Compiler Instruction-Level Parallelism

Introduction

1. How can a compiler facilitate out-of-order execution?

Can Compilers Help Improve IPC?

1. ILP can be limited due to dependence chains
    - Compiler can attempt to avoid dependence chains
2. Hardware has limited "window" into a program
    - Independent instructions are far apart, so we run out of ROB space
    - Compiler has full knowledge of the program

Tree Height Reduction

1. R8 = R2 + R3 + R4 + R5
    - This can be done as follows, but has a long dependence chain
        - ADD R8, R2, R3
        - ADD R8, R8, R4
        - ADD R8, R8, R5
    - Alternatively, use associativity to facilitate parallelism
        - ADD R8, R2, R3
        - ADD R7, R4, R5
        - ADD R8, R8, R7
    - However, not all operations all associative

Tree Height Reduction Quiz

1. Consider the following program:
    - ADD R10, R1, R2
    - SUB R10, R10, R3
    - ADD R10, R10, R4
    - SUB R10, R10, R5
    - ADD R10, R10, R6
    - SUB R10, R10, R7
2. Rewrite reducing the tree height
    - ADD R10, R1, R2
    - ADD R11, R4, R6
    - ADD R10, R10, R11
    - ADD R11, R3, R5
    - ADD R11, R11, R7
    - SUB R10, R10, R10
3. Original ILP = 6 instructions / 6 cycles = 1
4. New ILP = 6 instructions / 3 cycles = 2

Make Independent Instructions Easier to Find

1. Real processor can only look so far ahead
    - Ideal processor can examine the entire program
2. Three techniques:
    - Instruction scheduling
    - Loop unrolling
    - Trace scheduling

Instruction Scheduling

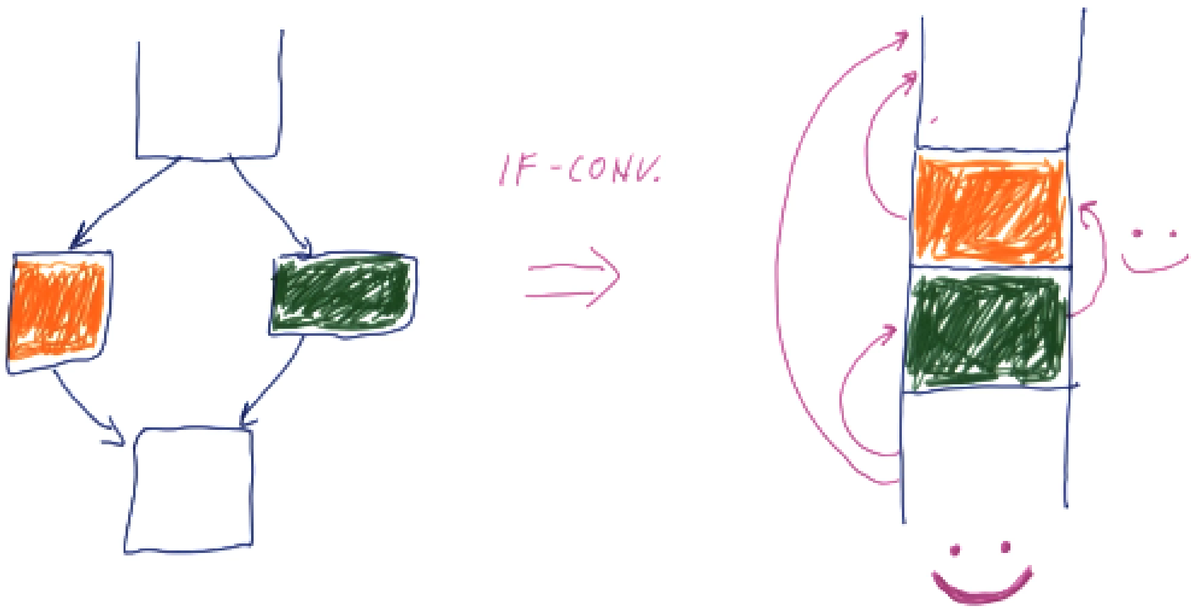
1. Consider the following program:
   - Loop: LW R2, 0(R1)
   - ADD R2, R2, R0
   - SW R2, 0(R1)
   - ADDI R1, R1, R4
   - BNE R1, R3, Loop
2. If a processor can only look at the next instruction, it must insert stalls between every instruction until they finish
3. Compiler could rewrite the program as follows to reduce the number of stalls
   - Loop: LW R2, 0(R1)
   - ADDI R1, R1, 4
   - ADD R2, R2, R0
   - SW R2, -4(R1)
   - BNE R1, R3, Loop
4. Idea of instruction scheduling is to find instructions that can be moved in place of stalls to reduce the number of idle cycles

