Detecting and Enhancing Loop Level Parallelism

- Find all dependencies in the following loop (5) and eliminate as many as you can:

```c
for(i=1; i<100; i=i+1) {
    Z[i] = X[i] + c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = Y[i] + c; /* S4 */
}
```

Solution at page 325

---

Code Transformation

- Eliminating dependent computations
  - **Copy propagation**
    - `DADDUI R1, R2, #4` → `DADDUI R1, R2, #8`
  - **Tree height reduction**
    - `ADD R8, R4, R7` → `ADD R8, R4, R6` → `ADD R8, R4, R7`
    - `ADD R1, R2, R3` → `ADD R1, R2, R3` → `ADD R4, R6, R7`
    - `L.D F0, 0(R1)` → `L.D F0, 0(R1)` → `L.D F0, 0(R1)`
    - `S.D F4, 0(R1)` → `S.D F4, 0(R1)` → `S.D F4, 0(R1)`

Can be done in parallel

Can be done in parallel

Must be done sequentially

---

Software Pipelining

- Combining instructions from different loop iterations to separate dependent instructions within an iteration

---

Software Pipelining vs. Loop Unrolling

- Loop unrolling eliminates loop maintenance overhead exposing parallelism **between** iterations
  - Creates larger code
  - Software pipelining enables some loop iterations to run at top speed by eliminating RAW hazards that create latencies **within** iteration
  - Requires more complex transformations

---

Global Code Scheduling

- Sometimes loop iterations contain control flow instructions that complicate loop transformations

 RAW dependence so we would like to schedule some instruction in between


T[0] = T[i] + F[i]

More common flow
Global Code Scheduling

We can move assignment to B here if B is not used in X or after C

C\[i\] = …

Unroll the loop (which contains some conditional branches) and determine most frequently used path – critical path

Trace selection determines highly probable outcomes of branches and generates sequential portion of the code

Trace compaction then parallelizes the sequential portion

Trace Scheduling

A\[i\] = 0?

Many entrances and exits create a problem

Critical path

Hardware support can help us expose parallelism in these cases

Superblocks

Unroll the loop (which contains some conditional branches) and determine most frequently used path – critical path

Form only one entrance into a path but many exits

When we exit we will need to duplicate some code

Hardware Support for ILP wrt Loops

Loop unrolling, software pipelining and global code scheduling make sense if we can accurately predict loop behavior, i.e. branch outcome

Sometimes this is not the case

Hardware support can help us expose parallelism in these cases
Conditional Instructions

- Conditional instructions convert control dependence into data dependence

```
BNEZ R1, L
...
L: ADDU R2, R3, R0
```

In this case we had to resolve branch quickly to decide which instruction to fetch; if we used prediction and made mistake we had to flush several instructions.

In this case there is only one instruction that may be wasted if branch is taken, we can then simply not write into R2.

Conditional Instructions

- Conditional instructions waste resources when condition is not met
- Conditional instructions may take longer to execute and may create additional data hazards
- Conditional instructions are convenient for small branch body but not for a large one

Compiler Speculation with Hardware Support

- Compiler speculates by moving instructions from after the \( \text{if} \) branch (\( \text{then or else} \) part) to a code before the branch
- If it was wrong, we must be able to stop execution and to ignore exceptions created by speculative code
- It also must take care of data hazards that may be violated by moving loads across stores