Review: (Write on the board before class)
1) Dynamic frequencies of operations if designing a new instruction set
2) Register based machines
3) Fallacies: Not using time: Peak performance, throughput, code size,

Today Ch 3: Finish Instruction Set Architecture & projects

Role of the compiler:

Today ISA/compiler are inseparable; can't do good research in ISA without compiler (toy programs, hand compilation not trusted); say that in this decade research will have to have be at the state of the art in both fields, classically independent (courses, books, conferences)

Goals: correct compilation & speed of code

Figure 3.24: structure of optimizing compiler
Some optimizations done before we know the code
(expand procedure calls inline before know the compiler)
=> *phase ordering*
Global common subexpression assumes saved term in a register, or pesimizes performance

Graph coloring: powerful tool for register allocation (3.25)
- interference graph: node = candidate for register,
  - arc = ranges of usage that overlap
  - color graph so that nodes connectd by a single arc have different colors
  - draw code on board and show live ranges as bars
- 2 colors so 2 registers
- At least 16 registers

Categories of optimization (3.26)
Collective impact of combinations (3.27)
Impact on instruction mix (3.30 and 3.31)

General Advice to ISA architect to help compiler writer
1. regular: all addressing modes apply (orthogonal)
2. primitives, not solutions (e.g., calls is a bad example)
3. simplify tradeoffs (which sequence is best); mem-mem arch when use a register?

4. bind quantities known at compile time as constants (call mask in calls)

Pitfalls: Hardware vs. software? (decimal & string)
Pitfall
Closing semantic gap results in semantic clash
(length of string in Pascal vs. end mark in C)
Overkill: too general for frequent case; read calls (callee determines registers but caller does saving)

1. Align the stack if needed.
2. Push the argument count on the stack.
3. Save the registers indicated by the procedure call mask on the stack. The mask is kept in the called procedure's code—this permits callee-save to be done by the caller even with separate compilation.
4. Push the return address on the stack, then push the top and base of stack pointers for the activation record.
5. Clear the condition codes, which sets the trap enables to a known state.
6. Push a word for status information and a zero word on the stack.
7. Update the two stack pointers.
8. Branch to the first instruction of the procedure.

Overkill: level of functionality exceeds what is needed
Intel 432 always creates a new protected context
But most calls within same module and don't need protection
20% speedup

PIAT: Note how load-store differ from mem-mem
(3.32 MIX: % lower of branches because more of others)
(3.33 counts: fewer data access, more instruction accesses)

Dicussion: **3.9** [Discussion] <3.2–3.8> What are the **economic** arguments (i.e., more machines sold) **for and against** changing instruction set architecture?

Break
Find a new partner for HW: how many need a partner
Discussion questions?

Summary: Compiler technology dominates ISA

Next time: Chapter 4

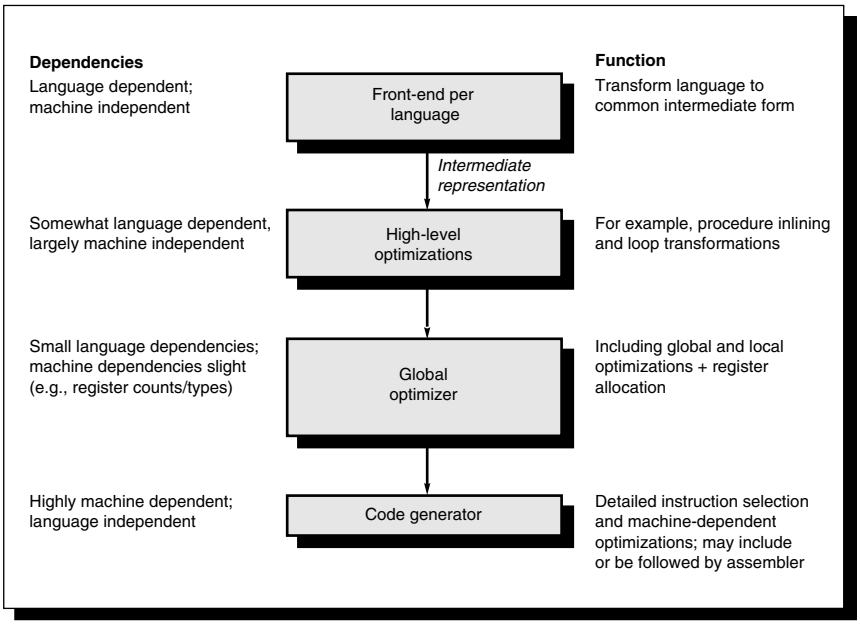

FIGURE 3.24 Current compilers typically consist of two to four passes, with more highly optimizing compilers having more passes.

