Instruction Level Parallelism

M. Sonza Reorda

Politecnico di Torino
Dipartimento di Automatica e Informatica

INSTRUCTION-LEVEL PARALLELISM

Pipelines exploit the parallelism existing among instructions (*Instruction-Level Parallelism*, or ILP), which allows their execution in parallel.

The highest the amount of ILP that can be found and exploited, the better the performance of the pipeline.

Approaches

There are two approaches to exploit ILP:

- ***Dynamic*, depending on the hardware to locate parallelism**
- ***Static*, depending on the software (i.e., the compiler).**

The two approaches can partly be combined.

Dynamic approach

It dominates the desktop and server markets, with products such as

- **Pentium III and IV**
- **Athlon**
- **MIPS R10000/12000**
- **Sun UltraSPARC III**
- **PowerPC 603, G3, G4**
- **Alpha 21264.**

Static approach

It can mainly be found in products for the embedded market.

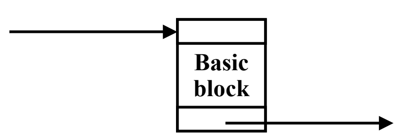
However, both the Intel IA-64 and Itanium use this approach.

Basic blocks

The first kind of ILP is the one among instructions belonging to the same basic block.

A *basic block* is a sequence of instructions with

- No branches in, except to the entry
- No branches out, except at the exit.



Rescheduling

Within a basic block, the compiler may reschedule instructions to optimize the code.

Example

Consider the following high-level code

```
a = b + c;
d = e - f;
```

Assume load instructions have a latency of one clock cycle.

Example (I)

The plain code implementing the required computation is

```
LD  Rb, b
LD  Rc, c
ADD Ra, Rb, Rc
SD  a, Ra
LD  Re, e
LD  Rf, f
SUB Rd, Re, Rf
SD  d, Rd
```

Example (I)

```
LD   Rb, b       IF ID EX MEM WB
LD   Rc, c          IF ID EX  MEM WB
ADD  Ra, Rb, Rc        IF ID  st  EX  MEM WB
SD   a, Ra                IF  st  ID  EX  MEM WB
LD   Re, e                        IF  ID  EX  MEM WB
LD   Rf, f                            IF  ID  EX  MEM WB
SUB  Rd, Re, Rf                           IF  ID  st  EX MEM  WB
SD   d, Rd                                    IF  st  ID EX   MEM WB
```

14 clock cycles are required.

Example (II)

The optimally scheduled code is

```
LD    Rb, b
LD    Rc, c
LD    Re, e
ADD   Ra, Rb, Rc
LD    Rf, f
SD    a, Ra
SUB   Rd, Re, Rf
SD    d, Rd
```

No load interlock are required. Instruction scheduling may also require a clever register allocation (and a higher number of registers).

Example (II)

```
LD   Rb, b      IF ID EX MEM WB
LD   Rc, c         IF ID EX  MEM WB
LD   Re, e            IF ID  EX  MEM WB
ADD  Ra, Rb, Rc          IF  ID  EX  MEM WB
LD   Rf, f                   IF  ID  EX  MEM WB
SD   a, Ra                       IF  ID  EX  MEM WB
SUB  Rd, Re, Rf                      IF  ID  EX  MEM WB
SD   d, Rd                               IF  ID  EX  MEM   WB
```

12 clock cycles are required.

ILP in basic blocks

For typical MIPS program the typical size of a basic block is between 4 and 7 instructions.

Since these instructions are likely to be dependent one from the other, the amount of parallelism existing within a basic block is normally rather small.

To further increase the available parallelism, the parallelism among iterations of a loop is considered.

Loop-level parallelism

Example

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

Any iteration of the loop is independent on the others, so that they can be overlapped.

There are several ways for exploiting the loop-level parallelism:

- **loop enrolling (either static or dynamic)**
- **vector instructions.**

Loop unrolling

It is a technique that unroll the loops, by explicitly replicating the loop body multiple times.

```
for (i=0;i<N;i++ )
{
    body
}
```

```
for (i=0;i<N/4;i++ )
{
    body
    body
    body
    body
}
```

Advantages

In this way

- The relative overhead due to iteration is reduced
- The loop body is made wider, thus increasing the chance for the compiler to exploit rescheduling to eliminate stalls.

Vector instructions

A vector instruction operates on a set of data, instead of on a scalar data (as a normal instruction).

