Static Scheduling

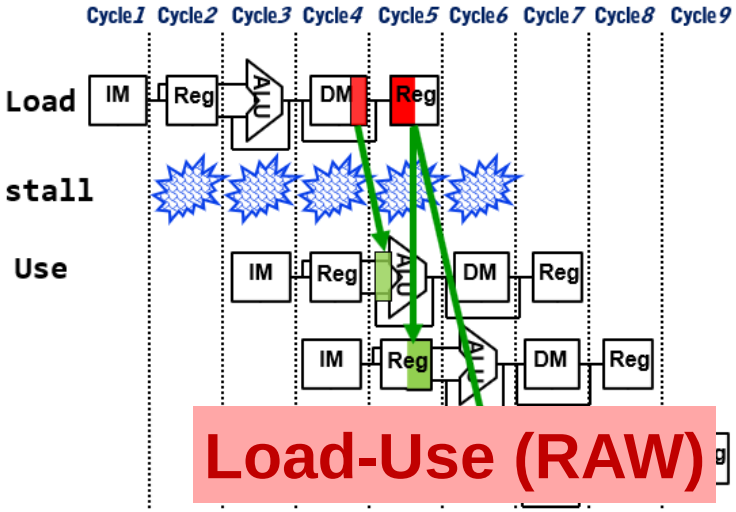
Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory
KAIST EE
CAMELab

Recall: Data-Dependence Stalls

- Previously, we tried to reduce the program execution time with
- But, there are **limits** due to *data-dependency*

Single-issue pipeline (Lecture 5)

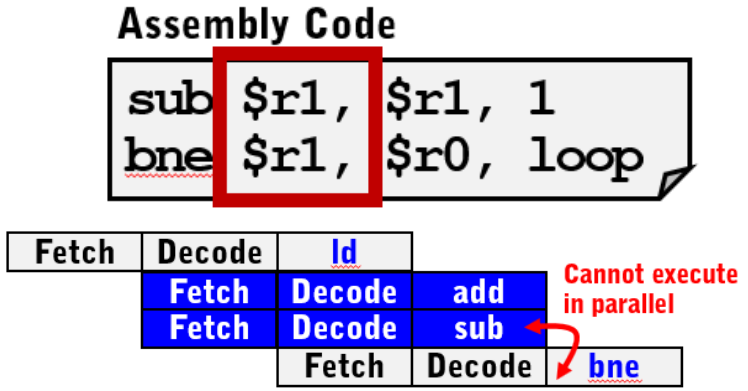


Load-Use (RAW)

What else?

- When no bypassing exists
- Long-latency instructions

Multiple-issue; Superscalar (Lecture 6)

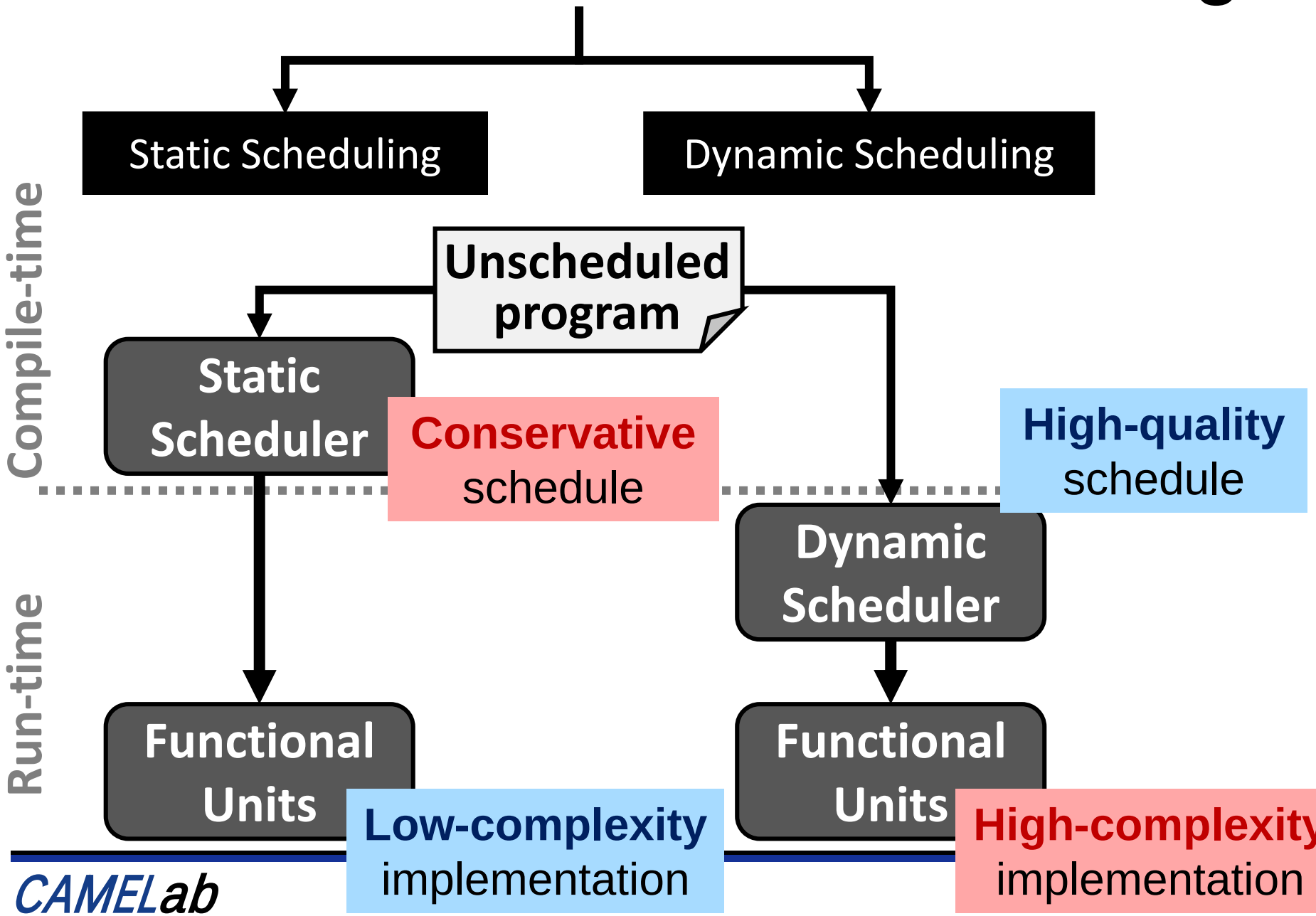


Limits on WAW

What else?

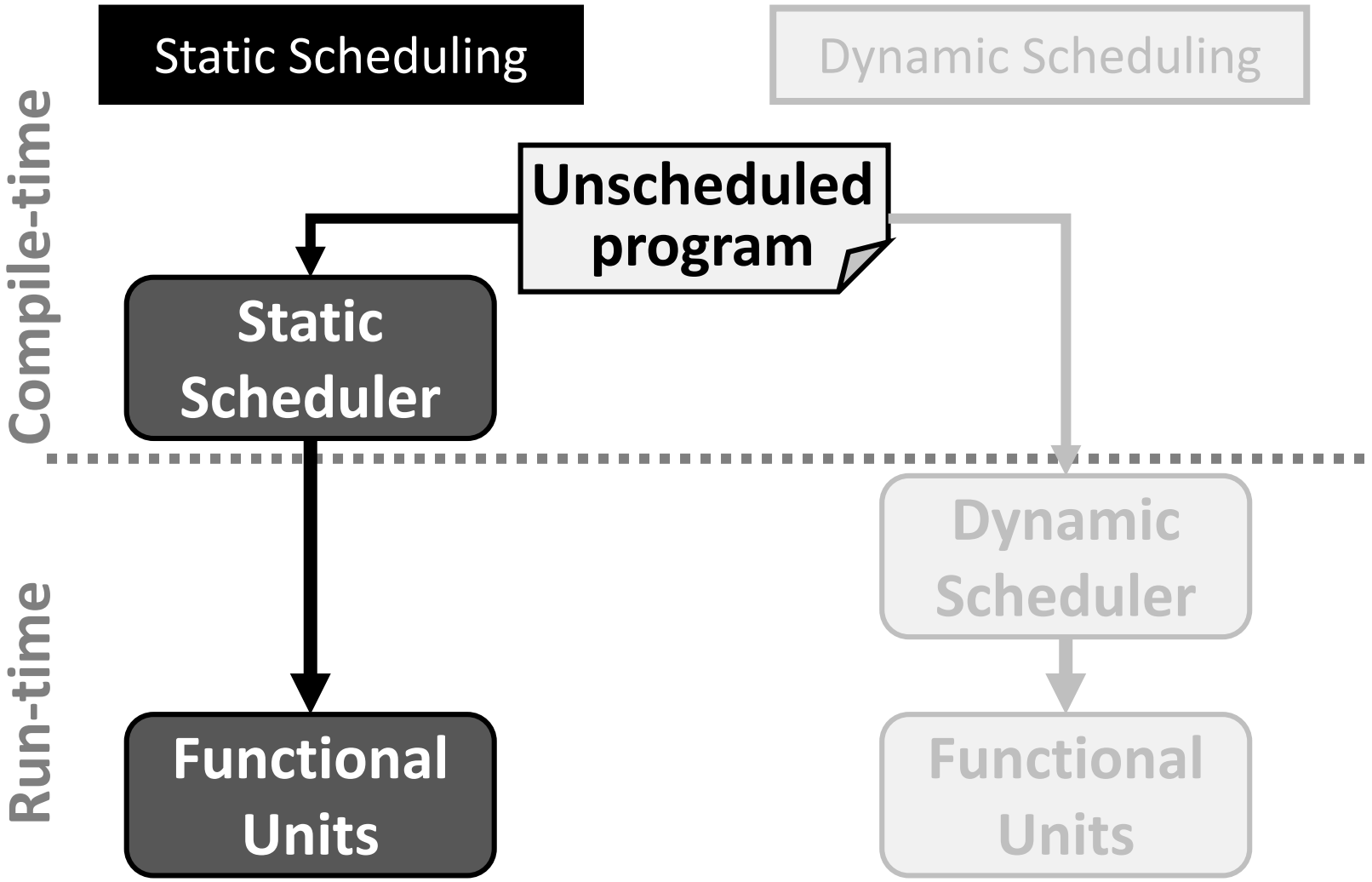
- Instructions executing in a same cycle shouldn't have RAW/WAR

