Lecture 6: Static ILP

• Topics: loop analysis, SW pipelining, predication, speculation (Section 2.2, Appendix G)

• Assignment 2 posted; due in a week
Loop Dependences

• If a loop only has dependences within an iteration, the loop is considered parallel → multiple iterations can be executed together so long as order within an iteration is preserved.

• If a loop has dependences across iterations, it is not parallel and these dependences are referred to as “loop-carried”.

• Not all loop-carried dependences imply lack of parallelism.
Examples

For \((i=1000; \ i>0; \ i=i-1)\)
\[
x[i] = x[i] + s;
\]

For \((i=1; \ i<=100; \ i=i+1)\) \{ 
  \[
  A[i+1] = A[i] + C[i]; \quad \text{S1} \\
  B[i+1] = B[i] + A[i+1]; \quad \text{S2}
  \]
\}

For \((i=1; \ i<=100; \ i=i+1)\) \{ 
  \[
  A[i] = A[i] + B[i]; \quad \text{S1} \\
  B[i+1] = C[i] + D[i]; \quad \text{S2}
  \]
\}

For \((i=1000; \ i>0; \ i=i-1)\)
\[
x[i] = x[i-3] + s; \quad \text{S1}
\]
Examples

For \( i=1000; \ i>0; \ i=i-1 \)
\[
x[i] = x[i-1] + s;
\]
No dependences

For \( i=1; \ i<=100; \ i=i+1 \) {
\[
A[i+1] = A[i] + C[i]; \quad S1
B[i+1] = B[i] + A[i+1]; \quad S2
\]
S2 depends on S1 in the same iteration
S1 depends on S1 from prev iteration
S2 depends on S2 from prev iteration

For \( i=1; \ i<=100; \ i=i+1 \) {
\[
A[i] = A[i] + B[i]; \quad S1
B[i+1] = C[i] + D[i]; \quad S2
\]
S1 depends on S2 from prev iteration

For \( i=1000; \ i>0; \ i=i-1 \)
\[
x[i] = x[i-3] + s; \quad S1
\]
S1 depends on S1 from 3 prev iterations
Referred to as a recursion
Dependence distance 3; limited parallelism
Constructing Parallel Loops

If loop-carried dependences are not cyclic (S1 depending on S1 is cyclic), loops can be restructured to be parallel

```
For (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i];          S1
    B[i+1] = C[i] + D[i];        S2
}
```

S1 depends on S2 from prev iteration

```
For (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];       S3
    A[i+1] = A[i+1] + B[i+1];   S4
}
```

S4 depends on S3 of same iteration

```
B[101] = C[100] + D[100];
```
Finding Dependences – the GCD Test

• Do $A[ai + b]$ and $A[ci + d]$ refer to the same element?

• Restrict ourselves to affine array indices (expressible as $ai + b$, where $i$ is the loop index, $a$ and $b$ are constants) – example of non-affine index: $x[y[i]]$

• For a dependence to exist, must have two indices $j$ and $k$ that are within the loop bounds, such that
  
  $aj + b = ck + d;$
  
  $aj – ck = d – b;$
  
  $G = GCD(a,c);$
  
  $(aj/G - ck/G) = (d-b)/G;$

• If $(d-b)/G$ is not an integer, the initial equality can not be true
Software Pipeline?!

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

- Advantages: achieves nearly the same effect as loop unrolling, but without the code expansion – an unrolled loop may have inefficiencies at the start and end of each iteration, while a sw-pipelined loop is almost always in steady state – a sw-pipelined loop can also be unrolled to reduce loop overhead

- Disadvantages: does not reduce loop overhead, may require more registers
Predication

- A branch within a loop can be problematic to schedule.
- Control dependences are a problem because of the need to re-fetch on a mispredict.
- For short loop bodies, control dependences can be converted to data dependences by using predicated/conditional instructions.
Predicated or Conditional Instructions

• The instruction has an additional operand that determines whether the instr completes or gets converted into a no-op

• Example: `lwc R1, 0(R2), R3` (load-word-conditional)
  will load the word at address (R2) into R1 if R3 is non-zero; if R3 is zero, the instruction becomes a no-op

• Replaces a control dependence with a data dependence (branches disappear); may need register copies for the condition or for values used by both directions

```
if (R1 == 0)
  R2 = R2 + R4
else
  R6 = R3 + R5
R4 = R2 + R3
```

```
R7 = !R1 ; R8 = R2 ;
R2 = R2 + R4  (predicated on R7)
R6 = R3 + R5  (predicated on R1)
R4 = R8 + R3  (predicated on R1)
```
Complications

• Each instruction has one more input operand – more register ports/bypassing

• If the branch condition is not known, the instruction stalls (remember, these are in-order processors)

• Some implementations allow the instruction to continue without the branch condition and squash/complete later in the pipeline – wasted work

• Increases register pressure, activity on functional units

• Does not help if the br-condition takes a while to evaluate
Support for Speculation

• In general, when we re-order instructions, register renaming can ensure we do not violate register data dependences.

• However, we need hardware support
  ➢ to ensure that an exception is raised at the correct point
  ➢ to ensure that we do not violate memory dependences
Detecting Exceptions

• Some exceptions require that the program be terminated (memory protection violation), while other exceptions require execution to resume (page faults)

• For a speculative instruction, in the latter case, servicing the exception only implies potential performance loss

• In the former case, you want to defer servicing the exception until you are sure the instruction is not speculative

• Note that a speculative instruction needs a special opcode to indicate that it is speculative
Program-Terminate Exceptions

• When a speculative instruction experiences an exception, instead of servicing it, it writes a special NotAThing value (NAT) in the destination register.

• If a non-speculative instruction reads a NAT, it flags the exception and the program terminates (it may not be desirable that the error is caused by an array access, but the core-dump happens two procedures later).

• Alternatively, an instruction (the *sentinel*) in the speculative instruction’s original location checks the register value and initiates recovery.
Memory Dependence Detection

• If a load is moved before a preceding store, we must ensure that the store writes to a non-conflicting address, else, the load has to re-execute

• When the speculative load issues, it stores its address in a table (Advanced Load Address Table in the IA-64)

• If a store finds its address in the ALAT, it indicates that a violation occurred for that address

• A special instruction (the sentinel) in the load’s original location checks to see if the address had a violation and re-executes the load if necessary
Dynamic Vs. Static ILP

• Static ILP:
  + The compiler finds parallelism → no scoreboarding → higher clock speeds and lower power
  + Compiler knows what is next → better global schedule
  - Compiler can not react to dynamic events (cache misses)
  - Can not re-order instructions unless you provide hardware and extra instructions to detect violations (eats into the low complexity/power argument)
  - Static branch prediction is poor → even statically scheduled processors use hardware branch predictors
  - Building an optimizing compiler is easier said than done

• A comparison of the Alpha, Pentium 4, and Itanium (statically scheduled IA-64 architecture) shows that the Itanium is not much better in terms of performance, clock speed or power
Title

• Bullet