Compiler Support for Branch Handling

Using compilers for hard to predict branches

Predication:

In modern processors, penalty for mispredicted branch is ~~huge~~. huge.

A pipeline may be filled with instructions executed in parallel — say a 12 stage pipe with 4 IPC.
$\sim$ 48 instructions in the pipe

→ will go waste if prediction is wrong.

Workaround → Do the work both ways [taken & not taken]

In a branch you always ~~waste~~ waste exactly 50% of the ~~prediction~~ executed instruction if you use predication.

Where useful??

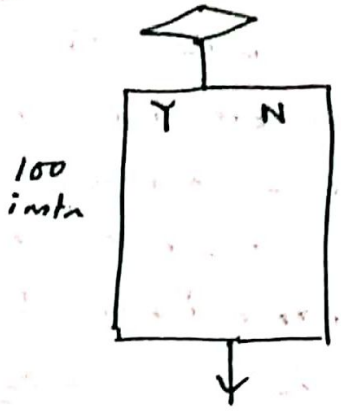
Consider a loop → in every iteration you execute loop instr & also outside loop.

→ lot of work wasted ~~in~~ if loop is deep

Better to predict

Fn call / return: Unless the call is conditional it is always taken, ~~but~~ hence no point of predication.

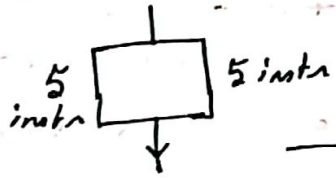
A large if-then-else



~ amount of waste may be too much w.r.t. B.P.

∴ for large if-then-else we should be predicting.

Small if-then-else → less wastage
∴ makes sense to do predication.



→ wastage 5 instr

comider 100 instances
→ we waste 500 instr
using BP with 90% accuracy & penalty of 50 instr
→ we waste $\underset{\text{wrong instances}}{10} \times 50 = 500$

∴ for smaller if-then-else, we want to predicate.

if conversion

```
if (cond) {
    x = arr[i];
    y = x+1;
}
else {
    x = arr[j];
    y = x-1;
}
```

⟹

```
x1 = arr[i];
x2 = arr[j];
y1 = y0 + 1;
y2 = y - 1;
x = cond? x1 : x2
y = cond ? y1 : y2
```

still need a branch instr

alternate idea
have hw support for a conditional 'mov' instr ⟸

Conditional Mov

Simplest indication → mov X X_1 flag

MIPS → MOVZ R_d R_s R_t

⇒ if (R_t == 0) $R_d = R_s$;

↔ there is not branching to anywhere.

MOVN

if (R_t != 0)

X = cond ? X1 : X2;

↓↓

R_3 = cond;

result

R1 = X1 expression result

R2 = X2

MOVN X, R1, R3

MOVZ X, R2, R3

X86 ! → CMOV instructions

Example on MOVN / MOVZ

80% → BEQZ R1, Else

40 instr penalty

ADDI R2, R2, 1

100% → B END

Else:

ADDI R3, R3, 1

END:

if conv

ADDI R4, R2, 1

ADDI R5, R3, 1

MOVN R2, R4, R1

MOVZ R3, R5, R1

↓

4 instr

Performance

↓

3 instr on one path

2 instr on other path

Consider 50% taken & 50% not taken

2.5 instr + 0.2 × 40

= 10.5 instr on average

Need compiler support for if conversion.
Such conversion requires more registers.

A branch hard to predict as ~~define~~ can be found by profiling → we can perform if ~~conversion~~ conversion.

In general case → if conv requires more instructions.

full predication → make all instr is conditional,

this alleviates the problem of requiring more instructions or extra registers.

Full predication HW support

EX:→ ITANIUM ISA
- every instr has conditional bits.
- 6 bits specify condition.
  - tells which of 64 registers (1 bit) to use to determine execution.

Earlier example

(i) MPEQZ P1, P2 R1
predicate set instr, P1 & P2 are always set opposite
if R1 == 0, P1 true, P2 false
& vice versa.

(ii) (P2) ADDI R2, R2, 1

(iii) (P1) ADDI R3, R3, 1

(5)

Compiler Support for ILP

HW has a limited window of view of the program → cannot track dependency beyond the window
limited by ROB size.

Compiler has an overall view.

Tree height Reduction technique :→

R8 = R2 + R3 + R4 + R5

```
ADD  R8, R2, R3
ADD  R8, R8, R4
ADD  R8, R8, R5
```

3 cycles

Tree height Reduction opt →

```
ADD  R8, R2, R3
ADD  R7, R4, R5
ADD  R8, R8, R7
```

2 cycles.

exploit associativity of '+'.

