Review: (Write on the board before class)
1) Price vs. Cost: 8 to 15% for engineering,
2) low volume, high investment => poorer price/perf supercomputers
3) Fallacies: Not using time: Peak performance, throughput, code size,

Today Ch 3: Instruction Set Architecture Review, will go over quickly

Implementation vs. Architecture =>
Computer Architecture = Instruction set architecture

Also, hard to figure best order; instruction sets and then fast execution or vice versa; if reread this chapter first, will see difference

static vs. dynamic

Why measurements? Designing a new instruction set

First concentrate on register based machines, since dominate and likely to stay dominant; stacks of historical interest

Operand location: register-register or load/store vs. register-memory vs. memory-memory (see figures 3.5 & 3.6)

Memory addressing:
Endian (3.7), Alignment in memory
List of addressing modes (3.11) & frequency (3.12)

Rest of statistics like to see as answer to questions?

What is impact of different sized offsets in instruction set? (3.13)

How important are immediates? (3.14)
Size of immediates? (3.15) (0 not included here)

Operations: what is in the 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

What is relation of call/return, jump, branch? (3.18) (speed)

What is branch distance? (3.19) (branch offset)

Which comparisons most popular? (3.21) (branch operations)

What percentage of branches branch? (3.22) (which speedup)

Operands:
Which operands most popular? (3.23) program dependent
Note how dangerous it would be to base design on a single program

Role of the compiler:

In last decade virtually all programming in HLL
(How many people program in assembly language routinely?
How many know people who routinely program in assembly language?
Why has it changed in PCs?
ISA is compiler target, not human interface

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)

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)

Break
Homework assignments: 3.8, 4.2, 4.3 Friday in 533 Evans
Find a new partner for HW
be prepared to discuss in class next Wed 3.9
Discussion questions?

Summary: Compiler technology dominates ISA

Next time: Projects and finish chapter 4

Number of memory addresses per typical ALU instruction	Maximum number of operands allowed for a typical ALU instruction	Examples
0	2 3	IBM RT-PC SPARC, MIPS, HP Precision Architecture
1	2 3	PDP-10, Motorola 68000, IBM 360 Part of IBM 360 (RS instructions)
2	2 3	PDP-11, National 32x32, part of IBM 360 (SS instructions)
3	3	VAX (also has two operand formats)

FIGURE 3.5 Combinations of memory and total operands

Type	Advantages	Disadvantages
Register–register (0,3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see Chapter 6).	Higher instruction count than architectures with memory references in instructions. Some instructions are short and bit encoding may be wasteful.
Register–memory (1,2)	Data can be accessed without loading first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. CPI varies by operand location.
Memory–memory (3,3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. Also, large variation in work per instruction. Memory accesses create memory bottleneck.

FIGURE 3.6 Advantages and disadvantages of the three most common types of general-purpose register machines. The notation (*m, n*) means *m* memory operands and *n* total operands.

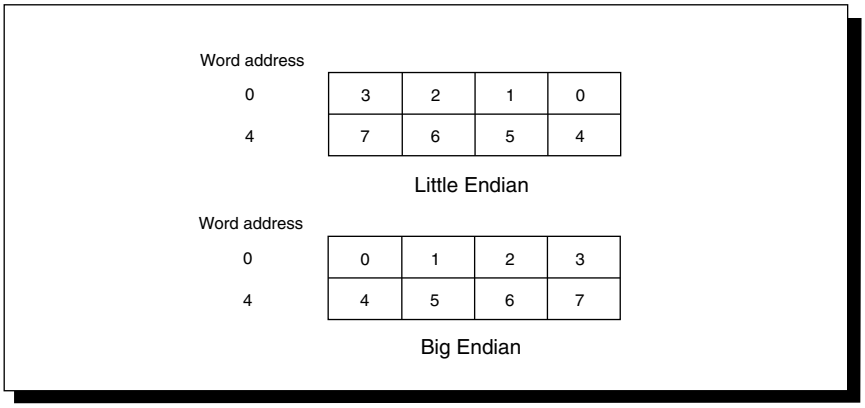

FIGURE 3.7 The two different conventions for ordering bytes within a word. The names "Big Endian" and "Little Endian" come from a famous paper by Cohen [1981].

