Chapter 3 (Cont III): Exploiting ILP with Software Approaches

Exposing ILP (3.2)

- Want to find sequences of unrelated instructions that can be overlapped in the pipeline.
- Separate dependent instructions by a distance in clock cycles equal to the latency of the source instruction.

- Assumptions: basic 5-stage pipeline, 1-cycle delay branches, and fully pipelined FUs or replicated
- Note: In Figure 3.2, latency means “number of intervening instructions”

Loop Unrolling

- Example page 158-160
- Unrolling:
  - replicate the loop body several times, adjusting the loop termination code
  - use different regs in every iteration, thus increasing register pressure
- In real life, we do not know the number of iterations:
  - assume n iterations, k bodies per iteration
  - separate into two loops:
    - One executes (n mod k) times and has the original body
    - Other executes (n/k) times and has k bodies

Loop Unrolling Summary

- Advantages:
  - more ILP
  - fewer overheard instructions
- Disadvantages:
  - code size increases
  - register pressure increases
  - problem becomes worse in multiple issue processors

- Example: went from 9 cycles/elem to 3.5 cycles

VLIW Approach

- Superscalar hardware is tough
- E.g. for the two issue superscalar: every cycle we examine the opcode of 2 instr, the 6 registers specifiers and we dynamically determine whether one or two instructions can issue and dispatch them to the appropriate FUs
- VLIW:
  - packages multiple independent ops into one very long instruction
  - Burden of finding indep instructions is on the compiler
  - No deps within issue package or at least indicate when a dep is present
  - Hardware simpler
- Example VLIW instruction: 2 int, 2 fp, 2 mem, 1 branch
- An instruction has a set of fields for each FU → 112-168 bits per instruction

VLIW Approach

- Early VLIW:
  - rigid in their instruction formats
  - require recompilation of programs for different versions of the HW
- Newer VLIWs:
  - still require compiler to do most of the work to find/schedule instruc
  - however, add innovations to increase flexibility

VLIW Approach (3.7)

- To keep all FU busy → need enough parallelism in code
- How to uncover parallelism?
  - Unrolling loops →eliminates branches
  - Scheduling code within basic block → local scheduling
  - Scheduling code across basic blocks → global scheduling

- One global scheduling technique: Trace Scheduling

Example: 2 mem + 2 fp + 1 int or branch per instr

ignore the branch delay slot

unroll the typical loop to eliminate all stalls (issue 1 VLIW instruction per cycle)

Figure 3.16: seven copies of the body take 9 cycles, or 1.29 cycles/result, which is about twice as fast as superscalar.

Note: Efficiency is 60% (slots with operation)

Requires a LARGE number of registers!

Problems in the Original VLIW Model

- Increase in code size
  - Generating enough parallelism requires wide loop unrolling
  - Many instructions are not full
  - If no operation can be scheduled, whole instruction empty
- Limitations of lockstep operation
  - Any stall in any FU pipeline must cause entire processor to stall (all FUs kept synchronized)
  - Easy for deterministic FUs, but not for cache misses
  - Cache misses caused all FU to stall!

Problems in the Original VLIW Model

- Binary code compatibility
  - Code generation needs to know the detailed pipeline structure, including FUs and latencies
  - Migrating between successive implementations or between implementations with different issue width requires recompilation (unlike dynamic superscalars)
- Finding enough parallelism (problem for all multiple issue processors)
- EPIC (IA-64) solves some of these problems

Combating Code Size Increase

- Use clever encoding
  - Have one large immediate field used by any FU
  - Compress the instructions in main memory and expand them when read into the cache or decoded

Lock Step Operation

- Lock step operation is not acceptable. Recent VLIW processors:
  - FU operate more independently
  - Hardware checks for hazards and allows unsynchronized execution

Solving Migration Problem

- Use object code translation or emulation

Finding Enough Parallelism

- If need loop unrolling in FP programs: Vector processors can do just as well
- However, advantages of multiple issue procs over vector procs:
  - Ability to extract some parallelism from less structured codes
  - Ability to use a conventional, cheap cache-based mem system
- Consequently: use multiple issue procs and add a vector extension to them