Instruction Scheduling Quiz

1. Consider the following program:
   - LW R1, 0(R2)
   - ADD R1, R1, R3
   - SW R1, 0(R2)
   - LW R1, 0(R4)
   - ADD R1, R1, R5
   - SW R1, 0(R4)
2. Assumptions:
   - LW takes 2 cycles
   - ADD takes 1 cycle
   - SW takes 1 cycle
   - Processor can only complete one instruction per cycle and in-order
3. How many cycles does the program take to execute as-is?
   - 8 cycles
4. How many cycles does the program take to execute after instruction scheduling in the compiler?
   - 6 cycles

Scheduling and If Conversion

1. If-conversion helps with branch prediction by avoid branches
2. If-conversion can also help the compiler in improving instruction scheduling



If-Conversion

If-Conversion of Loops

1. Consider the following program:
   - Loop: LW R2, 0(R1)
   - ADD R2, R2, R3
   - SW R2, 0(R1)
   - ADDI R1, R1, 4
   - BNE R1, R2, Loop
   - Assume every instruction takes two cycles
2. The naive approach of predicating every iteration would be prohibitively expensive

Loop Unrolling

1. Consider the following program:
   - for(int i = 1000; i != 0; i–) { a[i] = a[i] + s; }
2. This is equivalent to:
   - Loop: LW R2, 0(R1)
   - ADD R2, R2, R3
   - SW R2, 0(R1)
   - ADDI R1, R1, 4
   - BNE R1, R2, Loop
3. Loop unrolling will optimize to the following:
   - for(i = 1000; i != 0; i-=2) { a[i] = a[i] + s; a[i-1] = a[i-1] + s; }
   - This has eliminated half of the conditional branches
   - Can unroll multiple times
   - IMPORTANT: This is unrolled only once; unrolling twice means another instruction inside the loop
4. This is equivalent to:
   - Loop: LW R1, 0(R1)
   - ADD R2, R2, R3
   - SW R2, 0(R1)

- LW R2, -4(R1)
- ADD R2, R2, R3
- SW R2, -4(R1)
- ASSI R1, R1, -8
- BNE R1, R5, Loop

Loop Unrolling Benefits ILP

1. Benefits of loop unrolling
    - Reduces the overall number of instructions to be executed
        - Previous example (original): 5 * 1000 = 5000 instructions
        - Previous example (unrolled): 8 * 500 = 4000 instructions
        - We've reduced the looping overhead (number of branches)
    - By reducing the number of instructions, this reduces the overall execution time
        - Iron Law: Execution time = # Insts * CPI * Cycle time

Loop Unrolling Benefits CPI

1. Assume we have a 4-issue in-order processor with perfect branch prediction
2. For the previous example, the CPI is 3/5
3. For the previous example with scheduling, the CPI is 2/5
4. For the previous example unrolled once, the CPI is 5/8
    - This is slightly worse, but with fewer total instructions
5. For the previous example unrolled once with scheduling, the CPI is 3/8
    - This is slightly better than the rolled example with scheduling
    - More unrolling allows for more parallelism due to more independent instructions

