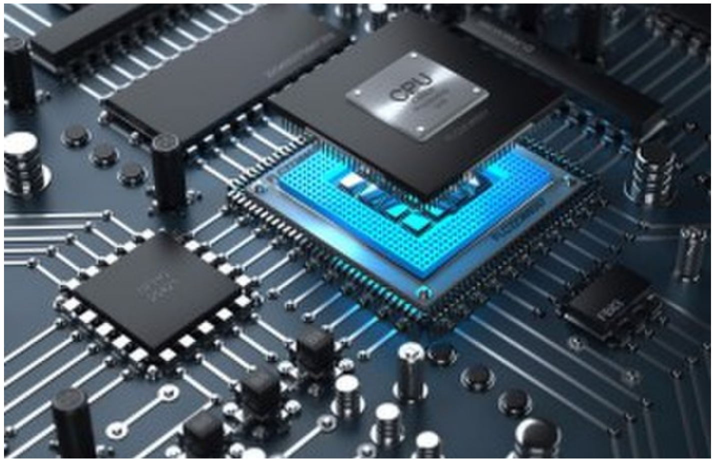


Instructions and Programs

CS 154: Computer Architecture
Lecture #7
Winter 2020

Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB

Administrative

- I got nada

Lecture Outline

- Branch and Jump Addressing
- Parallelism and Synchronization
- Going from File to Machine Code
- Relative Performance Comparisons

Branch Addressing

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

I-Type of instruction (beq , bne)

- Branch instructions specify:
  Opcode + 2 registers + target address
- Most branch targets are *near* the branch instruction in the *text* segment of memory
  - Either ahead or behind it
- Addressing can be done relative to the value in PC Reg. ("PC-Relative Addressing")
  - Target address = PC + offset (in words) x 4
  - **PC is already incremented by 4 by this time**

Branching Far Away

If branch target is too far to encode with 16-bit offset, then assembler will rewrite the code

- Example

```
        beq $s0, $s1, L1     # L1 is far away
                  ⬇
        bne $s0, $s1, L2     # rewritten…
        j L1
    L2: ...
```

Jump Addressing

op	address
6 bits	26 bits

J-Type of instruction **(`j , jal`)**

- Jump (j and jal) targets could be anywhere in *text* segment
- Encode full address in instruction
- Direct jump addressing
  - Target address = (address x 4) **OR** (PC[31: 28])
  - i.e. Take the **4** most sig. bits in PC
    and concatenate the **26** bits in “address” field
    and then concatenate another **00** (i.e x 4)

