:

ADD R1, R2, R3
ADD R4, R1, R5
LW F2, 40(R6)

could be changed to

ADD R1, R2, R3
LW F2, 40(R6)
ADD R4, R1, R5

With no loss of correctness, at least one of the stalls caused by waiting for R1 to be written to (or have executed the add, and forward) is thus removed. The problem of doing this consistently and well is twofold: first, how can you determine which instructions are dependant on each other, and which are 'merely' dependant on registers (that is, you could rename the target register in an instruction with no loss of correctness). For now, the examples are simple enough that you can test it by hand. Second, are there enough independant instructions to eliminate all stalls? For small loops, this may not be the case and you will need to stall a bit unless you unroll.

 

Prev Next

Okay, you say you want a longer example? Here it is. So far we've looked at unrolling and rescheduling on the simple 5-stage DLX pipelike pipe. Now we will look at an example using the Multi-cycle DLX.

- Assume Multi-cycle DLX.
- Assume Forwarding is implemented.
- Assume ignore BNEZ command in analysis of CPI.
- Assume ignore contributions to the branch.
- Assume use latencies given in the book. (Figure 3.43)

The DLX code is as follows:

```
ADDI R4, R0, #5200 ; make a float 5200
MVI2FP F4, R4 ;
CVTI2FP F4, F4 ; F4 has a float constant
ADD R1, R0, R0 ; init counter to 0
Loop: LF F2, 100(R1) ; F2 is array element, R1 has offset of lowest unused array element
      LF F3, 500(R1) ; F3 holds array element
      SUBF F5, F3, F2 ; perform subtraction
      ADDF F5, F5, F4 ; perform addition of a constant
      SF 1000(R1), F5 ; store the results
      ADDI R1, R1, #4 ; increment pointer
      SUBI R5, R1, #400 ; check pointer
      BNEZ R5, Loop ; branch while not done
```

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Loop: LF F2, 100(R1)	I	D	X	M	W													
LF F3, 500(R1)		I	D	X	M	W												
SUBF F5, F3, F2			I	D	**s**	A1	A2	A3	A4	M	W							
ADDF F5, F5, F4				I	**s**	D	**s**	**s**	**s**	A1	A2	A3	A4	M	W			
SF 1000(R1), F5						I	**s**	**s**	**s**	D	**s**	**s**	**s**	X	M	W		
ADDI R1, R1, #4										I	**s**	**s**	**s**	D	X	M	W	
SUBI R5, R1, #400														I	D	X	M	W

The average CPI for this loop is 18 clock cycles / 7 instructions = 2.571. Now lets see what happens when we unroll the loop 4 times, and reschedule to avoid stalls. The new code is as follows.

```
ADD R1, R0, RO
ADDI R4, R0, #5200 ; make a float 5200
MVI2FP F14, R4 ;
CVTI2FP F14, F14 ; F14 has a float constant
ADD R1, R0, R0 ; init counter to 0
```

```
Loop: LF F2, 100(R1);
      LF F6, 500(R1)
      LF F3, 100(R1);
      LF F7, 504(R1);
      SUBF F10, F6, F2;
      LF F4, 108(R1);
      SUBF F11, F7, F3;
      LF F8, 508(R1)
      LF F5, 112(R1)
      LF F9, 512(R1)
      SUBF F12, F8, F4
      SUBF F13, F9, F5
      ADDI R1, R1, #16
      ADDF F10, F10, F14
      ADDF F11, F11, F14
      ADDF F12, F12, F14
      ADDF F13, F13, F14
      SUBI R5, R1, #400
      SF 984(R1), F10
      SF 988(R1), F11
      SF 992(R1), F12
      SF 996(R1), F13
      BNEZ R5, Loop ; branch while not done
```

