

Instruction Level Parallelism

María Jesús Garzarán
CS 498: Compiler Optimizations
Fall 2006

University of Illinois at Urbana-Champaign



Outline

- ◆ Basic Pipeline Schedule
- ◆ Loop Unrolling
- ◆ Static Multiple Issue
- ◆ VLIW
- ◆ Software Pipelining
- ◆ Global Code Scheduling

Based on Chapter 4 (Exploiting Instruction Level Parallelism with Software Approaches) in Computer Architecture: A Quantitative Approach, by J.L. Hennessy and D. A. Patterson.



Basic Pipeline Scheduling

- ◆ Want to find sequences of unrelated instructions that can be overlapped in the pipeline.
- ◆ Separate dependent instructions by a distance in clock cycles equal to the latency of the source instruction.
- ◆ Assumptions: basic 5-stage pipeline, 1-cycle delay branches, and fully pipelined FUs or replicated

Inst. producing result	Inst. using result	latency in clock cycles
FP ALU op	FP ALU op	3
FP ALU op	Store	2
Load	FP ALU op	1
Load	Store	0



3

Example

```
for (i=1000;i>0;i=i-1)
    x[i]=x[i] +10
```

```
loop: LD    F0, 0(R1)    ;F0 = array element
      ADD   F4,F0,F2    ;Add scalar constant
      SD    0(R1), F4   ;Store result
      DADDUI R1,R1,-#8  ;Decrement array ptr.
      BNEZ  R1,R2,loop ;Branch if R1!=R2
```



4

Scheduling within each loop iteration

```
loop: LD    F0, 0(R1)
      stall
      ADD   F4,F0,F2
      stall
      stall
      SD    0(R1), F4
      DADDUI R1,R1,-#8.
      stall
      BNEZ  R1,R2,loop
      stall
```

Original loop: 10 cycles

```
loop: LD    F0, 0(R1)
      DADDUI R1,R1,-#8
      ADD   F4,F0,F2
      stall
      BNEZ  R1,R2,loop
      SD    8(R1), F4
```

Scheduled loop: 6 cycles

Can we do better by scheduling across iterations?



5

Loop unrolling

```
loop: LD    F0, 0(R1)
      ADD   F4,F0,F2
      SD    0(R1),F4 ; drop SUBI and BNEZ
      LD    F6, -8(R1)
      ADD   F8,F6,F2
      SD    -8(R1),F8 ; drop SUBI and BNEZ
      LD    F10, -16(R1)
      ADD   F12,F10,F2
      SD    -16(R1), F12 ; drop SUBI and BNEZ
      LD    F14, -24(R1)
      ADD   F16,F14,F2
      SD    -24(R1), F16
      DADDUI R1,R1, -#32
      BNEZ  R1, R2, loop
      nop
```

This loop will execute in 28 cycles: each LD has 1 stall, each ADD has 2 stalls, DADDUI has 1, and BNEZ has 1



6

Schedule unrolled Loop

```
loop: LD    F0, 0(R1)
      LD    F6, -8(R1)
      LD    F10, -16(R1)
      LD    F14, -24(R1)
      ADD   F4, F0, F2
      ADD   F8, F6, F2
      ADD   F12, F10, F2
      ADD   F16, F14, F2
      SD    0(R1), F4
      SD    -8(R1), F8
      DADDUI R1, R1, -#32
      SD    16(R1), F12
      BNEZ  R1, R2, loop
      SD    8(R1), F16
```

Important steps:

- Hoist loads
- Push stores down
- The displacement of the store instructions must be changed

Total: 14 cycles, or 3.5 cycles per element.

All penalties are eliminated. CPI = 1



7

Loop Unrolling

- ◆ Unrolling:
 - replicate the loop body several times, adjusting the loop termination code
 - use different regs in every iteration, thus increasing register pressure
- ◆ In real life, we do not know the number of iterations:
 - assume n iterations, k bodies per iteration
 - separate into two loops:
 - One executes $(n \bmod k)$ times and has the original body
 - Other executes (n/k) times and has k bodies



8

Loop Unrolling Summary

- ◆ Advantages:
 - more ILP
 - fewer overhead instructions
- ◆ Disadvantages:
 - code size increases
 - register pressure increases
 - problem becomes worse in multiple issue processors
- ◆ Example: went from 6 cycles/elem to 3.5 cycles