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`R4←R4+R3`	When a value is in a register.
Immediate or literal	`Add R4,#3`	`R4←R4+3`	For constants. In some machines, literal and immediate are two different addressing modes.
Displacement or based	`Add R4,100(R1)`	`R4←R4+M[100+R1]`	Accessing local variables.
Register defer-red or indirect	`Add R4,(R1)`	`R4←R4+M[R1]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1 + R2)`	`R3←R3+M[R1+R2]`	Sometimes useful in array addressing—`R1`=base of array; `R2`=index amount.
Direct or absolute	`Add R1,(1001)`	`R1←R1+M[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect or memory deferred	`Add R1,@(R3)`	`R1←R1+M[M[R3]]`	If `R3` is the address of a pointer p, then mode yields $*p$.
Auto-increment	`Add R1,(R2)+`	`R1←R1+M[R2]` `R2←R2+`d	Useful for stepping through arrays within a loop. `R2` points to start of array; each reference increments `R2` by size of an element, d.
Auto-decrement	`Add R1,-(R2)`	`R2←R2-`d `R1←R1+M[R2]`	Same use as autoincrement. Autoincrement/decrement can also be used to implement a stack as push and pop.
Scaled or index	`Add R1,100(R2)[R3]`	`R1← R1+M[100+R2+R3*`d`]`	Used to index arrays. May be applied to any base addressing mode in some machines.

FIGURE 3.11 Selection of addressing modes with examples, meaning, and usage.

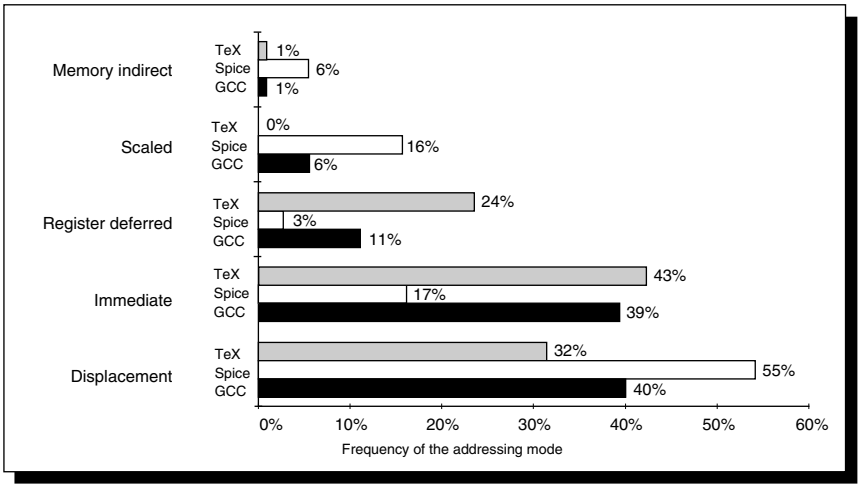

FIGURE 3.12 Summary of use of memory addressing modes (including immediates).