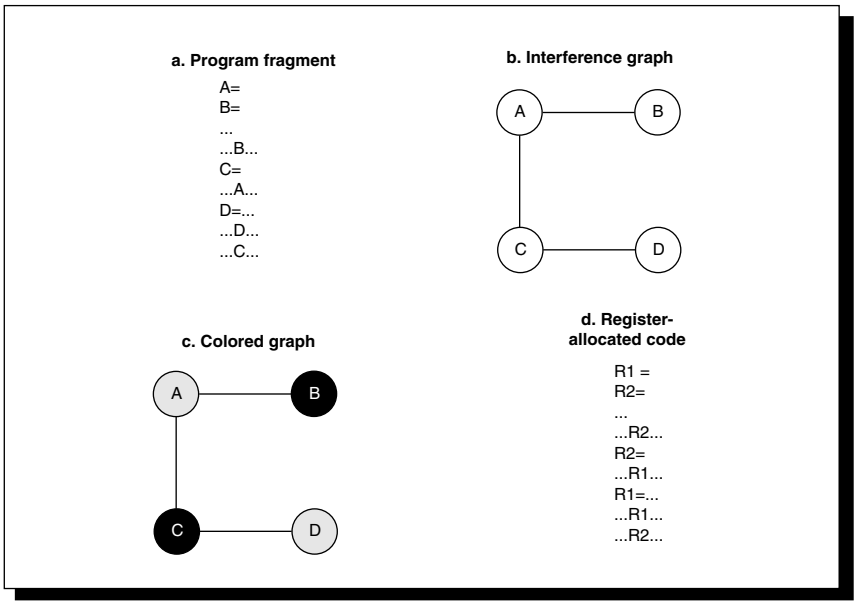

FIGURE 3.25 Graph coloring is used to allocate registers by constructing an interference graph that is colored heuristically using a number of colors corresponding to the register count.

Optimization name	Explanation	Percent of the total number of optimizing transforms
High-level	**At or near the source level; machine independent**	
Procedure integration	Replace procedure call by procedure body	N.M.
Local	**Within straightline code**	
Common subexpression elimination	Replace two instances of the same computation by single copy	18%
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	N.M.
Global	**Across a branch**	
Global subexpression elimination	Same as local, but this version crosses branches	13%
Copy propagation	Replace all instances of a variable *A* that has been assigned *X* (i.e., *A*=*X*) with *X*	11%
Code motion	Remove code from a loop that computes same value each iteration of the loop	16%
Induction variable elimination	Simplify/eliminate array-addressing calculations within loops	2%
Machine dependent	**Depends on machine knowledge**	
Strength reduction	Many examples, such as replace multiply by a constant with adds and shifts	N.M.
Pipeline scheduling	Reorder instructions to improve pipeline performance	N.M.
Branch offset optimization	Choose the shortest branch displacement that reaches target	N.M.

FIGURE 3.26 Major types of optimizations and examples in each class.

Optimizations performed	Percent faster
Procedure integration only	10%
Local optimizations only	5%
Local optimizations + register allocation	26%
Global and local optimizations	14%
Local and global optimizations + register allocation	63%
Local and global optimizations + procedure integration + register allocation	81%

FIGURE 3.27 Performance effects of various levels of optimization.

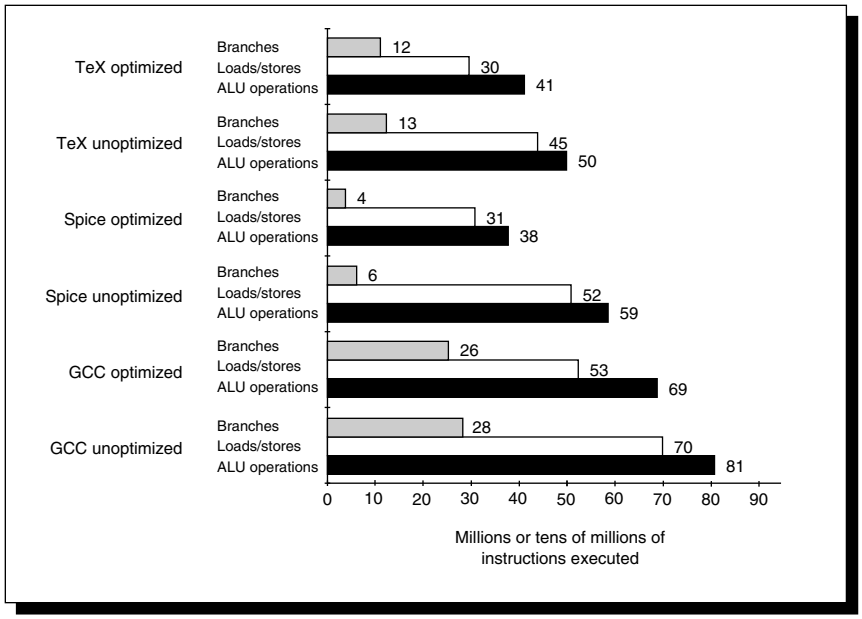

FIGURE 3.30 The effects of optimization in absolute instruction counts.

