



Exploiting Instruction-Level Parallelism with Software Approaches

Chapter 4

Scheduling Code for Pipelines

- To avoid pipeline stalls, instructions dependent on some instruction i must be separated from i by at least c cycles (where c denotes i 's pipeline latency).

In the following, assume the following latencies:

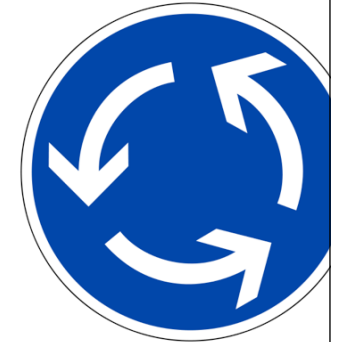
Producing Instruction	Consuming instruction	Latency
ALU Op	ALU Op	0
FP ALU Op	FP ALU Op	3
FP ALU Op	Store	2
Load	(FP) ALU Op	1
Load	Store	0*

Scheduling Code for Pipelines

- Further, assume the simple standard 5-stage pipeline:
 - integer ALU ops with 0 cycle latency,
 - branch delay slot of one cycle,
 - functional units fully pipelined or replicated as many times as pipeline depth
(no structural hazards—any instruction can be issued on every clock cycle).

Loop Parallelism

```
double x[], s;  
  
for (i=1000; i>0; i=i-1)  
    x[i] = x[i]+s;
```



- This loop is obviously parallel: each loop iteration may be evaluated independently (independence can be detected at compile time).
- Can we exploit this parallelism/independence to improve loop performance in the MIPS pipeline?

Loop Parallelism

- Equivalent MIPS code:

$R1 \equiv$ address of $x[1000]$, $R2 \equiv$ address of $x[0]$, $F2 \equiv s$

```
loop: L.D    F0, 0(R1)    ; F0 ← x[i]
      ADD.D  F4, F0, F2   ; F4 ← F0 + s
      S.D    F4, 0(R1)   ; x[i] ← F4
      DADDUI R1, R1, #-8
      BNE    R1, R2, loop ; R1 ≠ R2?
```

- How well will this code run in the MIPS pipeline?

Unscheduled Loop Code

Cycle issued

Loop:	L.D	F0, 0(R1)	1
	stall		2
	ADD.D	F4, F0, F2	3
	stall		4
	stall		5
	S.D	F4, 0(R1)	6
	DADDUI	R1, R1, #-8	7
	stall		8
	BNE	R1, R2, loop	9
	stall		10

Loop Code Scheduled

loop:	L.D	F0, 0(R1)	1
	DADDUI	R1, R1, #-8	2
	ADD.D	F4, F0, F2	3
	stall		4
	BNE	R1, R2, loop	5
	S.D	F4, 8(R1)	6

- The stall separates **ADD.D** and the dependent **S.D**.
- Note: Effective store address now **8(R1)**, was **0(R1)**.
- Dependency chain **L.D** → **ADD.D** → **S.D**:
6 cycles is optimal. Saved through scheduling: 40%.

Loop Payload vs. Loop Control



- The scheduled loop code stores back one array element every 6 cycles.
- But the actual work—payload—of the loop accounts for 3 cycles (**L . D**, **ADD . D**, **S . D**) only, a mere 50%.
- Loop control contributes 2 cycles: **DADDUI**, **BNE .**

Try to increase the loop payload/control ratio.

Loop Unrolling

- Loop unrolling: *replicate loop body* multiple times (and adjust the loop termination code).
 - Helps scheduling: instructions from subsequent iterations may be scheduled together (replicated loop body supplies more independent instructions).
 - Rotate/alternate register usage to prevent data hazards.
 - Increases code size.

Unrolled Loop

```
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
```

- Body replicated 4 times.
- Registers not reused.
- Initially assume $(R1-R2) \bmod 32 = 0$.
- Saved loop control:
 $3 \times (\text{DADDUI} + \text{BNE})$.
- Code *not* scheduled yet!

Adapting Loop Control

```
loop: L.D      F0,0(R1)
      ADD.D   F4,F0,F2
      S.D     F4,0(R1)
      DADDUI R1,R1,#-8
      L.D     F6,0(R1)
      ADD.D   F8,F6,F2
      S.D     F8,0(R1)
      DADDUI  R1,R1,#-8
      L.D     F10,0(R1)
      ADD.D   F12,F10,F2
      S.D     F12,0(R1)
      DADDUI  R1,R1,#-8
      L.D     F14,0(R1)
      ADD.D   F16,F14,F2
      S.D     F16,0(R1)
      DADDUI  R1,R1,#-8
      BNE    R1,R2,loop
```

R1 → R1+8

Adapting Loop Control

- Displacement addressing ($d(R1)$) is used to compensate for the merged **DADDUI**.
 - Such code transformations are not trivial and require the compiler to perform symbolic substitution and simplification.
- Loop body B ,
of loop iterations: n ,
of replicated unrolled loop bodies (B'): k

Execute $B \ n \bmod k$ times; then execute $B' \ n/k$ times.