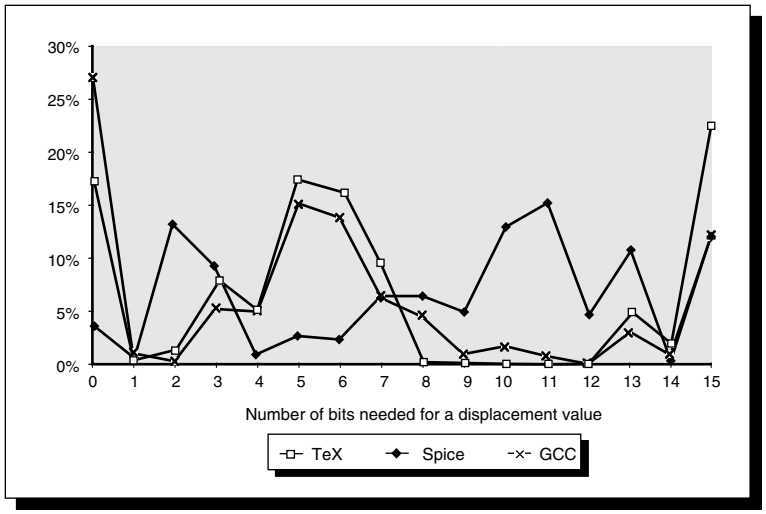

FIGURE 3.13 Displacement values are widely distributed.

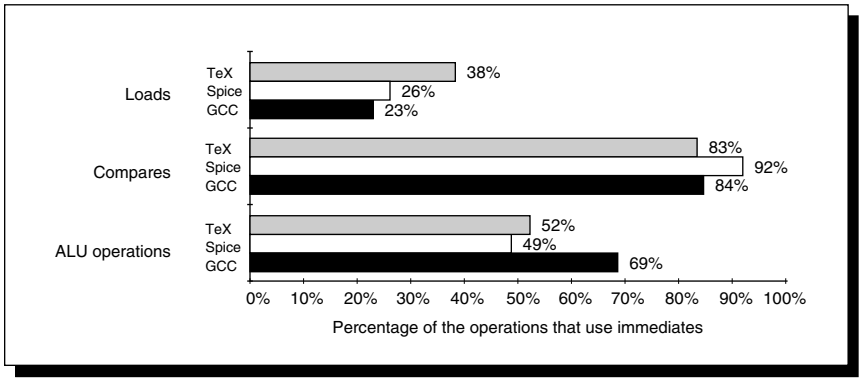

FIGURE 3.14 We see that for ALU operations about half the operations have an immediate operand, while for compares more than 85% of the occurrences use an immediate operand.

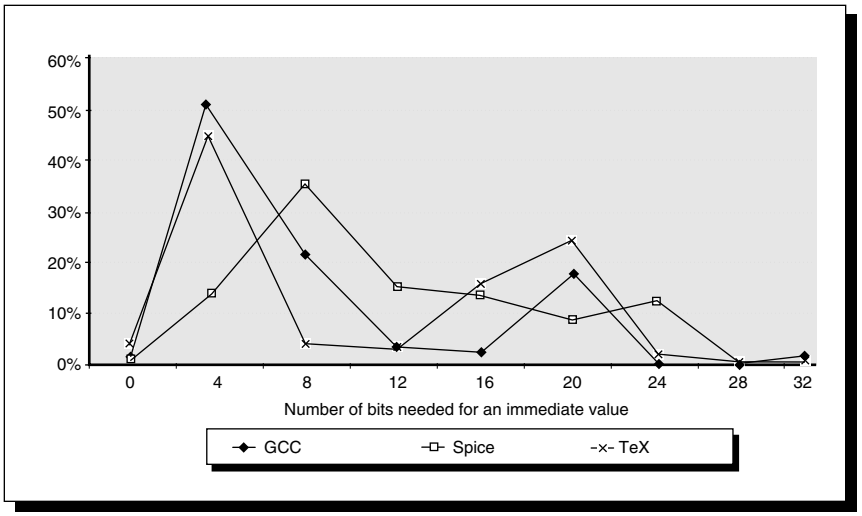

FIGURE 3.15 The distribution of immediate values is shown.

