Loop Unrolling
Static Branch Prediction
Static Multiple Issue: The VLIW Approach
Software Pipelining
Global Code Scheduling
Trace Scheduling
Superblock Scheduling
Conditional or Predicated Instructions
Fill stall slots of a pipeline with independent instructions to keep the pipeline full.

Example:

**Source**

```plaintext
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

**MIPS object code**

```plaintext
Loop
    LD F0,0(R1) ; F0 = array element
    ADD F4,F0,F2 ; add scalar in F2
    ST F4,0(R1) ; store result
    DADDUI R1,R1,#-8 ; decrement pointer
        ; 8 bytes (per DW)
    BNE R1,R2,Loop ; branch R1!=R2
```

<table>
<thead>
<tr>
<th>Instruction Producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>

Latencies of FP operations assumed

Clock cycle issued

- Loop: 10
- LD: 1
- ADD: 3
- ST: 5
- DADDUI: 7
- BNE: 9
- Stall: 10

Data dependency

Non-optimized code sequence
In the example, a critical path of a LD-ADD-Store chain needs at least 6 cycles because of dependencies and pipeline latencies.

But the actual work of operating on the array element takes just 3 of those 6 clock cycles.

Remaining 3 clock cycles consists of loop overhead and stall!

**Loop Unrolling**

- Enhance ILP and reduce the penalties of the control hazard
- Simply replicates the loop body multiple times, adjusting the loop termination code
- Register-reallocation is required to solve register conflicts among unrolled iteration bodies
Unrolling 4 times

<table>
<thead>
<tr>
<th>Loop</th>
<th>Instruction</th>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F0,0(R1)</td>
<td>Iteration 1</td>
</tr>
<tr>
<td>ADD</td>
<td>F4,F0,F2</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>F4,0(R1)</td>
<td>;drop DADDUI&amp;BNE</td>
</tr>
<tr>
<td>LD</td>
<td>F0,-8(R1)</td>
<td>Iteration 2</td>
</tr>
<tr>
<td>ADD</td>
<td>F4,F0,F2</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>F4,-8(R1)</td>
<td>;drop DADDUI&amp;BNE</td>
</tr>
<tr>
<td>LD</td>
<td>F0,-16(R1)</td>
<td>Iteration 3</td>
</tr>
<tr>
<td>ADD</td>
<td>F4,F0,F2</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>F4,-16(R1)</td>
<td>;drop DADDUI&amp;BNE</td>
</tr>
<tr>
<td>LD</td>
<td>F0,-24(R1)</td>
<td>Iteration 1</td>
</tr>
<tr>
<td>ADD</td>
<td>F4,F0,F2</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>F4,-24(R1)</td>
<td></td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1,R1,#-32</td>
<td>Loop overhead</td>
</tr>
<tr>
<td>BNE</td>
<td>R1,R2,Loop</td>
<td></td>
</tr>
</tbody>
</table>

Three branches and three decrement of R1 eliminated.
But 28(=6x4+2+2) cycles required if no scheduling adopted!

Rescheduling by using independent instructions in unrolled iterations

• Register renaming to avoid WAR and WAW hazards
• Index adjustment for data references and loop counting

Reduction of 28 cycles to 14, 3.5 cycles per element!
The key to most of the techniques that allow us to take advantage of ILP to fully utilize the potential of the function units in a processor is to know when and how the ordering among instructions may be changed.

In the case of the example, we had to make the following decisions and transformation:

1. Determine that it was legal to move the ST after the DADDUI and BNE, and find the amount to adjust the SD offset.
2. Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.
3. Use different registers to avoid unnecessary constraints that would be forced by using the same register for different computations.
4. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.
5. Determine that the loads and stores in the unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent. This transformation requires analyzing the memory addresses and finding that they do not refer to the same address.
6. Schedule the code, preserving any dependences needed to yield the same result as the original code.

Delayed Branches and Loads

- Its effectiveness partly depends on whether we correctly guess which way a branch will go.

Example

```
  LD R1,0(R2)
  DSUBU R1,R1,R3
  BEQZ R1,L
  OR R4,R5,R6
  DADDU R10,R4,R3

  L:
    DADDU R7,R8,R9
```

Case 1: The branch was almost always taken and the value of R7 was no needed on the fall-through path.