Instn scheduling by compiler

ϕ denotes address in R1.

Loop:

```
LW   R2, ϕ(R1)
ADD  R2, R2, R0
SW   R2, ϕ(R1)
ADDI R1, R1, 4
BNE  R1, R3, Loop
```

Consider a simple Processor, w/o R/s & o-o-o support.

cycle	
1	LW → 2 cycle with cache hit
2	Stall
3	ADD — 3 cycle ← dependency
4	Stall
5	Stall
6	SW ← dependency
7	ADDI — 3 cycles
8	Stall
9	Stall
10	BNE

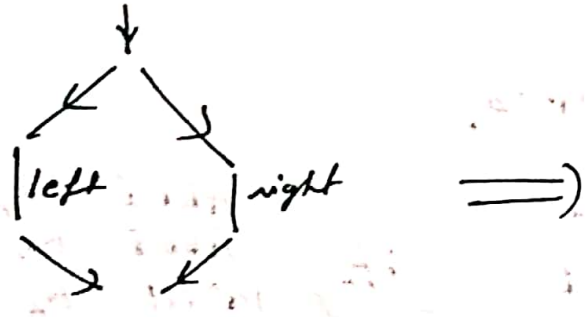
no dependency

A transformed but equivalent code

```
loop:
    LW   R2, 0(R1)
    ADDI R1, R1, 4
    ADD  R2, R2, R0
    SW   R2, -4(R1)
    BNE  R1, R3, Loop
```

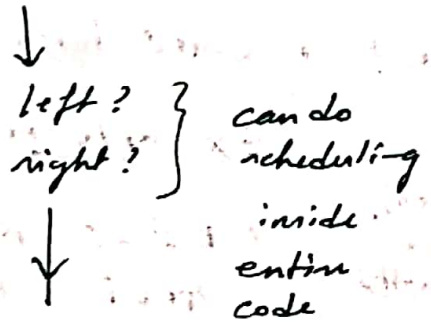
Cycle	
1.	LW
2.	ADDI
3.	ADD
4.	Stall
5.	Stall
6.	SW
7.	BNE

Scheduling & If-conversion



We can only do instr scheduling in individual branches

⟹



if-convert a loop

```
Loop:
    LW   R2, 0(R1)   <- 1 stall cycle
    ADD  R2, R2, R3  <- 1 stall
    SW   R2, 0(R1)   <- no stall, addI dont depend on SW
    ADDI R1, R1, 4   <- 1 stall
    BNE  R2, R1, loop
```

assume all instr has 2 cycle exec time

(7)

after scheduling →

```
Loop: LW   R2, 0(R1)
      ADDI R1, R1, 4
      ADD  R2, R2, R3
      SW   R2, -4(R1)   ← 1 stall
      BNE  R1, R5, Loop
```

A potential if-conversion can eliminate the branch & then some code after BNE can fill in the stall cycle, but as we have seen, if conversion of loops insert 1 predicate per iteration which is very costly.

Consider a loop which sometimes execute 10^4 times but sometimes only once, in predication it is always exponentially many predicates. considering all the possibilities.

Alternative is to unroll the loop

```
for(i=1000; i!=0; i--)
    a[i] = a[i]+5;
```

```
Loop: LW   R2, 0(R1)
      ADD  R2, R2, R3
      SW   R2, 0(R1)
      ADDI R1, R1, -4
      BNE  R1, R5, Loop
```

No. of instn = 5×1000 = 5000

Unroll Once

```
for( ; ; i=i-2)
{ a[i] = a[i]+5;
  a[i-1] = a[i-1]+5;
}
```

```
Loop: LW   R2, 0(R1)
      ADD  R2, R2, R3
      SW   R2, 0(R1)
      LW   R2, -4(R1)
      ADD  R2, R2, R3
      SW   R2, -4(R1)
      ADDI R1, R1, -8
      BNE  R1, R5, Loop
```

