Loop Unrolling – Defined

• Loop unrolling
  
  - A simple scheme for increasing the number of instructions relative to the branch and overhead instructions
  
  - Scheme simply replicates the loop body multiple times, adjusting the loop termination code.

Example 1: Loop Unrolling

• Source code (with x an array of doubles):
  
  ```c
  for ( i=1000 ; i>0 ; i-- ) x[i] = x[i]+s;
  ```

• MIPS assembly (let scalar value be in F2):
  
  ```mips
  Loop:  L.D F0,0(R1) ;F0-array element
          ADD.D F4,F0,F2 ;add scalar in F2
          S.D F4,8(R1) ;store result
          DADDUI R1,R1,#-8 ;decrement pointer
          BNE R1,R2,Loop ;branch R1!=R2
  ```
Example 3: with Loop Unrolling

Note:
- This is a 4-fold unroll: an n-fold unroll is possible.
- SUBI & BNEZ needed for antidependence
- Multiple offsets used.
- Rescheduling has not yet been done; hence, there will be a lot of stalls.
- Use of different registers per unrolled iteration will ease rescheduling.

Example 6: Eliminating Name Dependences

Limits to Loop Unrolling

- There are three different types of limits to the gains that can be achieved by loop unrolling:
  - A decrease in the amount of overhead amortized with each unroll.
  - Growth in code size that results from unrolling.
  - Potential shortfalls in registers as register pressure created by aggressive unrolling and scheduling.

Example 5: Eliminating Data Dependences

Loop Unrolling & Scheduling with Static Multiple Issue

The following unrolled superscalar loop now runs in 12 clock cycles per iteration, or 24 clock cycles per element, versus 35.5 for the scheduled and unrolled loop on the ordinary MIPS pipeline.
Static Branch Prediction

- Earlier we discussed predict taken, predict not taken static prediction strategies
  - Applied uniformly across all branches in program
- Static analysis in compiler may be able to do better, if it can nonuniformly predict whether each specific branch is likely to be taken
  - One way: Backwards taken, forwards not taken.
- If we can do better, it can help with static code scheduling to reduce data hazard stalls...
  - Also may assist later dynamic prediction

Profile-Based Predictor Statistics

Profile-based predictors are generally better for FP programs, which have an average misprediction rate of 8% (4% standard deviation), than for integer programs, which have an average misprediction rate of 15% (5% standard deviation).

Some Static Prediction Schemes

- Always predict taken
  - 34% misprediction rate on SPEC (range 9% - 54%)
- Backwards predict taken, forwards not taken
  - In SPEC, more than 1/3 of forwards are taken!
    - This does worse than “always predict taken” strategy
    - Usually not better than 30/40% misprediction rate
- Better than either: Use profile information
  - Collect statistics on earlier program runs.
  - Works well because individual branches tend to be strongly biased (taken or not) given average data
  - Bias remains stable across multiple runs

More on VLIW

- A technique for multiple-issue
  - Staticaly scheduled by compiler
- Difference vs. statically scheduled superscalar:
  - Compiler pre-collects instructions into issue packets
    - Avoids or marks dependences within issue packet
    - Avoids need for dynamic dependence detection
CSCI 380: Computer Architecture 4

Dr. J's Lecture Slides

4/22/2003

More on VLIW

- Example (our earlier "add a scalar to vector" code)
  - Unrolled loop on a 5-way VLIW
  - 2 memory references, 2 FP ops, and 1 integer operation per clock
  - Achieves 9/7 = ~1.3 cycles per array element!
  - 60% efficiency vs. peak instruction issue rate

Difficulties with early VLIW

- Increased code size:
  - Aggressively unrolling loops to expand basic blocks
  - Unfilled instruction slots are wasted bits in VLIW
  - Can be dealt with by alternative encodings or in-memory compression
- Lockstep operation:
  - All instructions in packet proceed in lockstep
  - Entire pipe must stall if one functional unit does
  - Difficult to statically predict some stalls
    - e.g., cache misses
- Binary code incompatibilities
  - Code layout depends on microarchitecture version!

Compiler Support for ILP

- Detecting and eliminating dependences
- Software pipelining: Symbolic loop unrolling
- Trace scheduling: Critical path scheduling

VLIW Processor Schematic

Carter (2002), Computer Architecture, McGraw-Hill, Figure 7.6, p. 156

Difficulties with early VLIW

- Increased code size:
  - Aggressively unrolling loops to expand basic blocks
  - Unfilled instruction slots are wasted bits in VLIW
  - Can be dealt with by alternative encodings or in-memory compression
- Lockstep operation:
  - All instructions in packet proceed in lockstep
  - Entire pipe must stall if one functional unit does
  - Difficult to statically predict some stalls
    - e.g., cache misses
- Binary code incompatibilities
  - Code layout depends on microarchitecture version!