```
  LD R1,0(R2)
  DADDU R7,R8,R9
  DSUBU R1,R1,R3
  BEQZ R1,L
  OR R4,R5,R6
  DADDU R10,R4,R3

  L:
```

```
  Delay slot

  Fall-through path

  Taken path
```
Example

LD R1,0(R2)
DSUBU R1,R1,R3
BEQZ R1,L
OR R4,R5,R6
DADDU R10,R4,R3
L: DADDU R7,R8,R9

Delay slot

Fall-through path

Token path

Case 2: The branch was rarely taken and the value of R4 was not needed on the taken path.

LD R1,0(R2)
OR R4,R5,R6
DSUBU R1,R1,R3
BEQZ R1,L
DADDU R10,R4,R3
L: DADDU R7,R8,R9

- Always predict a branch as taken
  - The misprediction rate ranges from not very accurate (59%) to highly accurate (9%), depending on the individual SPEC programs (average 34%)

- Predict on the basis of branch direction
  - Choosing backward-going branches to be taken and forward-going branches to be not taken.
  - Direction-based prediction is unlikely to generate an overall misprediction rate of less than 30% to 40%; more than half of the forward-going branches are taken in the SPEC benchmarks.

- Predict branches on the basis of profile information collected from earlier runs.
  - An individual branch is often highly biased toward taken or not taken.
**Better prediction rate for the FP programs than integer ones**
- **FP Prgms**: Average misprediction rate of 9% with a standard deviation of 4%
- **INT Prgms**: Average misprediction rate of 15% with a standard deviation of 5%

**Profile-based predictor gives a longer span of instructions between mispredictions**
- 20 instructions in a predict-taken strategy and 110 instruction in the profile-based strategy.
  - Exploit higher ILP

**VLIW (Very Long Instruction Word) processors**
- Rely on compiler technology not only to minimize the potential data hazard stalls, but to actually format the instructions in a potential issue packet so that the hardware need not check explicitly for dependences.
  - *Statically scheduled superscalar processors*
- Simpler hardware for instruction issue mechanisms, while still exhibiting good performance through extensive compiler optimization
Today's Processors often 60% Idle

Execution Units Available Used Inefficiently

Increases Parallel Execution

More efficient use of execution resources

Compiler Views Wider Scope
VLIW processors
- Multiple, independent functional units are employed, but multiple independent instructions are not issued. Instead,
- A VLIW packages the multiple independent operations into one very long instruction, and a single long instruction is issued.
  - An instruction length of between 112 and 168 bits, 16 to 24 bits each operation; 5 to 6 operations packed into a VLIW instruction
- Since the burden for choosing the instructions to be issued simultaneously falls on the compiler, the hardware in a superscalar to make these issue decisions is unneeded.
  - Simple hardware
  - Wide scope for exploiting the ILP

Suppose a VLIW processor with 5 FUs for two memory references, two FP operations, and one integer operation or branch in every clock cycle.

Show an unrolled version of the loop $x[i]=x[i]+s$

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP operation 2</th>
<th>Integer operation/branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,0(R1)</td>
<td>LD F6,-8(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>LD F14,-24(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>LD F22,-40(R1)</td>
<td>ADD F4,F0,F2</td>
<td>ADD F6,F6,F2</td>
<td></td>
</tr>
<tr>
<td>LD F26,-48(R1)</td>
<td>ADD F12,F10,F2</td>
<td>ADD F16,F14,F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD F20,F18,F2</td>
<td>ADD F24,F22,F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST F4,0(R1)</td>
<td>ST F8,-8(R1)</td>
<td>ADD F28,F26,F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST F12,-16(R1)</td>
<td>ST F16,-24(R1)</td>
<td>DADDUI R1,R1,#-56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST F20,24(R1)</td>
<td>ST F24,16(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST F28,8(R1)</td>
<td>BNE R1, R2, Loop</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unrolled 7 times: 9 cycles for seven iterations (23 operations), 1.29 cycles per iteration (2.5 operations per cycle)
Increase in code size
- Ambitously unrolling loops is required to fill operation slots with effective operations as many as possible, but also increases code size
- Inefficient use of FUs
  - 60% of the FUs in the example used

Compressed format of VLIWs
- Instructions are stored in a compressed form without no-operations, and expanded when they are read into the cache or are decoded

Operations in lockstep
- No hazard detection hardware at all, and a stall in any functional unit pipeline must cause the entire processor to stall.
- Recent VLIW processors employ functional units that can operate independently with hardware mechanisms for hazard detections

Binary code compatibility
- Different numbers of functional units and unit latencies require different versions of the code, and it makes migration between successive different implementations very difficult.
- Binary translation (code morphing), emulation

Detecting and Enhancing Loop-Level Parallelism
