Exploitation of instruction level parallelism

Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Francisco Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid

Taking advantage of ILP

- ILP directly applicable to basic blocks.
  - **Basic block**: sequence of instructions without branching.
  - **Typical** program in MIPS:
    - Basic block average size from 3 to 6 instructions.
    - Low ILP exploitation within block.
  - Need to exploit ILP across basic blocks.

Example

```
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.

Scheduling and loop unrolling

- **Parallelism exploitation**:
  - Interleave execution of unrelated instructions.
  - Fill stalls with instructions.
  - Do not alter original program effects.
- Compiler can do this with detailed knowledge of the architecture.

ILP exploitation

Example

```
for (i=999;i>=0;i--) {
  x[i] = x[i] + s;
}
```

- Each iteration body is independent.

Latencies between instructions

Instruction producing result	Instruction using result	Latency (cycles)
FP ALU operation	FP ALU operation	3
FP ALU operation	Store double	2
Load double	FP ALU operation	1
Load double	Store double	0

Compiled code

- **R1** → Last array element.
- **F2** → Scalar **s**.
- **R2** → Precomputed so that **8(R2)** is the first element in array.

Assembler code

```
Loop:   L.D  F0, 0(R1)          ; F0 <- x[i]
        ADD.D  F4, F0, F2       ; F4 <- F0 + s
        S.D  F4, 0(R1)          ; x[i] <- F4
        DADDUI  R1, R1, #-8     ; i--
        BNE  R1, R2, Loop       ; Branch if R1!=R2
```

Stalls in execution

Original

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

Stalls

```
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 scheduling

Original

```
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
```

- 9 cycles per iteration.

Scheduled

```
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
```

- 7 cycles per iteration.

Loop unrolling

- **Idea**:
  - Replicate loop body several times.
  - Adjust termination code.
  - Use different registers for each iteration replica to reduce dependencies.
- **Effect**:
  - Increase basic block length.
  - Increase use of available ILP.

Unrolling

Unrolling (x4)

```
Loop:    L.D F0, 0(R1)
         ADD.D F4, F0, F2
         S.D F4, 0(R1)
         L.D F6, −8(R1)
         ADD.D F8, F6, F2
         S.D F8, −8(R1)
         L.D F10, −16(R1)
```

Unrolling (x4)

```
         ADD.D F12, F10, F2
         S.D F12, −16(R1)
         L.D F14, −24(R1)
         ADD.D F16, F14, F2
         S.D F16, −24(R1)
         DADDUI R1, R1, #−32
         BNE R1, R2, Loop
```

- 4 iterations require more registers.
- This example assumes that array size is multiple of 4.

Stalls and unrolling

Unrolling (x4)

```
Loop:   L.D F0, 0(R1)
        <stall>
        ADD.D F4, F0, F2
        <stall>
        <stall>
        S.D F4, 0(R1)
        L.D F6, −8(R1)
        <stall>
        ADD.D F8, F6, F2
        <stall>
        <stall>
        S.D F8, −8(R1)
        L.D F10, −16(R1)
        <stall>
```

Unrolling (x4)

```
        ADD.D F12, F10, F2
        <stall>
        <stall>
        S.D F12, −16(R1)
        L.D F14, −24(R1)
        <stall>
        ADD.D F16, F14, F2
        <stall>
        <stall>
        S.D F16, −24(R1)
        DADDUI R1, R1, #−32
        <stall>
        BNE R1, R2, Loop
```

- 27 cycles for every 4 iterations $\rightarrow$ 6.75 cycles per iteration.

Scheduling and unrolling

Unrolling (x4)

```
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,  R1,  #−32
        S.D  F16,  8(R1)
        BNE  R1,  R2,  Loop
```

- Code reorganization.
  - Preserve dependencies.
  - Semantically equivalent.
  - **Goal**: Make use of *stalls*.
- Update of **R1** at enough distance from **BNE**.
- 14 cycles for every 4 iterations → 3.5 cycles per iteration.

