Let the Compiler Do it - Pros and Cons

- **Pros**
  - No window size limitation, the whole program is there to see
  - Hardware is simple, so can push the clock rate
  - Pragmas and profile information can be used

- **Cons**
  - Binary code compatibility
  - Basic blocks are small - global code optimization is might hard
  - Lack of run-time knowledge - eg: memory dataflow problems

- Perhaps a mixture of the two?

Overview of the Chapter

- Basic Compiler Techniques for Exposing ILP
  - Loop Unrolling
  - Instruction Scheduling

- Static Branch Prediction

- Static Multiple Issue: VLIW Approach

Overview of the Chapter

- Advanced Compiler Support for Exposing and Exploiting ILP
  - Detecting and Enhancing Loop-Level Parallelism
  - Dependence Analysis
  - Software Pipelining
  - Global Code Scheduling

- Hardware Support for Exposing More Parallelism at Compile Time
  - Predicated Execution
  - Compiler Speculation with HW Support

- CASE STUDY: ITANIUM PROCESSOR
Overview of Software Approaches

<table>
<thead>
<tr>
<th>Technique</th>
<th>Reduces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop Threading</td>
<td>Control Stalls</td>
</tr>
<tr>
<td>Basic Pipeline Scheduling</td>
<td>RAW stalls</td>
</tr>
<tr>
<td>Dynamic Branch Prediction</td>
<td>Control Stalls</td>
</tr>
<tr>
<td>Issuing multiple instructions per cycle</td>
<td>Ideal CPI</td>
</tr>
<tr>
<td>Compiler dependence analysis</td>
<td>Ideal CPI and data stalls</td>
</tr>
<tr>
<td>Software pipelining and trace scheduling</td>
<td>RAW stalls and data stalls</td>
</tr>
<tr>
<td>Speculation</td>
<td>RAW stalls involving memory</td>
</tr>
</tbody>
</table>

Chapter 4 - VLIW

Assumptions for the Examples

- Default 5-stage MIPS pipeline structure

<table>
<thead>
<tr>
<th>Instruction Producing Value</th>
<th>Instruction Consuming Value</th>
<th>Intervening Instructions to Avoid Stalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF ALU Op</td>
<td>FPU ALU Op</td>
<td>3</td>
</tr>
<tr>
<td>FF ALU Op</td>
<td>Store Double</td>
<td>2</td>
</tr>
<tr>
<td>Load Double</td>
<td>FF ALU Op</td>
<td>1</td>
</tr>
<tr>
<td>Load Double</td>
<td>Store Double</td>
<td>0</td>
</tr>
</tbody>
</table>

Remember these are called latencies in your text.

Chapter 4 - VLIW

A Simple Loop

Consider adding a scalar s to a vector

```
for (i = 0; i < 1000; i++)
    s[i] += s[i] + s
```

Loop:

```
L.D, F0, 0(R1)  // R1 array ptr
ADD.D, F4, F0, F2  // F2 = s
DADDUI, R1, R1, A-8  // decr. R1
BNE  // R1, R2, loop again if not done
```

With no scheduling

- 10 cycles per iteration
  - L.D, load stall, ADD.D, 2 RAW stalls, S.D, DADDUI, RAW stall, BNE, branch delay control stall

Let's try and fill the delayed branch stall

Chapter 4 - VLIW

Scheduling the Instructions to minimize Stalls

Non-trivial and many compilers don't try this

```
L.D, ADD.D, S.D, DADDUI, BNE
```

Common View: SD depends on DADDUI, and therefore can't be moved

Smarter View: DADDUI is immed.

Note:

- promote DADDUI move SD to branch delay sbb solution exists
- but since it will now be after the SUB it will need an offset of 8
- remaining stall is for ADD.D to SD + 2 instructions

Result will run as:

```
L.D, DADDUI, ADD.D, BNE
```

40% improvement
10 cycles vs 6
but still a 5 cycle
loop + stall overhead

Chapter 4 - VLIW
How to do better?

- Unroll the loop, replicate the loop body and adjust the iteration counter appropriately so that the program semantics remains the same.

- What are the advantages?

  - Get rid of the ADDUI and BNE overhead.

- What is the downside?

  - Larger code size - bad for instruction caches.
  - Needs lots of registers.

Basic Idea

- take n loop bodies and catenate them into 1 basic block.
- adjust the new termination code
  - Let's say n was 4
  - Then modify the R1 pointer in the example by 4x of what it was before.
  - savings - 4 BNE's + 4 DADDUI's => just one of each.
  - Hence 75% improvement.
  - Problem - still have 4 load stalls per loop.

  - Can we do better?