Solution: Instruction Scheduling



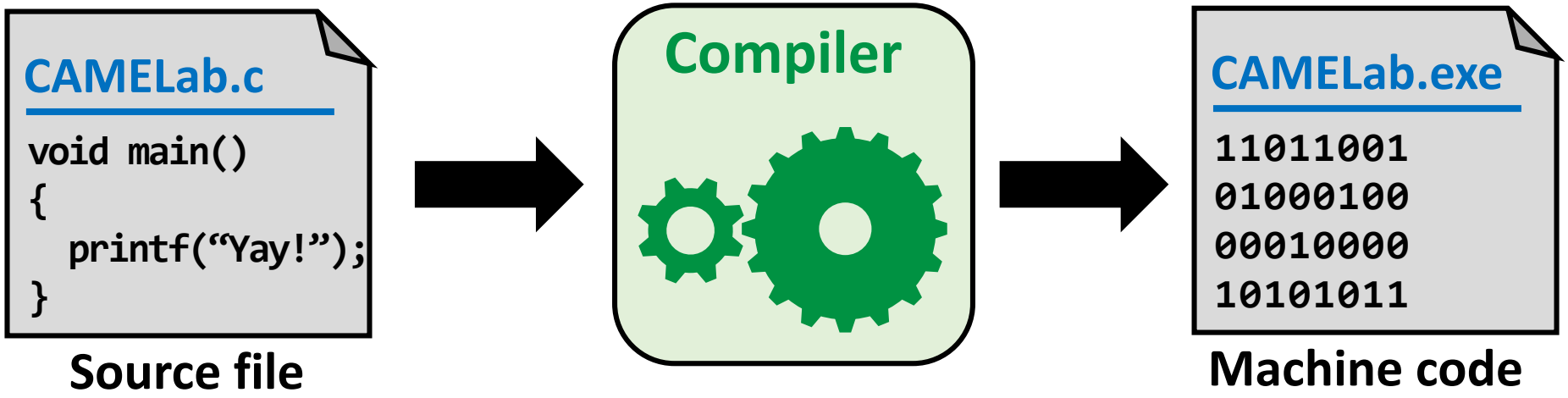
Static Scheduling

- Statically schedule inst. from the **compiler** angle! (for data-dep.)



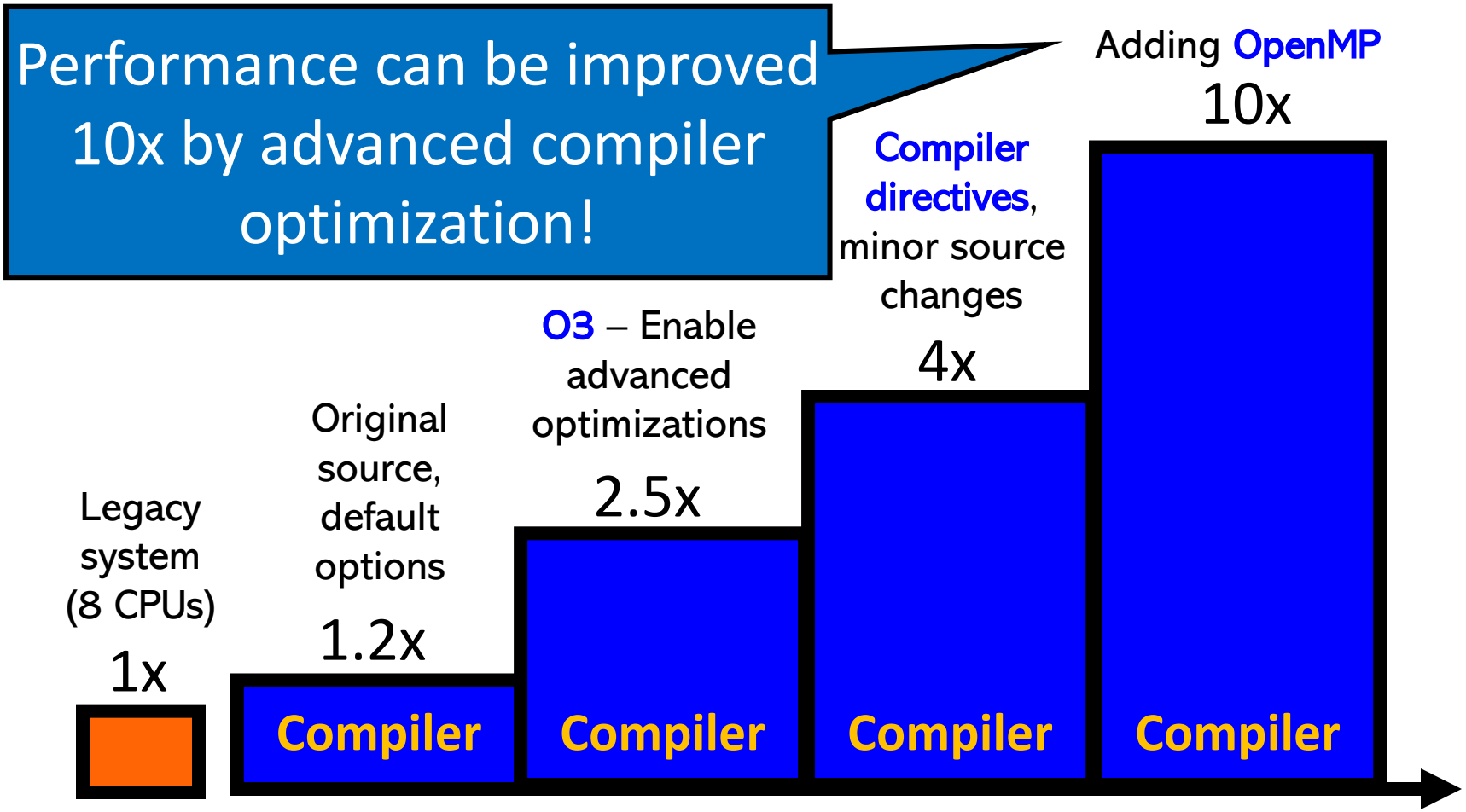
What is Compiler?

- Compiler *translates* a program written in a high-level language into an equivalent program in a target language



Performance Impacts of Compiler

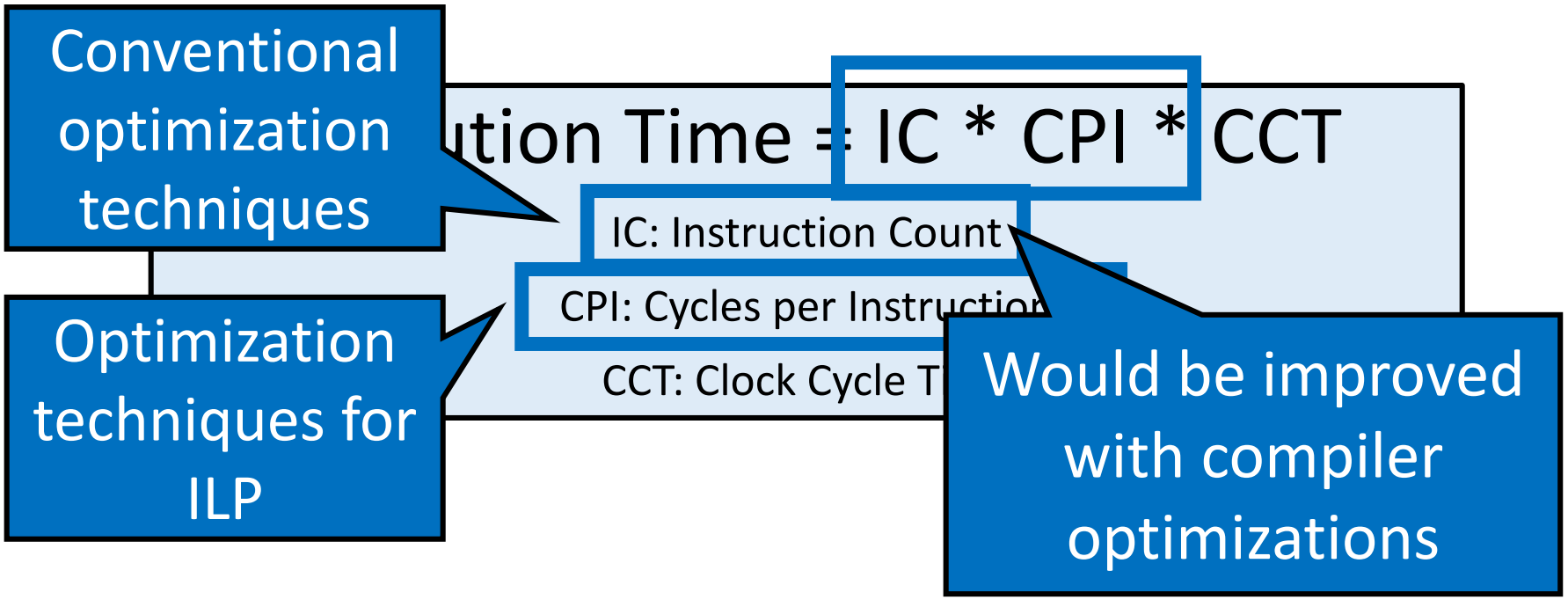
➢ ***Compiler optimizations*** may improve performance significantly



Source: Intel Corp. 2006 (Itanium® 2 4 CPU)

