Instruction Level Parallelism

E. Sanchez

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 be partly combined.

Dynamic approach

It dominates the desktop and server markets, but is also included in PMD, with products such as

- Intel Core Series
- ARM Cortex-A9
- Athlon
- MIPS R10000/12000
- Sun UltraSPARC III
- PowerPC 603, G3, G4
- Alpha 21264
- RISC-V.

Static approach

It can mainly be found in products for the embedded market. For example, the ARM Cortex-A8.

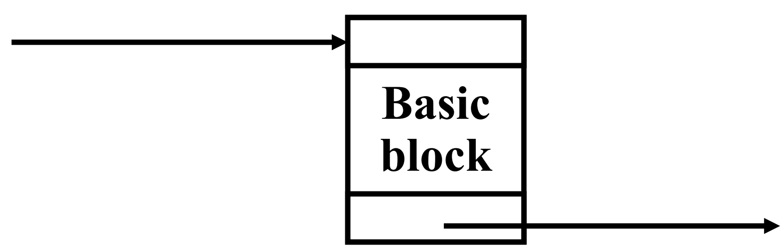
However, both the Intel IA-64 and Itanium use this approach.

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.



Example

```c
while (save[i] == k)
      i += 1;
```

```
LOOP:   sll     $t1, $s3, 2
        add     $t1, $t1, $s6
        lw      $t0, 0($t1)
        bne     $t0, $s5, EXIT
        addi    $s3, $s3, 1
        j       LOOP
EXIT:
```

```
LOOP
 |
 |
 |  +---------+
 +->| Block 1 | <---+
    | ------- |     |
    | sll     |     |
    | add     |     |
    | lw      |     |
    | bne     |     |
    +---------+     |
       |            |
 +-----+            |
 |     |            |
 |     \/           |
 |  +---------+     |
 |  | Block 2 |     |
 |  | ------- |     |
 |  | addi    |     |
 |  | j       |     |
 |  +---------+     |
 |     |            |
 |     +------------+
 |
 \/
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

Assuming that `x10 – x15` are properly set, the assembly code implementing the required computation is:

```
lw x1, 0(x10)
lw x2, 0(x11)
add x5, x1, x2
sw x5, 0(x14)
lw x3, 0(x12)
lw x4, 0(x13)
add x6, x3, x4
sw x6, 0(x15)
```

Example (I)

`lw x1, 0(x10)`	IF	ID	EX	MEM	WB											5
`lw x2, 0(x11)`		IF	ID	EX	MEM	WB										1
`add x5, x1, x2`			IF	ID	st	EX	MEM	WB								2
`sw x5, 0(x14)`				IF	st	ID	EX	MEM	WB							1
`lw x3, 0(x12)`						IF	ID	EX	MEM	WB						1
`lw x4, 0(x13)`							IF	ID	EX	MEM	WB					1
`add x6, x3, x4`								IF	ID	st	EX	MEM	WB			2
`sw x6, 0(x15)`									IF	st	ID	EX	MEM	WB		1

14 clock cycles are required.

Example (II)

The optimally scheduled code is

```
lw x1, 0(x10)
lw x2, 0(x11)
add x5, x1, x2
sw x5, 0(x14)
lw x3, 0(x12)
lw x4, 0(x13)
add x6, x3, x4
sw x6, 0(x15)
```

```
lw x1, 0(x10)
lw x2, 0(x11)
lw x3, 0(x12)
add x5, x1, x2
lw x4, 0(x13)
sw x5, 0(x14)
add x6, x3, x4
sw x6, 0(x15)
```

No load stalls are required.

Example (III)

```
lw x1, 0(x10)     IF ID EX MEM WB
lw x2, 0(x11)        IF ID EX MEM WB
add x5, x1, x2          IF ID EX MEM WB
sw x5, 0(x14)              IF ID EX MEM WB
lw x3, 0(x12)                 IF ID EX MEM WB
lw x4, 0(x13)                    IF ID EX MEM WB
add x6, x3, x4                      IF ID EX MEM   WB
sw x6, 0(x15)                          IF ID EX MEM WB
```

12 clock cycles are required.

ILP in basic blocks

For typical RISC-V programs, 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 two ways for exploiting the loop-level parallelism:

- **loop unrolling (either static or dynamic)**
- **SIMD.**

Loop unrolling

It is a technique that unrolls 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
     }
```

Loop unrolling

It is a technique that unrolls 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
}
```

If the iteration body corresponds to a basic block, after loop unrolling it is wider.

Example

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

