COMPUTER ARCHITECTURE

Instruction Level Parallelism Exploitation
Compiler techniques and ILP.
Advanced branch prediction techniques.
Introduction to dynamic scheduling.
Speculation.
Multiple issue techniques.
Limits of ILP.
Thread Level Parallelism.
3

Compiler Techniques and ILP
ILP directly applicable to basic blocks.

- **Basic block**: Sequence of instructions without branching.
- Typical MIPS program → Average basic block size 3 to 6.
  - Low ILP exploitation within block.
- Need to exploit ILP across basic blocks.

```
for (i=0;i<1000;i++) {
  x[i] = x[i] + y[i];
}
```

- Loop level parallelism.
  - Can be transformed to ILP.
    - By compiler or hardware.
  - Alternative:
    - Vector instructions.
    - SIMD instructions in processor.
Parallelism exploitation.
- Interleave execution of non related instructions.
- Fill with instruction stalls.
- Do not alter effects of original program.

Compiler can use detailed knowledge of architecture.
for (i=999; i>=0; i--) {
    x[i] = x[i] + s;
}

Each iteration body is independent

- Instructions latencies.

<table>
<thead>
<tr>
<th>Instruction producing the result</th>
<th>Instruction producing the result</th>
<th>Latency (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU operation</td>
<td>Another FP ALU operation</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU operation</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU operation</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>
- **R1**: Last element of array
- **F2**: Scalar s.
- **R2**: Precomputed to make 8(R2) the first element in array.

<table>
<thead>
<tr>
<th>Loop</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F0, 0(R1)</td>
<td>F0 ← x[i]</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4, F0, F2</td>
<td>F4 ← F0 + s</td>
</tr>
<tr>
<td>S.D</td>
<td>F4, 0(R1)</td>
<td>x[i] ← F4</td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1, R1, #8</td>
<td>i--</td>
</tr>
<tr>
<td>BNE</td>
<td>R1, R2, Loop</td>
<td>Branch if R1!=R2</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------</td>
</tr>
<tr>
<td>L.D F0, 0(R1)</td>
<td>; F0 ← x[i]</td>
<td>stall</td>
</tr>
<tr>
<td>ADD.D F4, F0, F2</td>
<td>; F4 ← F0 + s</td>
<td>stall</td>
</tr>
<tr>
<td>S.D F4, 0(R1)</td>
<td>; x[i] ← F4</td>
<td>stall</td>
</tr>
<tr>
<td>DADDUI R1, R1, #-8</td>
<td>; i--</td>
<td>stall</td>
</tr>
<tr>
<td>BNE R1, R2, Loop</td>
<td>; Branch if R1!=R2</td>
<td></td>
</tr>
</tbody>
</table>
Loop scheduling

Loop:
- L.D  F0, 0(R1)
- stall
- ADD.D  F4, F0, F2
- stall
- stall
- S.D  F4, 0(R1)
- DADDUI  R1, R1, #8
- stall
- BNE  R1, R2, Loop

Loop:
- L.D  F0, 0(R1)
- DADDUI  R1, R1, #8
- ADD.D  F4, F0, F2
- stall
- stall
- S.D  F4, 8(R1)
- BNE  R1, R2, Loop

9 cycles per iteration

7 ciclos por iteración
Loop unrolling

**Idea:**
- Replicate loop body several times.
- Adjust loop termination code.
- Use distinct registers for each replica to reduce dependencies.

**Effect:**
- Increase basic block length.
- Increase available ILP.
Loop unrolling

Assume size is a 4-multiple

Bucle:  
L.D  F0, 0(R1)  stall  
ADD.D  F4, F0, F2  stall  
ADD.D  F8, F6, F2  stall  
ADD.D  F12, F10, F2  stall  
S.D  F4, 0(R1)  stall  
L.D  F6, -8(R1)  stall  
S.D  F8, -8(R1)  stall  
L.D  F10, -16(R1)  stall  
S.D  F16, -24(R1)  stall  
ADD.D  F16, F14, F2  stall  
DADDUI  R1, F1, #-32  stall  
BNE  R1, R2, Bucle