Target Addressing Example

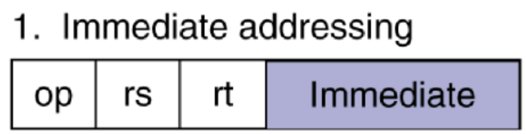
- Assume Loop is at location 80000

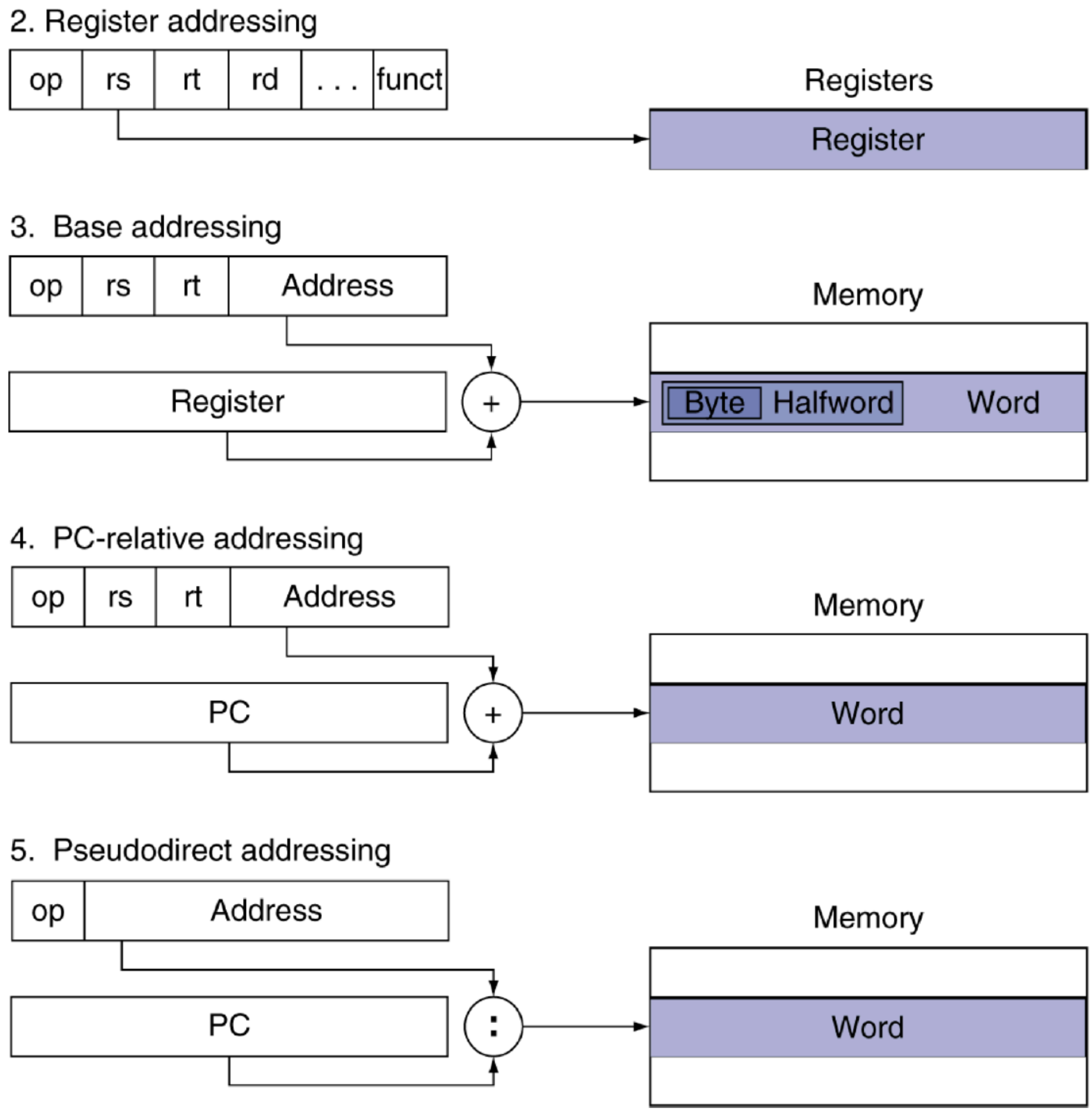
`Loop:`	`sll $t1, $s3, 2`	80000	0	0	19	9	4	0
	`add $t1, $t1, $s6`	80004	0	9	22	9	0	32
	`lw $t0, 0($t1)`	80008	35	9	8	0		
	`bne $t0, $s5, Exit`	80012	5	8	21	2		
	`addi $s3, $s3, 1`	80016	8	19	19	1		
	`j Loop`	80020	2	20000				
`Exit:`	…	80024						

Addressing Mode Summary

Examples:

```
addi $t0, $t0, 42

add $t0, $t1, $t3

lw $t0, 4($t1)

beq $t0, $t1, L1

j L1
```





Parallelism and Synchronization

- Consider: 2 processors sharing an area of memory
  - P1 writes, then P2 reads
- There may be a "data race" if P1 and P2 don't synchronize
  - Result depends of order of accesses
- Hardware support required
  - *"Atomic"* read/write memory operation,
    i.e. no other mem. access allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions (like `ll` & `sc`)

Synchronization in MIPS

- Load link: **`ll rt, offset(rs)`**
- Store conditional: **`sc rt, offset(rs)`**
  - Succeeds if location not changed since the **`ll`**: Returns 1 in **`rt`**
  - Fails if location is changed: Returns 0 in **`rt`**
- **`ll`** returns the current value of a memory location
- A subsequent **`sc`** to the same memory location will store a new value there ***only if*** no updates have occurred to that location since the **`ll`**.

Going From File to Machine Code

- There are 4 steps in transforming a program in a file into a program running on a computer

1. Compiler
    - Takes a program in a HLL and **translates to assembly language**
    - Some compilers have assemblers & linkers built-in

2. Assembler
    - Takes care of pseudoinstructions, number conversions (to hex)
    - **Produces an *object file*** (a combination of machine language instructions, data, and information needed to place instructions properly in memory)
    - This has to determine the addresses corresponding to all labels

Producing an Object Module

- **Header**: described contents of object module
- **Text segment**: translated instructions
- **Static data segment**: data allocated for the life of the program
- **Relocation info**: for contents that depend on absolute location of loaded program
- **Symbol table**: global definitions and external refs
- **Debug info**: for associating with source code