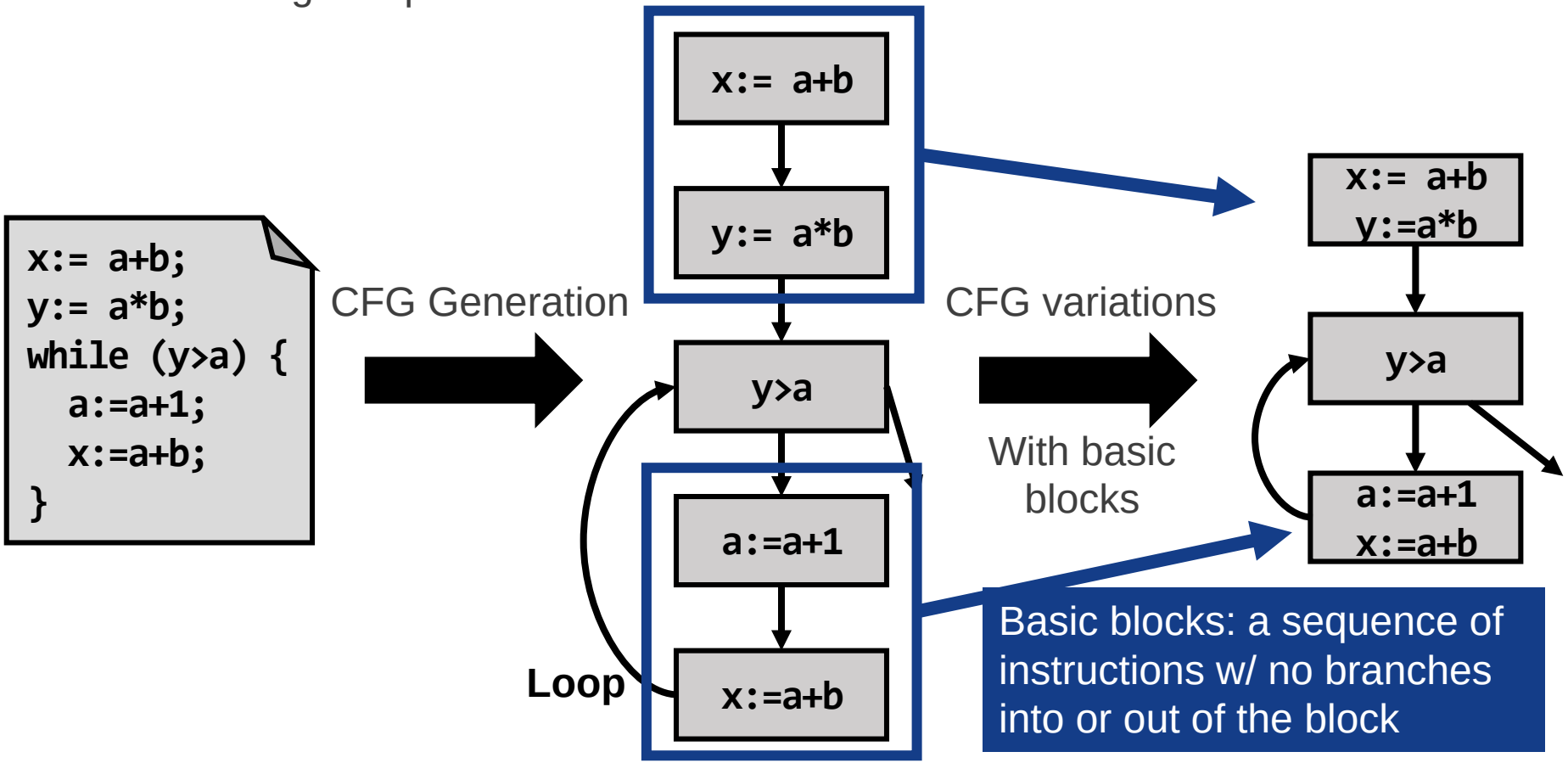
What is Compiler Optimization?

- The most common requirement is to *minimize the time* taken to execute a program with the restriction that the *result must be correct*



Compiler Optimization = Graph Problem!

- The input of optimization process is ***control flow graph (CFG)***
  - A directed graph where
    - Each node represents a statement
    - Edges represent control flow



Simple Loop Example

Simple loop:

```
for(i=1; i<=1000; i++)
 x[i]=x[i] + s;
```

Compilation w/ a vanilla compiler

```
Loop: L.D    F0, 0(R1)
      ADD.D  F4, F0, F2
      S.D    F4 0(R1)
      SUBI   R1, R1,#8
      BNEZ   R1, R2,Loop
```

Execute in machine

```
1.  Loop: LD     F0, 0(R1)
2.        Stall
3.        ADDD   F4,F0,F2
4.        Stall
5.        Stall
6.        SD     F4,0(R1)
7.        SUBI   R1,R1,#8
8.        Stall
9.        BNEZ   R1,R2,Loop
10.       Stall
```

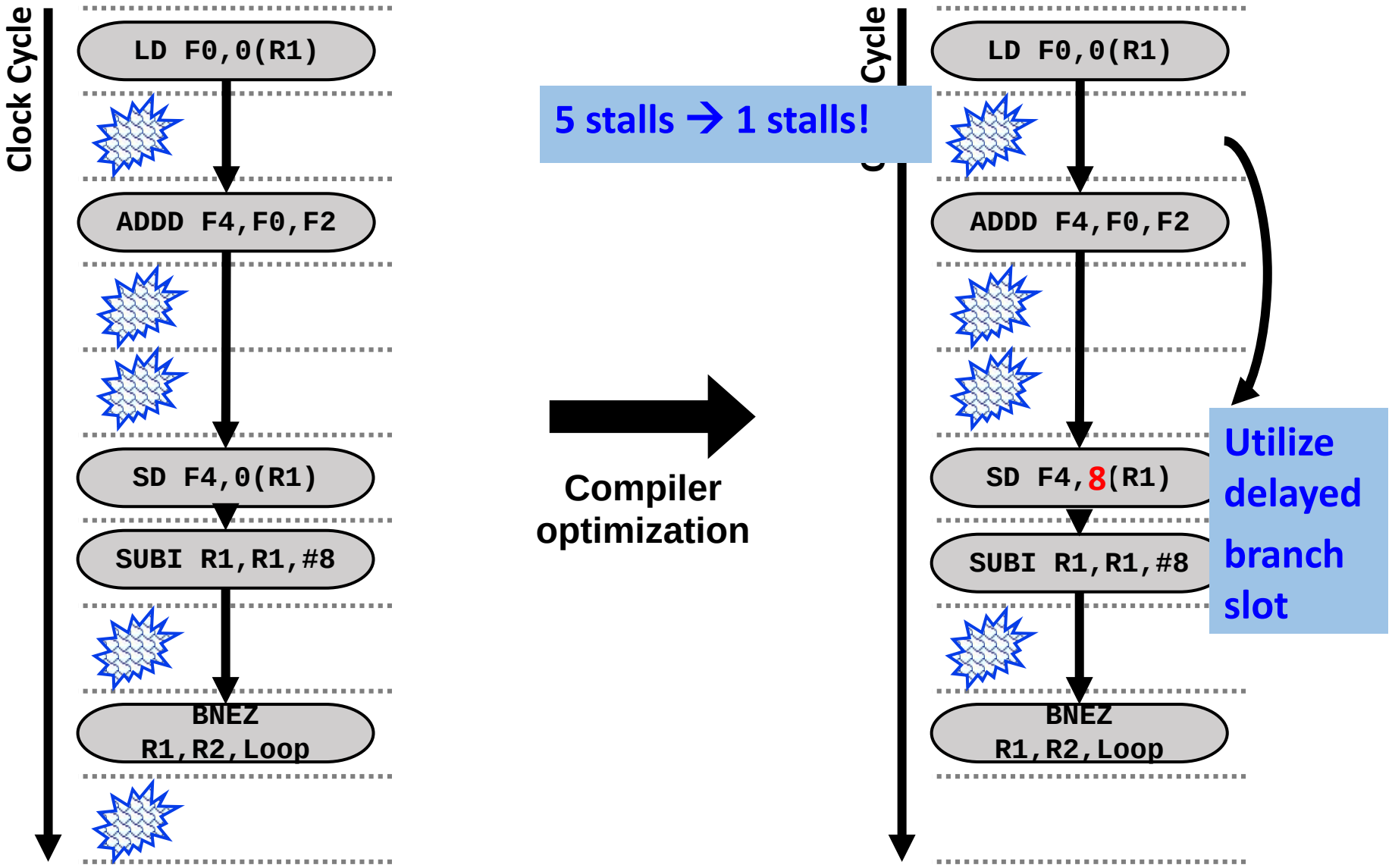
Our machine specification:

Instruction producing result	Instruction using the result	Delay 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
Integer op	Integer op	0
FP ALU op	Branch	1
Branch		1

10 clocks per iteration (5 stalls)

=> Can we rewrite the code to minimize stalls?