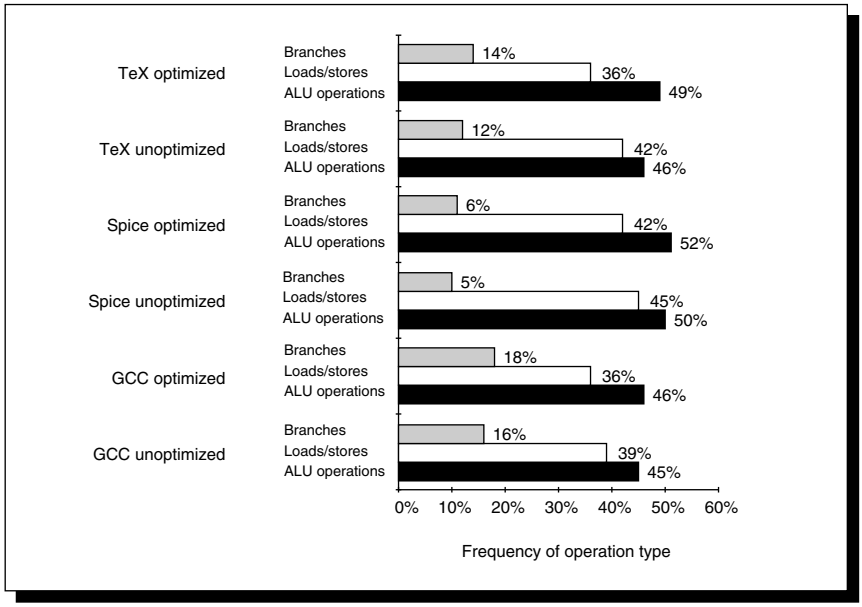

FIGURE 3.31 The effects of optimization on the relative mix of instructions for the data in Figure 3.30.

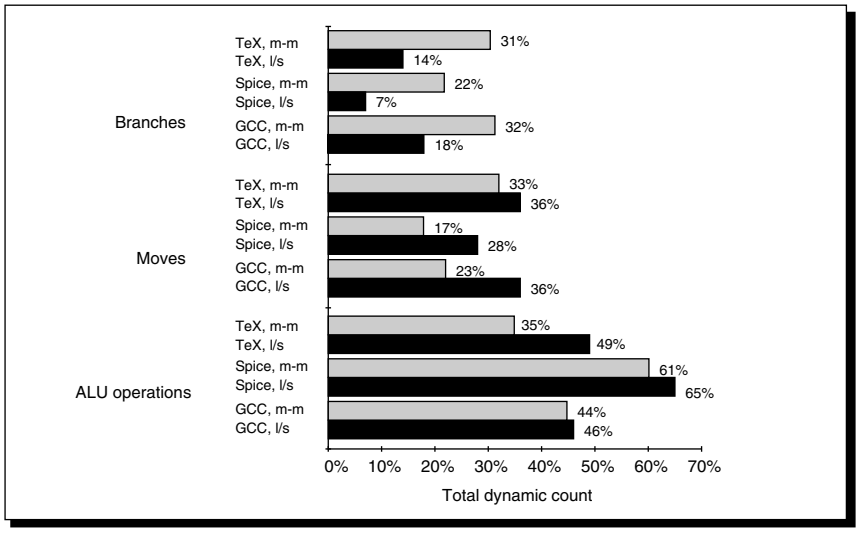

FIGURE 3.32 The instruction distributions for our benchmarks differ in straight forward ways when run on a load/store architecture (l/s) and on a memory–memory architecture (m–m).

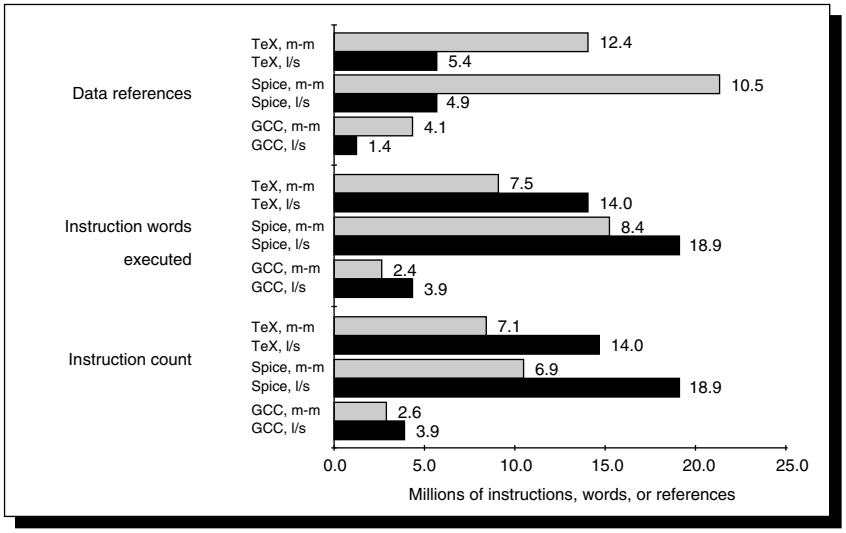

FIGURE 3.33 Absolute counts for dynamic events on a load/store and memory–memory machine.