Lots of Parallelism...

- Last unit: pipeline-level parallelism
  - Work on execute of one instruction in parallel with decode of next
- Next: instruction-level parallelism (ILP)
  - Execute multiple independent instructions fully in parallel
  - Today: limited multiple issue
  - Next Week: dynamic scheduling
    - Extract much more ILP via out-of-order processing
- Data-level parallelism (DLP)
  - Single-instruction, multiple data
    - Example: one instruction, four 16-bit adds (using 64-bit registers)
- Thread-level parallelism (TLP)
  - Multiple software threads running on multiple processors

This Unit: Multiple Issue/Static Scheduling

- Multiple issue scaling problems
  - Dependence-checks
  - Bypassing
- Multiple issue designs
  - Statically-scheduled superscalar
    - VLIW/EPIC (IA64)
- Advanced static scheduling
- Advanced hardware technique
  - Grid processor

Scalar Pipeline and the Flynn Bottleneck

- Scalar pipelines
  - One instruction per stage
    - Performance limit (aka "Flynn Bottleneck") is CPI = IPC = 1
    - Limit is never even achieved (hazards)
    - Diminishing returns from "super-pipelining" (hazards + overhead)

Superscalar Execution

<table>
<thead>
<tr>
<th>Single-issue</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>id [r1]=0</td>
<td>x2</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id [r1]=1</td>
<td>x3</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id [r1]=2</td>
<td>x4</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r3,r5</td>
<td>x6</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r4,r6</td>
<td>x7</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r5,r7</td>
<td>x8</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id [r2]=0</td>
<td>x9</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual-issue</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td></td>
</tr>
<tr>
<td>id [r1]=0</td>
<td>x2</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id [r1]=1</td>
<td>x3</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id [r1]=2</td>
<td>x4</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id [r1]=3</td>
<td>x5</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r3,r5</td>
<td>x6</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r4,r6</td>
<td>x7</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r5,r7</td>
<td>x8</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id [r2]=0</td>
<td>x9</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Superscalar Challenges - Front End

- Wide instruction fetch
  - Modest: need multiple instructions per cycle
  - Aggressive: predict multiple branches, trace cache
- Wide instruction decode
  - Replicate decoders
- Wide instruction issue
  - Determine when instructions can proceed in parallel
  - Not all combinations possible
  - More complex stall logic - order $N^2$ for $N$-wide machine
- Wide register read
  - One port for each register read
  - Example, 4-wide superscalar $\Rightarrow \geq 8$ read ports

Superscalar Challenges - Back End

- Wide instruction execution
  - Replicate arithmetic units
  - Multiple cache ports
- Wide instruction register writeback
  - One write port per instruction that writes a register
  - Example, 4-wide superscalar $\Rightarrow \geq 4$ write ports
- Wide bypass paths
  - More possible sources for data values
  - Order $(N^2 \times P)$ for $N$-wide machine with execute pipeline depth $P$

Fundamental challenge:
- Amount of ILP (instruction-level parallelism) in the program
- Compiler must schedule code and extract parallelism

Simple Dual-issue Pipeline

- Fetch an entire 16B or 32B cache block
  - 4 to 8 instructions (assuming 4-byte fixed length instructions)
  - Predict a single branch per cycle
- Parallel decode
  - Need to check for conflicting instructions
  - Output of $I_1$ is an input to $I_2$
  - Other stalls, too (for example, load-use delay)

Another Approach: Split Int/FP

- Split integer and floating point
- 1 integer + 1 FP
  - Limited modifications
  - Limited speedup

Four-issue pipeline (2 integer, 2 FP)

- 2 integer + 2 FP
- Similar to Alpha 21164
- Floating point loads execute in "integer" pipe
Superscalar Challenges

- Want 4-, 6-, 8-issue machines
- Hardware challenges
  - Wide instruction fetch
  - Wide instruction decode
  - Wide instruction issue
  - Wide register read
  - Wide instruction execution
  - Wide instruction register writeback
  - Wide bypass paths
- Extracting and exploiting available ILP
  - Hardware and software
- Let’s talk about some of these issues...