Scheduled Loop Body (with CFG)



Goal of Multi-Issue Scheduling

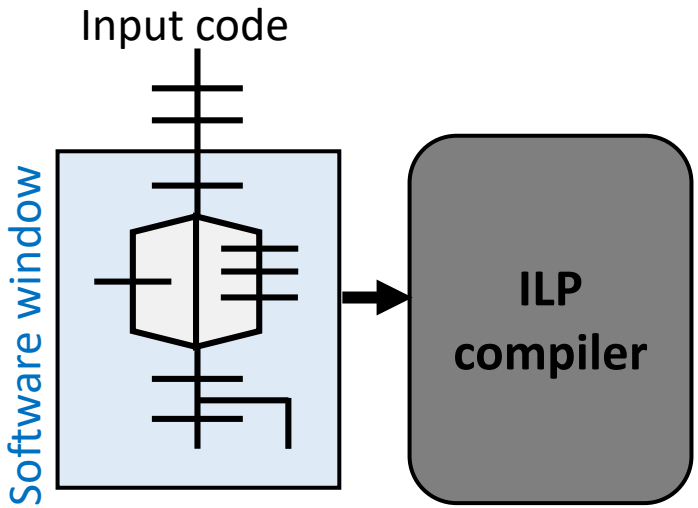
- Place as many *independent* instructions in sequence
  - "as many" → up to execution bandwidth
    - Don't need 7 independent instructions on a 3-wide machine
  - Avoid pipeline stalls

- If compiler is really good, we should be able to get high performance on an in-order superscalar processor
  - In-order superscalar provides execution bandwidth, compiler provides dependence scheduling

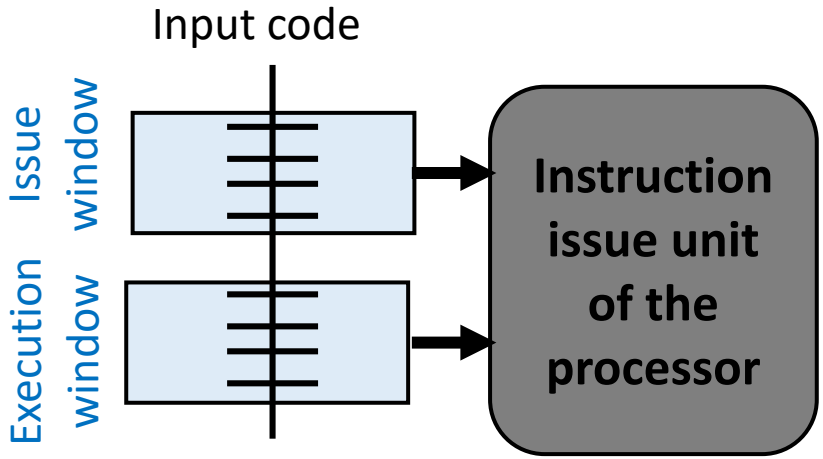
Why this Should work?

- Compiler has “all the time in the world” to analyze instructions
  - Hardware must do it in < 1ns
- Compiler can “see” a lot more
  - Compiler can do complex inter-procedural analysis, understand high-level behavior of code and programming language
  - Hardware can only see a small number of instructions at a time

Static detection & resolution of dependencies (Compiler)



Dynamic detection & resolution of dependencies (HW)



Why might this not work?

- Can't always schedule around branches
  - *limited* access to *dynamic information* (profile-based info)
  - Perhaps none at all, or not representative
  - Ex. Branch T in 1st ½ of program, NT in 2nd ½, looks like 50-50 branch in profile

- *Not all stalls* are predicable
  - Cannot react to dynamic events like data cache misses

Although there are limits of static scheduling (done by compiler), there is still a room to reap the benefits.

Let's check the detail techniques!

Conventional Optimization Techniques

Execution Time = IC * CPI * CCT

IC: Instruction Count

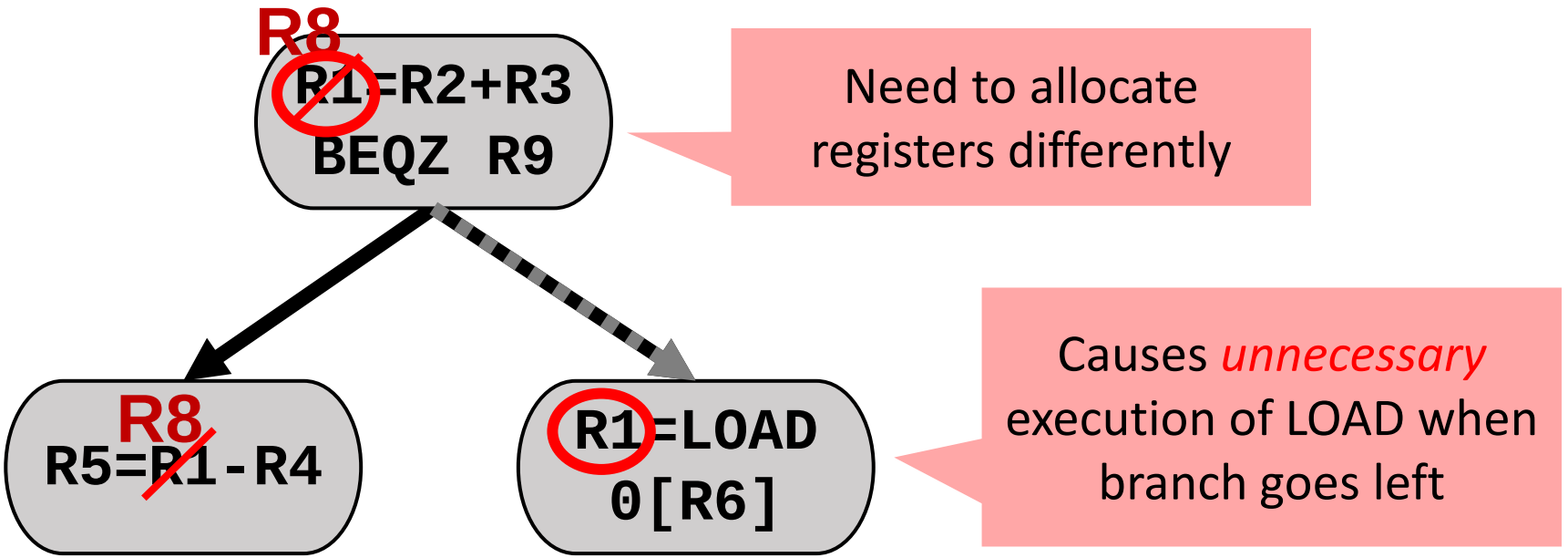
CPI: Cycles per Instruction

CCT: Clock Cycle Time

We mainly focus on Instruction Count!

Technique1: Register Renaming

- Observation1: weird register allocation
  - Largely limited by architected registers
  - Could possibly cause more spills/fills
- Observation2: Dynamic Dead Code (branches)
  - Code motion may be limited



Register Renaming & Scheduling

- Same functionality, no stalls

Schedule to remove stall

No need to same

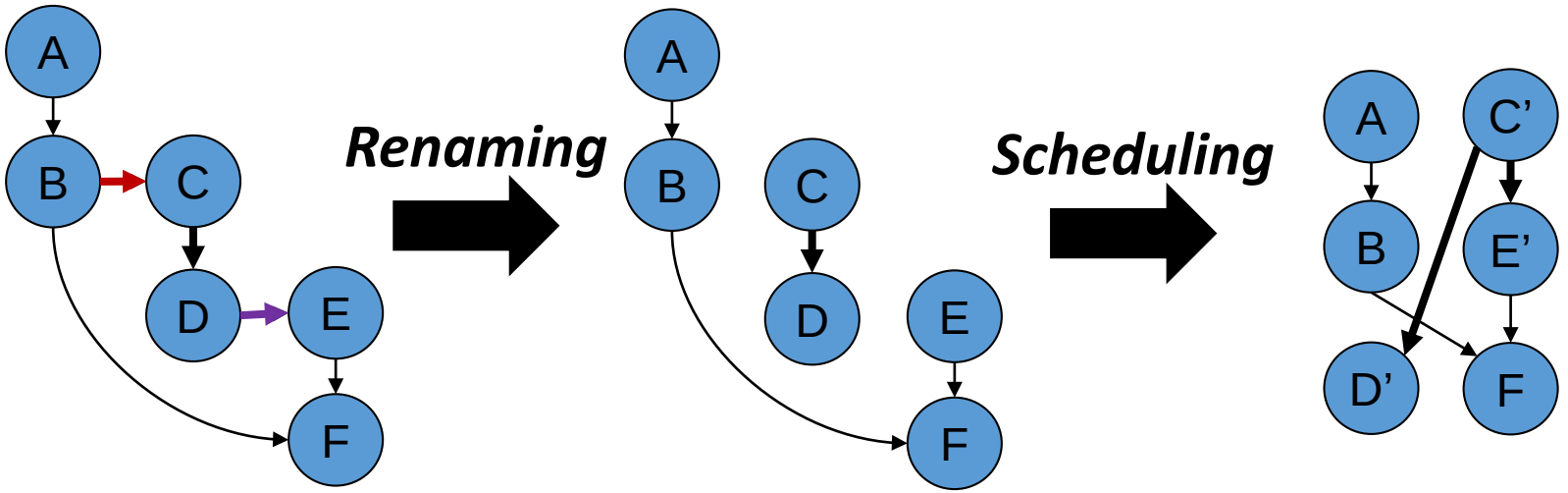
A: R1 = R2 + R3
B: R4 = R1 – R5
C: R1 = LOAD 0[R7]
D: R2 = R1 + R6
E: R6 = R3 + R5
F: R5 = R6 – R4

No need to same

Renaming

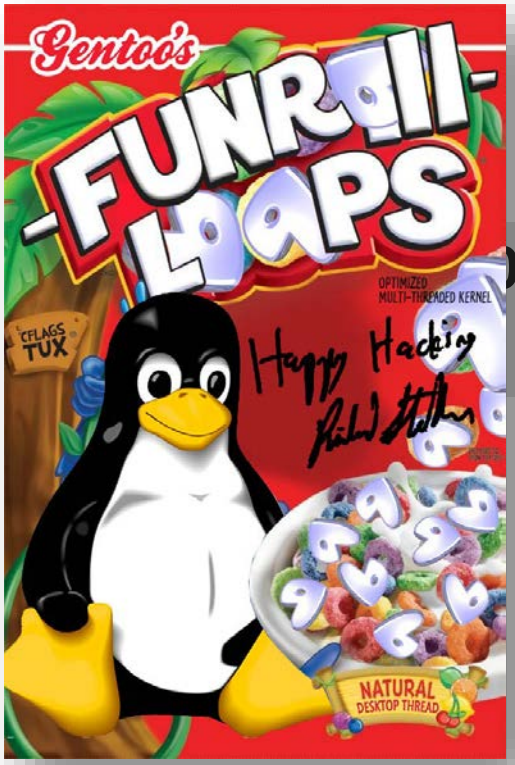
A: R1 = R2 + R3
C': R8 = LOAD 0[R7]
B: R4 = R1 – R5
E': R9 = R3 + R5
D': R2 = R8 + R6
F': R5 = R9 – R4