Duff's Device

```
send(short *to, short *from,
      int count)
{
    int n = (count+7)/8;

    switch (count%8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
                } while(--n > 0);
    }
}
```

Scheduling the 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)      ; 16-32 = -16
      BNE     R1, R2, loop
      S.D      F16, 8(R1)      ; 8-32 = -24
```

Loop-Carried Dependencies

- Up to here, we have studied loops in which iterations could be performed in any order or even in parallel.
- This changes, if later iterations of a loop depend on data computed in earlier iterations.
This leads to loop-carried dependencies.
- Because the analysis of loop-level parallelism involves recognizing structures like loop constructs, array references, and expressions over induction variables, compilers perform this analysis on the source level.

Loop-Carried Dependencies

```
for (i=1; i<=100; i=i+1) {  
    A[i+1] = A[i] + C[i];      S1  
    B[i+1] = B[i] + A[i+1];   S2  
}
```

- $S2$ depends on $S1$ in the *same* iteration—if this were the only dependency and the statements are kept in order, iterations may execute in *parallel*.
- The dependency of $A[i+1]$ on $A[i]$ spans *different* iterations (loop-carried). Since $S1$ depends on $S1$ itself, this forces successive iterations of the loop to execute *in series*. Analogously for $S2$.

Resolving a Loop-Carried Dependency

```
for (i=1; i<=100; i=i+1) {  
    A[i]    = A[i] + B[i];    S1  
    B[i+1] = C[i] + D[i];    S2  
}
```

- Loop-carried dependency from $S1$ to $S2$. But *no circular dependencies*.

Try to remove the loop-carried dependency to expose the parallelism in the loop.

Loop-Carried Dependency Resolved

```
A[1] = A[1] + B[1];  
for (i=1; i<=99; i=i+1) {  
    B[i+1] = C[i] + D[i];          S2  
    A[i+1] = A[i+1] + B[i+1];    S1  
}  
B[101] = C[100] + D[100]
```

- The only remaining dependency ($S1$ on $S2$) does not span iterations but merely orders the statements.

Rewriting Dependencies



- Back substitution:

```
DADDUI R1 , R2 , #4  
DADDUI R1 , R1 , #4
```

```
DADDUI R1 , R2 , #8
```

- Tree height reduction:

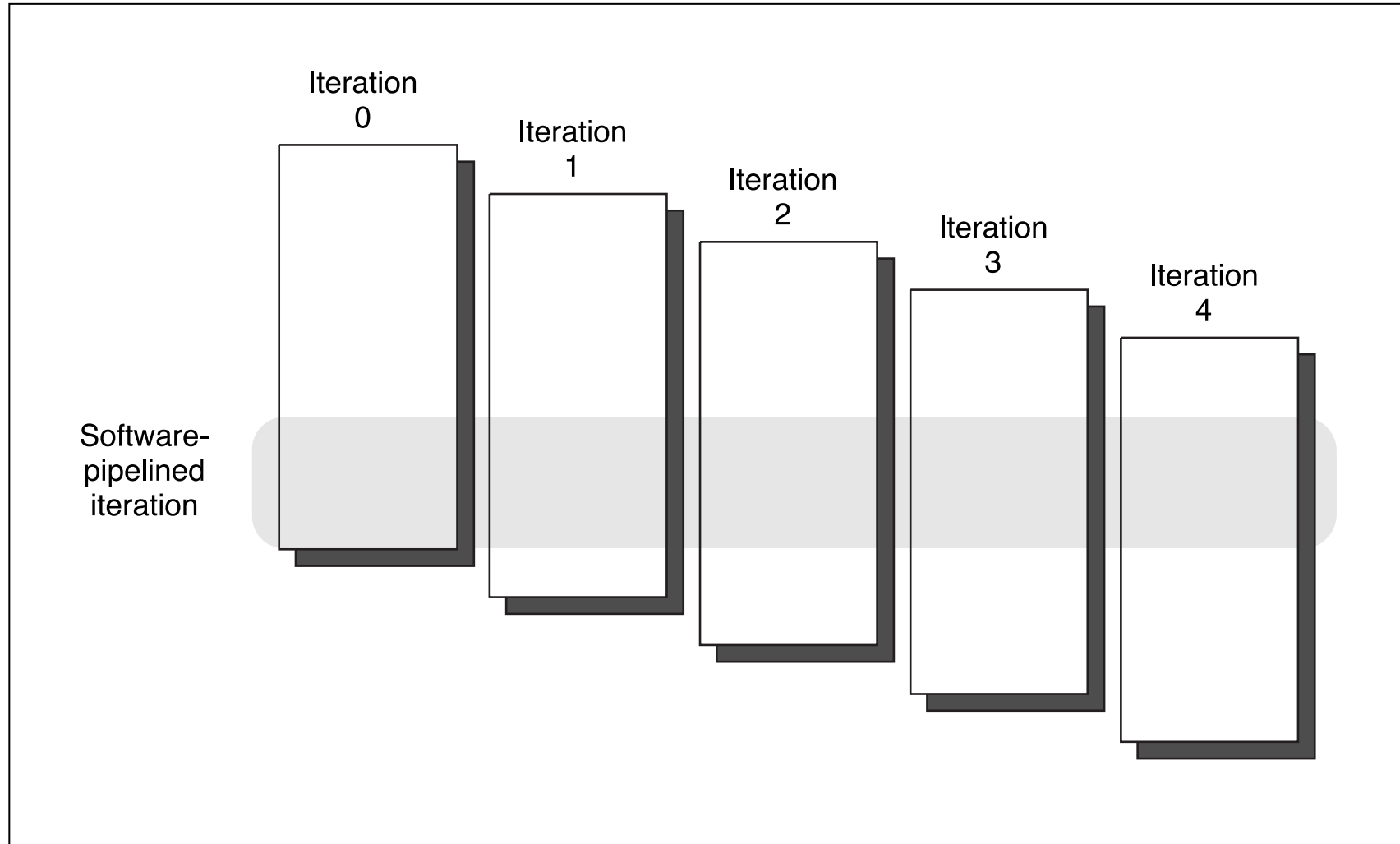
```
ADD R1 , R2 , R3  
ADD R4 , R1 , R6  
ADD R8 , R4 , R7
```