Wide Fetch - Sequential Instructions

- What is involved in fetching multiple instructions per cycle?
  - In same cache block? → no problem
  - Favors larger block size (independent of hit rate)
- Compilers align basic blocks to IS lines (pad with nops)
  - Reduces effective IS capacity
  - Increases fetch bandwidth utilization (more important)
- In multiple blocks? → Fetch block A and A+1 in parallel
  - Banked IS + combining network
  - May add latency (add pipeline stages to avoid slowing down clock)

Wide Fetch - Non-sequential

- Two related questions
  - How many branches predicted per cycle?
  - Can we fetch from multiple taken branches per cycle?
- Simplest, most common organization: "1" and "No"
  - One prediction, discard post-branch insns if prediction is "Taken"
  - Average number of instructions per taken branch?
    - Assumes 20% branches, 50% taken → ~10 instructions
  - Without smarter fetch, ILP is limited to 5 (not 8)
- Compiler can help
  - Unroll loops, reduce taken branch frequency

Parallel Non-Sequential Fetch

- Allowing "embedded" taken branches is possible
  - Requires smart branch predictor, multiple IS accesses in one cycle
  - Can try pipelining branch prediction and fetch
    - Branch prediction stage only needs PC
      - Transmits two PCs to fetch stage, PC and target PC
        - Elongates pipeline, increases branch penalty
        - Pentium II & III do something like this

Trace Cache

- Trace cache (T$) [Peleg+Weiser, Rotenberg+]
  - Overcomes serialization of prediction and fetch by combining them
  - New kind of IS that stores dynamic, not static, insn sequences
    - Blocks can contain statically non-contiguous insns
  - Tag: PC of first insn + N/T of embedded branches
  - Used in Pentium 4 (actually stores decoded µops)
- Coupled with trace predictor (TP)
  - Predicts next trace, not next branch

Trace Cache Example

- Traditional instruction cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data (insns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>addi r1,4,r1</td>
</tr>
<tr>
<td>1</td>
<td>beq r1,#4</td>
</tr>
<tr>
<td>4</td>
<td>st r1,4(sp)</td>
</tr>
<tr>
<td>5</td>
<td>call #32</td>
</tr>
</tbody>
</table>

- Trace cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data (insns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>addi r1,4,r1</td>
</tr>
<tr>
<td>1</td>
<td>beq r1,#4</td>
</tr>
<tr>
<td>2</td>
<td>addi r1,4</td>
</tr>
<tr>
<td>3</td>
<td>st r1,4(sp)</td>
</tr>
<tr>
<td>4</td>
<td>call #32</td>
</tr>
</tbody>
</table>

- Tracks can pre-decode dependence information (µops)
  - Helps fit the N dependence check problem calls #32
Aside: Multiple-issue CISC

- How do we apply superscalar techniques to CISC
  - Such as x86
  - Or CISCy ugly instructions in some RISC ISAs
- Break "macro-ops" into "micro-ops"
  - Also called "wops" or "RISC-ops"
- A typical CISCy instruction "add [r2], [r1] \[r3\]" becomes:
  - Load [r2] \[r1\] to E1 (E1 is a temp. register, not visible to software)
  - Load [r1] to E2
  - Add E1, E2 \[r3\]
- However, conversion is expensive (latency, area, power)
- Solution: cache converted instructions in trace cache
  - However, conversion is expensive (latency, area, power)
- How do we apply superscalar techniques to CISC
  - Normally some mix of functional units proportional to insn mix
  - Intel Pentium: 1 any + 1 ALU

Wide Memory Access

- How do we allow multiple loads/stores to execute?
  - Option#1: Extra read ports on data cache
    - Higher latency, etc.
  - Option#2: "Bank" the cache
    - Can support a load to an "odd" and an "even" address
    - Problem: address not known to execute stage
  - Option#3: "Double" the cache
    - Multiple read bandwidth only
    - Larger area, but no conflicts, can be faster than more ports
    - Independent reads to replicas, writes (stores) go to all replicas
  - Example: the Alpha 21164 uses option #3
    - 8KB LI-caches, supports two loads, but only one store

Wide Decode

- What is involved in decoding multiple (N) insns per cycle?
  - Actually doing the decoding?
  - Easy if fixed length (multiple decoders), doable if variable length
  - Reading input registers?
    - 2N register read ports (latency \(\propto\) dports)
    - Actually less than 2N, most values come from bypasses
  - What about the stall logic?

