Static Compiler Optimization Techniques

- We already examined the following static compiler techniques aimed at improving pipelined CPU performance:
  - Static pipeline scheduling (in ch 4.1).
  - Loop unrolling (ch 4.1).
  - Static branch prediction (in ch 4.2).
  - Static multiple instruction issue: VLIW (in ch 4.3).
  - Conditional or predicted instructions (in ch 4.5)

- Here we examine two additional static compiler-based techniques (in ch 4.4):
  - Loop-Level Parallelism (LLP) analysis:
    - Detecting and enhancing loop iteration parallelism
      - GCD test.
    - Software pipelining (Symbolic loop unrolling).
Loop-Level Parallelism (LLP) Analysis

- Loop-Level Parallelism (LLP) analysis focuses on whether data accesses in later iterations of a loop are data dependent on data values produced in earlier iterations.

  e.g. in

  ```c
  for (i=1; i<=1000; i++)
    x[i] = x[i] + s;
  ```

  the computation in each iteration is independent of the previous iterations and the loop is thus parallel. The use of `X[i]` twice is within a single iteration.

  ⇒ Thus loop iterations are parallel (or independent from each other).

- Loop-carried Dependence: A data dependence between different loop iterations (data produced in earlier iteration used in a later one).

- LLP analysis is normally done at the source code level or close to it since assembly language and target machine code generation introduces a loop-carried name dependence in the registers used for addressing and incrementing.

- Instruction level parallelism (ILP) analysis, on the other hand, is usually done when instructions are generated by the compiler.
LLP Analysis Example 1

• In the loop:

```c
for (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i]; /* S1 */
    B[i+1] = B[i] + A[i+1];} /* S2 */
}
```

(Where A, B, C are distinct non-overlapping arrays)

- S2 uses the value A[i+1], computed by S1 in the same iteration. This data dependence is within the same iteration (not a loop-carried dependence).
  ⇒ does not prevent loop iteration parallelism.

- S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] read in iteration i+1 (loop-carried dependence, prevents parallelism). The same applies for S2 for B[i] and B[i+1]
  ⇒ These two dependences are loop-carried spanning more than one iteration preventing loop parallelism.
LLP Analysis Example 2

• In the loop:

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

- **S1** uses the value B[i] computed by **S2** in the previous iteration (loop-carried dependence)
- This dependence is not circular:
  - **S1** depends on **S2** but **S2** does not depend on **S1**.
- Can be made parallel by replacing the code with the following:

```c
for (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];  /* Loop Completion code */
```
LLP Analysis Example 2

Original Loop:

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

Modified Parallel Loop:

```
for (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```
ILP Compiler Support: Loop-Carried Dependence Detection

- Compilers can increase the utilization of ILP by better detection of instruction dependencies.
- To detect loop-carried dependence in a loop, the GCD test can be used by the compiler, which is based on the following:
- If an array element with index: \( a \times i + b \) is stored and element: \( c \times i + d \) of the same array is loaded where index runs from \( m \) to \( n \), a dependence exist if the following two conditions hold:
  1. There are two iteration indices, \( j \) and \( k \), \( m \leq j \), \( K \leq n \) (within iteration limits)
  2. The loop stores into an array element indexed by:
     \[ a \times j + b \]
     and later loads from the same array the element indexed by:
     \[ c \times k + d \]
     Thus:
     \[ a \times j + b = c \times k + d \]
The Greatest Common Divisor (GCD) Test

- If a loop carried dependence exists, then:

$$\text{GCD}(c, a) \text{ must divide } (d-b)$$

Example:

```c
for(i=1; i<=100; i=i+1) {
    x[2*i+3] = x[2*i] * 5.0;
}
```

\[ a = 2 \quad b = 3 \quad c = 2 \quad d = 0 \]

\[ \text{GCD}(a, c) = 2 \]

\[ d - b = -3 \]

2 does not divide -3 \( \Rightarrow \) No dependence possible.
ILP Compiler Support: Software Pipelining (Symbolic Loop Unrolling)

- A compiler technique where loops are reorganized:
  - Each new iteration is made from instructions selected from a number of iterations of the original loop.
  - The instructions are selected to separate dependent instructions within the original loop iteration.
  - No actual loop-unrolling is performed.
  - A software equivalent to the Tomasulo approach.

- Requires:
  - Additional **start-up code** to execute code left out from the first original loop iterations.
  - Additional **finish code** to execute instructions left out from the last original loop iterations.
A software-pipelined loop chooses instructions from different loop iterations, thus separating the dependent instructions within one iteration of the original loop.
Software Pipelining Example

Before: Unrolled 3 times
1. L.D F0,0(R1)
2. ADD.D F4,F0,F2
3. S.D F4,0(R1)
4. L.D F6,-8(R1)
5. ADD.D F6,F6,F2
6. S.D F8,-8(R1)
7. L.D F10,-16(R1)
8. ADD.D F12,F10,F2
9. S.D F12,-16(R1)
10. DADDUI R1,R1,#-24
11. BNE R1,R2,LOOP

After: Software Pipelined
1. S.D F4,0(R1) ; Stores M[i]
2. ADD.D F4,F0,F2 ; Adds to M[i-1]
3. L.D F0,-16(R1); Loads M[i-2]
4. DADDUI R1,R1,#-8
5. BNE R1,R2,LOOP

Software Pipeline
Loop Unrolled

Time