No. of instn
= 8×500
= 4000

Reducing Loop overhead

exectime
= #instn × CPI × Cycle time

reduced definitely

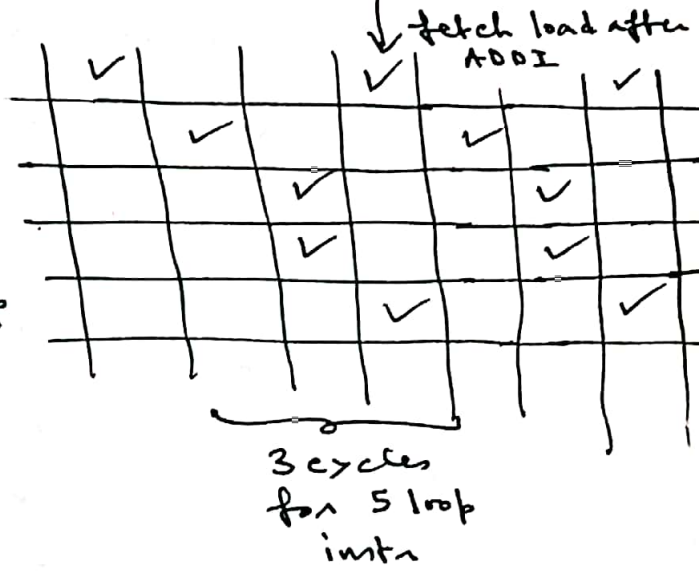
same or may change

same

Loop unrolling benefit ⟶ CPI ↓

Consider 4 issue in-order, perfect BP

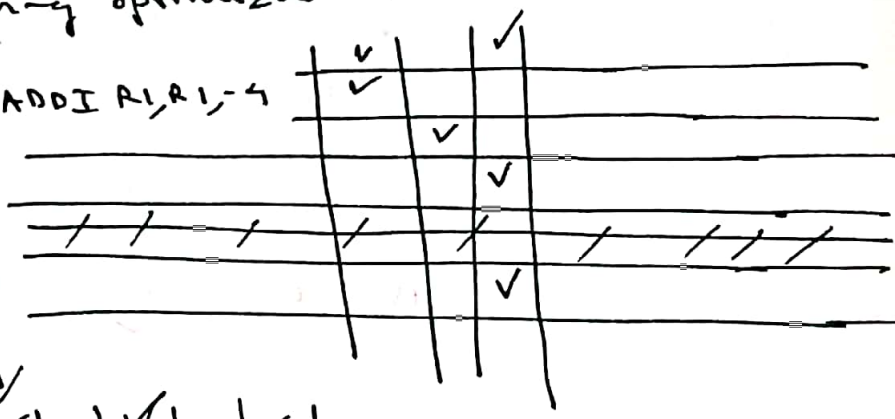
```
Loop: LW   R2, 0(R1)
      ADD  R2, R2, R3
      SW   R2, 0(R1)
      ADDI R1, R1, -4
      BNE  R1, R5, Loop
```



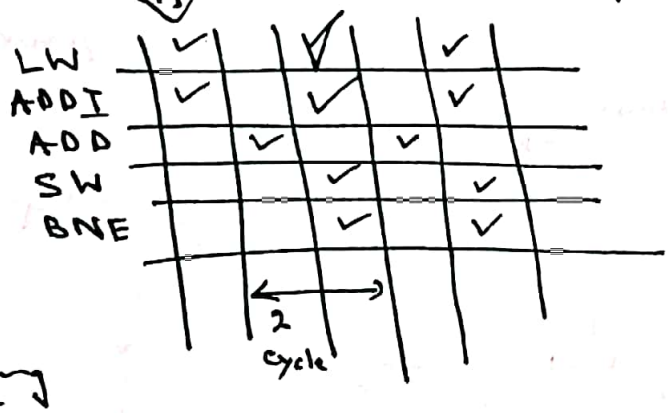
CPI = 3/5 = 0.6

perform previous re-ordering optimization

```
Loop: LW   R2, 0(R1)  ⟶ ADDI R1, R1, -4
      ADD  R2, R2, R3
      SW   R2, 0(R1)
      ADDI R1, R1, -4
      BNE  R1, R5, Loop
```



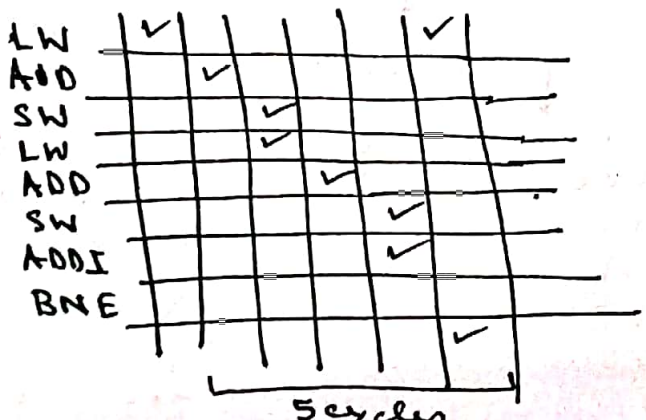
```
loop:
LW   R2, 0(R1)
ADDI R1, R1, -4
ADD  R2, R2, R3
SW   R2, 0(R1)
BNE  R1, R5, loop
```



