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 and possibly making loop iterations independent.

  e.g. in
  
  ```
  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 important in software optimizations such as loop unrolling since it usually requires loop iterations to be independent.

- 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.

(In Chapter 4.4)
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).
  
  \( \Rightarrow \) 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 \( B[i+1] \)

  \( \Rightarrow \) These two dependencies 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
  ```

Loop Completion code
LLP Analysis Example 2

Original 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 */
}
```

Modified Parallel Loop:

```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 Start-up code
- **Iteration 1:**

### Iteration 1

### Iteration 98
- B[100] = C[99] + D[99];

### Iteration 99
- B[101] = C[100] + D[100];

---

### Loop-completion code
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 exists 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:

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

The GCD test is sufficient to guarantee no dependence
However there are cases where GCD test succeeds but no dependence exits because GCD test does not take loop bounds into account

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.
Showing Example Loop Iterations to Be Independent

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

<table>
<thead>
<tr>
<th>Iteration i</th>
<th>Index of x loaded</th>
<th>Index of x stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>17</td>
</tr>
</tbody>
</table>

Index of element stored: \(a \times i + b\)

Index of element loaded: \(c \times i + d\)

\(a = 2\), \(b = 3\), \(c = 2\), \(d = 0\)

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

\(d - b = -3\)

2 does not divide -3

\(\Rightarrow\) No dependence possible.

**What if GCD \((a, c)\) divided \(d - b\)?**
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 independent 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.

(In Chapter 4.4)
Software Pipelining: Symbolic Loop Unrolling

A software-pipelined loop chooses instructions from different loop iterations, thus separating the dependent instructions within one iteration of the original loop.
Show a software-pipelined version of the code:

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 F0,-8 (R1)
5. ADD.D F4,F0,F2
6. S.D F4,-8 (R1)
7. L.D F0,-16 (R1)
8. ADD.D F4,F0,F2
9. S.D F4,-16 (R1)
10. DADDUI R1,R1,#-24
11. BNE R1,R2,LOOP

After: Software Pipelined

1. L.D F0,0 (R1)
2. ADD.D F4,F0,F2
3. L.D F0,-8 (R1)
4. S.D F4,0 (R1) ;Stores M[i]
5. ADD.D F4,F0,F2 ;Adds to M[i-1]
6. L.D F0,-16 (R1);Loads M[i-2]
7. DADDUI R1,R1,#-8
8. BNE R1,R2,LOOP
9. S.D F4,0 (R1)
10. ADD.D F4,F0,F2
11. S.D F4,-8 (R1)

2 fewer loop iterations
Software Pipelining Example Illustrated

Assuming 6 original iterations for illustration purposes:

L.D F0, 0 (R1)
ADD.D F4, F0, F2
S.D F4, 0 (R1)

1                      2                     3                        4                          5                         6

1                         2                     3                     4

start-up code

L.D L.D L.D L.D L.D L.D L.D
ADD.D ADD.D ADD.D ADD.D ADD.D ADD.D
S.D S.D S.D S.D S.D S.D S.D

4 Software Pipelined loop iterations (2 iterations fewer)

finish code