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Loop: LF F2, 100(R1)	I	D	X	M	W											
LF F6, 500(R1)		I	D	X	M	W										
LF F3, 104(R1)			I	D	X	M	W									
LF F7, 504(R1)				I	D	X	M	W								
SUBF F10, F6, F2					I	D	A1	A2	A3	A4	M	W				
LF F4, 108(R1)						I	D	X	M	W						
SUBF F11, F7, F3							I	D	A1	A2	A3	A4	M	W		
LF F8, 508(R1)								I	D	X	**s**	M	W			
LF F5, 112(R1)									I	D	**s**	X	**s**	M	W	
LF F9, 512(R1)										I	**s**	D	**s**	X	M	W
SUBF F12, F8, F4												I	**s**	D	A1	A2
SUBF F13, F9, F5														I	D	A1
ADDI R1, R1, #16															I	D
ADDF F10, F10, F14																I

Instruction	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
SUBF F12, F8, F4	A3	A4	M	W												
SUBF F13, F9, F5	A2	A3	A4	M	W											
ADDI R1, R1, #16	X	M	W													
ADDF F10, F10, F14	D	A1	A2	A3	A4	M	W									
ADDF F11, F11, F14	I	D	A1	A2	A3	A4	M	W								
ADDF F12, F12, F14		I	D	A1	A2	A3	A4	M	W							
ADDF F13, F13, F14			I	D	A1	A2	A3	A4	M	W						
SUBI R5, R1, #400				I	D	X	**s**	**s**	**s**	M	W					
SF 984(R1), F10					I	D	**s**	**s**	**s**	X	M	W				
SF 988(R1), F11						I	**s**	**s**	**s**	D	X	M	W			
SF 992(R1), F12										I	D	X	M	W		
SF 996(R1), F13											I	D	X	M	W	

The average CPI for this loop is now 31 clock cycles / 22 instructions = 1.409. This is significantly less then the original 2.571 CPI. Also you hit the BNEZ only 25% of the time of the original.

 

Prev Next

Ideally in a pipelined machine you want to get a CPI as close to 1 as possible. The first example showed the simple 5 stage DLX pipe-like-pipe and the improvements with unrolling and rescheduling. The longer example, using Multi-Cycle DLX, shows a better view of real world code, as just having integer capablity is not practical in this day an age. As the previous examples have shown with unrolling and rescheduling lowers the CPI. You have to remember though, that besides lowering the CPI, unrolling allows you to lessen the number of times you hit a control hazard which causes stalls.

Now it is time to try the self-test. Remember, there are usually more then one way to reschedule commands, and the only limitation on unrolling is the number of registers you have available to use.

 

Prev Next

For some basic practice, unroll the following code three times, and reschedule it to avoid as many stalls as you can. One possible answer will be found off the 'Answers' link below.

```
Loop: LW R3, 0(R1) ; load in an array entry
      ADDI R4, R3, #50 ; add a constant
      MULT R4, R4, R4 ; square the new value
      SW R4, 600(R1) ; store the new value
      ADDI R1, R1, #4 ; increment pointer
      SUBI R5, R1, #300 ; check whether ended
      BNEZ R5, Loop ; branch
```

Essentially, this loop adds fifty to a word in an array, and squares that, then stores the result into another array. An answer.

 

Prev Next

Important terms

control hazard
: *Hazard caused by the need to wait until a branch target is computed and resolved.*

compiler
: *Software that translates high-level code into assembly or machine code. Can (and usually does) perform many optimizations in the process.*

loop
: *Section of code that will be iterated more than one time through. Implies linear execution; that is, many times through that section of code without executing other instructions.*

optimization
: *Any of a number of methods used to improve performance by eliminating stalls, eliminating unnecessary instructions, etc.*

partial ordering
: *Series of instructions where the later are dependant on the earlier completing.*

rescheduling
: *Optimization with the goal of eliminating stalls.*

stall
: *That which keeps a pipelined machine from having a full pipe. Hazards can cause stalls.*

unrolling
: *Another optimization, this one done by putting multiple iterations of a high-level loop into the loop body in assembly and machine code.*

Prev Home