This may not have all the references/labels resolved yet

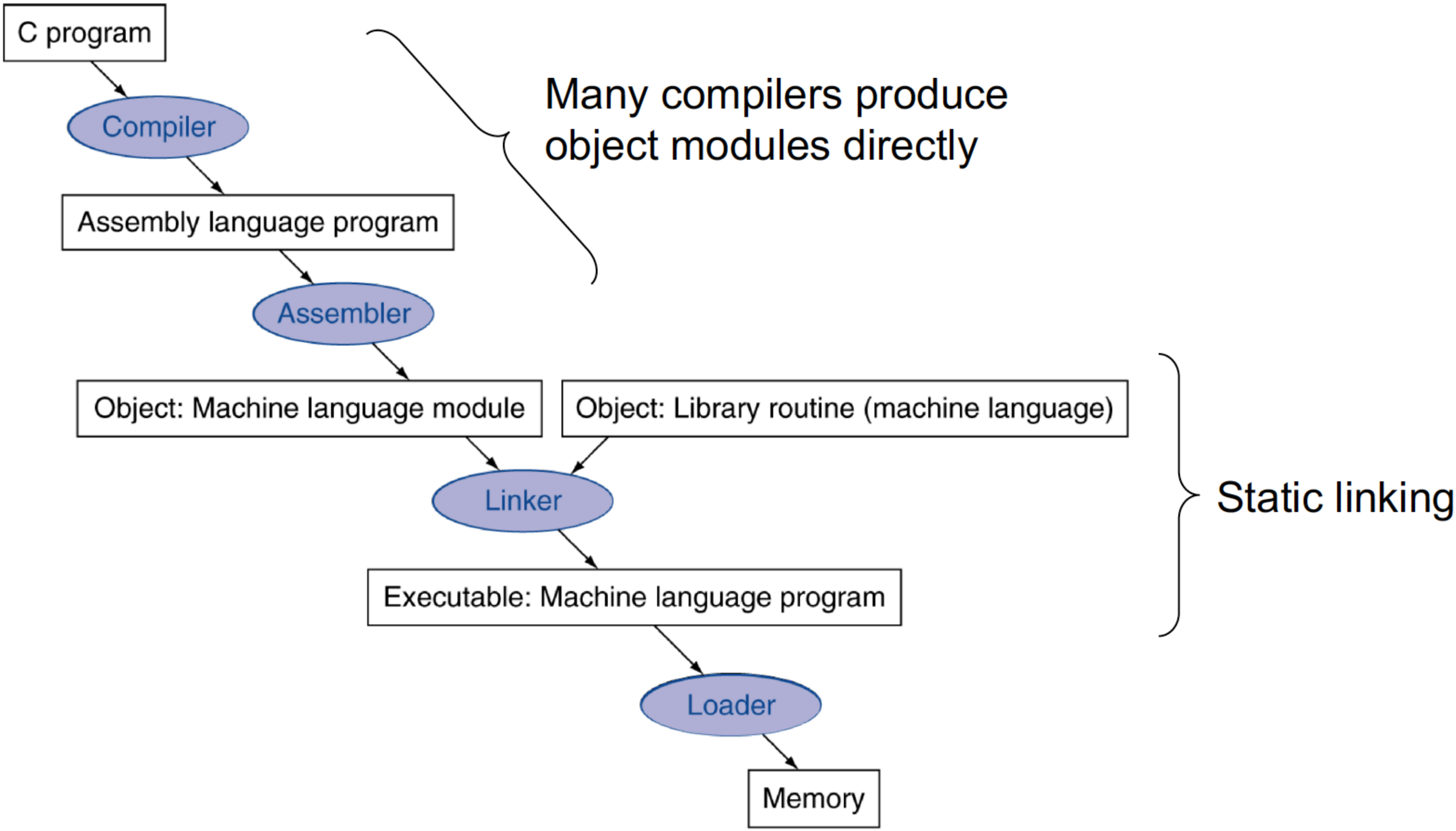
Going From File to Machine Code (cont…)

3. Linker
   - When a program comprises multiple object files, the linker combines these files into a unified executable program, resolving the *symbols* (references) as it goes along.
   - There are 3 steps for the linker:
     1. Place code and data modules symbolically in memory.
     2. Determine the addresses of data and instruction labels.
     3. Patch both the internal and external references.
   - **This produces one executable file with machine language instructions.**

4. Loader
   - OS program that takes the executable code, sets up CPU memory for it, copies over the instructions to CPU memory, initializes all registers, jumps to the start-up routine (i.e. usually `main:`)

4 steps in transforming a program in a file into a program running on a computer

Translation and Startup



Dynamic Linking

- Only finish linking a library procedure *when it is called.*

Pros:

- Often-used libraries need to be stored in only one location, not duplicated in every single executable file.
  - Saves memory and disk space

- Updates/fixes to one library can be done modularly. Cuts down on compiling time.