CFG view



Technique2: Loop Unrolling

- Transforms an M-iterations loop into a loop with M/N iterations
  - We say that the loop has been unrolled N times
- Some compilers can do this (gcc –funroll-loops) or you can do it manually (above)




```
);i++)
```

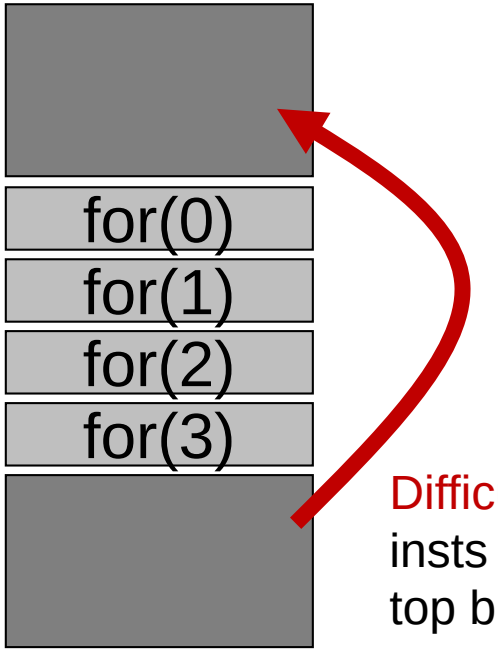
```
for(i=0;i<100;i+=4){
  a[i]*=2;
  a[i+1]*=2;
  a[i+2]*=2;
  a[i+3]*=2;
}
```

Why Loop Unrolling? (1)

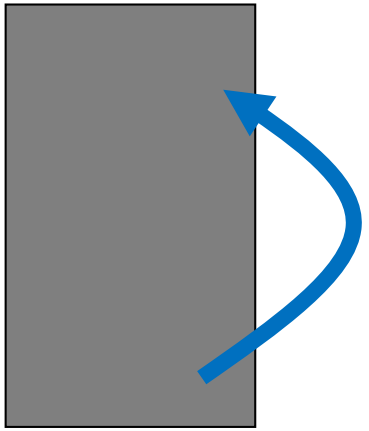
- Get rid of small loops

```
for(i=0;i<4;i++)
  a[i]*=2;
```

```
a[0]*=2;
a[1]*=2;
a[2]*=2;
a[3]*=2;
```



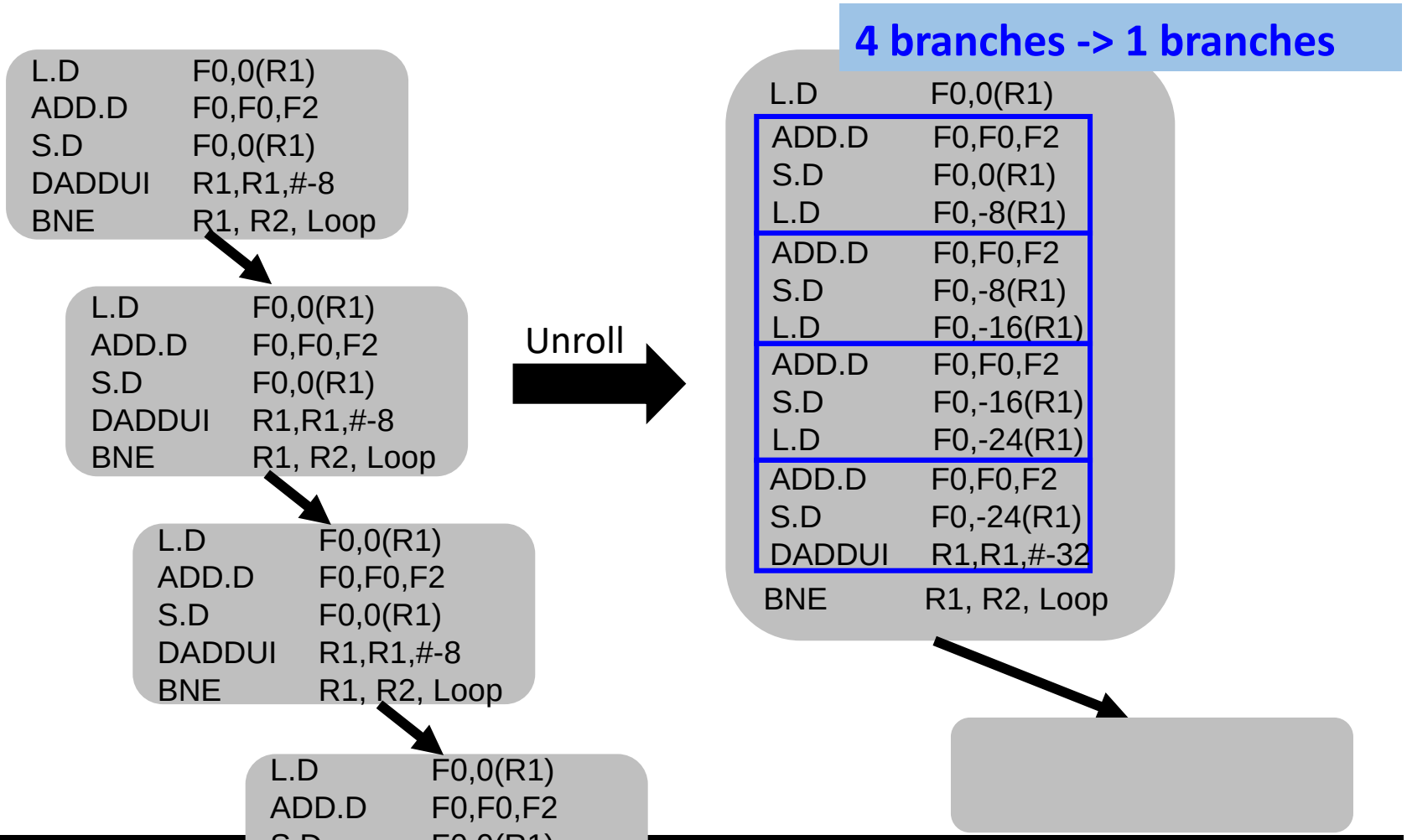
Difficult to schedule/hoist insts from bottom block to top block due to branches



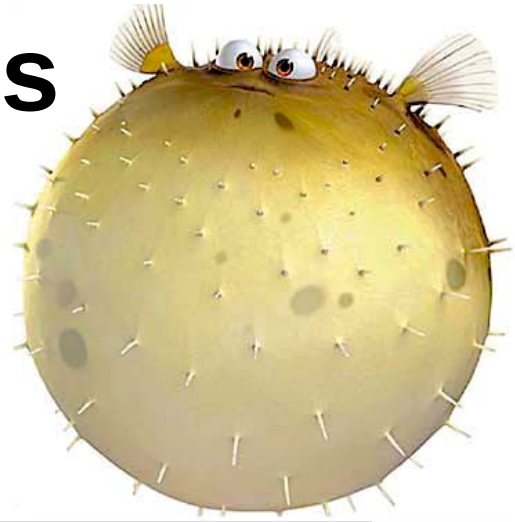
Easier: no branches in the way

Why Loop Unrolling? (2)

- Less loop overhead
- Allow better scheduling of instructions



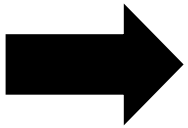
Loop Unrolling: Problems



- Program size is larger (code bloat)

Q1. What if N is not a multiple of M?
Q2. Or What if N is unknown at compiler time?
Q3. Or What if it is a while loop?

```
for(i=0;i<j;i++)
  a[i]*=2;
```



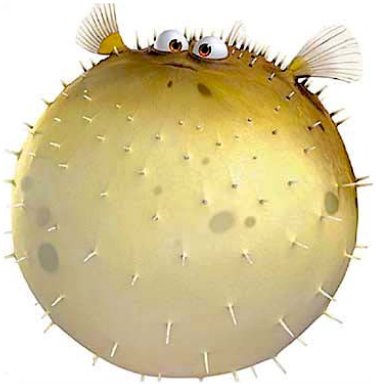
```
j1=j-j%4;
for(i=0;i<j1;i+=4)
{
  a[i]*=2;
  a[i+1]*=2;
  a[i+2]*=2;
  a[i+3]*=2;
}
for(i=j1;i<j;i++)
  a[i]*=2;
```