Vector processors implement vector instructions resorting to pipelines.

Although common in the past, vector processors are rare today.

They could be exploited for graphic, DSP, and multimedia applications in the near future.

Example

Let consider the code fragment:

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

This could be transformed in the following sequence of vector instructions:

- Load vector `x` from memory
- Load vector `y` from memory
- Add the two vectors
- Store the resulting vector.

DEPENDENCIES

If two instructions are not dependent, they can be executed in parallel without any stall.

If they are dependent, they have to be executed in order (although partly overlapped).

Therefore, exploiting the parallelism among instructions requires identifying *dependencies* existing among them.

There are three kinds of dependencies:

- data dependencies
- name dependencies
- control dependencies.

Data dependencies

An instruction *i* is data dependent on instruction *j* if either of the following conditions holds:

- instruction i produces a result that is used by instruction j, or
- instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.

<u>Example</u>

```
Loop:   L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
```


Data dependencies

An instruction *i* is data dependent or
either of the following conditions holds:

- instruction i produces a result t
instruction j, or
- instruction j is data dependent o
instruction k is data depende

First dependence

<u>Example</u>

```
Loop:   L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
```

Data dependencies

An instruction *i* is data dependent or
either of the following conditions holds:

- instruction i produces a result t
instruction j, or
- instruction j is data dependent or
instruction k is data dependent

Second dependence

<u>Example</u>

```
Loop:   L.D      F0, 0(R1)
        ADD.D    F4, F0, F2
        S.D      F4, 0(R1)
```


Dependencies and hazards

Dependencies are properties of the program.

Hazards are properties of the pipeline organization.

A dependency can cause a hazard or not (e.g., forwarding can avoid the hazard).

Dependencies

- create the possibility for a hazard
- determine the order in which results must be calculated
- set an upper bound on the amount of parallelism that can be exploited.

Memory dependencies

Detecting dependencies involving registers is easy.

Detecting dependencies involving memory cells is much more difficult, because accesses to the same cell can look very different.

Dependencies involving memory cells can only be detected at run time, when the addresses are known.

Name dependencies

A name dependency occurs when two instructions refer to the same register or memory location (*name*) but there is no flow of data associated to the name.

There are two kinds of name dependencies between an instruction i and an instruction j that follows:

- ***antidependence*: instruction j writes a register or memory location that instruction i reads, and instruction i is executed first.**
- ***output dependence*: both instruction i and instruction j write the same register or memory location.**

Example

```
Loop:   L.D      F0, 0(R1)
        ADD.D    F4, F0, F2
        S.D      F4, 0(R1)
        L.D      F0, -8(R1)
        ADD.D    F4, F0, F2
        S.D      F4, -8(R1)
        L.D      F0, -16(R1)
        …
```


Example

Output dependence

```
Loop:   L.D      F0, 0(R1)
        ADD.D    F4, F0, F2
        S.D      F4, 0(R1)
        L.D      F0, -8(R1)
        ADD.D    F4, F0, F2
        S.D      F4, -8(R1)
        L.D      F0, -16(R1)
        …
```

Example

Antidependence

```
Loop:   L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        L.D     F0, -8(R1)
        ADD.D   F4, F0, F2
        S.D     F4, -8(R1)
        L.D     F0, -16(R1)
        …
```


Register renaming

Name dependencies do not prevent from reordering involved instructions, provided that we change the register used by one of the two instructions.

This operation can be performed

- **Statically, by the compiler**
- **Dynamically, by the processor.**

A similar method (although more difficult to implement) can be followed for name dependencies involving memory locations.

Example

```
DIV.D    F0, F2, F4
ADD.D    F6, F0, F8
S.D      F6, 0(R1)
SUB.D    F8, F10, F14
MUL.D    F6, F10, F8
```

- Antidependence
- Could lead to an hazard

Example

```
DIV.D    F0, F2, F4
ADD.D    F6, F0, F8
S.D      F6, 0(R1)
SUB.D    T, F10, F14
MUL.D    F6, F10, T
```

Using a temporary register T eliminates the antidependence

Example

DIV.D F0, F2, F4
ADD.D F6, F0, F8
S.D F6, 0(R1)
SUB.D F8, F10, F14
MUL.D F6, F10, F8

- Output dependence
- Could lead to an hazard

Example

DIV.D F0, F2, F4
ADD.D S, F0, F8
S.D S, 0(R1)
SUB.D F8, F10, F14
MUL.D F6, F10, F8

Using a temporary register S eliminates the output dependence

Static register renaming

Some compilers perform register renaming to reduce the number of hazards (i.e., stalls).