```
for (i=0;i<N;i=i+4 )
{
    x[i] = x[i]+ y[i];
    x[i+1] = x[i+1]+ y[i+1];
    x[i+2] = x[i+2]+ y[i+2];
    x[i+3] = x[i+3]+ y[i+3];
}
```

Advantages

In this way

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

Disadvantages

Loop unrolling increases the size of the code.

SIMD

Single instruction stream, multiple data streams (SIMD) may be exploited in

- Vector processors

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

- Graphics Processing Units (GPUs)

  Different functional units perform similar tasks in parallel acting on multiple data.

Example

Let consider the following code fragment to be executed in a vector computer:

```
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:   fld     f0, 0(x1)
        fadd.d  f4, f0, f2
        fsd     f4, 0(x1)
```

Data dependencies

An instruction *i* is data dependent on
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 dependen

First dependence

Example

```
Loop:   fld     f0, 0(x1)
        fadd.d  f4, f0, f2
        fsd     f4, 0(x1)
```

Data dependencies

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

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

Second dependence

<u>Example</u>

```
Loop:   fld      f0, 0(x1)
        fadd.d   f4, f0, f2
        fsd      f4, 0(x1)
```

Dependencies and hazards

Dependencies are properties of the program:

- 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.

Hazards are potential problems stemming from dependecies in a specific pipeline organization.

Stalls depend on the program and the pipeline: a dependency can cause a hazard or not, and the hazard can cause a stall or not (e.g., forwarding can avoid the stall).

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.

If static techniques are used, the compiler must adopt a conservative approach, assuming that any load instruction refers to the same cell of a previous store.

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:   fld     f0, 0(x1)
        fadd.d  f4, f0, f2
        fsd     f4, 0(x1)
        fld     f0, -8(x1)
        fadd.d  f4, F0, F2
        fsd     f4, -8(x1)
        fld     f0, -16(x1)
        …
```

Example

Antidependence

```
Loop:   fld     f0, 0(x1)
        fadd.d  f4, f0, f2
        fsd     f4, 0(x1)
        fld     f0, -8(x1)
        fadd.d  f4, F0, F2
        fsd     f4, -8(x1)
        fld     f0, -16(x1)
        …
```

Example

Output dependence

```
Loop:   fld      f0, 0(x1)
        fadd.d   f4, f0, f2
        fsd      f4, 0(x1)
        fld      f0, -8(x1)
        fadd.d   f4, F0, F2
        fsd      f4, -8(x1)
        fld      f0, -16(x1)
        …
```

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, i.e., by the compiler
- Dynamically, i.e., by the processor.

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

Example

```
fdiv.d    f0, f2, f4
fadd.d    f6, f0, f8
fsd       f6, 0(x1)
fsub.d    f8, f10, f14
fmul.d    f6, f10, f8
```

- **Antidependence**
- **Could lead to a hazard**

Example

```
fdiv.d   f0, f2, f4
fadd.d   f6, f0, f8
fsd      f6, 0(x1)
fsub.d   T, f10, f14
fmul.d   f6, f10, T
```

Using a temporary register T eliminates the antidependence

Example

```
fdiv.d   f0, f2, f4
fadd.d   f6, f0, f8
fsd      f6, 0(x1)
fsub.d   f8, f10, f14
fmul.d   f6, f10, f8
```

- **Output dependence**
- **Could lead to a hazard**

Example

```
fdiv.d    f0, f2, f4
fadd.d    S, f0, f8
fsd       S, 0(x1)
fsub.d    f8, f10, f14
fmul.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>

```
add     x1, x2, x3
sub     x4, x5, x1
```

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 or processed by a stage for more than one clock cycle.

<u>Example</u>

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

`lw`	`x1, 0(x2)`	IF	ID	EX	MEM1	MEM2	MEM3	<u>WB</u>
`add`	`x1, x2, x3`		IF	ID	EX	MEM	<u>WB</u>	

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.

<u>Example</u>

```
    add     x2, x3, x4
    beq     x2, x0, L1
    lw      x1, 0(x2)
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>

```
    add   x1, x3, x4
    beq   x2, x0, L1
    sub   x1, x5, x6
L1: …
    or    x7, x1, x8
```

The value of `x1` used by the `OR` instruction must be that produced by the `add` if the branch is taken, or that produced by the `sub` if it is not taken.

Control dependence violation

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>

```
    add   x1, x2, x3
    beq   x12, x0, L
    sub   x4, x5, x6
    add   x5, x4, x9
L:  or    x7, x8, x9
```

Let assume that x4 is not used any more after L.

Control depende

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

Example

```
    add   x1, x2, x3
    beq   x12, x0, L
    sub   x4, x5, x6
    add   x5, x4, x9
L:  or    x7, x8, x9
```

Let assume that x4 is not used any more after L.

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

- the `SUB` 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.