27 cycles for every 4 iterations.  
6.75 cycles per iteration.
Scheduling and unrolling

Loop:

L.D F0, 0(R1)
L.D F6, -8(R1)
L.D F10, -16(R1)
L.D F14, -24(R1)
ADD.D F4, F0, F2
ADD.D F8, F6, F2
ADD.D F12, F10, F2
ADD.D F16, F14, F2
S.D F4, 0(R1)
S.D F8, -8(R1)
S.D F12, -16(R1)
DADDUI R1, F1, #32
S.D F16, -24(R1)
BNE R1, R2, Loop

14 cycles for every 4 iterations.
3.5 cycles per iteration.
- Improvement decreased with each unrolling.
  - Improvement limited to stalls elimination.
  - Overhead amortized among iterations.

- Code size increase:
  - Could affect instruction cache miss rate.

- Registers pressure with aggressive unrolling and scheduling.
  - May lead to shortage of registers.
  - Advantages lost if not enough available registers.
Advanced branch prediction techniques
High impact of branches on applications performance.

Impact mitigation:
- Loop unrolling.
- Branch prediction:
  - Compile-time.
  - Each branch handled isolated.
- Advanced branch prediction:
  - Branch predictors correlation.
  - Tournament predictors.
- Hardware reorders instructions execution to reduce stalls maintaining data flow and exceptions.

- Able to handle unknown cases at compile time:
  - Cache misses/hits.

- Code less dependent on a concrete pipeline.
  - Simplifies compiler.

- Allows for **hardware speculation**.
if (a==2) { a=0; }
if (b==2) { b=0; }
if (a!=b) {
  ...
}

When first and second branches are taken, Third branch is NOT taken.

- Maintains last branches history to select among several predictors.

- A (m,n) predictor:
  - Uses result of $m$ last branches to select among $2^m$ predictors.
  - Each predictor uses $n$ bits.

- (1,2) Predictor:
  - Uses result of last branch to select among 2 predictors each using 2 bits.
A \((m,n)\) predictor has several entries per branch address.

**Total size:**
- \(S = 2^m \times n \times \text{entries per branch}\)

**Examples:**
- \((0,2)\) with 4K entries → 8Kb
- \((2,2)\) with 4K entries → 32 Kb
- \((2,2)\) with 1K entries → 8Kb
Correlated predictor has less misses than simple predictor when same size.

Correlated predictor has less misses than simple predictor with unlimited number of entries.
Combines two predictors:
- Global information based predictor.
- Local information based predictor.

Use selector to choose between predictors.
- Change between predictor uses a saturated counter (2 bits).

**Advantage:**
- Allows different behavior for integer and FP.

**SPEC:**
- Integer benchmark → global predictor 40%.
- FP benchmark → global predictor 15%.

**Usos:** Alpha y AMD Opeteron.
Two-level predictor:
- Smaller first level predictor.
- Larger second level predictor as backup.

Each predictor combines 3 predictors:
- Simple 2-bits predictor.
- Global history predictor.
- Loop-exit predictor (iterations counter).

Besides:
- Predictor for indirect jumps.
- Return address predictor.
Introduction to dynamic scheduling
Dynamic scheduling

- **Idea:** Hardware reorder instructions execution to reduce stalls.

- **Advantages:**
  - Compiled code optimized for a pipeline runs efficiently in a different pipeline.
  - Correctly manages dependencies unknown at compile time.
  - Allows to tolerate delays that cannot be predicted (e.g. cache misses).

- **Drawback:**
  - More complex hardware.
Effects:
- Out-of-order (OOO) execution.
- Out-of-order instruction finalization.
- May introduce WAR and WAW hazards.

Separation of ID stage into different stages:
- **Issue**: Decodes instruction and checks for structural hazards.
- **Operands fetch**: Waits until no data hazard and fetches operands.

Instruction Fetch (IF):
- Fetches from instruction register or instruction queue.
Scoreboard:
- Stalls issued instructions until enough resources available and no data hazard.
- CDC 6600, ARM A8.