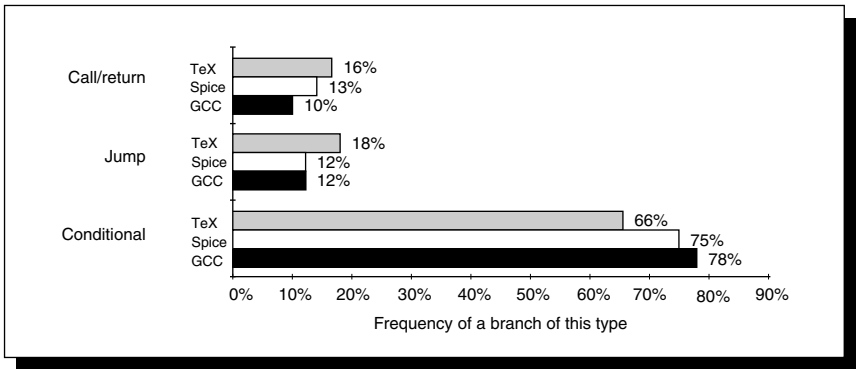

FIGURE 3.18 Breakdown of branches into three classes.

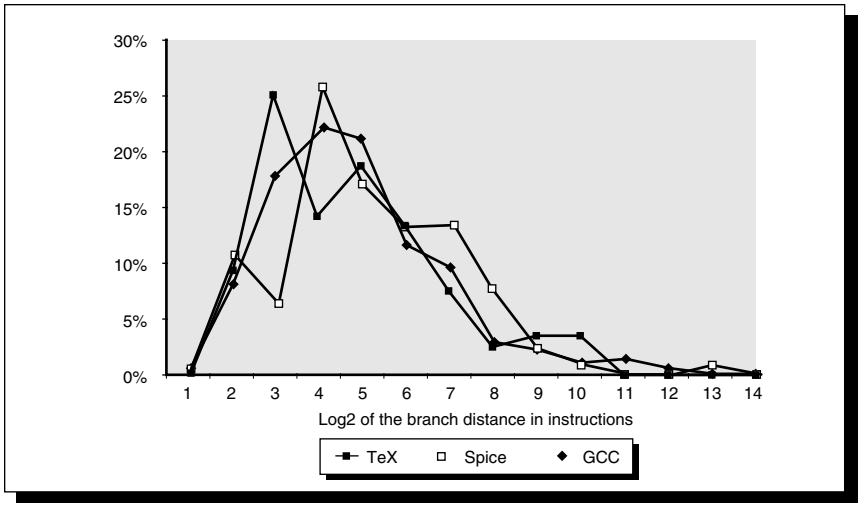

FIGURE 3.19 Branch distances in terms of number of instructions between the target and the branch instruction.

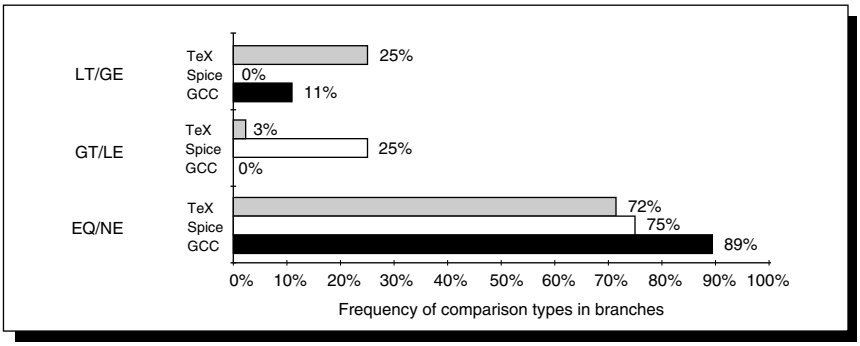

FIGURE 3.21 Frequency of different types of compares in conditional branches.

Program	% of backward branches	% taken branches	% of all control instructions that actually branch
GCC	26%	54%	63%
Spice	31%	51%	63%
TeX	17%	54%	70%
Average	25%	53%	65%

FIGURE 3.22 Branch direction, branch-taken frequency, and frequency that the PC is changed.