9

Schedule with static multiple issue

- The processor can issue two instructions per cycle
- load, store, branch, or integer ALU
 - any floating point operation

```
loop: LD    F0, 0(R1)
      LD    F6, -8(R1)
      LD    F10, -16(R1)    ADD    F4, F0, F2
      LD    F14, -24(R1)   ADD    F8, F6, F2
      LD    F18, -32(R1)   ADD    F12, F10, F2
      SD    0(R1), F4       ADD    F16, F14, F2
      SD    -8(R1), F8     ADD    F20, F18, F2
      SD    -16(R1), F12
      DADDUI R1, R1, -#40
      SD    16(R1), F16
      BNE   R1, R2, loop
      SD    8(R1), F20
```



Runs in 12 cycles per iteration, or 2.4 cycles per element (versus 3.5 cycles)

10

VLIW Approach

- ◆ Superscalar hardware is tough
- ◆ E.g. for the two issue superscalar: every cycle we examine the opcode of 2 instr, the 6 registers specifiers and we dynamically determine whether one or two instructions can issue and dispatch them to the appropriate Functional Units(FU)
- ◆ VLIW approach
 - Packages multiple independent instructions into one very long instruction
 - Burden of finding independent instructions is on the compiler
 - No dependence within issue package or at least indicate when a dependence is present
 - Simpler hardware



11

VLIW Approach

- ◆ To keep all FU busy → need enough parallelism in code
- ◆ How to uncover parallelism?
 - Unrolling loops → eliminates branches
 - Scheduling code within basic block → local scheduling
 - Scheduling code across basic blocks → global scheduling
- ◆ One global scheduling technique: Trace Scheduling



12

VLIW approach

- ◆ Example: 2 mem + 2 FP + 1 int or branch
- ◆ ignore the branch delay slot
- ◆ Unroll the loop to eliminate stalls (issue 1 VLIW per cycle)

Mem ref 1	Mem ref 2	FP op 1	FP op 2	int op/ branch	Clock
LD F0,0(R1)	LD F8,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADD F4,F0,F2	ADD F8,F8,F2		3
LD F26,-48(R1)		ADD F12,F10,F2	ADD F16,F14,F2		4
		ADD F20,F18,F2	ADD F24,F22,F2		5
SD 0(R1), F4	SD -8(R1), F8	ADD F28,F26,F2			6
SD -16(R1), F12	SD -24(R1), F16				7
SD -32(R1), F20	SD -40(R1), F24			SUBI R1,R1,#48	8
SD 0(R1), F28				BNEZ R1,LOOP	9

- ◆ 7 copies of the body take 9 cycles (1.29 cycles per element)
- ◆ Note: Efficiency is 60% (slots with operation)
 - Requires a LARGE number of registers!



13

Compiler Support for Exploiting ILP

- ◆ If there is no loop-carried dependence in a loop, we can execute the iterations in parallel

```
for (i=1;i<=100;i=i+1)
    A[i]=B[i]+C[i]
```

- ◆ Recurrences have loop-carried dependences

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



14

Recurrences May Have ILP

- ◆ Recurrence with dependence distance of 5
for ($i=6; i \leq 100; i=i+1$)
 $Y[i]=Y[i-5]+Y[i]$
- ◆ Can unroll this loop and find 5 independent instructions per iteration \rightarrow good ILP
for ($i=6; i \leq 100; i=i+5$)
 $Y[i]=Y[i-5]+Y[i]$
 $Y[I+1]=Y[i-4]+Y[I+1]$
 $Y[I+2]=Y[i-3]+Y[I+2]$
 $Y[I+3]=Y[i-2]+Y[I+3]$
 $Y[I+4]=Y[i-1]+Y[I+4]$



15

Conditions for Detection of Data Dependences

- ◆ Easier if array indices are affine, i.e., can be written as $a \cdot i + b$, where i is the loop index variable.
- ◆ Determining if there is a dependence between two references to the same array in a loop \Leftrightarrow whether two affine functions can have the same value for different indices between the bounds of the loop

for ($i=1; i \leq 100; i=i+1$)
 $X[2*i+3] = X[2*i] * 5.0$

- ◆ Simple and sufficient test for the absence of dependences is the Greatest Common Divisor (GCD) test



16

GCD Test

- ◆ Given 2 affine references: $a*i+b$, $c*i+d$
- ◆ If a loop carried dependence exists, then the greatest common divisor of (c,a) must divide $(d-b)$
- ◆ If $\text{GCD}(a,b)$ does not divide $(d-b)$, then no loop carried dependence exists

for ($i=1;i\leq 100;i=i+1$)

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

- ◆ $\text{GCD}(2,2)=2$, $d-b = -3$, so no loop carried dependence
- ◆ Sometimes the GCD test succeeds and no dependence present (out of bounds)