- Analyzed at the source level or close to it, instead of the instruction level
- Involves determining what dependences exist among the operand in a loop across the iteration of that loop.
  - Loop-carried dependence
    - Data accesses in later iterations are dependent on data values produced in earlier iterations

Example: Consider a loop like this one:
```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 */
}
```
Loop-carried dependence
Intra-iteration dependence

What are the data dependences among the statements S1 and S2 in the loop?
For \(i=1; \ i<=100; \ i=i+1\) {
\[
A[i]=A[i]+B[i]; \quad /* \text{S1} */ \\
B[i+1]=C[i]+D[i]; \quad /* \text{S2} */
\]
}

- A loop-carried dependence between S1 and S2, but
- Neither statement depends on itself, and S2 does not depend on S1, although S1 depends on S2.

A loop-carried dependence between S1 and S2, but neither statement depends on itself. S2 does not depend on S1, although S1 depends on S2.

The absence of a cycle in the dependences means that dependences give a partial ordering on the statements and a loop is parallel.

There is no dependence from S1 to S2. If there were, then there would be a cycle in the dependences and the loop would not be parallel. Since this other dependence is absent, interchanging the two statements will not affect the execution of S2.

On the first iteration of the loop, statement S1 depends on the value of \(B[1]\) computed prior to initiating the loop. The dependence between the two statements is no longer loop carried, so that iterations of the loop may be overlapped, provided the statements in each iteration are kept in order.
Loop-carried dependences in the form of a recurrence.

\[
\text{for } (i=2; i<=100; i=i+1) \{ \\
    Y[i] = Y[i-1] + Y[i]; \\
\}
\]

Dependence distance \(k\)

- On the iteration \(i\), the loop references element \(i-k\)
- A recurrence with a dependence distance of 5
- For \((i=6; i<=100; i=i+1) \{ \\
    Y[i] = Y[i-5] + Y[i]; \\
\}\)

The larger distance, the more potential parallelism can be obtained by unrolling the loop.
- If we unroll the loop with a dependence distance of 5, there is a sequence of five statements that have no dependences.

Finding the dependences in a program is an important part of three tasks:

1. Good scheduling of code,
2. Determining which loops might contain parallelism, and
3. Eliminating name dependences.

How does the compiler detect dependences in general?

- Assume that array indices are affine, where an indices can be written in the form \(a \times i + b\)
- Determining whether there is a dependence between two references to the same array in a loop is thus equivalent to determining whether two affine functions can have the same value for different indices between the bounds of the loop.
A dependence exists if two conditions hold:

1. There are two iteration indices, j and k, both within the limits of the for loop.
   - \( m \leq j \leq n, m \leq k \leq n \)

2. The loop stores into an array element indexed by \( a \times j + b \) and later fetches from that same array element when it is indexed by \( c \times k + d \).
   - \( a \times j + b = c \times k + d \)

As many programs contain primarily simple indices where \( a, b, c, \) and \( d \) are all constants, it is possible to devise reasonable compile time tests for dependence.

- GCD test for finding dependences
  - If a loop-carried dependence exists, then GCD\((a,c)\) must divide \((d-b)\)
  - An integer, \( x \) divides another integer, \( y \), if we get an integer quotient when we do the division \( y/x \) and there is no remainder.

Use the GCD test to determine whether dependences exist in the following loop:

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

- Given the values \( a=2, b=3, \) and \( d=0 \), then GCD\((a,c)\)=2, and \( d-b=3 \).
- Since 2 does not divide -3, no dependence is possible.

The CGD test is sufficient to guarantee that no dependence exists, however there are cases whether the test succeeds but no dependence exists.

In general, determining whether a dependence actually exists is NP-complete, but many common cases can be analyzed precisely at low cost!
Dependence analysis is a critical technology for exploiting parallelism, and used effectively to compile programs to either vector computers or multiprocessors.

- We cannot detect array-oriented dependences in the following cases:
  - When objects are referenced via pointers rather than array indices
  - When array indexing is indirect through another array, which happens with many representations of sparse arrays
  - When a dependence may exist for some value of the inputs, but does not exist in actuality when the code is run since the inputs never take on those values

- But it applies only under the limited circumstances, namely, among references within a single loop nest and using affine index function.

Copy propagation

- DADDUI R1,R2,#4
- DADDUI R1,R1,#4
- DADDUI R1,R2,#8

Tree height reduction

- Make the height of the tree structure representing a computation **wider but shorter**.
Example:

```c
for (i=1; i<=100; i++) sum = sum +x;
```

Unrolling five times

```c
sum = sum +x1+x2+x3+x4+x5;
```

five operations needed.

```c
sum = ((sum +x1)+(x2+x3))+(x4+x5);
```

Only three dependent operations needed.

---

A technique for reorganizing loops such that each iteration in the software-pipelined code is made from instructions chosen from different iterations of the original loop.

- The scheduler essentially interleaves instructions from different loop iterations, so as to separate the dependent instructions that occur within a single loop iteration.

Software pipeline is the software-counterpart to what Tomasulo’s algorithm does in hardware.
Symbolically unrolling in a steady state

Iteration i:
LD  F0,0(R1)
ADD F4,F0,F2
ST  F4,0(R1)

Iteration i+1:
LD  F0,0(R1)
ADD F4,F0,F2
ST  F4,0(R1)

Iteration i+2:
LD  F0,0(R1)
ADD F4,F0,F2
ST  F4,0(R1)

Loop ST  F4,16(R1) ; stores into M[i]
ADD F4,F0,F2 ; adds to M[i-1]
LD  F0,0(R1) ; loads M[i-2]
DADDUI R1,R1,#-8
BNE R1,R2,Loop

Final software-pipelined code in a steady state

Loop ST  F4,16(R1) ; stores into M[i]
DADDUI R1,R1,#-8
ADD F4,F0,F2 ; adds to M[i-1]
BNE R1,R2,Loop
LD  F0,0(R1) ; loads M[i-2]

Five cycles per results without any stall in a steady state!

- Before the steady state code; LD for iterations 1 and 2, and Add for iteration 1 are needed. (*Start-up code, or also called prologue*)

- After the steady state code; ADD for the last iteration and SD for last two iterations are needed. (*Finish-up code, or also called epilogue*)
• Code scheduling beyond the boundaries of basic blocks to increase the ILP
  • Of course, preserves the data and control dependences
  • Global code motion requires estimates of the relative frequency of different paths, since moving code across branches will often affect the frequency of execution of such code.

How can we make global scheduling for this code fragment?
Can we move the assignments to B and C to earlier in the execution sequence, before the test of A???

Global code motion must satisfy a set of constraints to be legal.

Speculative code motion should be applied to the cases in which the path containing the code would be taken
- otherwise it will not speed the computation up!
If $C$ is moved to before the if test, the copy of $C$ in the else branch can usually be eliminated, since it will be redundant.

- What are the relative execution frequencies of the then case and the else case in the branch?
  - If the then case is much more frequent, the code motion may be beneficial. If not, it is less likely, although not impossible, to consider moving the code.
- What is the cost of executing the computation and assignment to $B$ above the branch?
  - It may be that there are a number of empty instruction issue slots in the code above the branch and that the instructions for $B$ can be placed into these slots that would otherwise go empty. This opportunity makes the computation of $B$ "free" at least first order.
- How will the movement of $B$ change the execution time for the then case?
  - If $B$ is at the start of the critical path for the then case, moving it may be highly beneficial.
- Is $B$ the best code fragment that can be moved above the branch? How does it compare with moving $C$ or other statements within the then case?
- What is the cost of the compensation code that may be necessary for the else case? How effectively can this code be scheduled, and what is its impact on execution time?