Cons:

- "DLL hell": newer version of library is not backward compatible.

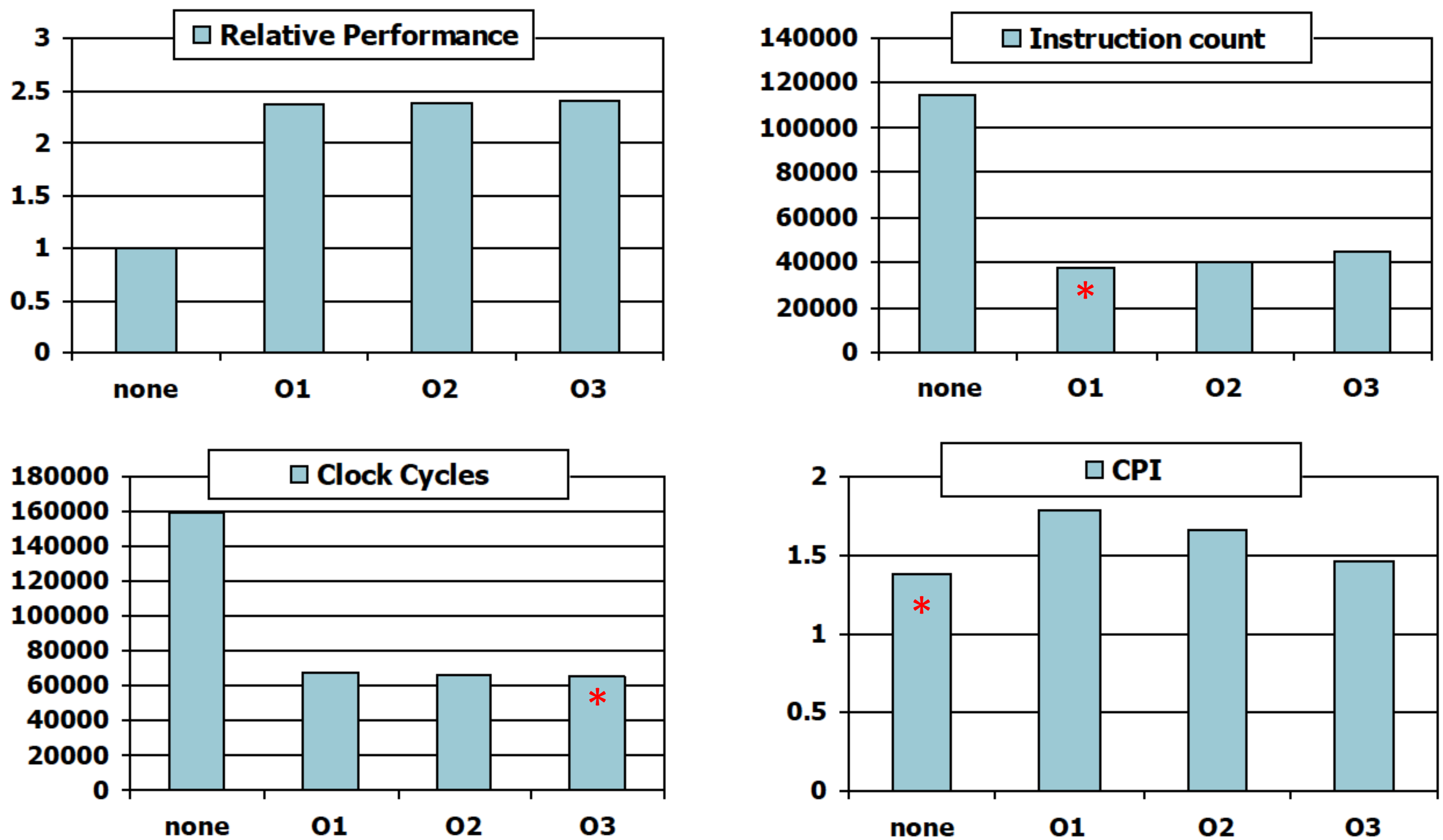
Java

- Java was invented to be different than C/C++
  - Intended to let application developers “write once, run anywhere”

