Instruction Level parallelism

ILP :- overlapping inst during the execution.

There are 2 approaches to exploit ILP

1) An approach that relies on h/w to help discover & exploit parallelism dynamically

2) An approach that relies on software technology to find parallelism Statistically during compile time.

many Techniques are used to Exploit parallelism among instructions in ~~blocks~~ different blocks. The block may contain straight line codes, many ALU & I/O instructions usually

~~35-30%~~ 15%-25% of code in a pgm has branch instructions.

Since these inst are likely to depend on each other the amount of ~~par~~ overlap we can exploit within a basic block is likely to be less than the avg block size. To obtain Substantial performance enhancement, we must exploit ILP across multiple basic blocks.

The Simple way to exploit parallelism is iterations in a loop.

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

This is also known as loop level parallelism.

Every iteration can overlap with any other iteration, although within each loop iteration there is little or no opportunity for overlap.

There are different techniques are used to convert loop level parallelism into instruction level parallelism.

Data dependence and Hazards

Determining how one instruction depends on another is critica to determining how much parallelism exists in a program and how much parallelism can be exploited in a pgm.

In particular, to exploit instruction - level, we have to determine which instruction can be exected in parallel.

If two inst are in parallel, they can executed ~~p~~ simultan -ously without causing stalls, then they are independent inst.

If two inst have to be exected, then it is not possible to execute completly overlapping 2 inst, then they are dependent inst

Dependents inst are executed in order by partially overlapping

There are differnt types of dependence among the instructions

1) Data dependence:

An inst j is data dependence on inst i such that.

1) inst i produces the result that can be used by inst j

2) inst k is dependent on j & j is dependent on i.

known as chain of dependences. The dependence chain can be as long as the entire program.

For Eg consider the mips code

```
loop:   LD      R0, 0(R1)
        ADD     R2, R0, R3
        SD      R2, 0(R1)
        DADDUI  R1, R1, #-8
        BNE     R1, R2, loop
```

Data dependence

There is data dependence b/w LD R0, 0(R1)

```
ADD   R2, R0, R3
SD    R2, 0(R1)
```

Data dependence

nce dependence must be preserved for the correct execution
If 2 inst are dependent, they must be executed in order
& can't execute Simultaneously
Executing Simultaneously causes pipeline interlocks to detect hazard & stall, there by reducing the overlap.
Dependence are the properties of programe, whether a dependence results in actual hazard being detected or whether that hazard actually causes a stall, these are the properties of the pipeline organization.

A data dependence conveys 3 things
1) The possibility of a hazard 2) The order in which results must be calculated. 3) The upper bound on ~~[illegible]~~ how much parallelism can be exploited.
A dependence can limit the amount of ILP. A data dependence can over come 2 ways
1) maintaining the dependence but avoiding a hazard
2) Eliminating the dependence by transforming the code

Name dependence :-
A name dependence occurs when two inst uses same register or memory location called a name, but there is no flow of data b/w those inst associated with that name.
There are 2 types of name dependence between i that preced inst j.
i) Antidependence :- b/w inst i and j occurs when j write a regr or memory that inst i reads, the original order must be preserved. Eg SD R1, 0(R2)
DADDUI R2, R2, #8.

Scanned with CamScanner

2) output dependence:- when both inst tries to write to a same reg / memory, the ordering must be preserved such first value gets updated by i then j.

They are just name dependences ~~not~~ opposed to true data dependences, since there is no value being transmitted b/w the instructions. So these inst can be reordered & even be executed parallely.

Data Hazards:-

A exists whenever there is data dependence b/w two instructions, they are so close that ~~changing the order~~ overlapping the inst may change the order of accessing the operands involved in the dependence.

program order must be maintained while preserving the dependence. Executing the inst in the order in which it was executed sequentially.

The main goal is
1) Preserve the pgming order. if it's output affects the result of the program.

2) maintaing the data dependence.
3) Detecting and avoiding the hazards & stalls.

The possible data hazards are.

1) RAW:- inst j tries to read the operand which ~~was~~ before inst i writes it.

ADD R1, R2, R3
~~ld R4, 10(R~~ SUB R4, R1, R5.

pgm order shld be preserved such that sub reads the updated data

Scanned with CamScanner

WAW:- inst j ~~write to a~~ tries to write an operand, where inst ~~is~~ i is writing. The write end up being performed in the wrong order, leaving the value i instead of j in the destination. WAW occurs in pipelines that write in more than one pipe stage

3) WAR:-(write after read):- inst j tries to write an operand before inst i reads it. So i incorrectly gets a new value. It occurs in pipeline where inst performs write in pipestage early before other inst reads it

RAR is not a hazard

Control dependence:-

It defines ordering of inst i related to branch inst in correct order and when it so that inst i shld be executed in correct order, and when it should be. Every inst, except the first basic block of the pgm is control dependent.

inst under the branch should should be executed in order and branch dependence should be preserved.

if P1 { S1; S1 is control dependent on P1
}

if P2 { S2; } S2 is control dependent on P2

There are 2 constraints imposed by control dependence

i) inst i inside the branch can not be put before the branch, so that it's pgming order is violated

ii) Any inst above/below the branch should not be executed inside the branch ~~[illegible]~~ because those inst are not control dependent.