N² Dependence Cross-Check

- Stall logic for 1-wide pipeline with full bypassing
  - Full bypassing = load/use stalls only
    - \(X/M,rp\) \& \((D/X,rs1==X/M,rd)\) \(\propto\) \(D/X,rs2==X/M,rd\)
  - Two "terms": \(\propto\) 2N
- Now: same logic for a 2-wide pipeline
  - \(X/M,rp\) \& \((D/X,rs1==X/M,rd)\) \(\propto\) \(D/X,rs2==X/M,rd\)
  - \(X/M,rp\) \& \((D/X,rs1==X/M,rd)\) \(\propto\) \(D/X,rs2==X/M,rd\)
  - Eight "terms": \(\propto\) 2N
  - This is the \(N^2\) dependence cross-check
  - Not quite done, also need
    - \(D/X,rs1==X/M,rd\) \(\propto\) \(D/X,rs2==X/M,rd\)

Superscalar Stalls

- Invariant: stalls propagate upstream to younger insns
  - If older insn in pair stalls, younger insns must stall too
  - What if younger insn stalls?
    - Can older insn from younger group move up?
      - Fluid: yes, but requires some muxing
      - Rigid: no
      - Fluid is a little, reduces clock a little
  - Higher latency, etc.

Wide Execute

- What is involved in executing multiple (N) insns per cycle?
  - Multiple execution units ... N of every kind?
    - N ALUs? OK, ALUs are small
    - N FP dividers? No, FP dividers are huge and \(X/M\) is uncommon
    - How many branches per cycle?
    - How many loads/stores per cycle?
    - Typically some mix of functional units proportional to insn mix
      - Intel Pentium: 1 any + 1 ALU
N\(^2\) Bypass Network

- N\(^2\) stall and bypass logic
  - Actually OK
  - 5-bit and 1-bit quantities

- N\(^2\) bypass network
  - 32-bit (or 64-bit) quantities
  - Routing lengths wires
  - Expensive metal layer crossings
  - N+1 input muxes at each ALU input
  - And this is just one bypassing stage!
  - Bit-slicing
    - Mitigates routing problem somewhat
    - 32 or 64 1-bit bypass networks

Clustering

- Clustering: mitigates N\(^2\) bypass
  - Group FUs into K clusters
  - Full bypassing within a cluster
  - Limited bypassing between clusters
    - With a one cycle delay
    - (N/K) + 1 inputs at each mux
    - (N/K)\(^2\) bypass paths in each cluster
  - Steering: key to performance
    - Steer dependent insns to same cluster
    - Statically (compiler) or dynamically
    - E.g., Alpha 21264
      - Bypass wouldn’t fit into clock cycle
      - 4-wide, 2 clusters, static steering
      - Replicates register file, too

Multiple-Issue Implementations

- Statically-scheduled (in-order) superscalar
  - Executes unmodified sequential programs
  - Hardware must figure out what can be done in parallel
    - E.g., Pentium (2-wide), UltraSPARC (4-wide), Alpha 21164 (4-wide)
- Very Long Instruction Word (VLIW)
  - Hardware can be dumb and low power
  - Compiler must group parallel insns, requires new binaries
    - E.g., Transmeta Crusoe (4-wide)
- Explicitly Parallel Instruction Computing (EPIC)
  - A compromise: compiler does some, hardware does the rest
    - E.g., Intel Itanium (6-wide)
  - Dynamically-scheduled superscalar
    - Pentium Pro/II/III (3-wide), Alpha 21264 (4-wide)
  - We’ll already talked about statically-scheduled superscalar

Wide Writeback

- What is involved in multiple (N) writebacks per cycle?
  - N register file write ports (latency \(\propto\) #ports)
    - Usually less than N, stores and branches don’t do writeback
    - But some ISAs have update or auto-incr/decr addressing modes
  - Multiple exceptions per cycle?
    - No just the oldest one

VLIW

- Hardware-centric multiple issue problems
  - Wide fetch+branch prediction, N\(^2\) bypass, N\(^2\) dependence checks
  - Hardware solutions have been proposed: clustering, trace cache
- Software-centric: very long insn word (VLIW)
  - Effectively, a 1-wide pipeline, but unit is an N-insn group
  - Compiler guarantees insns within a VLIW group are independent
    - If no independent insns, slots filled with nops
  - Group travels down pipeline as a unit
    - Simplifies pipeline control (no rigid vs. fluid business)
    - Cross-checks within a group un-necessary
    - Downstream cross-checks (maybe) still necessary
  - Typically “stuffed”: 1st insn must be ALU, 2nd mem, etc.
  - Further simplification

History of VLIW

- Started with ”horizontal microcode”
  - Culler-Harrison array processors (’72–’91)
  - Floating Point Systems FPS-1208
  - Academic projects
    - Yale ELF-512 (Fisher, ’85)
    - Illinois IMPACT (Hwu, ’91)
  - Commercial attempts
    - Multiflow (Colwell+Fisher, ’85) → failed
    - Cydrome (Rau, ’95) → failed
    - Motorola/TI embedded processors → successful
    - Intel Itanium (Colwell+Fisher+Rau, ’97) → ??
    - Transmeta Crusoe (Ditzel, ’99) → failed
Pure and Tainted VLIW