- Rather than compile to the assembly language of a target computer, Java is compiled first to the *Java bytecode instruction set*
  - These run on any *Java virtual machine* (JVM) regardless of the underlying computer architecture
  - JVM is a software interpreter that simulates an ISA
  - Advantage: portability
    - JVMs are found in hundreds of millions of devices (cell phones, Internet browsers, etc…)
- Performance can be enhanced with “Just-in-Time” compilation (JIT)
- Java is very popular, but still generally slower than C/C++

Program Performance:
Effect of Compiler Optimization on *sort* Program

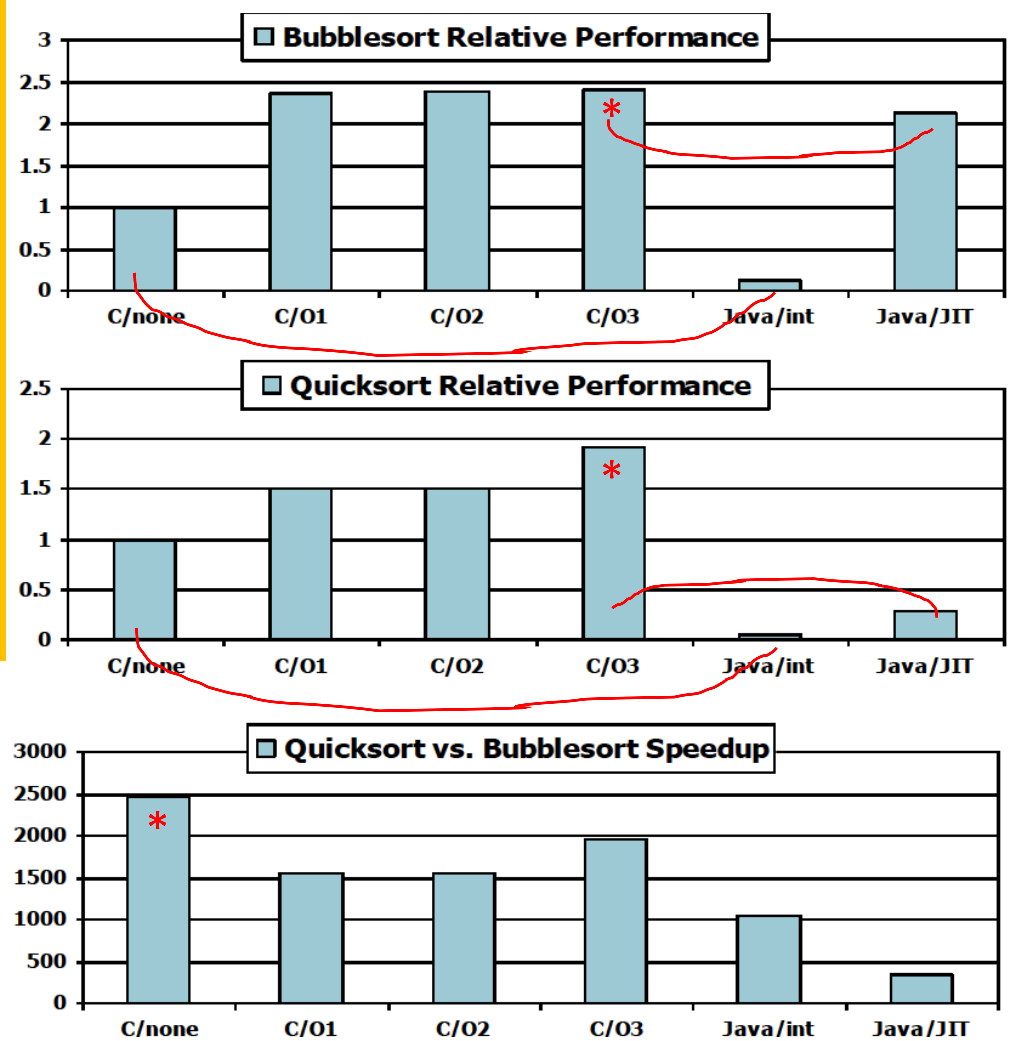
Compiled with gcc for Pentium 4 under Linux



Ultimately, O3 runs the fastest.
Instruction count and CPI are not good performance indicators in isolation

Program Performance: Effect of Language and Algorithm

1. *Compiler optimizations are sensitive to the algorithm*
2. *Java/JIT compiled code is significantly faster than JVM interpreted*
3. *Nothing can fix a dumb algorithm!*



YOUR TO-DOs for the Week

- Readings!

- Work on Lab 4!

</LECTURE>