Loop Unrolling Quiz

1. Consider the following program:
    - Loop: LW R1, 0(R2)
    - ADD R3, R3, R1
    - ADDI R2, R2, 4
    - BNE R2, R4, Loop
2. Assumptions:
    - In-order, 1 instruction/cycle
    - LW: 3 cycles
    - ADD/ADDI: 2 cycles
3. As is, how many cycles per 1000 iterations?
    - (3 + 2 + 2) * 1000 = 7000
4. After scheduling, how many cycles per 1000 iterations?
    - ADDI moves up once. There's one stall cycle as the ADD waits for LW
    - (3 + 2) * 1000 = 5000
5. After unrolling once and scheduling, how many cycles per 1000 iterations?
    - Code after unrolling is as follows:
        - Loop: LW R1, 0(R2)
        - LW R8, 4(R2)
        - ADD R3, R3, R1
        - ADDI R2, R2, 8
        - ADD R3, R3, R8
        - BNE R2, R4, Loop
    - We save 1000 cycles by removing instructions (3000 instead of 4000)
    - We save 500 instructions by allowing better scheduling
    - Total = 5000 - 1000 - 500 = 3500

Unrolling Downside

1. Code bloat
    - Body of loop might be larger
    - More unrolling results in proportionally larger code
2. What if the number of iterations is unknown?
3. What if the number of iterations is not a multiple of N?
4. Answers to these questions exist, but are outside the scope of this class

Function Call Inlining

1. Normal function call:
    - Work…
    - Prepare function parameters
    - Call function
    - Jump, execute, return
    - More work…
2. With function call inlining, we can eliminate the overhead of calling a function
    - Work…
    - Execute
    - More work…
3. Also provides the opportunity for more scheduling
    - Previously can only schedule previous block, function call, and next block
4. Benefits:
    - Eliminates call/return overheads (decrease number of instructions)
    - Better scheduling (improved CPI)
    - Results in decreased execution time
    - Works better for small functions since overhead is high relative to the amount of work being done

Function Call Inlining Downside

1. Primary downside is code bloat
    - Entire purpose of function is to reduce duplicate code
2. Need to be judicious about when function inlining is performed
    - Ideally, pick small functions (overhead is relatively greater)

Function Inlining Quiz

1. Consider the following program:
    - LW A0, 0(R1)
    - CALL AddSq
    - SW RV, 0(R2)
    - AddSq: MUL A0, A0, A0
    - ADD Rv, A0, A1
    - RET
2. Assumptions:
    - LW: 2 cycles
    - CALL: 2 cycles
    - RET: 2 cycles
    - SW: 1 cycle
    - ADD: 1 cycle
    - MUL: 3 cycles
3. After scheduling, how many cycles will this program take to execute?
    - LW A0, 0(R1): Starts in cycle 1
    - CALL AddSq: Starts in cycle 2

- SW RV, 0(R2): Starts (and finishes) in cycle 10
- AddSq: MUL A0, A0, A0: Starts in cycle 4
- ADD Rv, A0, A1: Starts in cycle 7
- RET: Starts in cycle 8
  - Store finishes in cycle 10
  - Scheduling can't help in this case

4. After scheduling and inlining, how many cycles will this program take to execute?
   - LW A0, 0(R1): Starts in cycle 1
   - MUL A0, A0, A0: Starts in cycle 3
   - ADD Rv, A0, A1: Starts in cycle 6
   - SW RV, 0(R2): Starts (and finishes) in cycle 7
     - Store finishes in 7 cycles
     - Scheduling can't help in this case either

Other IPC Enhancing Compiler Stuff

1. Software pipelining: Schedule loops in a way that doesn't greatly increase code bloat while reaping the benefits of unrolling many times
   - Essentially, execute different iterations of the loop in a pipeline to improve parallelism
2. Trace scheduling: If-conversion on steroids; identify the common path
   - Blocks on common path are combined and scheduled freely
   - When a branch needs to happen, we have to fix the assumptions that were made in our re-ordering

Conclusion

1. Explained how compilers work with the branch-predicted, out-of-order, mulitple instruction-per-cycle processors present in modern systems