Memory reference speculation with hardware support

- To move loads across stores, when compiler can’t be sure it is legal
- Use a **speculative load** instruction
  - hardware saves address of memory location
  - if a subsequent store changes that location before the check (to end the speculation), then the speculation failed, otherwise it succeeded
  - on failure, need to redo load and re-execute all speculated instructions after the speculative load

Superscalar execution

- Predication helps with scheduling
- Example: superscalar that can issue 1 memory reference and 1 ALU op per cycle, or just 1 branch

<table>
<thead>
<tr>
<th>1st instruction</th>
<th>2nd instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1,40(R2)</td>
<td>ADD R3,R4,R5</td>
</tr>
<tr>
<td></td>
<td>ADD R6,R3,R7</td>
</tr>
<tr>
<td>BEQZ R10,L</td>
<td></td>
</tr>
<tr>
<td>LW R8,0(R10)</td>
<td></td>
</tr>
<tr>
<td>LW R9,0(R8)</td>
<td></td>
</tr>
<tr>
<td>LWC loads if 3rd operand not 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1st instruction</th>
<th>2nd instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1,40(R2)</td>
<td>ADD R3,R4,R5</td>
</tr>
<tr>
<td>LWC R8,0(R10),R10</td>
<td>ADD R6,R3,R7</td>
</tr>
<tr>
<td>BEQZ R10,L</td>
<td></td>
</tr>
<tr>
<td>LW R9,0(R8)</td>
<td></td>
</tr>
</tbody>
</table>
**Limitations to ILP**

- Conflicting studies of amount
  - Benchmarks (vectorized Fortran FP vs. integer C programs)
  - Hardware sophistication
  - Compiler sophistication

- How much ILP is available using existing mechanisms with increasing HW budgets?

- Do we need to invent new HW/SW mechanisms to keep on processor performance curve?
  - Intel MMX, SSE (Streaming SIMD Extensions): 64 bit ints
  - Intel SSE2: 128 bit, including 2 64-bit Fl. Pt. per clock
  - Motorola AltiVec: 128 bit ints and FPs
  - Supersparc Multimedia ops, etc.

**Overcoming Limits**

- Advances in compiler technology + significantly new and different hardware techniques may be able to overcome limitations assumed in studies

- However, unlikely such advances when coupled with realistic hardware will overcome these limits in near future

**Limits to ILP**

- Initial HW Model here; MIPS compilers.

Assumptions for ideal/perfect machine to start:

1. **Register renaming** – infinite virtual registers
   => all register WAW & WAR hazards are avoided

2. **Branch prediction** – perfect; no mispredictions

3. **Jump prediction** – all jumps perfectly predicted (returns, case statements)
   2 & 3 => no control dependencies; perfect speculation & an unbounded buffer of instructions available

4. **Memory-address alias analysis** – addresses known & a load can be moved before a store provided addresses not equal; 1&4 eliminates all but RAW

Also: perfect caches; 1 cycle latency for all instructions (FP *,/); unlimited instructions issued/clock cycle;
Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th></th>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions Issued per clock</td>
<td>Infinite</td>
<td>4</td>
</tr>
<tr>
<td>Instruction Window Size</td>
<td>Infinite</td>
<td>200</td>
</tr>
<tr>
<td>Renaming Registers</td>
<td>Infinite</td>
<td>48 integer + 40 Fl. Pt.</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>Perfect</td>
<td>2% to 6% misprediction (Tournament Branch Predictor)</td>
</tr>
<tr>
<td>Cache</td>
<td>Perfect</td>
<td>64KI, 32KD, 1.92MB</td>
</tr>
<tr>
<td>Memory Alias Analysis</td>
<td>Perfect</td>
<td>??</td>
</tr>
</tbody>
</table>