Unroll until value`i` is multiple of 4

Remained for another for loop

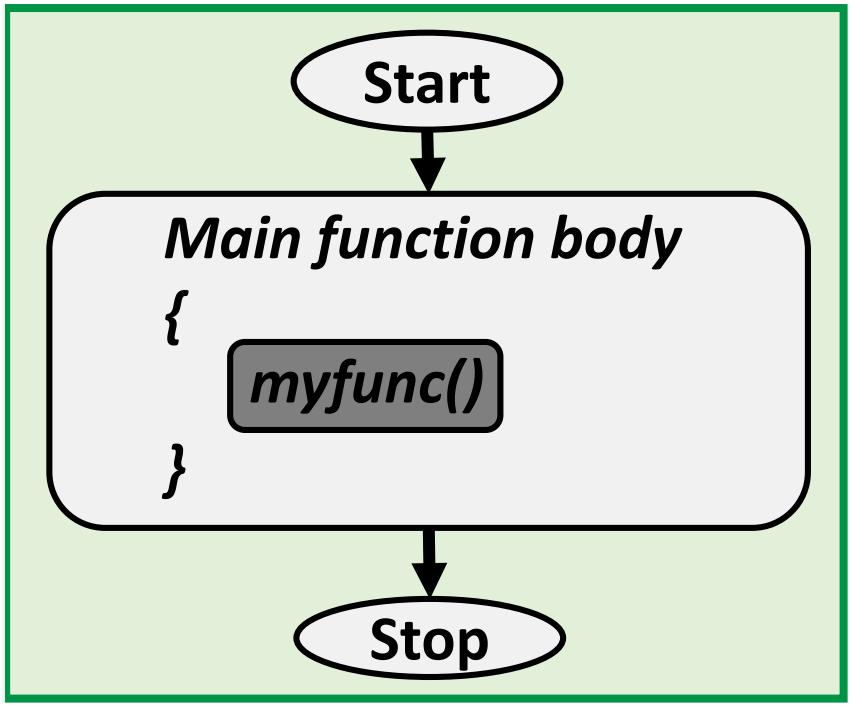
Technique3: Function Inlining

- Goal: sort of like “unrolling” a function
- Problems: primarily code bloat



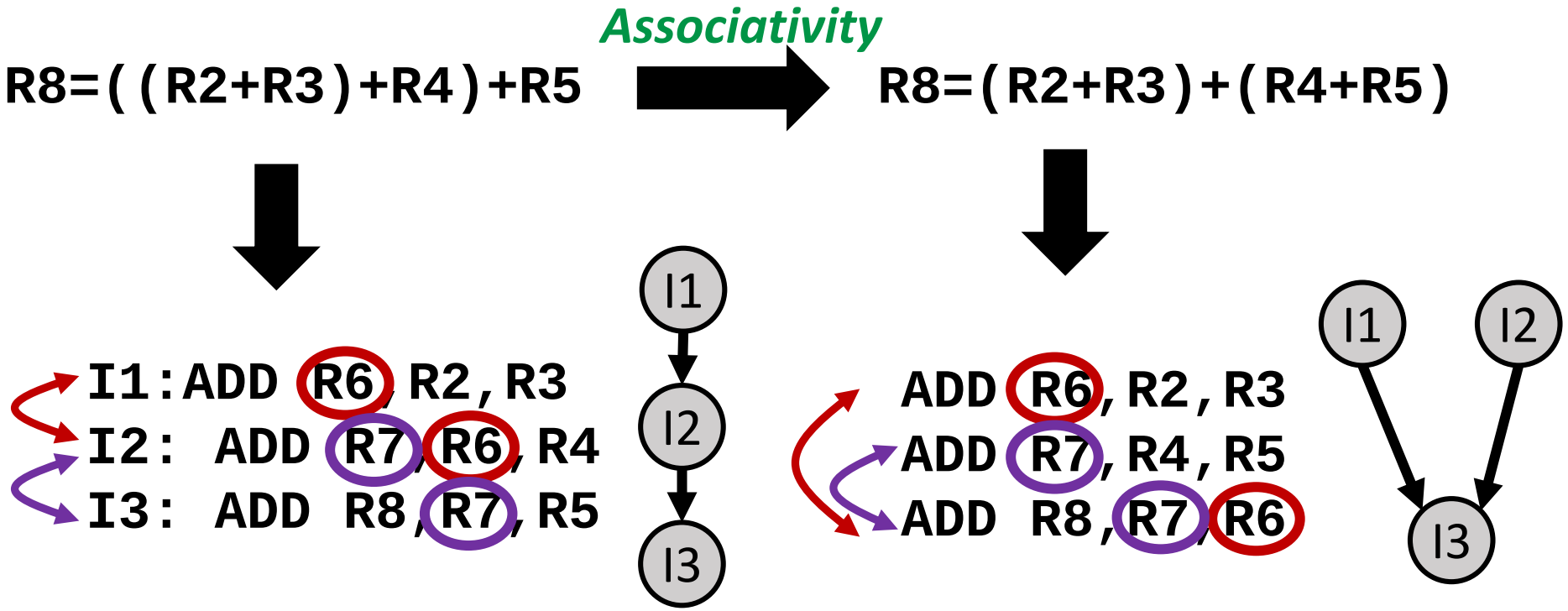
- Remove function call overhead
  - CALL/RETN (and possible branch misprds)
  - Argument/ret-val passing, stack allocation, and associated spills/fills of caller/calle-save regs
- Larger block of instructions for scheduling

Inline Function



Technique4: Tree Height Reduction

- Goal: shorten critical path(s) using associativity law
- Limitations: not all math operations are associative!
- C defines L-to-R semantics for most arithmetic



Optimization Techniques for ILP

Execution Time = IC * CPI * CCT

IC: Instruction Count

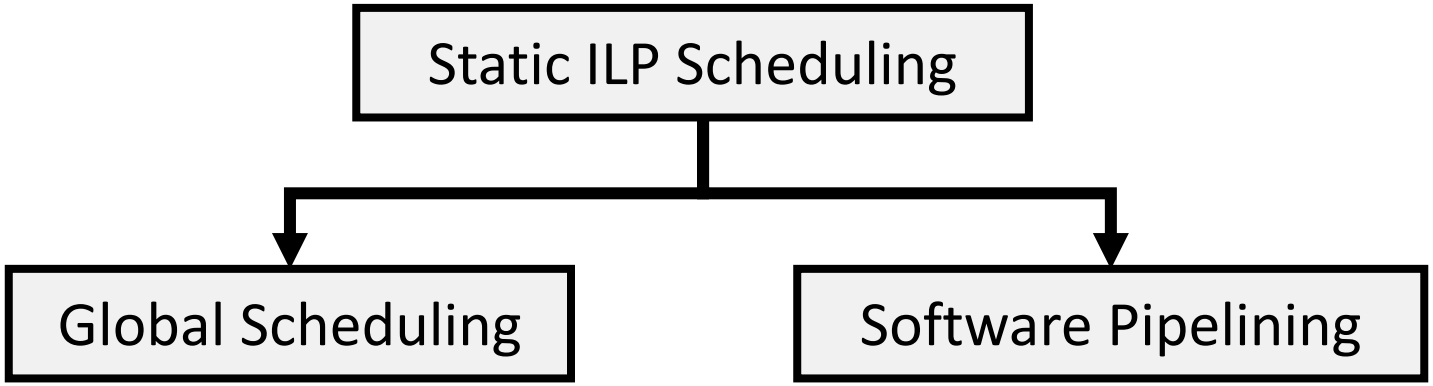
CPI: Cycles per Instruction

CCT: Clock Cycle Time

We mainly focus on Cycles/Instruction!

Scheduling for Inst-Level Parallelism

- Goal: Schedule instructions to finish whole program as soon as possible
- Two types of static ILP scheduling



Target: DAG (Directed Acyclic Graph) of a general purpose integer program with many conditional branches

Target: loop in any code

Technique1: Global Scheduling

Q. What is the general size of basic block?

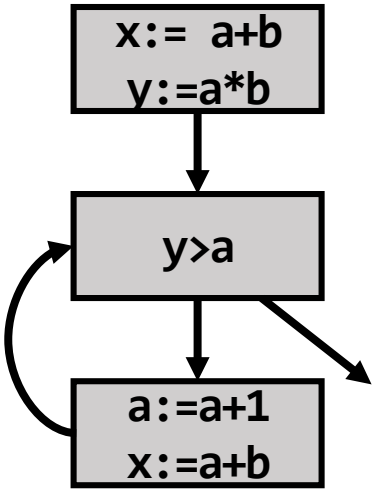
A. In general, the basic block size of non-numeric computation program is 5~20 instructions

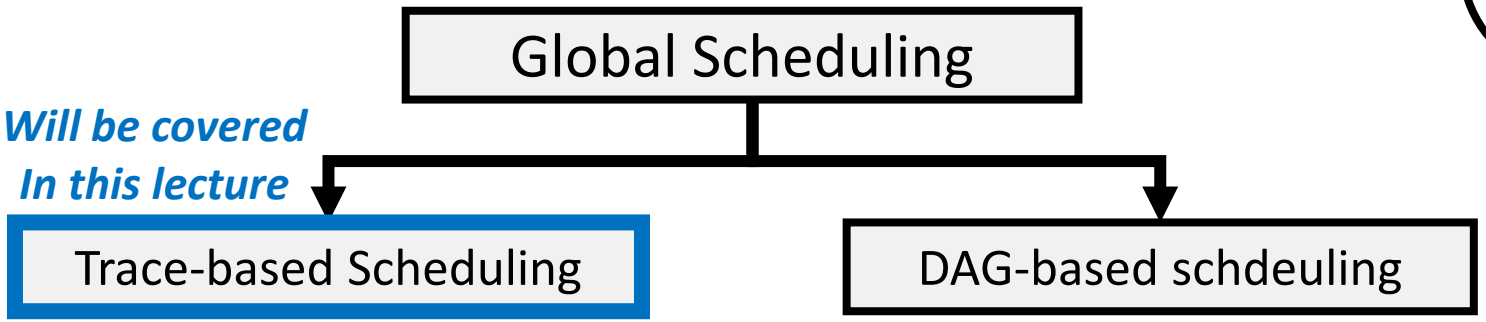
➔ No good enough # of instructions which can be processed in parallel

[REMIND] Basic blocks: a sequence of instructions w/ no branches into or out of the block

Q. Why global scheduling?

A. As the basic block size is too small to find out independent instructions, let's schedule the instructions that exist across all other basic blocks (e.g., in global)