Limits of loop unrolling

- **Improvement** is **decreased** with each additional unrolling.
  - Improvement limited to stalls removal.
  - Overhead amortized among iterations.
- **Increase** in code **size**.
  - May affect to instruction cache miss rate.
- **Pressure** on **register file**.
  - May generate shortage of registers.
  - Advantages are lost if there are not enough available registers.

Branch prediction

- High impact of **branches** on programs performance.

- To **reduce** impact:
  - Loop unrolling.
  - Branch prediction:
    - Compile time.
    - Each branch handled isolated.
  - Advanced branch prediction:
    - Correlated predictors.
    - Tournament predictors.

Dynamic scheduling

- Hardware **reorders** instructions execution to *reduce* stalls while keeping data flow and exceptions.
- Able to handle unknown cases at compile time:
  - Cache misses/hits.
- Code less dependent on a concrete pipeline.
  - Simplifies compiler.
- Permits the **hardware speculation**.

Correlated prediction

- If first and second branch are taken, third is NOT-taken.

example

```
if (a==2) { a=0; }
if (b==2) { b=0; }
if (a!=b) { f(); }
```

- Maintains last branches **history** to select among several predictors.
- A (m, n) predictor:
  - Uses the result of m last branches to select among 2^m predictors.
  - Each predictor has n bits.
- Predictor $(1, 2)$:
  - Result of last branch to select among 2 predictors.

Size of predictor

- A predictor (m, n) has several entries for each branch address.
- Total size:

$$S = 2^m \times n \times entries_{\text{address}}$$

- Examples:
  - $(0, 2)$ with 4K entries $\rightarrow$ 8 Kb
  - $(2, 2)$ with 4K entries $\rightarrow$ 32 Kb
  - $(2, 2)$ with 1K entries $\rightarrow$ 8 Kb

Miss rate

- Correlated predictor has less misses that simple predictor with same size.

- Correlated predictor has less misses than simple predictor with unlimited number of entries.

Tournament prediction

- **Combines two predictors**:
  - **Global information** based predictor.
  - **Local information** based predictor.
- Uses a selector to choose between predictors.
  - Change among two selectors uses a saturation counter (2 bits).
- **Advantage**:
  - Allows different behavior for integer and FP.
- **SPEC**:
  - **Integer benchmarks** $\rightarrow$ global predictor 40%
  - **FP benchmarks** $\rightarrow$ global predictor 15%.
- **Uses**: Alpha and AMD Opteron.

Intel Core i7

- Predictor with **two levels**:
  - Smaller first level predictor.
  - Larger second level predictor as backup.
- Each predictor combines **3 predictors**:
  - Simple 2-bits predictor.
  - Global history predictor.
  - Exit-loop predictor (iterations counter).
- Besides:
  - **Indirect jumps** predictor.
  - **Return address** predictor.

Dynamic scheduling

- **Idea**: hardware reorders instruction execution to decrease stalls.
- **Advantages**:
  - Compiled code optimized for one pipeline runs efficiently in another pipeline.
  - Correctly manages dependencies that are unknown at compile time.
  - Allows to tolerate unpredictable delays (e.g. cache misses).
- **Drawback**:
  - More complex hardware.

Dynamic scheduling

- **Effects**:
  - Out of Order execution (OoO).
  - Out of Order instruction finalization.
  - May introduce WAR and WAW **hazards**.
- Separation of **ID** stage into **two** different stages:
  - **Issue**: Decodes instruction and checks for structural hazards.
  - **Operands fetch**: Waits until there is no data hazard and fetches operands.
- **Instruction Fetch** (**IF**):
  - Fetches into instruction register or instruction queue.

Dynamic scheduling techniques

- **Scoreboard**:
  - Stalls issued instructions until enough resources are available and there is no data hazard.
  - Examples: CDC 6600, ARM A8.

- **Tomasulo Algorithm**:
  - Removes WAR and WAW dependencies with register renaming.
  - Examples: IBM 360, Intel Core i7.

Branches and parallelism limits

- As parallelism increases, **control dependencies** become a harder problem.
  - Branch prediction is not enough.

