Instruction Level Parallelism 1
(Compiler Techniques)

Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- Tomasulo Algorithm
- Conclusion

Recall from Pipelining

- Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls
  - Ideal pipeline CPI: measure of the maximum performance attainable by the implementation
  - Structural hazards: HW cannot support this combination of instructions
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

Instruction-Level Parallelism

- Instruction-Level Parallelism (ILP)
  - Overlap the execution of instructions to improve performance
- 2 approaches to exploit ILP
  1. Rely on hardware to help discover and exploit the parallelism dynamically
     - Pentium 4, AMD Opteron, IBM Power
  2. Rely on software technology to find parallelism, statically at compile-time
     - Itanium 2 / IA-64

Instruction-Level Parallelism (ILP)

- Basic Block (BB) ILP is quite small
  - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - average dynamic branch frequency 15% to 25% => 4 to 7 instructions execute between a pair of branches
  - Plus instructions in BB likely to depend on each other
- Need ILP across multiple basic blocks

Loop-Level Parallelism

- Simplest: loop-level parallelism to exploit parallelism among iterations of a loop.
  - Example

```
for (i=1; i<=1000; i=i+1)
        x[i] = x[i] + y[i];
```

- Exploit loop-level parallelism by "unrolling loop" either by
  - dynamic via branch prediction or
  - static via loop unrolling by compiler

(Another way is vectors, to be covered later)

Loop-Level Parallelism

- **Determining dependences critical**
- **If 2 instructions are**
  - **parallel, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls (assuming no structural hazards)**
  - **dependent, they are not parallel and must be executed in order, although they may often be partially overlapped**

Data Dependence and Hazards

- $Instr_J$ is data dependent (aka true dependence) on $Instr_I$:
  1. $Instr_J$ tries to read operand before $Instr_I$ writes it

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

  2. or $Instr_J$ is data dependent on $Instr_I$
- If two instructions are data dependent, *they cannot execute simultaneously* or be completely overlapped
- Data dependence in instruction sequence
  ⇒ data dependence in source code
  ⇒ effect of original data dependence must be preserved
- If data dependence caused a hazard in pipeline, that's a Read After Write (RAW) hazard

ILP and Data Dependencies, Hazards

- HW/SW must preserve *illusion* of program order: order instructions would execute in if executed sequentially as determined by original source program
  - dependences are a property of programs
- Presence of dependence indicates potential for a hazard, but
  - actual hazard and length of any stall is property of the pipeline
- Importance of the data dependencies
  1) indicates the possibility of a hazard
  2) determines order in which results must be calculated
  3) sets an upper bound on how much parallelism can possibly be exploited
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program

Name Dependence #1: Anti-dependence

- Name dependence: when 2 instructions use same register or memory location, called a name, but no flow of data between the instructions associated with that name; 2 versions of name dependence
- $Instr_J$ writes operand *before* $Instr_I$ reads it

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

Called an "anti-dependence" by compiler writers.
This results from reuse of the name "r1"

- If anti-dependence caused a hazard in the pipeline, that's a Write After Read (WAR) hazard

Name Dependence #2: Output dependence

- $Instr_J$ writes operand *before* $Instr_I$ writes it.

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1"
- If anti-dependence caused a hazard in the pipeline, that's a Write After Write (WAW) hazard
- Instructions involved in a name dependence can execute simultaneously if name used in instructions is changed so instructions do not conflict
  - Register renaming resolves name dependence for registers
  - Either by compiler or by HW

Control Dependencies

- **Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order**

```
if p1 {
  S1;
};
if p2 {
  S2;
}
```

- **S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.**

Control Dependence Ignored

- Control dependence need not be preserved
  - willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program
- Instead, 2 properties critical to program correctness are
  - exception behavior and
  - data flow

Exception Behavior

- Preserving exception behavior
  - any changes in instruction execution order must not change how exceptions are raised in program (no new exceptions)
- Example:

```
      DADDU          R2,R3,R4
      BEQZ       R2,L1
      LW         R1,0(R2)
L1:
```

  - (Assume branches not delayed)
- Problem with moving `LW` before `BEQZ`?