17

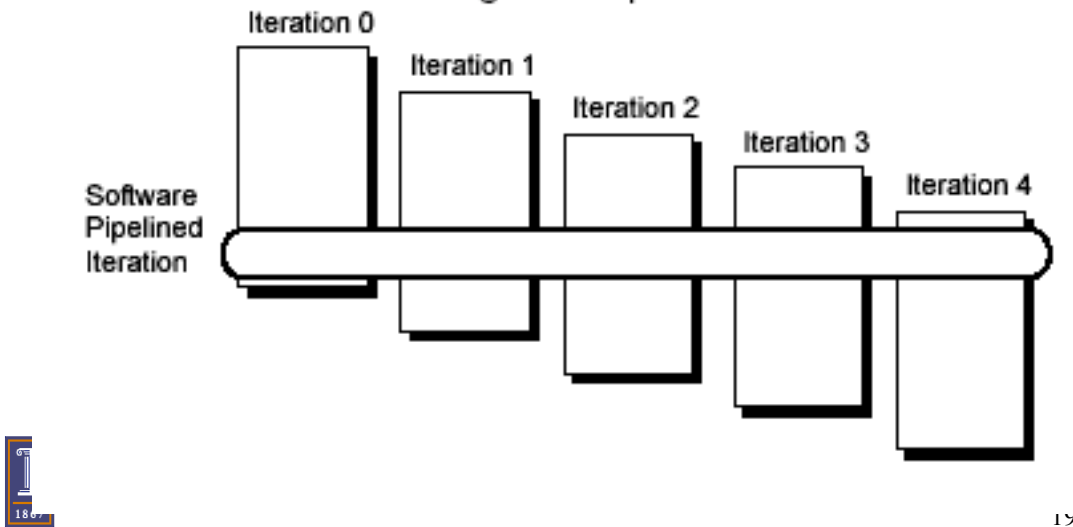
Software Pipelining

- ◆ Code reorganization technique to uncover parallelism
- ◆ Idea: each iteration contains instructions from several different iterations in the original loop
- ◆ The reason: separate the dependent instructions that occur within a single loop iteration
- ◆ We need some start-up code before the loop begins and some code to finish up after the loop is completed



18

- ◆ The instructions in a loop are taken from several iterations in the original loop



19

Software Pipelining

```

Loop: LD    F0,0(R1)
      ADDD  F4,F0,F2
      SD    F4,0(R1)
      DADDUI R1,R1,#-8
      BNE  R1,R2,Loop
  
```

```

Loop: SD F4,16(R1) ; stores into M[i]
      ADDD F4,F0,F2 ; adds to M[i-1]
      LD F0,0(R1) ; loads M[i-2]
      DADDUI R1,R1,#-8
      BNE R1,R2,Loop
  
```

```

It i:  LD F0,0(R1)
      ADDD F4,F0,F2
      SD F4,0(R1)
It I+1: LD F0,0(R1)
      ADDD F4,F0,F2
      SD F4,0(R1)
It I+2: LD F0,0(R1)
      ADDD F4,F0,F2
      SD F4,0(R1)
  
```



20

Software Pipelining

- ◆ Every 5 cycles, we get a result (ignoring the startup and cleanup portions) if we schedule appropriately
- ◆ Notice that there are no true dependences
- ◆ Because the load and store are separated by two iterations:
 - The loop should run for two fewer iterations
 - The startup code is: LD of iterations 1 and 2, ADDD of iteration 1
 - The cleanup code is: ADDD for last iteration and SD for the last two iterations



21

Software Pipelining

- ◆ Register management can be tricky
- ◆ Example shown is not hard: registers that are written in one iteration are read in the next one
- ◆ If we have long latencies of the dependences:
 - May need to increase the number of iterations between when we write a register and use it
 - May have to manage the register use
 - May have to combine software pipelining and loop unrolling



22

Software Pipelining vs Loop Unrolling

- ◆ Software pipelining consumes less code space
- ◆ Both yield a better scheduled inner loop
- ◆ Each reduces a different type of overhead:
 - Loop Unroll: branch and counter update code
 - Software Pipelining: reduces the time when the loop is not running at peak speed (only once at the beginning and once at the end)



23

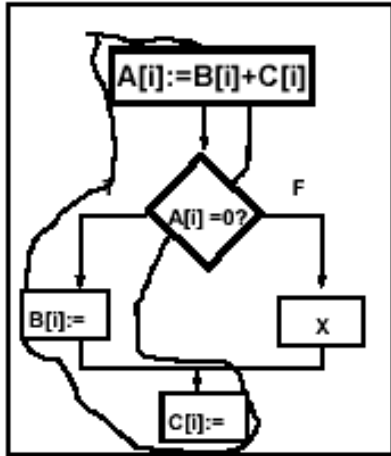
Global Code Scheduling

- ◆ Move instructions across branches (to improve the code scheduling)
- ◆ Goal: compact a code segment with internal control structure into the shortest sequence that preserves the data and control dependences --> critical path
 - Data dependences force partial order on operations.
 - Control dependences dictate instructions across which code cannot be easily moved.