- **Pure VLIW**: no hardware dependence checks at all
  - Not even between VLIW groups
  - Very simple and low power hardware
  - Compiler responsible for scheduling stall cycles
  - Requires precise knowledge of pipeline depth and structure
  - Must be fixed for compatibility
  - Doesn’t support caches well
  - Used in some cache-less micro-controllers and signal processors
    - Not useful for general-purpose computation

- **Tainted (more realistic) VLIW**: inter-group checks
  - Compiler doesn’t schedule stall cycles
  - Precise pipeline depth and latencies not needed, can be changed
  - Supports caches
  - TransMeta Crusoe

What Does VLIW Actually Buy Us?

- Simpler I$\backslash$branch prediction
  - No trace cache necessary
- Simpler dependence check logic
- Bypasses are the same
  - Clustering can help VLIW, too
  - Compiler can schedule for limited bypass networks
  - Not compatible across machines of different widths
  - Is non-compatibility worth all of this?

- PS how does TransMeta deal with compatibility problem?
  - Dynamically translates x86 to internal VLIW

EPIC

- Tainted VLIW
  - Compatible across pipeline depths
  - But not across pipeline widths and slot structures
  - Must re-compile if going from 4-wide to 8-wide
  - TransMeta sidesteps this problem by re-compiling transparently

- **EPIC (Explicitly Parallel Insn Computing)**
  - New VLIW (Variable Length Insn Words)
  - Implemented as “bundles” with explicit dependence bits
  - Code is compatible with different “bundle” width machines
  - Compiler discovers as much parallelism as it can, hardware does rest
  - E.g., Intel Itanium (IA-64)
    - 128-bit bundles (3 41-bit insns + 4 dependence bits)

ILP and Static Scheduling

- No point to having an N-wide pipeline...
  - ...if average number of parallel insns per cycle (ILP) << N
- How can the compiler help extract parallelism?
  - These techniques applicable to regular superscalar
  - These techniques critical for VLIW/EPIC

Code Example: SAXPY

- **SAXPY** (Single-precision A X Plus Y)
  - Linear algebra routine (used in solving systems of equations)
  - Part of early “Livermore Loops” benchmark suite
  - for (i=0;i<N;i++)
  - \[ Z[i]=A\times X[i] + Y[i]; \]

\[
\begin{align*}
0: & \text{ldf } X[r1],f1 & \text{// loop} \\
1: & \text{mulf } f0,f1,f2 & \text{// A in f0} \\
2: & \text{ldf } Y[r1],f3 & \text{// X,Y,Z are constant addresses} \\
3: & \text{addf } f2,f3,f4 & \\
4: & \text{stf } f4,Z[r1] & \\
5: & \text{addi } r1,4,r1 & \text{// i in r1} \\
6: & \text{bli } r1,r2,0 & \text{// *4 in r2}
\end{align*}
\]

SAXPY Performance and Utilization

- **Scalar pipeline**
  - Full bypassing, 5-cycle E*, 2-cycle E+, branches predicted taken
  - Single iteration (7 insns) latency: 16–5 = 11 cycles
  - **Performance**: 7 insns / 11 cycles = 0.64 IPC
  - **Utilization**: 0.64 actual IPC / 1 peak IPC = 66%
**SAXPY Performance and Utilization**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ldf X(r1),f1</code></td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>mulf c0,r1,f2</code></td>
<td>F</td>
<td>d+X</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>ldf Y(r1),f3</code></td>
<td>D</td>
<td>p+X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>addf r2,f3,f4</code></td>
<td>F</td>
<td>p+X</td>
<td>p+X</td>
<td>p+X</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td><code>stf f4,Z(r1)</code></td>
<td>F</td>
<td>p+X</td>
<td>p+X</td>
<td>p+X</td>
<td>p+X</td>
<td>p+X</td>
<td>p+X</td>
</tr>
</tbody>
</table>

- Dual issue pipeline (fluid)
  - Same e any two insns per cycle + embedded branch
  - Performance: 7 insns / 10 cycles = 0.70 IPC
  - Utilization: 0.70 actual IPC / 2 peak IPC = 35%
  - More hazards -> more stalls (why?)
  - Each stall is more expensive (why?)

