CS 223 Computer Organization & Architecture

Lecture 29 [21.04.2020]

Compiler Techniques to Explore ILP

John Jose

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam.

Introduction

- Pipelining overlaps execution of instructions
- Exploits Instruction Level Parallelism (ILP)
- There are two main approaches:
  - Compiler-based static approaches
  - Hardware-based dynamic approaches
- **Exploiting ILP, goal is to minimize CPI**
  - Pipeline CPI =Ideal (base) CPI + Structural stalls +Data hazard stalls +Control stalls

Parallelism limitation within Basic Block

- The basic block- a straight-line code sequence without branches in except to the entry and no branches out except at the exit.
- Parallelism with basic block is limited. Typical size of basic block few instructions only. Must optimize across multiple blocks (branches)

```
w = 0;
x = x + y;
y = 0;
if( x > z)
  {
   y = x;
   x++;
  }
else
  {
   y = z;
   z++;
  }
w = x + z;
```

Source Code

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

```
y = z;
z++;
```

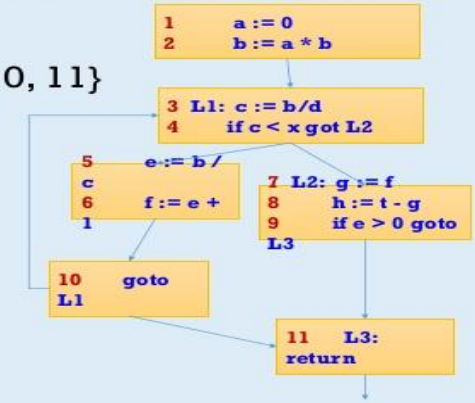
```
w = x + z;
```

Basic Blocks

Building basic blocks algorithm

```
1    a := 0
2    b := a * b
3 L1: c := b/d
4    if c < x goto L2
5    e := b / c
6    f := e + 1
7 L2: g := f
8    h := t - g
9    if e > 0 goto L3
10   goto L1
11   L3: return
```

Leaders?
– {1, 3, 5, 7, 10, 11}
Blocks?
– {1, 2}
– {3, 4}
– {5, 6}
– {7, 8, 9}
– {10}
– {11}



Data Dependence

- Loop-Level Parallelism
  - Unroll loop statically or dynamically
- Challenges→ Data dependency
- Data dependence conveys possibility of a hazard
- Dependent instructions cannot be executed simultaneously
- Pipeline determines if dependence is detected and if it causes a stall or not
- Data dependence conveys upper bound on exploitable instruction level parallelism

Name Dependence & Output dependence

- Two instructions use the same name but no flow of information.
- Not a true data dependence, but is a problem when reordering instructions
  - **Antidependence:** instruction j writes a register or memory location that instruction i reads
    - Initial ordering (i before j) must be preserved
  - **Output dependence:** instruction i and instruction j write the same register or memory location
    - Ordering must be preserved
- To resolve, use renaming techniques

Control Dependence

- Ordering of instruction with respect to a branch instruction
  - Instruction that is control dependent on a branch cannot be moved **before** the branch so that its execution is no longer controller by the branch
  - An instruction that is not control dependent on a branch cannot be moved **after** the branch so that its execution is controlled by the branch.

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

Control Dependence

- Instruction that is control dependent on a branch cannot be moved **before** the branch so that its execution is no longer controller by the branch
- An instruction that is not control dependent on a branch cannot be moved **after** the branch so that its execution is controlled by the branch.

Example 1:

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

Example 2:

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

Example 3:

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

Compiler Techniques for Exposing ILP

- Find and overlap sequence of unrelated instruction
- Pipeline scheduling
  - Separate dependent instruction from the source instruction by pipeline latency of the source instruction
- Example:

```
for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Pipeline 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 (assume integer load latency is 1)
        BNE R1,R2,Loop
```

```
for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Pipeline 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 (assume integer load latency is 1)
        BNE R1,R2,Loop
```

Scheduled code:

```
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
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Loop Unrolling

- **Loop unrolling**

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

Scheduled code:

```
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
```

number of live registers ?

Loop Unrolling/Pipeline Scheduling

❖ Loop unrolling

```
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)
        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
```

❖ Scheduled unrolled loop:

```
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)
        DADDUI R1,R1,#-32
        S.D F12,16(R1)
        S.D F16,8(R1)
        BNE R1,R2,Loop
```

Strip Mining

- Unknown number of loop iterations?
  - Goal: make k copies of the loop body Number of iterations = n
  - Generate pair of loops:
    - First executes n mod k times
    - Second executes n / k times
    - Strip mining
  - Example : Let n=35, k=4
    - Loop 1 execute 3 times
    - Loop 2 execute 8 times by unrolling 4 copies per iteration

Steps in Loop Unrolling and Scheduling

- Determine that unrolling the loop would be useful.
- Identify independency of loop iterations.
- Use different registers to avoid unnecessary constraints put in on same computations.
- Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.
- Determine whether the loads and stores from different iterations are independent.
- Schedule the code, preserving any dependences needed to yield the same result as the original code.

Loop Unrolling & Pipeline Scheduling

- **Limitations of loop unrolling:**
  - Code size limitations – I-cache miss
  - Compiler limitations – register pressure

johnjose@iitg.ac.in
http://www.iitg.ac.in/johnjose/