```
ADD R1 , R2 , R3  
ADD R4 , R6 , R7  
ADD R8 , R1 , R4
```

Software Pipelining

- Software pipelining reorganizes loops such that each iteration is made from instructions chosen from *different* iterations.
 - Rationale: Dependent instructions are separated from another by an entire loop body—helps to schedule without stalls.
- Software pipelining *symbolically* unrolls the loop. Code size does *not* increase.
- In general, start-up and finish-up code is needed.

Software Pipelining



Software Pipelining

Original Loop

```
loop: L.D      F0, 0(R1)
      ADD.D   F4, F0, F2
      S.D     F4, 0(R1)
      DADDUI  R1, R1, #-8
      BNE    R1, R2, loop
```

- Pick instructions from each iteration, avoid dependencies.
- Compensate for omitted loop index maintenance.

Symbolic Unrolling

```
L.D      F0, 0(R1)
ADD.D   F4, F0, F2
S.D     F4, 0(R1)
DADDUI  R1, R1, #-8
BNE    R1, R2, loop
L.D      F0, 0(R1)
ADD.D   F4, F0, F2
S.D     F4, 0(R1)
DADDUI  R1, R1, #-8
BNE    R1, R2, loop
L.D      F0, 0(R1)
ADD.D   F4, F0, F2
S.D     F4, 0(R1)
```

Software-Pipelined Loop

Start-up code

```
L.D    F0, 0(R1)
ADD.D  F4, F0, F2
L.D    F0, -8(R1)
DADDUI R1, R1, #-16
```

Software-Pipelined Loop

```
loop:  S.D    F4, 16(R1)
      ADD.D  F4, F0, F2
      L.D    F0, 0(R1)
      DADDUI R1, R1, #-8
      BNE   R1, R2, loop
```

```
loop:  S.D    F4, 16(R1)
      DADDUI R1, R1, #-8
      ADD.D  F4, F0, F2
      BNE   R1, R2, loop
      L.D    F0, 8(R1)
```

```
S.D    F4, 16(R1)
ADD.D  F4, F0, F2
S.D    F4, 8(R1)
```

Finish-up code

Predicated Instructions

- If the behavior of a branch is hard to predict, compiler techniques may be unable to uncover much ILP.
 - This is especially so for data-dependent branches
(`SELECT * FROM R WHERE X.A < c`).
- Modern CPUs offer predicated instructions—that *turn control into data dependencies*—to help address this challenge.

Predicated Instructions

- A predicated instruction i refers to a condition p which is evaluated as part of instruction execution.
 - $p = true$: Execute i normally.
 - $p = false$: Turn i into **NOP**.
- Commonly implemented in terms of conditional move (if zero):

$$p \equiv (R3 = 0)$$

```
CMOVZ R1, R2, R3
```

Control to Data Dependence

- Implement the *absolute value* function $A = \text{abs}(B)$ (assume $B \equiv R1, A \equiv R2$):

```
if (B < 0) A = -B;  
else A = B;
```

Branch:

```
ADDU R2, R0, R1  
BGTZ R1, pos  
SUBU R2, R0, R1  
pos: ...
```

Conditional move:

```
SUBU R2, R0, R1  
SLT R3, R1, R0  
CMOVZ R2, R1, R3
```

- Dependence has moved from front of pipeline (branch, stage ID) to end of pipeline (register write, stage WB).