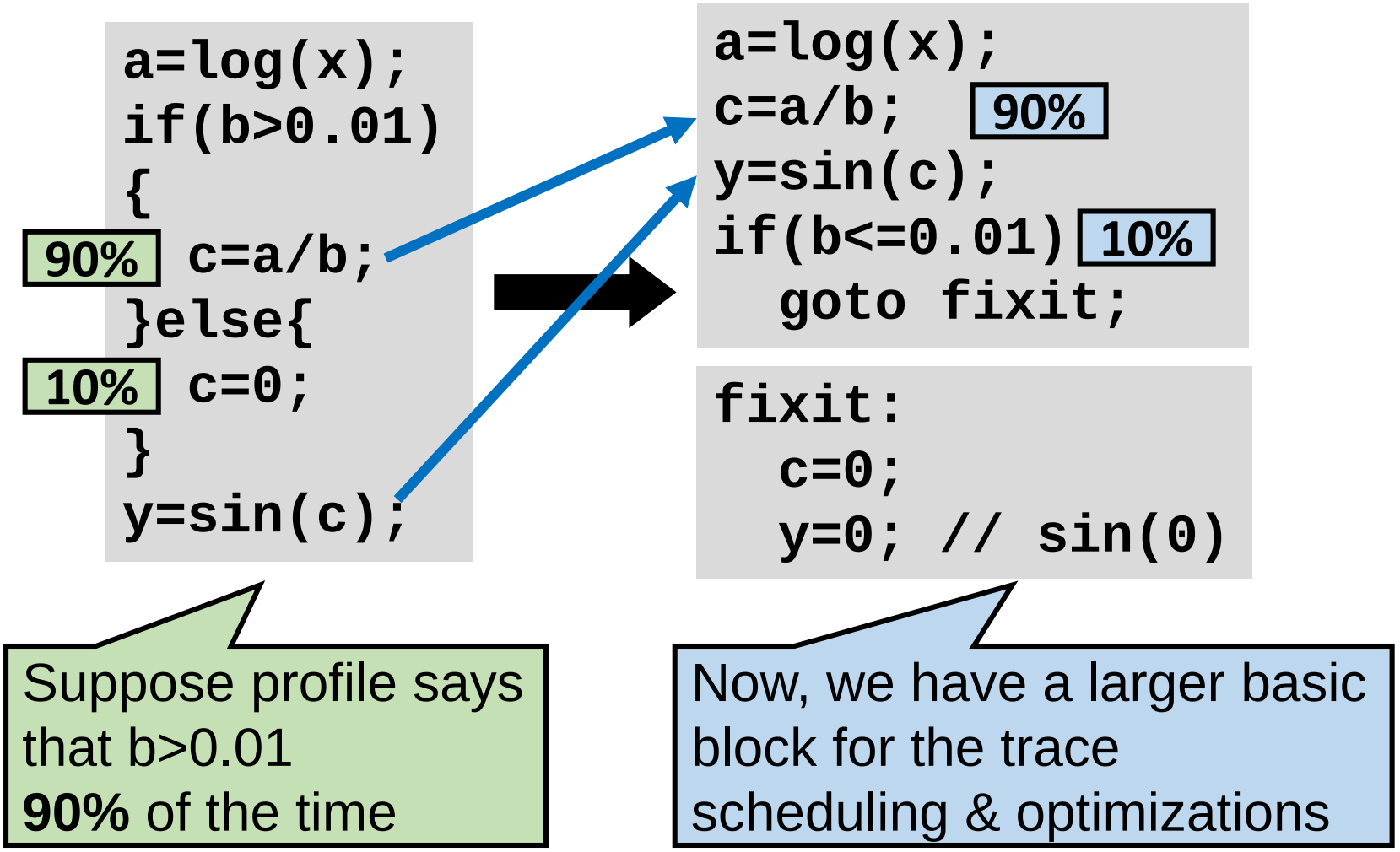
Trace-based Scheduling

- This is one technique for *global scheduling*
  - Works on all code, not just loops
  - Take an execution trace of the common case
  - Schedule code as if it had no branches
  - Check branch condition when convenient
  - If mispredicted, clean up the mess

Q. How do we find the “common case”

A. Program analysis or profiling

Example of Trace Scheduling



Pay Attention to Cost of Fixing

➢ [REMIND] Amdahl's law

$$\frac{1}{(1-P)+\frac{P}{S}}$$

= System performance
P: Fraction of enhanced component
S: Speedup of enhanced component

```
a=log(x);
c=a/b;
y=sin(c);
if(b<=0.01)
  goto fixit;
fixit:
  c=0;
  y=0; // sin(0)
```

- Assume the code for b > 0.01 accounts for **80%** of the *time*
- Optimized trace runs **15%** faster

- But, fix-up code may cause the remaining 20% of the time to be even slower!
- Assume fixup code is **30%** slower

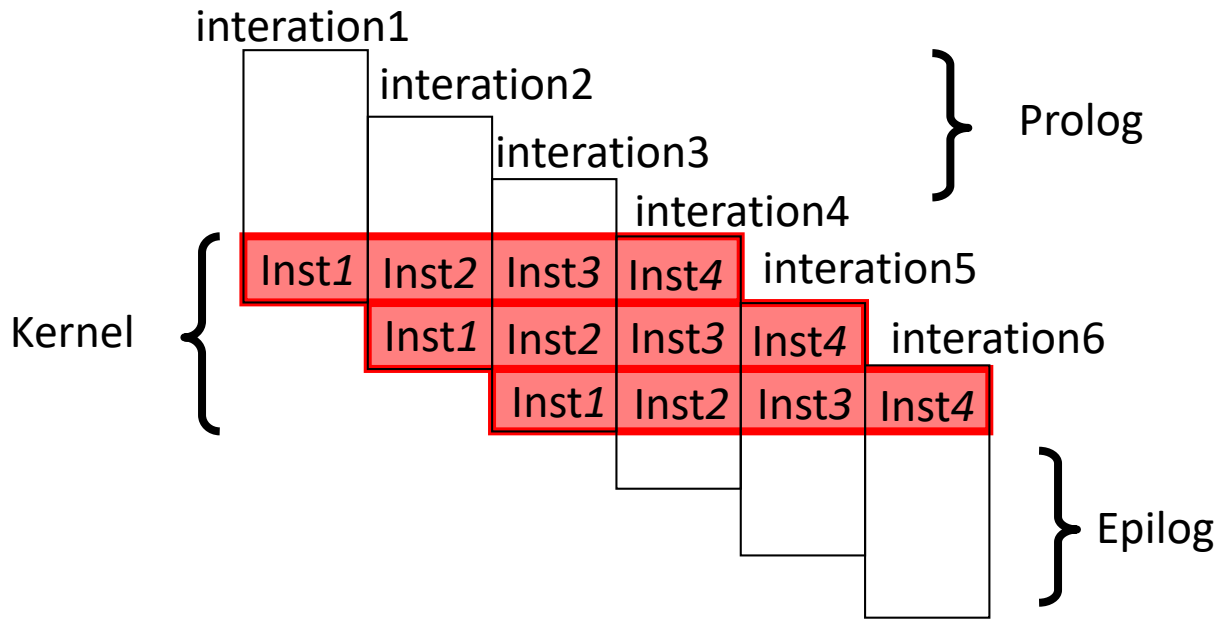
$$\frac{1}{(1-0.8)+\frac{0.8}{0.85}} = 1.176$$ **17.6%**

$$\frac{1}{(1-0.8)*1.3+\frac{0.8}{0.85}} = 1.10$$ **11%**

Over 1/3 of the benefit removed

Technique2: Software Pipelining

- Design: overlapping different iterations by starting the next iteration before the current iteration in the loop ends
  - Consider a graphical view of the overlay of iterations:



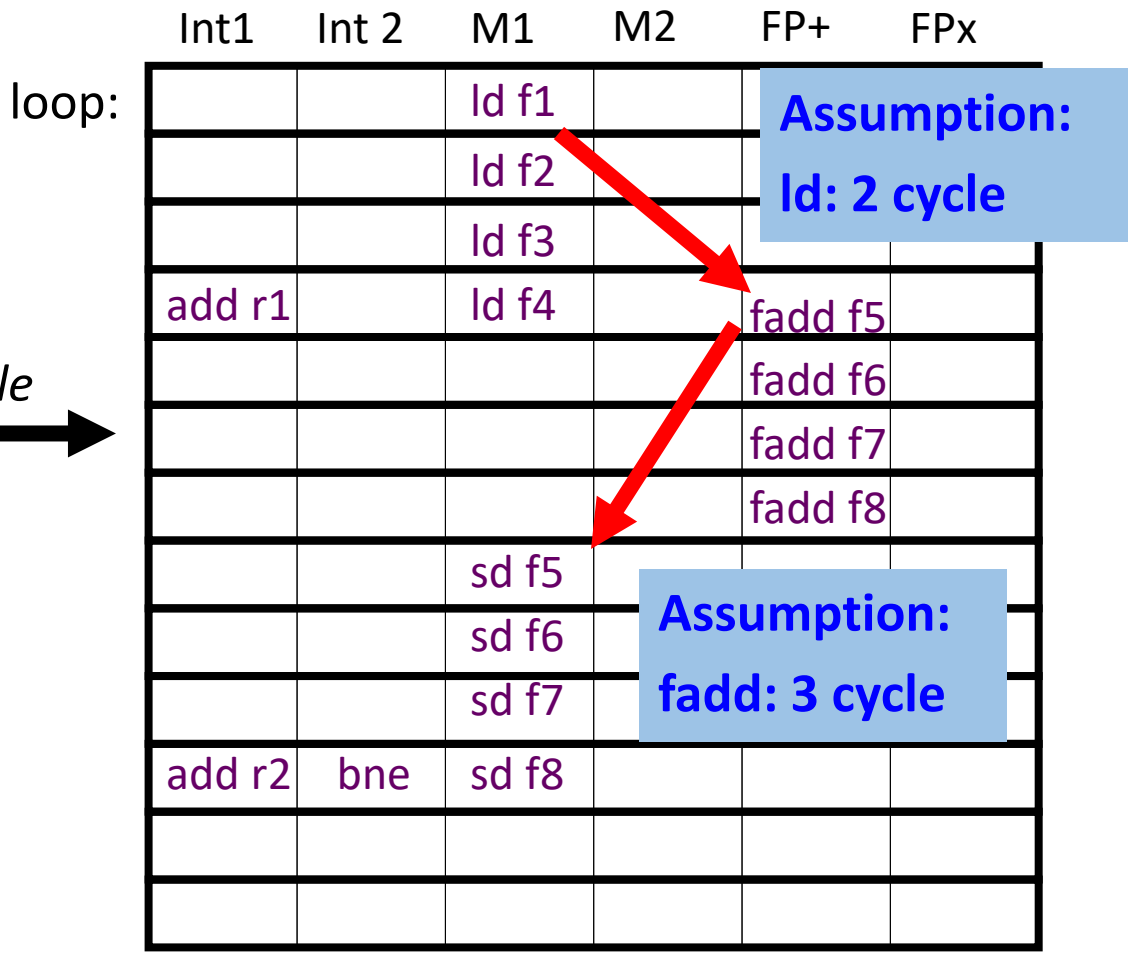
  - Only the shaded part, the loop kernel, involves executing the full width of the VLIW instruction.
    - The loop prolog and epilog contain only a subset of the instructions.
      - "ramp up" and "ramp down" of the parallelism.