Upper Limit to ILP: Ideal Machine

Programs

Integer: 18 - 60
FP: 75 - 150

More Realistic HW: Window Impact

Change from Infinite window 2048, 512, 128, 32
FP: 9 - 150

Integer: 8 - 63
Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th></th>
<th>New Model</th>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions Issued per clock</td>
<td>64</td>
<td>Infinite</td>
<td>4</td>
</tr>
<tr>
<td>Instruction Window Size</td>
<td>2048</td>
<td>Infinite</td>
<td>200</td>
</tr>
<tr>
<td>Renaming Registers</td>
<td>Infinite</td>
<td>Infinite</td>
<td>48 integer + 40 Fl. Pt.</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>Perfect vs. 8K Tournament vs. 512 2-bit vs. profile vs. none</td>
<td>Perfect</td>
<td>2% to 6% misprediction (Tournament Branch Predictor)</td>
</tr>
<tr>
<td>Cache</td>
<td>Perfect</td>
<td>Perfect</td>
<td>64KI, 32KD, 1.92MB L2, 36 MB L3</td>
</tr>
<tr>
<td>Memory Alias</td>
<td>Perfect</td>
<td>Perfect</td>
<td>??</td>
</tr>
</tbody>
</table>

More Realistic HW: Branch Impact

![Graph showing IPC vs. Program]

Misprediction Rates

![Graph showing Misprediction Rate vs. Program]

Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th></th>
<th>New Model</th>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions Issued per clock</td>
<td>64</td>
<td>Infinite</td>
<td>4</td>
</tr>
<tr>
<td>Instruction Window Size</td>
<td>2048</td>
<td>Infinite</td>
<td>200</td>
</tr>
<tr>
<td>Renaming Registers</td>
<td>Infinite v. 256, 128, 64, 32, none</td>
<td>Infinite</td>
<td>48 integer + 40 Fl. Pt.</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>8K 2-bit</td>
<td>Perfect</td>
<td>Tournament Branch Predictor</td>
</tr>
<tr>
<td>Cache</td>
<td>Perfect</td>
<td>Perfect</td>
<td>64KI, 32KD, 1.92MB L2, 36 MB L3</td>
</tr>
<tr>
<td>Memory Alias</td>
<td>Perfect</td>
<td>Perfect</td>
<td>Perfect</td>
</tr>
</tbody>
</table>
More Realistic HW:
Renaming Register Impact (N int + N fp)

Figure 3.5 – SPEC92

Change 2048 instr window, 64 instr issue, 8K 2 level Prediction

FP: 11 - 45

Integer: 5 - 15

Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th></th>
<th>New Model</th>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Issued per</td>
<td>64</td>
<td>Infinite</td>
<td>4</td>
</tr>
<tr>
<td>clock</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>2048</td>
<td>Infinite</td>
<td>200</td>
</tr>
<tr>
<td>Window Size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Renaming</td>
<td>256 Int + 256 FP</td>
<td>Infinite</td>
<td>48 integer + 40 Fl. Pt.</td>
</tr>
<tr>
<td>Registers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>8K 2-bit</td>
<td>Perfect</td>
<td>Tournament</td>
</tr>
<tr>
<td>Prediction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache</td>
<td>Perfect</td>
<td>Perfect</td>
<td>64KI, 32KD, 1.92MB L2, 36 MB L3</td>
</tr>
<tr>
<td>MemoryAlias</td>
<td>Perfect v. Stack v. Inspect v. none</td>
<td>Perfect</td>
<td>Perfect</td>
</tr>
</tbody>
</table>

More Realistic HW:
Memory Address Alias Impact

Figure 3.6 – SPEC92

Change 2048 instr window, 64 instr issue, 8K 2 level Prediction, 256 renaming registers

FP: 4 - 45 (Fortran, no heap)

Integer: 4 - 9

How to Exceed ILP Limits of this study?