- Next step is **speculation** on **branch outcome** and **execution** assuming that **speculation was right**.
  - Instructions fetched, issued and executed.
  - Need of a mechanism to handle wrong speculations.

Components

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

- To achieve this, must **separate**:
  - **passing instruction result** to another instruction using it.
  - Instruction **finalization**.

- **IMPORTANT**: Processor state (register file / memory) **not updated** until changes are confirmed.

Solution

- **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.

CPI < 1

- 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 instructions into a packet.
    - Static scheduling.
    - Explicit *ILP* by the compiler.

Approaches to multiple issue

- Several approaches to multiple issue.
  - **Static superscalar**.
  - **Dynamic superscalar**.
  - **Speculative superscalar**.
  - **VLIW/LIW**.
  - **EPIC**.

Static superscalar

- **Issue**: Dynamic.
- **Hazards detection**: Hardware.
- **Scheduling**: Static.
- **Discriminating feature**:
  - In-order execution.

- **Examples**:
  - MIPS.
  - ARM Cortex-A7.

Dynamic Superscalar

- **Issue**: Dynamic.
- **Hazards detection**: Hardware.
- **Scheduling**: Dynamic.
- **Discriminating feature**:
  - Out-of-Order execution with no speculation.
- **Examples**: None.

Speculative superscalar

- **Issue**: Dynamic.
- **Hazards detection**: Hardware.
- **Scheduling**: Speculative dynamic.
- **Discriminating feature**:
  - Out-of-Order execution with speculation.

- **Examples**:
  - Intel Core i3, i5, i7.
  - AMD Phenom.
  - IBM Power 7

VLIW

- Packs several operations into a single instruction.
- Example instruction in ISA VLIW:
  - One integer instruction or a branch.
  - Two independent floating point operations.
  - Two independent memory references.
- **IMPORTANT**: Code must exhibit enough parallelism.

VLIW / LIW

- **Issue**: Static.
- **Hazards detection**: Mostly software.
- **Scheduling**: Static.
- **Discriminating feature**:
  - All hazards determined and specified by the compiler.

- **Examples**:
  - DSPs (e.g. TI C6x).

Problems with VLIW

- **Drawbacks** from original **VLIW** model:
  - Complexity of finding statically 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.

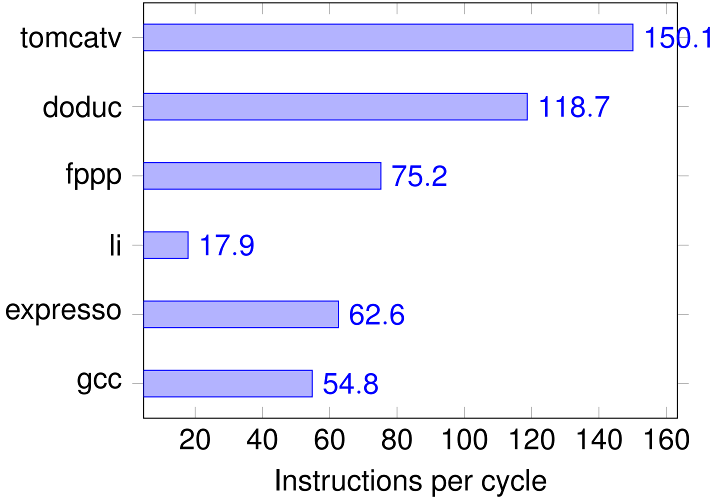
EPIC

- **Issue**: Mostly static.
- **Hazards detection**: Mostly software.
- **Scheduling**: Mostly static.
- **Discriminating feature**:
  - All hazards determined and specified by compiler.
- **Examples**:
  - Itanium.

ILP limits

- To study maximum **ILP** we model an **ideal processor**.
- **Ideal processor**:
  - **Infinite register renaming**: All WAR and WAW hazards can be avoided.
  - **Perfect branch prediction**: All conditional branch predictions are a hit.
  - **Perfect jump prediction**: All jumps (include returns) are correctly predicted.
  - **Perfect memory address alias analysis**: A load can be safely moved before a store if address is not identical
  - **Perfect caches**: All cache accesses require one clock cycle (always hit).