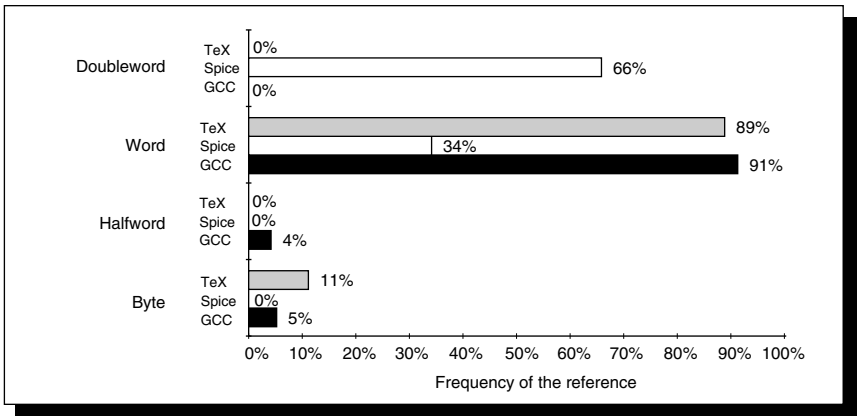

FIGURE 3.23 Distribution of data accesses by size for the benchmark programs.

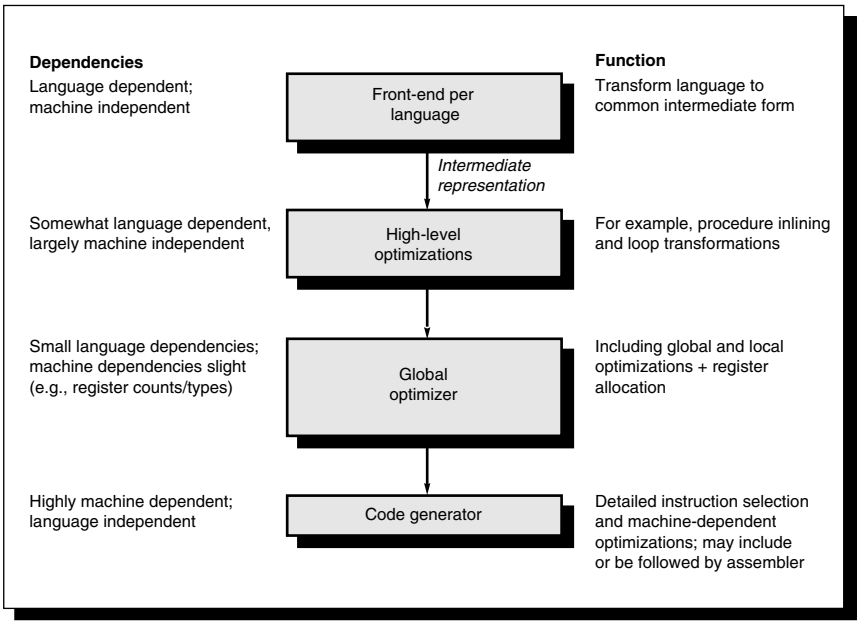

FIGURE 3.24 Current compilers typically consist of two to four passes, with more highly optimizing compilers having more passes.