---

**Schedule and Issue**

- **Issue**: time at which insns begin execution
  - Want to maintain issue rate of N
- **Schedule**: order in which insns execute
  - In in-order pipeline, schedule + stalls determine issue
  - A good schedule that minimizes stalls is important
    - For both performance and utilization
  - Schedule/issue combinations
    - Pure VLIW: static schedule, static issue
    - Tainted VLIW: static schedule, partly dynamic issue
    - Superscalar, EPIC: static schedule, dynamic issue

---

**Instruction Scheduling**

- **Idea**: place independent insns between slow ops and uses
  - Otherwise, pipeline stalls while waiting for RAW hazards to resolve
  - Have already seen pipeline scheduling
- **To schedule well need**: independent insns
- **Scheduling scope**: code region we are scheduling
  - The bigger the better (more independent insns to choose from)
  - Once scope is defined, schedule is pretty obvious
  - Trick is creating a large scope (must schedule across branches)
- **Compiler scheduling (really scope enlarging) techniques**
  - Loop unrolling (for loops)
  - Trace scheduling (for non-loop control flow)

---

**Aside: Profiling**

- **Profile**: statistical information about program tendencies
  - Software’s answer to everything
    - Collected from previous program runs (different inputs)
    - Works OK depending on information
      - Identities of frequently missing loads stable across inputs
      - But are tied to cache configuration
      - Memory dependencies
      - Stable across inputs
      - But exploiting this information is hard (need hw help)
      - Branch outcomes
        - Not so stable across inputs
        - Exploiting this information is hard (need hw help)
      - Software’s answer to everything
        - Could be useful
        - But are tied to cache configuration
      - Memory dependencies
        - Stable across inputs
        - But exploiting this information is hard (need hw help)
      - Branch outcomes
        - Not so stable across inputs
        - Exploiting this information is hard (need hw help)
      - Software’s answer to everything
        - Could be useful
      - But are tied to cache configuration
      - Memory dependencies
      - Stable across inputs
      - But exploiting this information is hard (need hw help)
      - Branch outcomes
        - Not so stable across inputs
        - Exploiting this information is hard (need hw help)
      - Software’s answer to everything
        - Could be useful
      - But are tied to cache configuration
      - Memory dependencies
      - Stable across inputs
      - But exploiting this information is hard (need hw help)
      - Branch outcomes
        - Not so stable across inputs
        - Exploiting this information is hard (need hw help)
      - Software’s answer to everything
        - Could be useful
      - But are tied to cache configuration
      - Memory dependencies
      - Stable across inputs
      - But exploiting this information is hard (need hw help)
      - Branch outcomes
        - Not so stable across inputs
        - Exploiting this information is hard (need hw help)
      - Software’s answer to everything
        - Could be useful
      - But are tied to cache configuration
      - Memory dependencies
      - Stable across inputs
      - But exploiting this information is hard (need hw help)
      - Branch outcomes
        - Not so stable across inputs
        - Exploiting this information is hard (need hw help)
      - Software’s answer to everything
        - Could be useful
      - But are tied to cache configuration
      - Memory dependencies
      - Stable across inputs
      - But exploiting this information is hard (need hw help)
      - Branch outcomes
        - Not so stable across inputs
        - Exploiting this information is hard (need hw help)
      - Software’s answer to everything
        - Could be useful
      - But are tied to cache configuration
      - Memory dependencies
      - Stable across inputs
      - But exploiting this information is hard (need hw help)
      - Branch outcomes
        - Not so stable across inputs
        - Exploiting this information is hard (need hw help)
      - Software’s answer to everything
        - Could be useful
      - But are tied to cache configuration
      - Memory dependencies
      - Stable across inputs
      - But exploiting this information is hard (need hw help)
      - Branch outcomes
        - Not so stable across inputs
        - Exploiting this information is hard (need hw help)
      - Software’s answer to everything
        - Could be useful
      - But are tied to cache configuration
      - Memory dependencies
      - Stable across inputs
      - But exploiting this information is hard (need hw help)
      - Branch outcomes
        - Not so stable across inputs
        - Exploiting this information is hard (need hw help)
      - Software’s answer to everything
        - Could be useful
      - But are tied to cache configuration
      - Memory dependencies
      - Stable across inputs
      - But exploiting this information is hard (need hw help)
      - Branch outcomes
        - Not so stable across inputs
        - Exploiting this information is hard (need hw help)
      - Software’s answer to everything
        - Could be useful
      - But are tied to cache configuration
      - Memory dependencies
      - Stable across inputs
      - But exploiting this information is hard (need hw help)
      - Branch outcomes
        - Not so stable across inputs
        - Exploiting this information is hard (need hw help)
      - Software’s answer to everything
        - Could be useful
      - But are tied to cache configuration
      - Memory dependencies
      - Stable across inputs
      - But exploiting this information is hard (need hw help)
      - Branch outcomes
        - Not so stable across inputs
        - Exploiting this information is hard (need hw help)
      - Software’s answer to everything
        - Could be useful
      - But are tied to cache configuration
      - Memory dependencies
      - Stable across inputs
      - But exploiting this information is hard (need hw help)
      - Branch outcomes
        - Not so stable across inputs
        - Exploiting this information is hard (need hw help)
      - Software’s answer to everything
        - Could be useful
      - But are tied to cache configuration
      - Memory dependencies
      - Stable across inputs
      - But exploiting this information is hard (need hw help)
      - Branch outcomes
        - Not so stable across inputs
        - Exploiting this information is hard (need hw help)
Unrolling SAXPY II: Pipeline Schedule
- Pipeline schedule to reduce RAW stalls
  - Have already seen this: pipeline scheduling