Tomasulo Algorithm:
- Removes WAR and WAW dependencies with register renaming.
- IBM 360, Intel Core i7.
Speculation
- As parallelism increases, control dependencies become a harder problem.
  - Branch prediction is not enough.

- Next step is **speculation** on branch outcome and run assuming speculation was right.
  - Instructions fetched, issued, executed.
  - Mechanism needed to handle wrong speculation.
**Ideas:**

- **Dynamic branch prediction:** Selects instructions to be executed.
- **Speculation:** Executes before control dependencies are resolved and may eventually undo.
- **Dynamic scheduling.**

To achieve this, separation among:

- Passing an instruction result to another instruction using it.
- Instruction finalization.

- Processor state (register file / memory) not updated until changes confirmed.

**Computer Architecture - 2014 - J. Daniel García**
Reorder Buffer (ROB):
- When an instruction is finalized ROB is written.
- When execution is confirmed real target is written.
- Instructions read modified data from ROB.

ROB entries:
- Instruction type: branch, store, register operation.
- Target: Register id or memory address.
- Value: Instruction result value.
- Ready: Indication of instruction completion.
Tomasulo with speculative execution

- Instruction queue
- Load-store operations
- Address unit
- Load buffers
- Store address
- Store data
- Memory unit
- FP adders
- FP multipliers
- Operation bus
- Common data bus (CDB)
- Reservation stations
- Floating-point operations
- FP registers
- Reg #
- Data
- Reorder buffer
- From instruction unit

© 2017 Elsevier Inc. All rights reserved.

Computer Architecture - 2014 - J. Daniel García
Multiple issue techniques
CPI $\geq 1$ $\rightarrow$ Issue one instruction per cycle.

Multiple issue processors (CPI $< 1$ $\rightarrow$ IPC $> 1$):

- Statically scheduled superscalar processors.
  - In-order execution.
  - Variable number of instructions per cycle.

- Dynamically scheduled superscalar processors.
  - Out-of-order execution.
  - Variable number of instructions per cycle.

- VLIW Processors (Very Long Instruction Word).
  - Several instruction into a packet.
  - Static scheduling.
  - Explicit Instruction Level Parallelism by the compiler.
<table>
<thead>
<tr>
<th>Name</th>
<th>Issue</th>
<th>Hazard detection</th>
<th>Scheduling</th>
<th>Discriminating feature</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static superscalar</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Static</td>
<td>In order execution</td>
<td>MIPS y ARM</td>
</tr>
<tr>
<td>Dynamic superscalar</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic</td>
<td>Out of order without speculation</td>
<td>Ninguno</td>
</tr>
<tr>
<td>Speculative superscalar</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Speculative dynamic</td>
<td>Out of order with speculation</td>
<td>Intel Core i3, i5, i7, AMD Phenom. IBM Power 7</td>
</tr>
<tr>
<td>VLIW/LIW</td>
<td>Static</td>
<td>Mostly software</td>
<td>Static</td>
<td>All hazards determined by compiler</td>
<td>Signal processing, e.g. TI C6x.</td>
</tr>
<tr>
<td>EPIC</td>
<td>Mostly static</td>
<td>Mostly software</td>
<td>Mostly static</td>
<td>All hazards determined by compiler</td>
<td>Itanium</td>
</tr>
</tbody>
</table>
- Packs several operations into a single instruction.

- Example instruction in VLIW ISA:
  - One integer or branch instruction.
  - Two independent floating point operations.
  - Two independent memory references.

- Code must exhibit enough parallelism.
Original VLIW model drawbacks:

- Complexity of statically finding enough parallelism.
- Generated code size.
- No hazard detection hardware.
- More binary compatibility problems than in regular superscalar designs.

EPIC tries to solve most of this problems.
Limits of ILP
To study maximum ILP we model an ideal processor.

