



ECSE 425 Lecture 11: Loop Unrolling

H&P Chapter 2

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer
Textbook figures © 2007 Elsevier Science

Last Time

- ILP is small within basic blocks
- Dependence and Hazards
- Change code, but preserve program correctness

Administrative Notes

- Wikinotes.ca
- No office hours tomorrow
 - Email me
 - Make an appointment for Thursday or Friday
- Homework
 - Homework 2 due Monday
 - Homework 3 out Monday, due October 17
- Midterm 1
 - 50 minutes, in class, October 12
 - Chapter 1, Appendix A, Chapter 2.1-2.3

Today

- Chapter 2.2
- Loop Unrolling

Stall Mitigation with Pipeline Scheduling

- *Goal*: keep the pipeline full
 - Avoid stalls due to hazards!
- *Approach*: Compile-time Pipeline Scheduling
 - Compiler sequences instructions to minimize stalls
 - Hardware executes instructions in their new order
- What can we do to keep the pipeline full?

Basic Pipeline Scheduling

- Compiler must separate dependent instructions from source instruction by the pipeline latency
- For example, in a pipeline with forwarding
 - EX stage (ALU) latency is 0
 - MEM latency is 1 (due to load interlock)
- Compiler is limited by
 - The amount of ILP in the program
 - The latencies of the functional units

Scheduling the MIPS Pipeline

- Assume the classic 5-stage integer pipeline
 - Integer ALU latency is 0 CC
 - Integer load latency is 1 CC
 - Branch delay is 1 CC
- Fully pipelined FUs (assume no structural hazards)
- Assume the following FP latencies (averages):

Producer	Consumer	Latency (CCs)
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Loop Example

- Adding a scalar to a vector (loop is parallel since the body of each iteration is independent)

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

```
Loop: L.D      F0, 0(R1)      ;F0=array element
      ADD.D    F4, F0, F2     ;add scalar from F2
      S.D      F4, 0(R1)     ;store result
      DADDUI   R1, R1, #-8    ;decrement ptr 8B
      BNE      R1, R2, Loop  ;branch R1!=R2
```

Loop Example: Unscheduled Code

- Ignore delayed branches for now
- Unscheduled code requires nine clock cycles

```
1  Loop: L.D      F0, 0(R1)
2      stall                ;load interlock
3      ADD.D     F4, F0, F2
4      stall                ;data hazard (F4)
5      stall                ;data hazard (F4)
6      S.D      F4, 0(R1)
7      DADDUI    R1, R1, #-8
8      stall                ;branch interlock
9      BNE      R1, R2, Loop
```

Loop Example: Scheduled Code

- Scheduled code: seven cycles!
- Not trivial: S.D depends on DAADUI!
 - Swap them but change S.D addressing

```
1  Loop: L.D      F0, 0(R1)
2          DADDUI  R1, R1, #-8
3          ADD.D   F4, F0, F2
4          stall                ;data hazard (F4)
5          stall                ;data hazard (F4)
6          S.D     F4, 8(R1)    ;0+8 = 8
7          BNE     R1, R2, Loop
```

Loop Example: Delayed Branch Slot

- One delayed branch slot
- Unscheduled code: 10 clock cycles

```
1  Loop: L.D      F0, 0(R1)
2      stall                ;load interlock
3      ADD.D      F4, F0, F2
4      stall                ;data hazard (F4)
5      stall                ;data hazard (F4)
6      S.D        F4, 0(R1)
7      DADDUI     R1, R1, #-8
8      stall                ;branch interlock
9      BNE        R1, R2, Loop
10     stall                ;delayed branch slot
```

Loop Example: Branch Slot Scheduling

- Scheduled code: 6 cycles

```
1  Loop: L.D      F0, 0(R1)
2          DADDUI  R1, R1, #-8
3          ADD.D   F4, F0, F2
4          stall                               ;data hazard (F4)
5          BNE     R1, R2, Loop
6          S.D     F4, 8(R1)      ; 0+8 = 8
```

But half of the instructions executed are loop overhead!

Loop Unrolling

- Unroll the loop
 - Replicate the body of the loop many times
 - Adjust the loop termination code
- Eliminating the branch allows instructions from different iterations to be scheduled together
 - In this case we can eliminate the data stall
 - We also increase the ratio of useful work to overhead
 - Doing so requires more registers

First Step: Copy the Loop

```
1 Loop: L.D      F0, 0(R1)
2      ADD.D     F4, F0, F2
3      S.D      F4, 0(R1)      ;drop DADDUI & BNE
4      L.D      F0, -8(R1)
5      ADD.D     F4, F0, F2
6      S.D      F4, -8(R1)     ;drop DADDUI & BNE
7      L.D      F0, -16(R1)
8      ADD.D     F4, F0, F2
9      S.D      F4, -16(R1)    ;drop DADDUI & BNE
10     L.D      F0, -24(R1)
11     ADD.D     F4, F0, F2
12     S.D      F4, -24(R1)
13     DADDUI    R1, R1, #-32   ; -8*4 = 32
14     BNE      R1, R2, LOOP
15     NOP
```

Second Step: Find Name Dependencies

```
1 Loop: L.D      F0, 0(R1)      ;1st iteration
2      ADD.D     F4, F0, F2
3      S.D       F4, 0(R1)
4      L.D       F0, -8(R1)     ;2nd iteration
5      ADD.D     F4, F0, F2
6      S.D       F4, -8(R1)
7      L.D       F0, -16(R1)   ;3rd iteration
8      ADD.D     F4, F0, F2
9      S.D       F4, -16(R1)
10     L.D       F0, -24(R1)
11     ADD.D     F4, F0, F2     ;4th iteration
12     S.D       F4, -24(R1)
13     DADDUI    R1, R1, #-32
14     BNE       R1, R2, LOOP
15     NOP
```

How do we go about removing them?