Scheduling Loop Unrolled Code

Unroll 4 ways

```
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
```

Schedule



Software Pipelining

Unroll 4 ways first

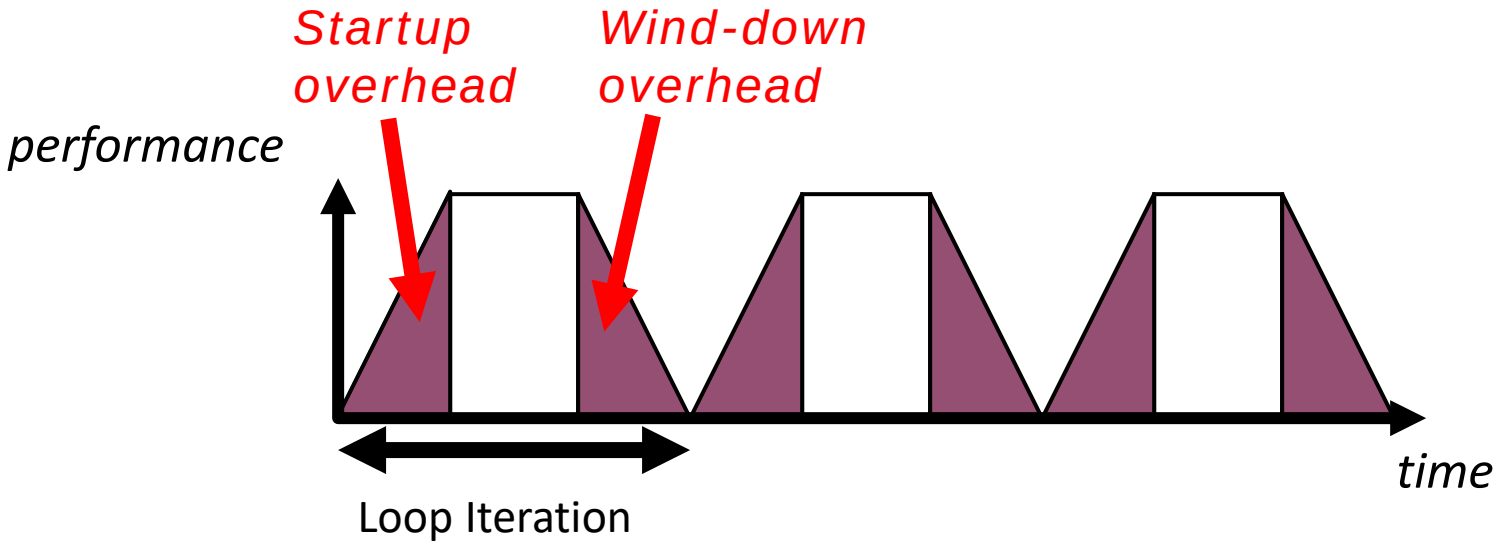
```
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
```

Schedule

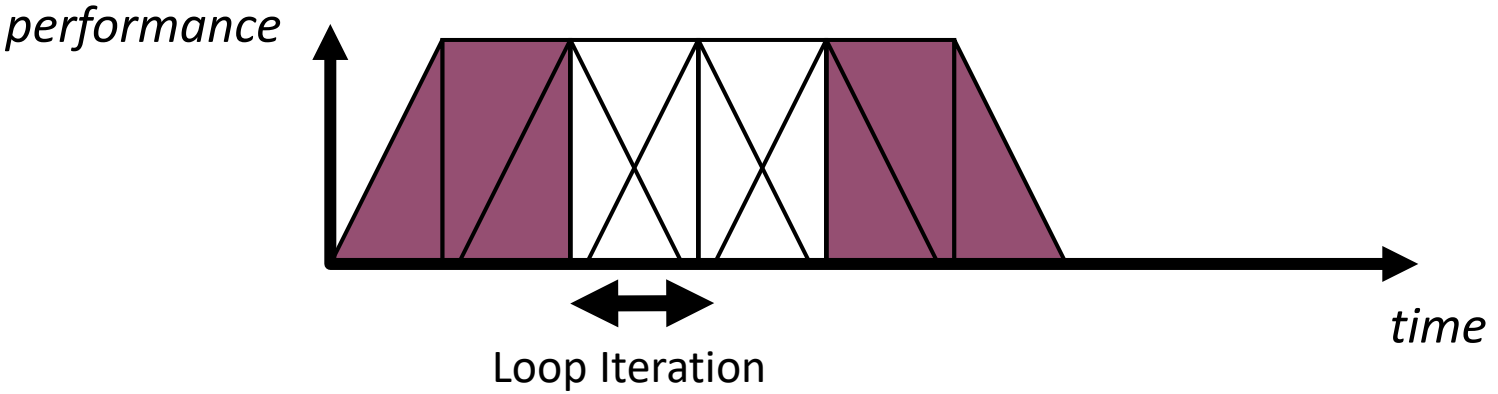
	Int1	Int 2	M1	M2	FP+	FPx
loop:			ld f1			
			ld f2			
			ld f3			
	add r1		ld f4			
			ld f1		fadd f5	
			ld f2		fadd f6	
			ld f3		fadd f7	
	add r1		ld f4		fadd f8	
			ld f1	sd f5	fadd f5	
			ld f2	sd f6	fadd f6	
		add r2	ld f3	sd f7	fadd f7	
		bne	ld f4	sd f8	fadd f8	
				sd f5	fadd f5	
				sd f6	fadd f6	
		add r2		sd f7	fadd f7	
		bne		sd f8	fadd f8	
				sd f5		

Loop Unrolling vs. Software Pipelining

Loop Unrolling



Software Pipelining



Software pipelining pays startup/wind-down costs only once per loop, not once per iteration

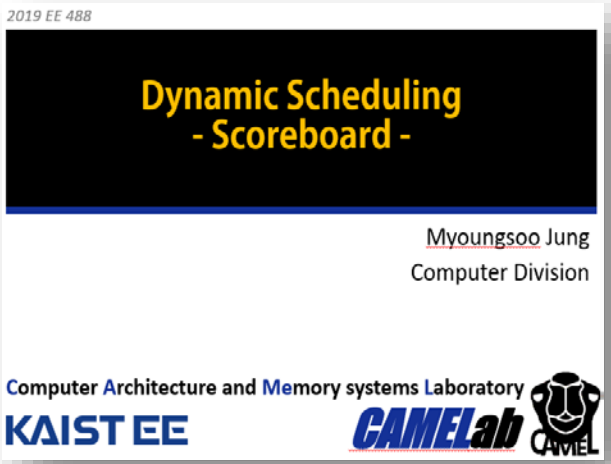
Recall: Why Compiler Might not Work

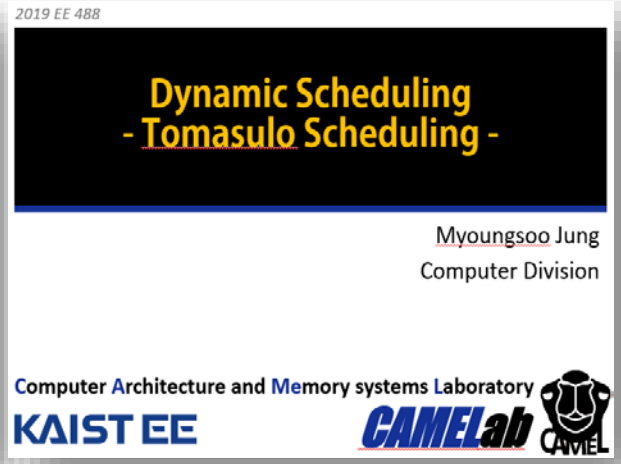
- Can't always schedule around branches
  - *limited* access to *dynamic information* (profile-based info)
  - Perhaps none at all, or not representative
  - Ex. Branch T in 1st ½ of program, NT in 2nd ½, looks like 50-50 branch in profile
- *Not all stalls* are predicable
  - Cannot react to dynamic events like data cache misses

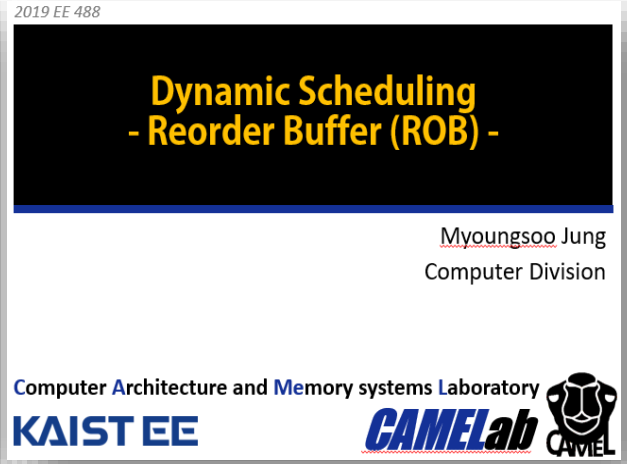
Solution: Dynamic Scheduling

➢ Solve data-dependency dynamically with **hardware**

- **Scoreboarding (Lecture8)**
- Tomasulo scheduling (Lecture 9)
- Reorder buffer (Lecture 10)







Static Scheduling

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory
KAIST EE
CAMELab