When processor executes these inst, compiler shld make sure that program is executing in the order and also control dependence must be preserved.

but if changing the order of the program doesn't the o/p of the program then we can compromise. for the Control dependence

The main critical properties to maintain control dependence are i) data flow ii) Exceptions

Preserving the exceptions ~~doesn't~~ means that during the Execution inst shld not cause new exception

Eg

```
DADDU  R1, R2, R3
  BEQZ R1, L1
  LD   R4, 10(R1)

L1:
```

here eventhough, there is no data depence that exist directly since after ADD R1 value is not changed but putting LD before BEQZ cause memory protection violation exception

it is only the name dependence. here Ld shld execute only if branch is not taken

because to preserve control dependence, we shld not put LD before BEQZ.

2) Data flow:

```
    SUB  R1, R2, R3

    ~~LD~~ BEQZ R4, L1
    ADD R1, R6, R5
L1: ! _ _ _ _
LP: _ ! _ _ _ .

    OR R7, R1, R3
```

here. BEQ is testing R4 when R4 is not at all used by any inst.
& but OR inst is data dependent on both SUB & ADD i.e if branch takes place, then OR shld hold the R1 of SUB else the result produced by ADD.
So here the control dependence is not the main criteria but preserving data dependence and Pgm's order is important

ADD R1, R2, R3
Sub R4, R1, R2
BEQZ R, loop
Sub R5, R6, R7
loop: &
loop ADD R7, R5, R7

if Branch ~~wa~~ taken then
Sub can be executed
before branch but speculation S/w ~~[illegible]~~ is required

Basic compiler Techniques for exploiting ILP.

These Techniques can be used to Schedul instructions to exploit parallelism.

Basic pipelining Scheduling and loop unrolling:-

To keep the pipeline full, parallelism among inst can be obtaind by executing the sequence of unrelated inst that can be overlapped in the pipeline.

To avoid pipeline Stall, dependent inst must be ~~executed~~ separated from the source inst by a distance equal to the pipeline latency of that source instruction.

The compiler ability to schedule the instructions depends on the amount of ILP available in the program and on the latency of the functional units in the pipeline.

Eg consider for(i=999; i>=0; i--)

x[i]= x[i]+5;

Each iteration is independent but dependence exist inside the instruction itself. To ~~see~~ chec how this iteration execute first we will convert this into assembly level language.

```
L ~~mov~~  F4, 0(R1)      highest addres
ADD    F4, F4, R2      x[i]+s
SD     F4  0(R1)
~~BNE  R3~~
DADDUI  R1, R1, #-8.
BNE  R1, R2, Loop   lowest element address
```

The unscheduled loop with stalls is given below. For this we consider the latency of FP operations

Inst producing result	inst using result	latency in CC
FP ALU OP	FP ALU	3
FP ALU OP	Store	2
load	FP ALU operation	1
Load	Store.	0

```
1   LD     F4, 0(R1)
2     Stall
3   ADD.D   F4, F4, R3
4     Stall
5     Stall
6   S.D   F4, 0(R1)
      ~~Stall~~
7   DADDUI  ~~R~~ F4, F4, #-8.
8     Stall
9   BNE   R1, R2
```

This will take ~~10~~ 7 CC for 1 iteration 1 Store
for 4 = ~~10~~ 3x4 = 12 . Each element is stored in 7 CC but actual work
LD, DADD, SD latency 3 CC
So DADDUI & BNE ~~are~~ + ~~2~~ stalls are overhead

load]3 1
add]2
store 3 4 CC is extra
only 3 is enough

Scheduled loop

```
LD  F4, 0(R1)        — 1
DADDUI  R1, R1, #-8. — 2
ADD.D F4, F4, R3     — 3
  Stall              — 4
  Stall              — 5
S.D  F4, 0(R1)       — 6
BNE  R1, R2          — 7
```

Here we can complete within 7 CC.

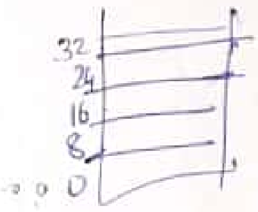
but 2 stalls & DADD is a overhead & BNE

Another scheme to increase the no of instructions relative to branch & overhead inst is loop unrolling. Unrolling simply replicates the loop body multiple times, by adjusting the loop termination code correctly.