∴ CPI = 2/5 = 0.4

Unroll w/o scheduling

Loop:



CPI = 5/8 = 0.62

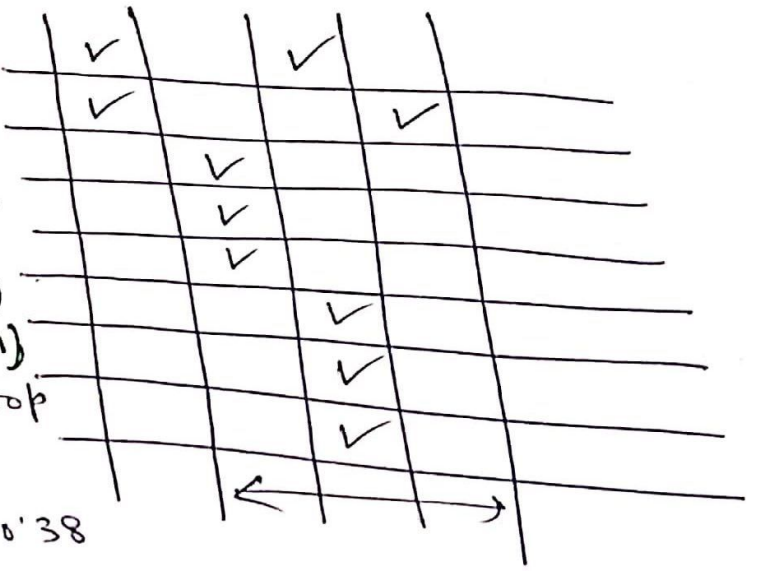
similar to original.

Unroll once + schedule

```
LW R2, 0(R1)
LW R10, -4(R1)  ⟶ parallel load in new register
```

↓

```
loop:
LW R2, 0(R1)
LW R10, -4(R1)
ADD R2, R2, R3
ADD R10, R10, R3
ADDI R1, R1, -8
SW R2, 8(R1)
SW R10, 4(R1)
BNE R1, R5, loop
```



CPI $= \frac{3}{8} = 0.38$

#instr reduced ↓ ← Unrolling

CPI reduced ↓ ← more scheduling scope due to unrolling.

Downside to Unrolling

Code bloating ⟶ increased code size

What if no. of iterations is unknown, we exit after executing extra instr.

Also, what if no. of iterations is not a multiple of the unroll factor.

Fn call inlining

Fn call

```
code
parameter set up
fn call  ⟶ Fn. . .
code     ⟵
:
```

preparing fn params is not required with inlining.

⟶ enables instr scheduling for larger code sequence

Also, less instr ⟶ CPI ↓.

Good for small fns which have less scope of scheduling inside fn body.

Fn call inlining downsize ⟶ code bloat

If too many calls exist ⟶ inlining increase instr count very much.

Other IPC enhancing techniques.

Software pipelining

```
loop {  LW   a[i]
        ADD  s
        SW   a[i]
        ADDI i, 1
        BNE
```

⟶ think of loop as a pipeline

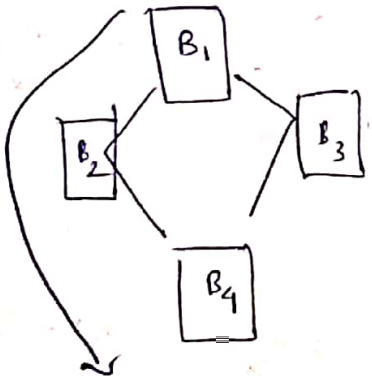
```
        SW   a[i]
        ADD  s, a[i+1]
        LW   a[i+2]
```

added in prev iteration

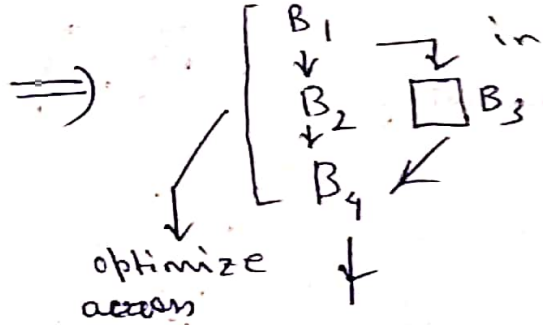
NO Dependency

Value added has been loaded in previous iteration of the loop

Trace Scheduling



Say, $B_1 \rightarrow B_2 \rightarrow B_4$ is common path



optimize across Basic Blocks

in case of a jump to B_3, undo effect of wrongfully executed instructions & proceed.

↓

execute some compensatory code.