    - don't catenate the unrolled segments - shuffle them instead.
    - 4 LD's then 4 ADDD's then 3 SD's, SUBI, BNE, then fill the branch delay slot with the final SD.
    - VTOLA - no more stalls since LD to dependent ADDD path now has 3 instructions in it.

Result of Unrolling

<table>
<thead>
<tr>
<th>Loop</th>
<th>LD</th>
<th>F6, 0(R1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F6, 0(R1)</td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>F10, 16(R1)</td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>F14, 24(R1)</td>
<td></td>
</tr>
<tr>
<td>ADDD</td>
<td>F4, F5, F2</td>
<td></td>
</tr>
<tr>
<td>ADDD</td>
<td>F6, F5, F2</td>
<td></td>
</tr>
<tr>
<td>ADDD</td>
<td>F12, F10, F2</td>
<td></td>
</tr>
<tr>
<td>ADDD</td>
<td>F16, F14, F2</td>
<td></td>
</tr>
<tr>
<td>S.D.</td>
<td>F4, 0(R1)</td>
<td></td>
</tr>
<tr>
<td>S.D.</td>
<td>F6, 0(R1)</td>
<td></td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1, R1, 0, 12</td>
<td></td>
</tr>
<tr>
<td>S.D.</td>
<td>F12, 16(R1)</td>
<td></td>
</tr>
<tr>
<td>S.D.</td>
<td>F16, 8(R1)</td>
<td></td>
</tr>
</tbody>
</table>

Note: still didn't run out of registers - so we could have improved on this.

Result

- 14 cycles for 4 elements => 3.5 cycles per element.
- Compared with:
  - 16 cycles with no scheduling.
  - 8 cycles per element with scheduling but no unrolling.

Caveats of Loop Unrolling

- 8 more unused register pairs
  - hence
  - we could have gone to an 8 block unroll without register conflict.
  - no problem since the 1900 element array would still have broken cleanly (4000 = 128)
  - what if it had not - say with remainder R
  - so what just put 8 blocks (shuffled of course) in front of the loop then start for real.
  - you'll notice that you run out of registers but by cutting through the names you can still remove any stalls in this case.

- Still (most compilers unroll early to expose code for later opt's)
  - there were many things the compiler had to notice to figure this one out (crickiest was the S.D/DADDUI swap - why?)
  - key was the independent nature of each loop body.
  - REMEMBER: 3 types of dependence: data, name, and control.
How does unrolling help interact with dependencies

- **Data**
  - Unrolling provides more independent instructions
  - Scheduling then removes RAW dependencies

- **Name**
  - Renaming removes WAX hazards ⇒ antidependencies (WAR) and output dependencies (RAW)

- **Control**
  - Scheduling across branches is very tricky
  - Unrolling offers a simple solution
  - More body instructions per control instruction
  - Works when iteration count is static AND independent loop bodies

---

Static Branch Prediction

- Static branch behavior is useful for scheduling instructions when the branch delays are exposed by the architecture (e.g., delayed branches or canceling branches)
- They can also be used to predict which code paths are more plausible which is key for global code optimization

- Predict Taken or Not Taken - Poor accuracy from 9% to 59% misprediction rates
- Backward taken and forward not-taken, context based prediction
- Profile information from earlier runs

---

Misprediction Rate of Profile-based predictor on SpEC92

- Better for FP programs than integer program
- Actual performance varies depends on prediction accuracy and branch frequency which varies from 3% to 24%

---

Number of instructions between mispredicted branches (log scale)

- Average = 20 for predict-taken and 110 for profile-based
- Compare this to 4-5 instructions between a branch without prediction
- The variation is due to the nature of the program and the branch frequency
VLIW Processors

- Suitable for wide issue. N>4 when superscalar becomes unwieldy
- Assume VLIW with 2 mem references, 2 FP ops, 1 integer/br unit
- Show do all x[i] = x[i] + s
- Unroll as many times as needed to eliminate all stalls

Unrolled 7 times

<table>
<thead>
<tr>
<th>Mem Ref1</th>
<th>Mem Ref2</th>
<th>FP OP1</th>
<th>FP OP2</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F28, 0(r1)</td>
<td>LD F28, 0(r1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F20, -40(r1)</td>
<td>LD F20, -40(r1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F20, -32(r1)</td>
<td>ADD F20, F18, F2</td>
<td>ADD F18, F2</td>
<td>ADD F18, F2</td>
<td></td>
</tr>
<tr>
<td>LD F14, -24(r1)</td>
<td>ADD F16, F14, F2</td>
<td>ADD F16, F14, F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F10, -16(r1)</td>
<td>ADD F12, F10, F2</td>
<td>ADD F12, F10, F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F6, -8(r1)</td>
<td>ADD F8, F6, F2</td>
<td>ADD F8, F6, F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F0, 0(r1)</td>
<td>ADD F4, F0, F2</td>
<td>ADD F4, F0, F2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Performance