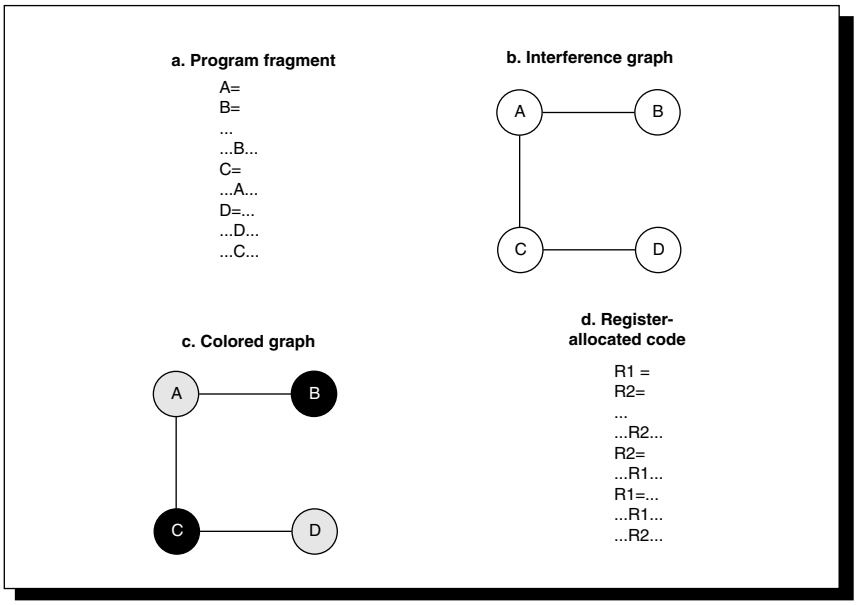

FIGURE 3.25 Graph coloring is used to allocate registers by constructing an interference graph that is colored heuristically using a number of colors corresponding to the register count.

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.

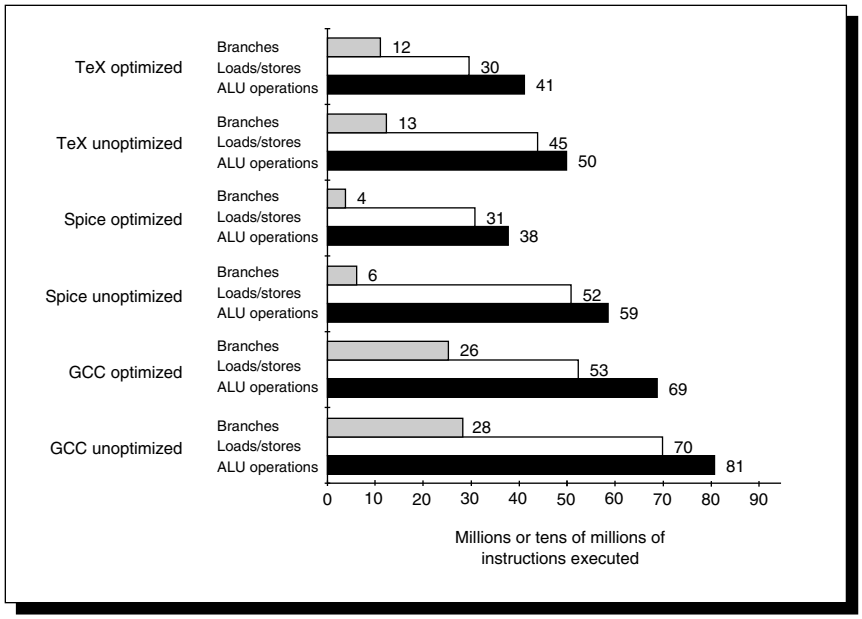

FIGURE 3.30 The effects of optimization in absolute instruction counts.

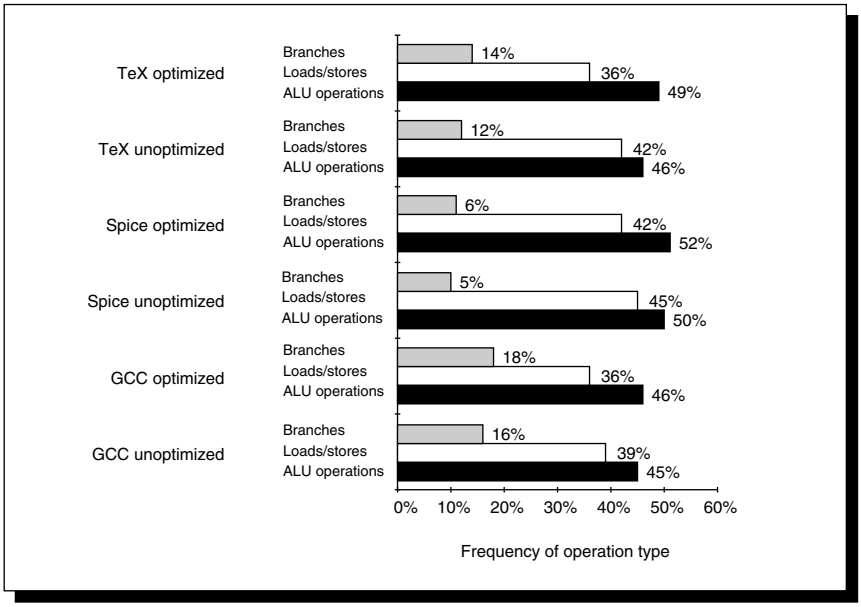

FIGURE 3.31 The effects of optimization on the relative mix of instructions for the data in Figure 3.30.