Unrolled SAXPY Performance/Utilization
- Performance: 12 instr / 13 cycles = 0.92 IPC
- Utilization: 0.92 actual IPC / 1 peak IPC = 92%
Superblocks

- First trace scheduling construct: superblock
  - Use when branch is highly biased
  - Fuse blocks from most frequent path: A,C,D
  - Schedule
  - Create repair code in case real path was A,B,D

Superblock and Repair Code

- What did we do?
  - Change sense (test) of branch 1
    - Original taken target now fall-thru
  - Created repair block
    - May need to duplicate some code (here basic-block D)
  - Haven't actually scheduled superblock yet

Superblocks Scheduling I

- First scheduling move: move insns 5 and 6 above insn 4
  - Hmm... moved load (5) above store (4)
  - We can tell this is OK, but can the compiler
    - If yes, fine
    - Otherwise, need to do something

ISA Support for Load/Store Speculation

- IA-64: change insn 5 to advanced load 
  - "Advanced" means advanced past some unknown store
  - Processor stores [address, reg] of advanced loads in table
    - Memory Conflict Buffer (MCB), Advanced Load Alias Table (ALAT)
  - Later stores search ALAT: matching address → invalidate ALAT entry
  - Insert check insn chk.a to make sure ALAT entry still valid
  - If not, jump to some more repair code (aghnhhh...)

Superblock Scheduling II

- Second scheduling move: move insns 5 and 6 above insn 1
  - That's OK, load did not depend on branch...
  - And would have executed anyway
- Scheduling non-move: don't move insn 4 above insn 1
  - Why? Hard (but possible) to undo a store in repair code
    - (What about in a multiprocessor or multithreaded workload?)

- Success: scheduled 3 insns between 6 and 7

What If...

- branch 1 had the opposite bias?
The Other Superblock and Repair Code

Superblock

<table>
<thead>
<tr>
<th>0: ldf Y(r1),f2</th>
<th>Repair code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: fmov f2,f4</td>
<td>4: atf f0.Y(r1)</td>
</tr>
<tr>
<td>2: ldf W(r1),f2</td>
<td>5: ldf X(r1),f4</td>
</tr>
<tr>
<td>5: ldf X(r1),f4</td>
<td>6: mulf f4,f2,f6</td>
</tr>
<tr>
<td>6: mulf f4,f2,f6</td>
<td>7: atf f6.Z(r1)</td>
</tr>
</tbody>
</table>

• Notice
  > Branch 1 sense (test) unchanged
    > Original taken target now in repair code

ISA Support for Load-Branch Speculation

Superblock

<table>
<thead>
<tr>
<th>0: ldf Y(r1),f2</th>
<th>Repair code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: fmov f2,f4</td>
<td>4: atf f0.Y(r1)</td>
</tr>
<tr>
<td>2: ldf W(r1),f2</td>
<td>5: ldf X(r1),f4</td>
</tr>
<tr>
<td>5: ldf X(r1),f4</td>
<td>6: mulf f4,f2,f6</td>
</tr>
<tr>
<td>6: mulf f4,f2,f6</td>
<td>7: atf f6.Z(r1)</td>
</tr>
</tbody>
</table>

• IA-64: change insn 2 to speculative load ldf.s
  > "Speculative" means advanced past some unknown branch
  > Processor keeps exception bit with register f8
  > Inserted insn cch s checks exception bit
  > If exception, jump to yet more repair code (arghhh...)

