

```

for (i = 0; i < n-1; i++) {
    for (j = i+1; j < n; j++) {
        for (k = n-1; k >= i; k--) {
            A[j][k] -= A[i][k] * A[j][i] / A[i][i];
        }
    }
}

```

(Gaussian elimination serves to triangularize a matrix. It is a key step in the solution of systems of linear equations.) What are the loop invariants in this code? What are the loop-carried dependences? Discuss how to optimize the code. Be sure to consider locality-improving loop transformations.

- 16.25** Modify the tiled matrix transpose of Example ©16.32 to eliminate the min calculations in the bounds of the inner loops. Perform the same modification on your answer to Exercise ©16.23.

16.11 Explorations

- 16.26** Investigate the back-end structure of your favorite compiler. What levels of optimization are available? What techniques are employed at each level? What is the default level? Does the compiler generate assembly language or object code?

Experiment with optimization in several program fragments. Instruct the compiler to generate assembly language, or use a disassembler or debugger to examine the generated object code. Evaluate the quality of this code at various levels of optimization.

If your compiler employs a separate assembler, compare the assembler input to its disassembled output. What optimizations, if any, are performed by the assembler?

- 16.27** As a general rule, a compiler can apply a program transformation only if it preserves the correctness of the code. In some circumstances, however, the correctness of a transformation may depend on information that will not be known until run time. In this case, a compiler may generate two (or more) versions of some body of code, together with a run-time check that chooses which version to use, or customizes a general, parameterized version.

Learn about the “inspector-executor” compilation paradigm pioneered by Saltz et al. [SMC91]. How general is this technique? Under what circumstances can the performance benefits be expected to outweigh the cost of the run-time check and the potential increase in code size?

- 16.28** A recent trend is the use of static compiler analysis to check for patterns of information flow that are likely (though not certain) to constitute

programming errors. Investigate the work of Guyer et al. [GL05], which performs analysis reminiscent of *taint mode* (Exploration 15.17) at compile time. In a similar vein, investigate the work of Yang et al. [YTEM04] and Chen et al. [CDW04], which use static *model checking* to catch high-level errors. What do you think of such efforts? How do they compare to taint mode or to *proof-carrying code* (Exploration 15.18)? Can static analysis be useful if it has both false negatives (errors it misses) and false positives (correct code it flags as erroneous)?

- 16.29** In a somewhat gloomy parody of Moore's Law, Todd Proebsting of Microsoft Research (himself an eminent compiler researcher) offers *Proebsting's Law*: "Compiler advances double computing power every 18 years." (See research.microsoft.com/toddpro/ for pointers.)

Survey the history of compiler technology. What have been the major innovations? Have there been important advances in areas other than speed? Is Proebsting's Law a fair assessment of the field?

16.12 Bibliographic Notes

Recent compiler textbooks (e.g., those of Cooper and Torczon [CT04], Grune et al. [GBJL01], or Appel [App97]) are an accessible source of information on back-end compiler technology. Much of the presentation here was inspired by Muchnick's *Advanced Compiler Design and Implementation*, which contains a wealth of detailed information and citations to related work [Muc97]. Much of the leading-edge compiler research appears in the annual *ACM Conference on Programming Language Design and Implementation* (PLDI). A compendium of "best papers" from the first 20 years of this conference was published in 2004 [McK04].

Throughout our study of code improvement, we concentrated our attention on the von Neumann family of languages. Analogous techniques for functional [App91; KKR⁺86; Pey87; Pey92; WM95, Chap. 3; App97, Chap. 15; GBJL01, Chap. 7]; object-oriented [AH95; GDDC97; WM95, Chap. 5; App97, Chap. 14; GBJL01, Chap. 6]; and logic languages [DRSS96; FSS83; Zho96; WM95, Chap. 4; GBJL01, Chap. 8] are an active topic of research, but are beyond the scope of this book. A key challenge in functional languages is to identify repetitive patterns of calls (e.g., tail recursion), for which loop-like optimizations can be performed. A key challenge in object-oriented languages is to predict the targets of virtual subroutine calls statically, to permit in-lining and interprocedural code improvement. The dominant challenge in logic languages is to better direct the underlying process of goal-directed search.

Local value numbering is originally due to Cocke and Schwartz [CS69]; the global algorithm described here is based on that of Alpern, Wegman, and Zadeck [AWZ88]. Chaitin et al. [CAC⁺81] popularized the use of graph coloring for