HW 2 is out!
Due 9/25!
CS 6290
Static Exploitation of ILP
Data-Dependence Stalls w/o OOO

- **Single-Issue Pipeline**
  - When no bypassing exists
  - Load-to-use
  - Long-latency instructions

- **Multiple-Issue (Superscalar), but *in-order***
  - Instructions executing in same cycle cannot have RAW
  - Limits on WAW
Solutions: Static Exploitation of ILP

• Code Transformations
  – Code scheduling, loop unrolling, tree height reduction, trace scheduling

• VLIW (later lecture)
Simple Loop Example

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

Loop:

L.D   F0,0(R1)       ; F0 = array element
ADD.D F4,F0,F2       ; add scalar in F2
S.D   F4,0(R1)       ; store result
DADDUI R1,R1,#-8     ; decrement pointer
8 bytes (per DW)

BNE   R1, R2, Loop   ; branch R1 != R2

Assume:

Single-Issue
FP ALU → Store    +2 cycles
Load DW → FP ALU  +1 cycle
Branch          +1 cycle
Scheduled Loop Body

Assume:
- FP ALU → Store: +2 cycles
- Load DW → FP ALU: +1 cycle
- Branch: +1 cycle

Loop:
- L.D F0,0(R1)
- DADDUI R1,R1,#-8
- ADD.D F4,F0,F2
- S.D F4,0(R1)
- BNE R1, R2, Loop

hoist the add
Scheduling for Multiple-Issue

A: \[ R1 = R2 + R3 \]
B: \[ R4 = R1 - R5 \]
C: \[ R1 = \text{LOAD 0[R7]} \]
D: \[ R2 = R1 + R6 \]
E: \[ R6 = R3 + R5 \]
F: \[ R5 = R6 - R4 \]

A: \[ R1 = R2 + R3 \]
C': \[ R8 = \text{LOAD 0[R7]} \]
B: \[ R4 = R1 - R5 \]
E': \[ R9 = R3 + R5 \]
D': \[ R2 = R8 + R6 \]
F': \[ R5 = R9 - R4 \]

Same functionality, no stalls
Interaction with RegAlloc and Branches

• Largely limited by architected registers
  – Weird interactions with register allocation ... could possibly cause more spills/fills

• Code motion may be limited:

R1 = R2 + R3
R8
BEQZ R9

R5 = R1 – R4
R8

R1 = LOAD 0[R6]

Need to allocate registers differently

Causes unnecessary execution of LOAD when branch goes left
(AKA Dynamic Dead Code)
Goal of Multi-Issue Scheduling

- Place as many independent instructions in sequence
  - “as many” → up to execution bandwidth
    - Don’t need 7 independent insts on a 3-wide machine
    - Avoid pipeline stalls
- If compiler is really good, we should be able to get high performance on an in-order superscalar processor
  - In-order superscalar provides execution B/W, compiler provides dependence scheduling
Why this Should Work

• Compiler has “all the time in the world” to analyze instructions
  – Hardware must do it in < 1ns

• Compiler can “see” a lot more
  – Compiler can do complex inter-procedural analysis, understand high-level behavior of code and programming language
  – Hardware can only see a small number of instructions at a time
Why this Might not Work

- Compiler has limited access to dynamic information
  - Profile-based information
  - Perhaps none at all, or not representative
  - Ex. Branch T in 1st ½ of program, NT in 2nd ½, looks like 50-50 branch in profile

- Compiler has to generate static code
  - Cannot react to dynamic events like data cache misses
Loop Unrolling

• Transforms an M-iteration loop into a loop with M/N iterations
  – We say that the loop has been unrolled N times

```c
for(i=0;i<100;i+=4){
a[i]*=2;
a[i+1]*=2;
a[i+2]*=2;
a[i+3]*=2;
}
```

Some compilers can do this (`gcc -funroll-loops`)
Or you can do it manually (above)
Why Loop Unrolling? (1)

• Less loop overhead

```c
for(i=0;i<100;i++)
a[i] += 2;
```

```c
for(i=0;i<100;i+=4){
a[i] += 2;
a[i+1] += 2;
a[i+2] += 2;
a[i+3] += 2;
}
```

How many branches?
Why Loop Unrolling? (2)

- Allows better scheduling of instructions
Why Loop Unrolling? (3)

• Get rid of small loops

```
for(i=0;i<4;i++)
a[i]*=2;
```

Difficult to schedule/hoist insts from bottom block to top block due to branches

Easier: no branches in the way

```
a[0]*=2;
a[1]*=2;
a[2]*=2;
a[3]*=2;
```
Loop Unrolling: Problems

- Program size is larger (code bloat)
- What if N not a multiple of M?
  - Or if N not known at compile time?
  - Or if it is a while loop?

```c
j1 = j - j % 4;
for (i = 0; i < j1; i += 4) {
    a[i] *= 2;
    a[i + 1] *= 2;
    a[i + 2] *= 2;
    a[i + 3] *= 2;
}
for (i = j1; i < j; i++)
    a[i] *= 2;
```
Function Inlining

• Sort of like “unrolling” a function
• Similar benefits to loop unrolling:
  – Remove function call overhead
    • CALL/RETN (and possible branch mispreds)
    • Argument/ret-val passing, stack allocation, and associated spills/fills of caller/callee-save regs
  – Larger block of instructions for scheduling

• Similar problems
  – Primarily code bloat
Tree Height Reduction

• Shorten critical path(s) using associativity

ADD R6, R2, R3
ADD R7, R6, R4
ADD R8, R7, R5

ADD R6, R2, R3
ADD R7, R4, R5
ADD R8, R7, R6

R8 = ((R2 + R3) + R4) + R5

R8 = (R2 + R3) + (R4 + R5)

Not all Math operations are associative!
C defines L-to-R semantics for most arithmetic
Trace Scheduling

• Works on all code, not just loops
  – Take an execution trace of the common case
  – Schedule code as if it had no branches
  – Check branch condition when convenient
  – If mispredicted, clean up the mess

• How do we find the “common case”
  – Program analysis or profiling
Trace Scheduling Example

\[
\begin{align*}
a &= \log(x); \\
\text{if}(b>0.01) & \{ \\
    c &= a/b; \\
    y &= \sin(c); \\
\} \text{else} & \\
    c &= 0; \\
\} \\
y &= \sin(c); \\
\end{align*}
\]

Suppose profile says that \( b>0.01 \) 90\% of the time

\[
\begin{align*}
a &= \log(x); \\
\text{fixit:} & \\
    c &= 0; \\
    y &= \sin(0); \quad // \sin(0)
\end{align*}
\]

Now we have larger basic block for our scheduling and optimizations
Pay Attention to Cost of Fixing

- Assume the code for \( b > 0.01 \) accounts for 80% of the time
- Optimized trace runs 15% faster
- But, fix-up code may cause the remaining 20% of the time to be even slower!
- Assume fixup code is 30% slower

By Amdahl’s Law:

\[
\text{Speedup} = \frac{1}{0.2 + 0.8 \times 0.85} = \frac{1}{0.2 + 0.68} = \frac{1}{0.88} = 1.1636
\]

\( = + 17.6\% \) performance

\[
\text{Speedup} = \frac{1}{0.2 \times 1.3 + 0.8 \times 0.85} = \frac{1}{0.26 + 0.68} = \frac{1}{0.94} = 1.0638
\]

Over 1/3 of the benefit removed!