- 9 cycles for 7 iterations
- 1.29 cycles per result
- Increases code size - unrolling + packing density in cache is poor (18 empty slots)
- Compression and Encoding is possible
- Register pressure
- Lack of binary compatibility
  - Dynamic Binary Translation or Emulation
  - New approaches such as IA64 relaxes the strictness
- VLIW permits simpler/standard memory such as caches as opposed to a Vector Processor

Compiler Techniques to Expose ILP

- ECS 243 EEC175 classes dedicated to this topic
- What can the compiler do?
  a) Detecting and Enhancing Loop Level Parallelism
  b) Source Level Optimizations
  c) Eliminating Dependent Computation
    a) Copy propagation
    b) Symbolic Substitution
    c) Tree Height Reduction
    d) Algebraic Simplification
  d) Software Pipelining -
  e) Global Scheduling
Loop Carried Dependencies

- LCD = whether data accesses in a particular iteration are dependent on values produced in an earlier iteration
- Eg1 - for (I=1000; I>0, I--)
  \[ X[I] = X[I] + s \]
  No LCD
  \( X(I) \) depends on the value of \( X[I] \) in current iteration

Loop Carried Dependencies II

For (I=1, I<100, I++)

\[
\begin{align*}
A[I+1] &= A[I] + C[I] & \text{--- S1} \\
\end{align*}
\]

There are two types of dependencies here:

S1 \( \rightarrow \) S1 -- LCD \( A[I] \) current iteration needs \( A[I] \) from previous iteration!!!
So iterations I and I+1 cannot proceed in parallel

S1 \( \rightarrow \) S2 on \( A[I+1] \) is not LCD

Finding Dependencies

- In general HARD problem because of implicit names and arrays and pointers
- General Case is \( \mathsf{NP} \)-complete
- Restricted cases are solvable in polynomial time
- Eg: Arrays indices that are AFFINE i.e can be expressed as \( A*I + B \) where \( A \) and \( B \) are constants and \( I \) is the loop index variable
- The problem can be formulated as checking if 2 affine functions can have the same value i.e.
For some values of \( a,b,c,d \) : \( a[I] + b = c[I] + d \)
- Apply GCD tests if \( a,b,c,d \) are constants
- \( \text{GCD}(c,a) \) mod \( (d-b) = 0 \) --- sufficient condition, not necessary
Chapter 4 – VLIW

Software Pipelining

- Symbolic Loop Unrolling
- Loop unrolling is good for uncovering parallelism among instructions by creating longer sequences of st. line code - increasing the window size
- Eg: repeat (n) [A;B;C] is replaced by repeat (n/2) [A1; B1; C1; A2; B2; C2]

Reorder instructions to avoid hazards, hence expose ILP - code size increases, some branches are eliminated

Software pipelining is different - it involves reorganizing a loop with instructions from different iterations:
- A: (repeat (n-1) [BCA]) B: C

B, C, A are instructions from DIFFERENT iterations of the loop

This does in SW what Tomasulo’s algorithm did in HW - interleaving instructions from different iterations of a loop and executing them in parallel

Prolog and Epilog code is to make sure that the same code is executed in both the cases I.e. semantics of the program are not altered.

A Concrete Example

Loop:LD F0, 0(R1)
ADD F4, F4, F2
SD F4, 0(R1)
DADDUI R1, R1, -8
BNE R1, R2, LOOP

LD F0, 0(R1)
STALL
ADD F4, F0, F2
LD F0, -8(R1)
DADDUI R1, R1, -8
Loop:SD F4, 16(R1) -- stores M[i]
ADD F4, F0, F2 --- adds M[i-1]
LD F0, 0(R1) ---- loads M[i-2]
DADDUI R1, R1, -8
STALL
BNE R1, LOOP

Instructions not executed
In the final 2 iterations

Loop Unrolling vs SW Pipelining

- SW is symbolic so does not increase the code size.
- Does not increase register pressure, however register management is tricky - lifetimes of registers values can be long
- Loop unrolling and SW pipelining attack different problems so in general they can be combined
- LU - reduces loop overhead I.e. branch and counter update
- SWP - reduces the time when the loop is not running at its peak
**Chapter 4 - VLIW**

**Figuratively Speaking**

Diagram showing a VLIW instruction set architecture.

**Going Beyond Basic Blocks**

What are the consequences of moving instruction K before the branch? Instr J is dependent on K instead of P. If branch is not taken C[I] will get wrong value. Add compensating code.

---

What are the consequences of moving instruction K before the branch? Instr J is dependent on K instead of P. If branch is not taken C[I] will get wrong value. Add compensating code. Welcome to Speculation!!