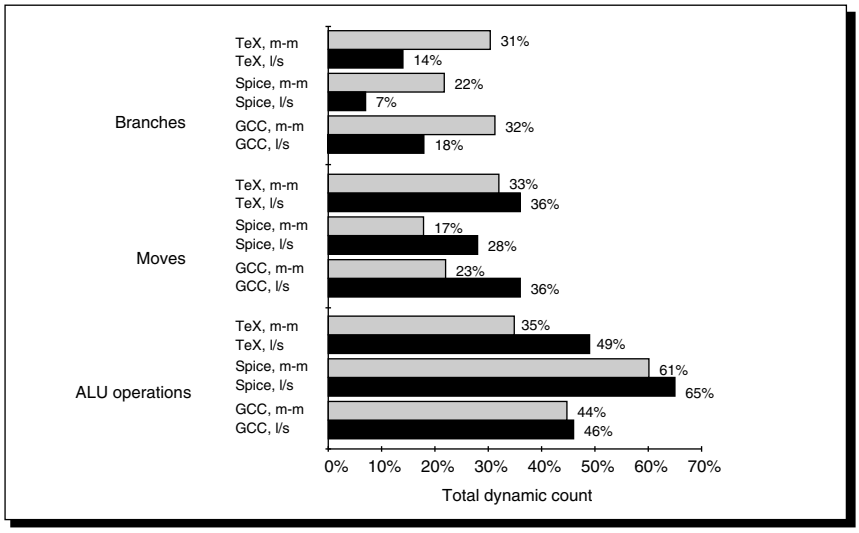

FIGURE 3.32 The instruction distributions for our benchmarks differ in straight forward ways when run on a load/store architecture (l/s) and on a memory–memory architecture (m–m).

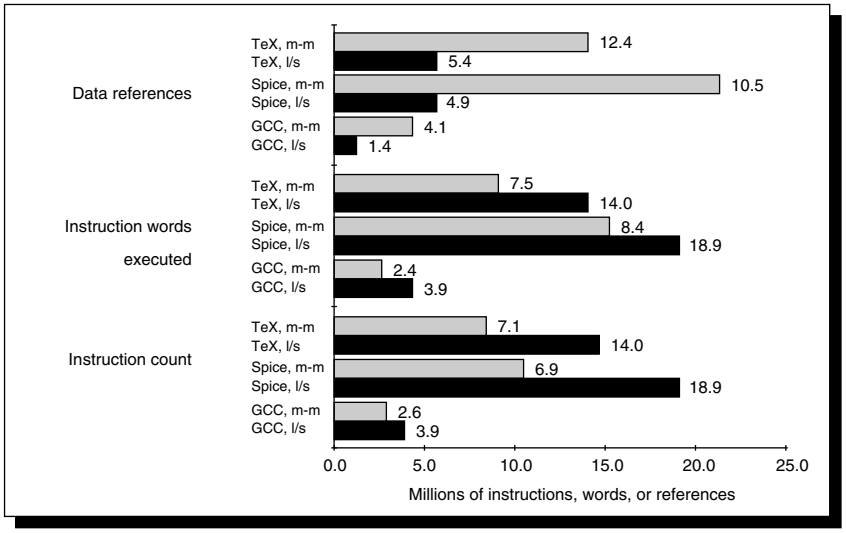

FIGURE 3.33 Absolute counts for dynamic events on a load/store and memory–memory machine.