Loop unrolling is used to improve the scheduling. It allows multiple iterations to be scheduled together. In this case, we can eliminate the data stall by creating additional independent instructions. To create successfully these addition inst is not possible if we use the same registers, because it may ~~the~~ affect the performance of the scheduling ∴ different registers can used for each iteration.

unrolled loop for the previous ex

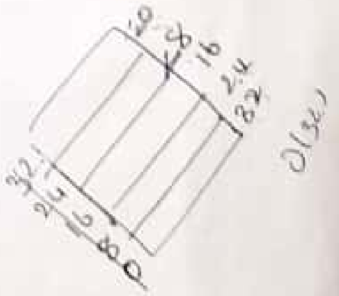
```
LD F0, 0(R1)
ADD.D F0, F0, R2
S.D F0, 0(R1)
LD F1, ~~0(R)~~ -8(R1)
ADD.D F1, F1, R3
SD F1, -8(R1)
LD F2, -16(R1)
ADD.D F2, F2, R4
S.D F2, -16(R1)
LD F3, -24(R1)
ADD F3, F3, R5
SD F3, -24(R1)
DADDUI R1, R1, #-32
BNE R1, R2, loop
```



32
24-32
08

1024
1016
1008
1000

3 10000
−32



ere the unrolled loop without Schedule is

```
Loop      LD    F0, 0(R1)
   1      ADD   F1, F0, F2
   2      SD    F1, 0(R1)

          LD    F3, -8(R1)
          ADD   F4, F3, F5
          SD    F4, -8(R1)

          LD    F6, -16(R1)
          ADD   F7, F6, F8
          SD    F7  -16(R1)

          LD    F9, -24(R1)
          ADD   F10, F9, F11
          SD    F10, -24(R1)
          DADDUI R1, R1, #32
   1      BNE   R1, R2, loop
```

LD - ADD - SD
 1   2

$3 \times 4 = 12$ Stalls inside loop
+ 14 instruction
+ 1 ~~Stalls~~ branch
27 CC is required

loop iteration i.e. Termination code is adjusted with ld & sd & different regs are used to avoid hazards

It optimizes the code by using Substitution and Simplifications

If Actually we do not know the upper bound on the array i.e it is difficult to unknown k, we can known during the compilation time

Suppose if the size is n & we should unroll k copies of each iteration, then instead of single large unrolling rod, sim generate a pair of consecutive loops first loop executes ~~of~~ n%k times. For large value of the second unrolled loop executes n/k times. For loop unrolling it most of execution time is spent for loop unrolling

Unrolling loops improves the performance of loop over pipelining & eventhough code size is large.

```
loop  LD  F0, 0(R1)        SD F1, 0(R1)
      LD  F3, -8(R1)       SD F4, -8(R1)
      LD  F6, -16(R1)      SD F7, -16(R1)
      LD  F9, -24(R1)      SD ~~F10, -24(R1)~~
      ADD F1, F0, F2       ~~DADDUI R1~~, R1, #32
      ADD F4, F3, F5       SD F10, -24(R1)
      ADD F7, F6, F8       BNE R1, R2, loop.
      ADD F10, F9, F11
```

This loop will work without any stalls.

~~ere~~ the unrolled loop without Schedule is

```
Loop    1 ┌ LD    F0, 0(R1)           LD — ADD — SD
          └ ADD   F1, F0, F2              1      2
        2 └ SD    F1, 0(R1)

            LD    F3, -8(R1)
            ADD   F4, F3, F5
            SD    F4, -8(R1)

            LD    F6, -16(R1)
            ADD   F7, F6, F8
            SD    F7  -16(R1)

            LD    F9, -24(R1)
            ADD   F10, F9, F11
            SD    F10, -24(R1)
            DADDUI R1, R1, #32
        1 — BNE   R1, R2, loop
```

3×4=12 stalls inside loop
+ 14 instruction
+ 1 ~~Sodary~~ Branch
――――
27 CC is required

loop iteration i.e. Termination code is adjusted with ld & sd & different regs are used to avoid hazards

It optimizes the code by using Substitution and Simplifications

If Actually we do not know the upper bound on the array i.e it is difficult to known; we can known during the compilation time

Suppose if the size is N & we should unroll k copies of each iteration, then instead of single large unrolling rool, sim generate a pair of consecutive loops first loop executes ~~n~~ n%k times For large value of the second unrolled loop executes n/k times.
If most of execution time is spent for loop unrolling
loop unrolling loops improves the performance of loop over pipeling even though code size is large.

```
loop  LD  F0, 0(R1)          SD F1, 0(R1)
      LD  F3, -8(R1)         SD F4, -8(R1)
      LD  F6, -16(R1)        SD F7, -16(R1)
      LD  F9, -24(R1)        ~~SD F10, -24(R1)~~
      ADD F1, F0, F2         ~~DADDUI R1, R1, #32~~
      ADD F4, F3, F5         SD F10, -24(Ri)
      ADD F7, F6, F8         BNE R1, R2, loop
      ADD F10, F9, F11
```

This loop will work without any stalls.