• IA-64 also contains ldf.sa

Predication

• Conventional control
  > Conditionally executed insns also conditionally fetched

• Predication
  > Conditionally executed insns unconditionally fetched
    > Full predication (ARM, IA-64)
      > Can tag every insn with predicate, but extra bits in instruction
    > Conditional moves (Alpha, IA-32)
    > Construct appearance of full predication from one primitive
      > mmove s1.s0,r1
        > if (r1=0) s0=r1;
      > May require some code duplication to achieve desired effect
    > Only good way of adding predication to an existing ISA
    > II-conversion: replacing control with predication
      > Good if branch is unpredictable (save mis-prediction)
      > But more instructions fetched and "executed"
Hyperblock Scheduling

- Second trace scheduling construct: hyperblock
  - Use when branch is not highly biased
  - Fuse all four blocks: A,B,C,D
  - Use predication to conditionally execute insns in B and C
  - Schedule

```
0: ldf Y(r1),f2
1: fbne f2,4
4: stf f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
2: ldf W(r1),f2
3: jump 5
```

Static Scheduling Summary

- Goal: increase scope to find more independent insns
- Loop unrolling
  - Simple
  - Expands code size, can't handle recurrences or non-loops
- Software pipelining
  - Handles recurrences
  - Complex prologue/epilogue code
  - Requires register copies (unless rotating register file...)
- Trace scheduling
  - Superblocks and hyperblocks
  - Works for non-loops
  - More complex, requires ISA support for speculation and predication
  - Requires nasty repair code

Multiple Issue Summary

- Problem spots
  - Wide fetch + branch prediction → trace cache?
  - N<sup>2</sup> dependence cross-check
  - N<sup>2</sup> bypass → clustering?

- Implementations
  - Statically scheduled superscalar
  - VLIW/EPIC
  - Research: Grid Processor

- What’s next:
  - Finding more ILP by relaxing the in-order execution requirement

Loop Unrolling Shortcomings

- Static code growth more IS misses (relatively minor)
- Poor scheduling along “seams” of unrolled copies
- Need more registers to resolve WAR hazards
- Doesn’t handle recurrences (inter-iteration dependences)

```
for (i=0;i<N;i++)
X[i]=A*X[i-1];
```

```
ldf X-4(r1),f1
mulf f0,f1,f2
stf f2,X(r1)
addi r1,4,r1
blt r1,r2,0
```

What About Leftover Iterations?

- What to do if N % K != 0
  - What to do with extra iterations?
  - Main unrolled loop executes N / K times
  - Add non-unrolled loop that executes N % K times
Software Pipelining

- **Software pipelining**: deals with these shortcomings
  - Also called "symbolic loop unrolling" or "poly-cyclic scheduling"
  - Reinvented a few times [Charlesworth, '81], [Rau, '85] [Lam, '88]
  - One physical iteration contains insns from multiple logical iterations

- The pipeline analogy
  - In a hardware pipeline, a single cycle contains:
    - Stage 3 of insn i, stage 2 of insn i+1, stage 1 of insn i+2
  - In a software pipeline, a single physical (SP) iteration contains:
    -Insn 3 from iter i, insn 2 from iter i+1, insn 1 from iter i+2

Software Pipelined Recurrence Example

- Goal: separate **mulf** from **stf**
- Physical iteration (box) contains
  - **stf** from original iteration i
  - ld, mulf from original iteration i+1
- Prologue: get pipeline started (ld, mulf from iteration 0)
- Epilogue: finish up leftovers (**stf** from iteration N–1)

```
ldf X-4(r1),f1
mulf f0,f1,f2
stf f2,X(r1)
addi r1,4,r1
blt r1,r2,0
```

Software Pipelined Example II

- Vary software pipelining structure to tolerate more latency
  - Example: physical iteration combines three logical iterations

```
ldf X(r1),f1
mulf f0,f1,f2
stf f2,X(r1)
addi r1,4,r1
blt r1,r2,0
```

Software Pipelining

- Doesn't increase code size
- Good scheduling at iteration "seams"
- Can vary degree of pipelining to tolerate longer latencies
  - "Software super-pipelining"
    - One physical iteration: insns from logical iterations i, i+2, i+4
  - Hard to do conditionals within loops
    - Easier with loop unrolling
Scheduling: Compiler or Hardware