• These are not laws of physics; just practical limits for today, and perhaps overcome via research

• Compiler and ISA advances could change results

• WAR and WAW hazards through memory: eliminated WAW and WAR hazards through register renaming, but not in memory usage
  – Can get conflicts via allocation of stack frames as a called procedure reuses the memory addresses of a previous frame on the stack
HW v. SW to increase ILP

- Memory disambiguation: HW best
- Speculation:
  - HW best when dynamic branch prediction better than compile time prediction
  - Exceptions easier for HW
  - HW doesn’t need bookkeeping code or compensation code
  - Very complicated to get right
- Scheduling: SW can look ahead to schedule better
- Compiler independence: does not require new compiler, recompilation to run well

Performance beyond single thread ILP

- There can be much higher natural parallelism in some applications (e.g., Database or Scientific codes)
- Explicit Thread Level Parallelism or Data Level Parallelism
- Thread: process with own instructions and data
  - thread may be a process part of a parallel program of multiple processes, or it may be an independent program
  - Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute
- Data Level Parallelism: Perform identical operations on data, and lots of data

Thread Level Parallelism (TLP)

- ILP exploits implicit parallel operations within a loop or straight-line code segment
- TLP explicitly represented by the use of multiple threads of execution that are inherently parallel
- Goal: Use multiple instruction streams to improve
  1. Throughput of computers that run many programs
  2. Execution time of multi-threaded programs
- TLP could be more cost-effective to exploit than ILP

New Approach: Multithreaded Execution

- Multithreading: multiple threads to share the functional units of 1 processor via overlapping
  - processor must duplicate independent state of each thread e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
  - memory shared through the virtual memory mechanisms, which already support multiple processes
  - HW for fast thread switch; much faster than full process switch \( \approx 100s \) to \( 1000s \) of clocks
- When to switch?
  - Alternate instruction per thread (fine grain)
  - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)
### Fine-Grained Multithreading

- Switches between threads on each instruction, causing the execution of multiple threads to be interleaved.
- Usually done in a round-robin fashion, skipping any stalled threads.
- CPU must be able to switch threads every clock.
- Advantage is it can hide both short and long stalls, since instructions from other threads executed when one thread stalls.
- Disadvantage is it slows down execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads.
- Used on Sun’s Niagara (will see later).

### Coarse-Grained Multithreading

- Switches threads only on costly stalls, such as L2 cache misses.
- **Advantages**
  - Relieves need to have very fast thread-switching.
  - Doesn’t slow down thread, since instructions from other threads issued only when the thread encounters a costly stall.
- **Disadvantage** is hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs.
  - Since CPU issues instructions from 1 thread, when a stall occurs, the pipeline must be emptied or frozen.
  - New thread must fill pipeline before instructions can complete.
- Because of this start-up overhead, coarse-grained multithreading is better for reducing penalty of high cost stalls, where pipeline refill << stall time.
- Used in IBM AS/400.

---

**For most apps, most execution units lie idle**

For an 8-way superscalar.

```
memory conflict  load delays  control hazards  branch misprediction  data miss  load miss  stlb miss  processor busy
```


---

**Do both ILP and TLP?**

- TLP and ILP exploit two different kinds of parallel structure in a program.
- Could a processor oriented at ILP exploit TLP?
  - Functional units are often idle in data path designed for ILP because of either stalls or dependences in the code.
- Could the TLP be used as a source of independent instructions that might keep the processor busy during stalls?
- Could TLP be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?
Simultaneous Multithreading (SMT)

- Simultaneous multithreading (SMT): insight that dynamically scheduled processor already has many HW mechanisms to support multithreading
  - Large set of virtual registers that can be used to hold the register sets of independent threads
  - Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in datapath without confusing sources and destinations across threads
  - Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW
- Just add a per thread renaming table and keep separate PCs
  - Independent commitment can be supported by logically keeping a separate reorder buffer for each thread

Multithreaded Categories

- Superscalar
- Fine-Grained
- Coarse-Grained
- Multiprocessing
- Simultaneous Multithreading

Design Challenges in SMT

- Since SMT makes sense only with fine-grained implementation, impact of fine-grained scheduling on single thread performance?
  - A preferred thread approach sacrifices neither throughput nor single-thread performance?
  - Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput, when preferred thread stalls
- Larger register file needed to hold multiple contexts
- Not affecting clock cycle time, especially in
  - Instruction issue - more candidate instructions need to be considered
  - Instruction completion - choosing which instructions to commit may be challenging
- Ensuring that cache and TLB conflicts generated by SMT do not degrade performance