24

Global Code Scheduling



- The leftmost sequence is chosen as the most likely trace
- The assignment to B is control dependent on the if statement.
- Trace compaction has to respect data dependences
- The rightmost (less likely) trace has to be augmented with fix up code



25

Global Code Scheduling

- ◆ Effectively scheduling the code may require that we move assignments to B and C before the branch
- ◆ The movement of the assignment to B is speculative: the branch may not be taken
- ◆ We can only move B if
 - We can ensure that it cannot cause an exception
 - We can ensure that it does not change the data flow (it will change the data flow if B is referenced before it is assigned, either in X or after the if statement) → for example, make a shadow copy of B before the assignment, and if the branch outcome is to go to X, use the shadow copy of B
- ◆ We move C before the test if the test does not depend on C and the X part does not use C
- ◆ This code movement will make sense only if the “then” case executes more frequently than the “else” case.



26

Trace Scheduling

- ◆ Way to organize the global code motion process
- ◆ Extends loop unrolling with a technique to find parallelism across conditional branches
- ◆ Useful for very wide issue processors: need a lot of ILP to keep processor busy
- ◆ It can have significant overheads in the infrequent path, so it is better to combine with profile information.
- ◆ Combination of two processes:
 - Trace Selection
 - Trace Compaction



27

Trace Selection

- ◆ Try to find likely sequence of basic blocks. This is called a Trace
- ◆ Uses loop unrolling to generate traces, since loop branches are taken with high probability
- ◆ Use static branch prediction to decide what blocks will likely execute in sequence



28

Trace Compaction

- ◆ Code scheduling : try to squeeze the trace into a small number of wide instructions
 - Moves operations as early as it can in the trace
 - Packs instructions into as few packets as possible
- ◆ Branches are viewed as jumps in and out of the selected trace, which is assumed to be the most probable path.
 - When exit and entry in the trace, we need special book-keeping code.



29

Hardware Support for Parallelism

- ◆ When branch behavior is not well known, compiler techniques may not be of much use
- ◆ In this case, use hardware techniques:
 - Add **Conditional** or **Predicated** instructions: used to eliminate branches and to assist the the compiler to move instructions up past branches
 - Support Speculation: allow the execution of an instruction before the processor knows that the instruction should execute:
 1. Static Speculation: Compiler chooses to make an instruction speculative
 2. Dynamic Speculation: Done by the HW using branch pred



30

Conditional or Predicated Instructions

- ◆ When the compiler is not enough...
- ◆ An instruction refers to a condition, which is evaluated as part of the instruction execution
- ◆ If condition is true: instruction is executed normally; if false: the instruction is a NO-OP
- ◆ New architectures include conditional instructions: e.g. conditional move: move a value from a reg to another one if a condition is true

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

BNEZ R1, L
ADDU R2,R3,R0

CMOVZ R2,R3, R1
;performs the move only if
; third op is 0

L:



31

Conditional or Predicated Instructions

- ◆ Allow us to convert control dep to data dep
- ◆ In a pipeline: moves the resolution from near the front of the pipeline to the end where the register write occurs!
- ◆ Another example: Conditional load, which loads only if the third operand is not zero

LWC R8, 20(R12), R10

- ◆ If this instruction used speculatively, we must ensure that it does not cause an exception
- ◆ Therefore, these conditional instructions, if condition not true:
 - No effect (not update any reg)
 - No exception



32

Conditional or Predicated Instructions

```
.... wasted ...      LW    R8, 20 (R10) , R10
BEQZ R10, L          BEQZ R10, L
LW    R8, 20 (R10)
```

- ◆ If R10 contains 0, it better not cause an exception
- ◆ Problems with conditional instructions:
 - Those that are annulled, still take time
 - Conditional instructions are most useful when the condition can be evaluated early
 - Sometimes it would be useful to have several conditions
 - Conditional instr may have some speed penalty relative to non conditional