Procesador ideal:

- **Infinite register renaming:** All WAR and WAW hazards avoided.
- **Perfect branch prediction:** All branch predictions are a hit.
- **Perfect jump prediction:** All jumps (including returns) predictions are a hit.
- **Perfect memory address alias analysis:** A load can be moved before a store if address is not identical.
- **Perfect caches:** All cache accesses require one cycle (always hit)
Available ILP

<table>
<thead>
<tr>
<th>Tool</th>
<th>ILP</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>54.8</td>
</tr>
<tr>
<td>expresso</td>
<td>62.6</td>
</tr>
<tr>
<td>li</td>
<td>17.9</td>
</tr>
<tr>
<td>fppp</td>
<td>75.2</td>
</tr>
<tr>
<td>doduc</td>
<td>118.7</td>
</tr>
<tr>
<td>tomcatv</td>
<td>150.1</td>
</tr>
</tbody>
</table>
More ILP implies more control logic:
- Smaller caches.
- Longer cycle.
- Higher energy consumption.

Practical limitation:
- Issue 3 to 6 instructions per cycle.
Thread level parallelism
Some applications with more natural parallelism than the achieved with ILP:
- Servers, Scientific applications, ...

Two models emerge:
- **Thread level Parallelism (TLP):**
  - **Thread**: Process with its own instructions and data.
  - Can be part of a program or an independent program.
  - Each thread has an associated state (instructions, data, PC, registers, ...).
- **Data Level Parallelism (DLP):**
  - Identical operations on data.
ILP exploits implicit parallelism within a basic block or a loop.

TLP uses multiple threads of execution inherently parallel.

TLP Goals:
- Use multiple instruction flows to improve:
  - Throughput in computers using many programs.
  - Execution time in multi-thread programs.
Multiple threads sharing processor functional units and overlapping their use.

- Need to replicate processor state n-times.
  - Register file, PC, page table (when threads do not belong to the same program).
  - Shared memory through virtual memory mechanisms.
  - Hardware for fast thread context switch.

Tipos:

- Fine grain: Thread switch every instruction.
- Coarse grain: Thread switch in stalls (e.g. cache miss).
- Simultaneous: Fine grain with multiple issue dynamically scheduled.
- Switches between threads in each cycle.
  - Interleaves thread execution.

- Usually round-robin.
  - Excludes stalled threads.

- Processor must be able to switch every clock cycle.

**Advantage:**
- Can hide short and long stalls.

**Drawback:**
- Delays individual thread execution due to switching.

- Examples: Sun Niagara, Nvidia GPUs.
Coarse Grained Multithreading

- Switch only on costly stalls.
  - Example: L2 or L3 cache miss.

- **Advantages:**
  - No need for almost free thread switching.
  - No slowdown for threads.

- **Drawbacks:**
  - Needs to flush the pipeline.
  - Needs to fill the pipeline with instructions from the new thread (latency).

- Appropriate when filling the pipeline has much lower cost than stall.
- Example: IBM AS/400.
**Idea:** Dynamically scheduled processors already have many mechanisms to support multithreading.

- Large sets of virtual registers.
  - Registers for multiple threads.
- Register renaming.
  - Avoids conflicts in access to registers from threads.
- Out of order finalization.

**Modifications:**

- Per-thread renaming table.
- Separate PC registers.
- Separate ROB.

**Examples:** Intel Core i7, IBM Power 7
<table>
<thead>
<tr>
<th>Superscalar</th>
<th>Fine MT</th>
<th>Coarse MT</th>
<th>Multiprocessor</th>
<th>SMT</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
<td><img src="image4.png" alt="Diagram" /></td>
<td><img src="image5.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

- **Thread 1**: Blue
- **Thread 2**: Red
- **Thread 3**: Yellow
- **Thread 4**: Pink
- **Thread 5**: Purple
- **Inactive**: Blank
Loop unrolling allows for hiding latencies in stalls, but offer a limited improvement.

Dynamic scheduling is able to handle stalls that are unknown at compile time.

Speculative execution techniques are supported by branch prediction and dynamic scheduling.

Multiple issue is limited in practice to a range from 3 to 6.

SMT approach to TLP within one core.
Hennessy y Patterson.
Sections: 3.1, 3.2, 3.3, 3.4, 3.6, 3.7, 3.10, 3.12

Ejercicios: 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.11, 3.14, 3.17