- Each has some advantages
- Compiler
  + Potentially large scheduling scope (full program)
  + Simple hardware — fast clock, short pipeline, and low power
  + Low branch prediction accuracy (profiling?)
  + Little information on memory dependences and latencies (profiling?)
  + Pain to speculate and recover from mis-speculation (h/w support?)
- Hardware
  + High branch prediction accuracy
  + Dynamic information about memory dependences and latencies
  + Easy to speculate and recover from mis-speculation
  - Finite buffering resources fundamentally limit scheduling scope
  - Scheduling machinery adds pipeline stages and consumes power

Research: Frames

- New experimental scheduling construct: frame
  - rePLay (Patel+Lumetta)
  - Frame: an atomic superblock
  + Atomic means all or nothing, i.e., transactional
  - Two new insns
    + begin_frame: start buffering insn results
    + commit_frame: make frame results permanent
  - High branch prediction accuracy
  - Dynamic information about memory dependences and latencies
  - Easy to speculate and recover from mis-speculation
  - Eliminates nastiest part of trace scheduling … nasty repair code
  - If frame path is wrong just jump to original basic block code
  - Repair code still exists, but it’s just the original code

Frames

<table>
<thead>
<tr>
<th>Frame</th>
<th>Repair Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin_frame</td>
<td>0: ldf Y(r1),f2</td>
</tr>
<tr>
<td></td>
<td>1: fsub f2,f4</td>
</tr>
<tr>
<td></td>
<td>2: ldf W(r1),f2</td>
</tr>
<tr>
<td></td>
<td>3: jump 5</td>
</tr>
<tr>
<td></td>
<td>4: mulf f4,f2,f6</td>
</tr>
<tr>
<td></td>
<td>5: ldf X(r1),f4</td>
</tr>
<tr>
<td></td>
<td>6: stf f6,Z(r1)</td>
</tr>
<tr>
<td>commit_frame</td>
<td>7: stf f6,Z(r1)</td>
</tr>
</tbody>
</table>

- What about frame optimizations?
  + Load-branch optimizations can be done without support
  + Load-store optimizations still require ISA support
    - Fixup code still simpler

Research: Grid Processor

- Grid processor architecture (aka TRIPS)
  - [Nagarajan, Sankaralingam, Burger+Keckler]
  - EDGE (Explicit Dataflow Graph Execution) execution model
  - Holistic attack on many fundamental superscalar problems
    + Specifically, the nastiest one: N² bypassing
    + But also N² dependence check
    + And wide-fetch + branch prediction
  - Two-dimensional VLIW
    - Horizontal dimension is insns in one parallel group
    - Vertical dimension is several vertical groups
    - Executes atomic hyperblocks
  - IBM looking into building it

Grid Processor

- Components
  + next h-block logic/predictor (NH), I$, D$, regfile
  + N x N ALU grid: here 4 x 4
- Pipeline stages
  + Fetch h-block to grid
  + Read registers
  + Execute/memory
  + Write registers
  + Block atomic
    + No intermediate regs
    + Grid limits size/shape

Grid Processor SAXPY

- An h-block for this Grid processor has 5 4-insn words
  - The unit is all 5
- Some notes about Grid ISA
  - read: read register from register file
  - pass: null operation
  - [1,0,1]: routing directives send result to next word
    + one insn left (-1), insn straight down (0), one insn right (1)
    + Directives specify value flow, no need for interior registers
Grid Processor SAXPY Cycle 1
- Map hyperblock to grid

Grid Processor SAXPY Cycle 2
- Read registers

Grid Processor SAXPY Cycle 3
- Execute first grid row
- Execution proceeds in "data flow" fashion
  - Not lock step

Grid Processor SAXPY
- When all instructions are done
  - Write registers and next hyperblock PC

Grid Processor SAXPY Performance
- Performance
  - 1 cycle fetch
  - 1 cycle read regs
  - 8 cycles execute
  - 1 cycle write regs
  - 11 cycles total (same)
- Utilization
  - 7 / (11 * 16) = 4%
- What's the point?
  - Simpler components
  - Faster clock?

Grid Processor Pros and Cons
+ Naturally aligned IS
+ No hardware dependence checks period
  - Insns explicitly encode rather than hardware reconstruct
  - Still get dynamic issue
+ Simple, forward only, short-wire bypassing
  - No wraparound routing, no metal layer crossings, low input muxes
- Code size
  - Lots of nop and pass operations
- Poor scheduling between hyperblocks
- Non-compatibility
  - Code assumes horizontal and vertical grid layout
  - Overcome with transparent dynamic translation? Like Transmeta?
- Utilization
  - Overcome by multiple concurrent executing hyperblocks