Data Flow

- Data flow: actual flow of data values among instructions that produce results and those that consume them
  - branches make flow dynamic, determine which instruction is supplier of data
- Example:

```
        DADDU   R1,R2,R3
        BEQZ    R4,L
        DSUBU   R1,R5,R6
L:      …
        OR      R7,R1,R8
```

- OR depends on DADDU or DSUBU?
  Must preserve data flow on execution

Instruction-Level Parallelism

- Basic Block (BB) ILP is quite small
  - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - Average dynamic branch frequency 15% to 25% => 4 to 7 instructions execute between a pair of branches
  - Plus instructions in BB likely to depend on each other
- Need ILP across multiple basic blocks

Loop-Level Parallelism

- **Simplest: loop-level parallelism to exploit parallelism among iterations of a loop.**
  - **Example**

```
for (i=1; i<=1000; i=i+1)
        x[i] = x[i] + y[i];
```

- **Exploit loop-level parallelism by "unrolling loop" either by**
  - **Dynamic via branch prediction or**
  - **Static via loop unrolling by compiler**
- **Another way is vectors, to be covered later**

Compiler Techniques - Example

- **This code, add a scalar to a vector:**

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

- **Assume following latencies for all examples**
  - **Ignore delayed branch in these examples**

Instruction producing result	*Instruction using result*	*Latency in cycles*	*stalls between in cycles*
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	3	2
Load double	FP ALU op	1	1
Load double	Store double	1	0
Integer op	Integer op	1	0

FP Loop: Where are the Hazards?

- First translate into MIPS code:
  - To simplify, assume 8 is lowest address

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

```
Loop:  L.D     F0,0(R1)  ;F0=vector element
       ADD.D   F4,F0,F2  ;add scalar from F2
       S.D     0(R1),F4  ;store result
       DADDUI  R1,R1,-8  ;decrement pointer 8B (DW)
       BNEZ    R1,Loop   ;branch R1!=zero
```


FP Loop Showing Stalls

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

```
1 Loop:  L.D     F0,0(R1)    ;F0=vector element
2        stall
3        ADD.D   F4,F0,F2    ;add scalar in F2
4        stall
5        stall
6        S.D     0(R1),F4    ;store result
7        DADDUI  R1,R1,-8    ;decrement pointer 8B (DW)
8        stall               ;assumes can't forward to branch
9        BNEZ    R1,Loop     ;branch R1!=zero
```

plus branch delay!

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

- **9 clock cycles: Rewrite code to minimize stalls?**

Revised FP Loop Minimizing Stalls

```
1 Loop:  L.D      F0,0(R1)
2        DADDUI   R1,R1,-8
3        ADD.D    F4,F0,F2
4        stall
5        stall
6        S.D      8(R1),F4;altered offset when move DADDUI
7        BNEZ     R1,Loop
```

Swap DADDUI and S.D by changing address of S.D

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

7 clock cycles, but just 3 for execution (L.D, ADD.D,S.D), 4 for loop overhead; How make faster?

Unroll Loop Four Times (straightforward way)

```
1  Loop:L.D     F0,0(R1)
3       ADD.D   F4,F0,F2
6       S.D     0(R1),F4      ;drop DADDUI & BNEZ
7       L.D     F6,-8(R1)
9       ADD.D   F8,F6,F2
12      S.D     -8(R1),F8     ;drop DADDUI & BNEZ
13      L.D     F10,-16(R1)
15      ADD.D   F12,F10,F2
18      S.D     -16(R1),F12   ;drop DADDUI & BNEZ
19      L.D     F14,-24(R1)
21      ADD.D   F16,F14,F2
24      S.D     -24(R1),F16
25      DADDUI  R1,R1,#-32    ;alter to 4*8
27      BNEZ    R1,LOOP
```

1 cycle stall

2 cycles stall

Rewrite loop to minimize stalls?

27 clock cycles, or 6.75 per iteration

(Assumes R1 is multiple of 4)

Unrolled Loop Detail

- Do not usually know upper bound of loop
- Suppose it is n, and we would like to unroll the loop to make k copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
  - 1st executes (n mod k) times and has a body that is the original loop
  - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
- For large values of n, most of the execution time will be spent in the unrolled loop

Unrolled Loop That Minimizes Stalls

```
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     0(R1),F4
10      S.D     -8(R1),F8
11      S.D     -16(R1),F12
12      DADDUI  R1,R1,#-32
13      S.D     8(R1),F16 ; 8-32 = -24
14      BNEZ    R1,LOOP
```

14 clock cycles, or 3.5 per iteration

5 Loop Unrolling Decisions

- Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:
  1. Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code)
  2. Use different registers to avoid unnecessary constraints forced by using same registers for different computations

5 Loop Unrolling Decisions (cont.)

3. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code
4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent
   - » Transformation requires analyzing memory addresses and finding that they do not refer to the same address
5. Schedule the code, preserving any dependences needed to yield the same result as the original code

3 Limits to Loop Unrolling

1. Decrease in amount of overhead amortized with each extra unrolling
   - Amdahl's Law
2. Growth in code size
   - For larger loops, concern it increases the instruction cache miss rate
3. Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
   - If not be possible to allocate all live values to registers, may lose some or all of its advantage

- Loop unrolling reduces impact of branches on pipeline; another way is branch prediction