Static Approaches to speculation:

- **Global Scheduling:**
  - Trace Scheduling
  - Superblocks

- **Predicated Execution:**
  - A Form of Speculation
  - Works when static branch prediction is poor
  - Wide issue m/c when multiple branches have to be resolved
  - ILP is limited due to control dependencies
  - Convert control dependency into data dependency - predicated/cond instr
  - This helps both hw based scheduling and SW pipelining and global scheduling
  - Every instruction refers to a cond.
  - If the condition is TRUE execute the instruction normally, if it is FALSE it becomes a NOP

---

**Global Scheduling - Fisher 1984**

- **Going Beyond BASIC BLOCKS**
- Exploit Parallelism ACROSS BB i.e. inter BB parallelism as opposed to INTRA BB parallelism with Loop Unroll and SWP
- Useful when there is no support for predication and loop unrolling doesn’t help because of conditional branches inside the loop body
- Pick a trace - a sequence of basic blocks (called trace) that is executed frequently
- **Trace Selection** can be done by profiling
- Loop unrolling and static branch prediction can be used
- **Trace Compaction** - Squeeze the trace into a small number of VLIW instructions by moving instructions as early as possible - packing with constraints
- Add Compensation Code for paths that will be taken if the prediction is wrong

---

**Predicated Execution - A Form of Speculation**

- Works when static branch prediction is poor
- Wide issue m/c when multiple branches have to be resolved
- ILP is limited due to control dependencies
- Convert control dependency into data dependency - predicated/cond instr
- This helps both hw based scheduling and SW pipelining and global scheduling
- Every instruction refers to a cond.
- If the condition is TRUE execute the instruction normally, if it is FALSE it becomes a NOP

---

**Chapter 4 - VLIW**

Diagram showing the flow of instructions and branch conditions.
Chapter 4 – VLIW

Example

- Example, if (A==0) S=T, Conditional move
Assume R1 = A; R2 = S and R3 = T

BNEZ R1, L
ADDD R2, R3, RD
CMOVZ R2, R3, R1
L:
- Another Example, A = abs (A) ... If B<0 A = -B else A = B
- Full Predication – Every instruction is controlled by a predicate, it allows to convert large blocks of code that are branch dependent
- Useful for global scheduling

WRINKLES
- Predicated Execution cannot generate an exception if the predicate is FALSE
- Annuled instructions consume valuable resources like FU, fetch bW
- Wastes power
- What happens if something depends on multiple branches - you need multiple conditions and that makes it complex

Chapter 4 – VLIW

HW Support to ASSIST Speculation

- HW and OS cooperatively ignore exceptions for speculative instructions - this approach preserves exception behavior for CORRECT programs but not for INCORRECT ones
- Speculative instructions that never raise exceptions are used and checks made to detect when exceptions can occur
- Poison bits in the result register - to invalidate results when a speculative instr causes an exception when a normal instruction tries to read the if a poison bit is set it will result in a fault
- Use something like a reorder buffer

Chapter 4 – VLIW

Another Scheme to preserve exception behavior

Special versions of Instructions are used in the speculative mode - these do not generate terminating exception
Eg: sLD R14, 0(R2)
- New instructions to check for such exceptions
Eg: SPECCK 0(r2) -- perform speculation check.
So, with these instruction the previous code becomes:

Ld R1, 0(r3): load A
sLD R14, 0(R2): speculative load B
Bnez R1, L1:; test A
DADD R14, R1, 4: the else clause
L1: DADD R1, R1, 4: -- else clause
L2: SD R14, 0(r3): non spec store
R14 is a temporary register to keep a copy of A in case branch is not taken
A will be in r14 after the code

Chapter 4 – VLIW

Exception Handling Rule:
- HW and OS simply handle all resumable exceptions, when the exceptions occur both for spec and non spec instructions - yes extra work for non spec instructions but works
- For terminating exceptions like protection faults, return an undefined value when the branch is not taken unless it comes from spec or no spec instr, and dealt with appropriately.

Chapter 4 – VLIW
Third Scheme to Preserve Exception Behavior

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1, 0(R3)</td>
<td>-- load A</td>
</tr>
<tr>
<td>LD R14, 0(R2)</td>
<td>-- speculative Load B</td>
</tr>
<tr>
<td>BEQZ R1, L3</td>
<td></td>
</tr>
<tr>
<td>DADDI R14, R1, 4</td>
<td></td>
</tr>
<tr>
<td>L3: SD R14, 0(R3)</td>
<td></td>
</tr>
</tbody>
</table>

R14 has a poison bit, which is turned on when
LD in BLUE I.e. spec LD causes an exception
When non speculative SD in GREEN tries to read R14, it
will cause an EXCEPTION

Note an extra bit in an instruction is needed to signify
is it a normal instr or a speculative instruction. This
is set by the compiler.