Trace scheduling is a way to organize the global code motion process, so as to simplify the code scheduling by incurring the costs of possible code motion on the less frequent paths.

- Because it can generate significant overheads on the designated infrequent path, it is best used where profile information indicates significant differences in frequency between different paths and where the profile information is highly indicative of program behavior independent of the inputs.

**First Step: trace selection**
- Tries to find a likely sequence of basic blocks whose operations will be put together into a smaller number of instructions (traces).
  - Loop unrolling used to generate long traces since loop branches are taken with high probability.
  - By using static branch prediction, other conditional branches are also chosen as taken or not taken, so that the resultant trace is a straight-line sequence resulting from concatenating many basic blocks.

**Second Step: trace compaction**
- Tries to squeeze the trace into a small number of wide instructions.
  - Trace compaction is code scheduling;
  - It attempts to move operations as early as it can in a sequence (trace), packing the operations into as few wide instructions (or issue packets) as possible.
  - When code is moved across such trace entry and exit points, additional bookkeeping code will often be needed on the entry or exit point.

This trace is generated by unwinding four times.
One of the major drawbacks of trace scheduling is that the entries and exits into the middle of the trace cause significant complications, requiring the compiler to generate and track the compensation code and often making it difficult to assess the cost of such code.

Superblocks are formed by a process similar to that used for traces, but are a form of extended basic blocks, which are restricted to a single entry point but allow multiple exits.

How can a superblock with only one entrance be constructed?

- Use tail duplication to create a separate block that corresponds to the portion of the trace after the entry.

The superblock approach reduces the complexity of bookkeeping and scheduling versus the more general trace generation approach, but may enlarge code size more than a trace-based approach.
Conditional or predicated instructions can be used to eliminate branches, converting a control dependence into a data dependence and potentially improving performance.

**Concept:** An instruction refers to a condition, which is evaluated as part of the instruction execution.

For a pipelined processor, this moves the place where the dependence must be solved from near the front of the pipeline, where it is resolved for branches, to the end of the pipeline, where the register write occurs.

Example: consider the following code:

```
if (A==0) {S=T;
```

Straightforward code:

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

```
L:
```

Code using a conditional move:

```
CMOVZ R2, R3, R1
```

Support full predication for all instruction:

- When the predicate is false, the instruction becomes a no-op.
- Full predication allows us to simply convert large blocks of code that are branch dependent.
- An if-then-else statement within a loop can be entirely converted to predicated execution, so that the code in the then case executes only if the value of the condition is true, and the code in the else case executes only if the value of the condition is false.

Predication is particularly valuable with global code scheduling, since it can eliminate nonloop branches, which significantly complicate instruction scheduling: Intel/HP IA-64.