Note that detecting all name dependencies requires carefully analyzing the code, taking also into account the effects of branches.

Hazards and data dependencies

Each time an operand involved in a dependency is accessed in a different order than the original one, there could be a hazard.

This means that the program output may become wrong.

Data hazards can be classified in three categories:

- RAW (*Read After Write*)
- WAW (*Write After Write*)
- WAR (*Write After Read*).

Data Hazard Classification

Consider an instruction *i* followed by an instruction *j*.

- RAW (*Read After Write*): *j* tries to read a source before *i* writes it
- WAW (*Write After Write*): *j* tries to write a destination before it is written by *i*
- WAR (*Write After Read*): *j* tries to write a destination before it is read by *i*.

RAR never corresponds to a hazard.

RAW hazards

They are the most common.

They correspond to a true data dependence.

<u>Example</u>

```
DADD    R1, R2, R3
DSUB    R4, R5, R1
```

WAW hazards

They stem from output dependences.

They are possible if

- **instructions may write in more than one stage, *or***
- **an instruction can proceed even if a previous instruction is stalled.**

<u>Example</u>

Suppose that load/store instructions require two memory cycles. The following situation causes a WAW hazard:

`LW`	`R1, 0(R2)`	IF	ID	EX	MEM1	MEM2	<u>WB</u>
`DADD`	`R1, R2, R3`		IF	ID	EX	<u>WB</u>	

37

WAR hazards

They stem from antidependence.

They are possible if there are instructions that write early in the pipeline, and others that read operands late.

The former case happen when implementing complex addressing modes, e.g., the autoincrement/autodecrement ones.

WAR hazards are quite rare.

Control dependencies

A control dependency occurs when an instruction depends on a branch.

Example

```
if p1 {
    S1;
};
if p2 {
    S2;
};
```

`S1` is control dependent on `p1`, and `S2` is control dependent on `p2`.

Constraints from control dependencies

- An instruction that is control dependent on a branch cannot be moved before the branch (so that its execution is no more controlled by the branch).
- An instruction that is not control dependent on a branch cannot be moved after the branch (so that its execution become dependent on the branch).

Control dependence and program correctness

Preserving control dependencies is a sufficient condition for preserving the program correctness.

But there are cases in which the reverse is not true.

The critical properties for program correctness are

- **exception behavior**
- **data flow.**

Exception behavior

Any change in the order of instruction execution must not change how exceptions are raised in the program.

Example

```
      DADDU    R2, R3, R4
      BEQZ     R2, L1
      LD       R1, 0(R2)
L1:  …
```

The `LD` instruction can cause an exception.

Therefore, moving the load instruction before the `BEQZ` is not allowed, because an exception caused by the load can then happen no matter whether the branch is taken or not.

Data flow

Data flow is the actual flow of data among instructions that produce results and consume them. Data flow must be preserved.

<u>Example</u>

```
      DADDU   R1, R2, R3
      BEQZ    R4, L
      DSUBU   R1, R5, R6
L:    …
      OR      R7, R1, R8
```

The value of `R1` used by the `OR` instruction must be that produced by the `DADDU` if the branch is taken, that produced by the `DSUBU` if it is not.

Example

There are cases in which it is possible to violate the control dependence without affecting the exception behavior or the data flow.

<u>Example</u>

```
      DADDU   R1, R2, R3
      BEQZ    R12, L
      DSUBU   R4, R5, R6
      DADDU   R5, R4, R9
L:    OR      R7, R8, R9
```

Let assume that R4 is not used any more after L.

Exam

There are cases in which it i
control dependance without
behavior or the data flow.

<u>Example</u>

```
    DADDU   R1, R2, R3
    BEQZ    R12, L
    DSUBU   R4, R5, R6
    DADDU   R5, R4, R9
L:  OR      R7, R8, R9
```

Let assume that R4 is not used any more after L.

The `DSUBU` instruction can be moved before the `BEQZ` instruction, since

- the `DSUBU` instruction cannot generate exceptions
- the program results are not changed anyway.

By doing this, the compiler *speculates*, i.e., bets on the branch not to be taken.