Third Step: Register Renaming

```
1 Loop: L.D      F0, 0(R1)
2      ADD.D     F4, F0, F2
3      S.D       F4, 0(R1)
4      L.D       F6, -8(R1)      ;renamed F0
5      ADD.D     F8, F6, F2      ;renamed F4,  F0
6      S.D       F8, -8(R1)      ;renamed F4
7      L.D       F10, -16(R1)    ;renamed F0
8      ADD.D     F12, F10, F2    ;renamed F4,  F0
9      S.D       F12, -16(R1)    ;renamed F4
10     L.D       F14, -24(R1)    ;renamed F0
11     ADD.D     F16, F14, F2    ;renamed F4,  F0
12     S.D       F16, -24(R1)    ;renamed F4
13     DADDUI    R1, R1, #-32
14     BNE       R1, R2, LOOP
15     NOP
```

Unrolled, Unscheduled

1	Loop:	L.D	F0, 0 (R1)	← 1 cycle stall (load interlock)
2		ADD.D	F4, F0, F2	← 2 cycles stall (data hazard)
3		S.D	F4, 0 (R1)	
4		L.D	F6, -8 (R1)	← 1 cycle stall (load interlock)
5		ADD.D	F8, F6, F2	← 2 cycles stall (data hazard)
6		S.D	F8, -8 (R1)	
7		L.D	F10, -16 (R1)	← 1 cycle stall (load interlock)
8		ADD.D	F12, F10, F2	← 2 cycles stall (data hazard)
9		S.D	F12, -16 (R1)	
10		L.D	F14, -24 (R1)	← 1 cycle stall (load interlock)
11		ADD.D	F16, F14, F2	← 2 cycles stall (data hazard)
12		S.D	F16, -24 (R1)	
13		DADDUI	R1, R1, #-32	
14		BNE	R1, R2, LOOP	← 1 cycle stall (branch interlock)
15		NOP		

15 + 4 (1 + 2) + 1 = 28 cycles, or 7 cycles per iteration!

Unrolled, Scheduled

```
1 Loop: L.D      F0, 0(R1)
2      L.D      F6, -8(R1)
3      L.D      F10, -16(R1)
4      L.D      F14, -24(R1)
5      ADD.D    F4, F0, F2
6      ADD.D    F8, F6, F2
7      ADD.D    F12, F10, F2
8      ADD.D    F16, F14, F2
9      S.D      F4, 0(R1)
10     S.D      F8, -8(R1)
11     DADDUI   R1, R1, #-32
12     S.D      F12, 16(R1)      ; -16+32 = 16
13     BNE     R1, R2, LOOP
14     S.D      F16, 8(R1)       ; -24+32 = 8
```

14 cycles, or 3.5 cycles per iteration!

Loop Unrolling Isn't Free

- Benefits decrease with additional unrolling
- Code length increases with additional unrolling
 - This is issue for embedded processors
 - This can increase instruction cache miss rates
- Uses lots of registers
 - Renaming requires many registers
 - When registers become scarce: “register pressure”
 - Aggressive unrolling and scheduling and cause a compiler to run out of registers to use for renaming

Compiler Perspectives: Unrolling Loops

- We don't usually know the upper bound of a loop
- Suppose it is n , and we want k copies of the loop
- Don't generate a single unrolled loop!
- Generate a pair of consecutive loops:
 - First executes the original loop $(n \bmod k)$ times
 - Second executes the unrolled loop body (n/k) times
 - For large n , most iterations occur in unrolled loop

Compiler Perspectives: Dependencies

- Compilers must preserve data dependencies
 - Determine if loop iterations are independent
 - Rename registers during unrolling
 - Eliminate extra test and branch instructions
 - Adjust loop maintenance and termination accordingly
 - Determine if loads and stores can be interchanged
 - Schedule the code, preserving dependencies

Compiler Perspectives: Renaming

- Dependent instructions can't execute in parallel
 - Easy to determine for registers (fixed names)
 - Much harder for memory
 - This is the “memory disambiguation” problem
 - Does $100(R4) = 20(R6)$?
 - In different loop iterations, does $20(R6) = 20(R6)$?
- In our example, compiler must determine that if R1 doesn't change then:
$$0(R1) \neq -8(R1) \neq -16(R1) \neq -24(R1)$$
- In this case, loads and stores can be interchanged

Summary

- Basic Pipeline Scheduling
- Looping Unrolling
 - Reduces loop overhead
 - Exposes additional instructions for scheduling
- Compiler unrolls a loop by
 - Copying the loop
 - Identifying and resolving name dependencies
 - Scheduling the loop
- Compiler must
 - Identify and preserve true data dependencies

Next Time

- Two Lectures on Branch Prediction
 - Chapter 2.3
 - Last material for the first Midterm