Available ILP



However . . .

- More ILP implies more control logic:
  - Smaller caches.
  - Longer clock cycle.
  - Higher energy consumption.

- **Practical limitation**:
  - Issue from 3 to 6 instructions per cycle.

Why TLP?

- Some applications exhibit 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.
    - May be either part of a program or an independent program.
    - Each thread has an associated **state** (instructions, data, PC, registers, . . .).
  - **Data level parallelism (DLP)**:
    - Identical operation on different data items.

TLP

- **ILP** exploits implicit parallelism within a basic block or a loop.

- **TLP** uses multiple threads of execution inherently parallel.

- **TLP Goal**:
  - Use multiple instruction flows to improve:
    - **Throughput** in computers using many programs.
    - **Execution time** of multi-threaded programs.

Multi-threaded execution

- Multiple threads share processor functional units overlapping its use.
  - Need to replicate state n-times.
    - Register file, PC, page table (when threads do note belong to the same program).
    - Shared memory through virtual memory mechanisms.
    - Hardware for fast thread context switch.
  - **Kinds**:
    - **Fine grain**: Thread switch in every instruction.
    - **Coarse grain**: Thread switch in stalls (e.g. Cache miss).
    - **Simultaneous**: Fine grain with multiple-issue and dynamic scheduling.

Fine-grain multithreading

- Switches between threads in each instruction.
  - Interleaves thread execution.
  - Usually does *round-robin*.
  - Threads in a *stall* are excluded from *round-robin*.
  - Processor must be able to switch every clock cycle.
- **Advantage**:
  - Can hide short and long stalls.
- **Drawback**:
  - Delays individual thread execution due to sharing.
- **Example**: Sun Niagara.

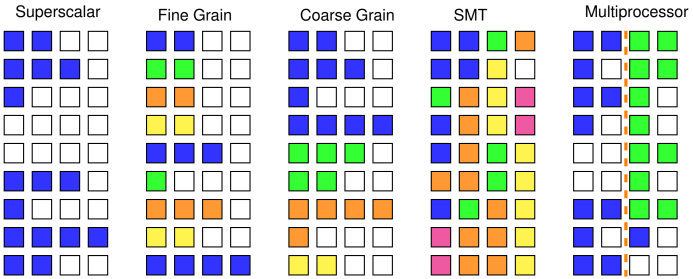
Coarse grain multithreading

- Switch thread only on long stalls.
    - **Example**: L2 cache miss.
- **Advantages**:
    - No need for a highly fast thread switch.
    - Does not delay individual threads.
- **Drawbacks**:
    - Must flush or freeze the pipeline.
    - Needs to fill pipeline with instructions from the new thread (latency).
- Appropriate when filling the pipeline takes much shorter than a stall.
    - **Example**: IBM AS/400.

SMT: Simultaneous multithreading

- **Idea**: Dynamically scheduled processors already have many mechanisms to support multithreading.
  - Large sets of virtual registers.
    - Registers for multiple threads.
  - Register renaming.
    - Avoid 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

TLP: Summary



🟦	Thread 1	🟩	Thread 2	🟧	Thread 3	🟨	Thread 4	🟪	Thread 5	⬜	stall

Summary

- Loop unrolling allows hiding stall latencies, but offers a limited improvement.
- Dynamic scheduling manages stalls unknown at compile-time.
- Speculative techniques built on branch prediction and dynamic scheduling.
- Multiple issue in ILP is limited in practice from 3 to 6 instructions.
- SMT is an approach to TLP within one core.

References

- **Computer Architecture. A Quantitative Approach**
  5th Ed.
  Hennessy and Patterson.
  Sections 3.1, 3.2, 3.3, 3.4, 3.6, 3.7, 3.10, 3.12.

- **Recommended exercises**:
  - 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.11, 3.14, 3.17.

Exploitation of instruction level parallelism

Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Francisco Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid