Outline of a Roadmap for Compiler Technology

David A. Padua
University of Illinois at Urbana-Champaign

Compiler technology has been a major subfield of computer science ever since the first compilers were developed in the late 1950s. Compilers made possible the development of today's efficient and sophisticated software at an affordable cost, thus playing a crucial role in popularizing computers. Although we have learned much over 40 years about compiler development tools, internal compiler organization, parsing techniques, and optimization algorithms, sustained progress in computer usability and performance will require much more research in this area.

What drives compiler technology?

High-level languages and machine architectures define the character of the compiler technology that connects them. New architectural features usually lead to new compiler algorithms, whose objective is to improve the performance of high-level language programs without introducing machine-dependent constructs that tend to hinder readability and portability. For example, compiler algorithms for register allocation allow high-level language programs to profit from multiple CPU registers.

Parallelism, in all its forms, is another important example of architectural features exploited via compiler algorithms. Techniques like trace scheduling, software pipelining, and their derivatives were developed and are universally applied to exploit (or to exploit more effectively) the functional unit parallelism available on practically all current machines. Parallelism is today one of the main concerns in computer architecture. In fact, widespread opinion holds that most computers of the future, from PCs to the most powerful machines, will be parallel. For this reason, coarse-grain parallelism detection, locality enhancement, and communication optimization will be central concerns of compiler technology in the foreseeable future.

The main characteristics of the most popular programming languages have evolved quite slowly since the late 1950s. A Fortran or Algol 60 programmer of that time would be familiar with many of today's programming constructs. The evolution of the widely accepted general-purpose languages probably will remain slow. Certainly, a paradigm shift—for example, to functional or logic programming—is unlikely. Nevertheless, programming languages will evolve, and two major forces driving this evolution will be parallel computing and problem-solving environments. New constructs for parallelism are already available for most parallel computers, and standard parallel extensions will most likely appear soon. Many application packages and problem-solving environments include some form of programming language for flexibility. Embedded languages will become even more common. They will include interfaces to powerful preexisting modules and will incorporate domain-specific language constructs and intrinsic data types. Already numerous examples of languages are embedded in problem-solving packages, including the languages of computer algebra packages, the Matlab language, and Java (if we categorize Web browsers as problem-solving environments).

1070-9924/96/$5.00 © 1996 IEEE

Compiler technology challenges

Much work lies ahead in compiler technology to continue improving the translation of conventional languages for uniprocessors and to support progress in machine architecture and programming language design. We will need a concerted effort involving algorithm development, evaluation, and implementation strategies to develop Fortran 90 and C++ compilers of quality equal to some previous Fortran compilers. Also, progress in compiling functional and logic programming languages should continue. These studies widen our understanding of language and compiler technology. At the same time, we will need sophisticated new algorithms enabling the accurate global program analysis necessary to generate efficient code from very high level constructs that are typical of embedded languages, and to effectively compile the code for highly parallel computers.

I'll briefly discuss a few important specific challenges in compiler technology.

A public-domain compiler infrastructure

We sorely need a high-quality, robust compiler infrastructure to improve the quality of compiler technology research and to accelerate computing progress in general. In fact, an important factor limiting experimental compiler research has been the lack of a public-domain infrastructure in which researchers could easily incorporate new translation algorithms and evaluate them in the context of a complete compiler. Fortunately it appears that this situation will soon be remedied. Several federal government agencies, led by DARPA, plan to initiate development of such a compiler infrastructure.

A related issue is the need for extensive, realistic benchmark collections. The Perfect and SPEC benchmarks and a few others have been widely used and very helpful in the past few years. However, we need collections that are more extensive in the number of codes, computational requirements, and dimensions of codes.

Effective compiler algorithms for parallel computers

The general area of compilers for parallel machines has two subtopics. One is the translation of conventional languages, such as Fortran and C++, for parallel execution (also called parallelization). The second is the compilation of explicitly parallel languages. Compiler techniques for explicitly parallel languages are needed not only to translate programs with user-specified parallelism but also to build the back end of parallelizers.

Although much progress has been made in the translation of conventional languages, much work remains. Accurate analysis algorithms to detect parallelism are an important parallelizer component, as are algorithms such as those for data layout selection, locality enhancement, and communication optimization. Progress in these analysis and translation techniques is important because explicitly specifying all relevant parallelism in a program and coding for efficient communication often increase software development costs, which are already too high.

We also need new compiler algorithms for explicitly parallel programming languages—some to deal with resource allocation and overhead and others to allow application of traditional optimizations to parallel languages. This is a largely unexplored subject, with many open problems. Unfortunately, the lack of standard parallel programming languages and parallel program benchmarks impedes the development of a solid experimental foundation in this area.

Compiler techniques for data-structure manipulation

Most compiler algorithms operate on control structures. However, manipulating data structures is very important in some cases. For example, array distribution across modules in a distributed-memory machine, and array layout on different classes of machines, may strongly affect performance. Past research has studied automatic selection of data structures to represent objects in the SETL language, and several ongoing projects are studying techniques to automatically select the data structure to represent sparse arrays. This issue will most likely gain greater attention as new languages embedded in problem-solving environments are developed.

Increased compiler efficiency

Fast compilation has always been a goal and will increase in importance as new machines and languages introduce more complex translation processes. Major areas of study include techniques that postpone part of the compilation until execution in order to decrease response time, and compiler algorithms that execute in parallel.

Adaptive code generation strategies

Compiler developers are increasingly using runtime information to speed up program execution. For example, branch frequency information has been used to improve the quality of code generated for multifunctional parallelism. Also, some experimental compilers generate code that determines at execution time how to distribute data or whether a loop is parallel. These two strategies illustrate an increasingly important general approach that might be called "adaptive compiling" or "runtime code transformation." ♦

David A. Padua is a professor of computer science at the University of Illinois. He is a principal investigator in the Polaris project, working to create an optimizing compiler that automatically puts conventional Fortran programs into parallel form. Readers can contact Padua at the University of Illinois at Urbana-Champaign, 3318 Digital Computer Laboratory, 1304 W. Springfield Ave., Urbana, IL 61801; e-mail, padua@csrd.uiuc.edu.