DOI:10.1145/1461928.1461946

Research and education in compiler technology is more important than ever.

BY MARY HALL, DAVID PADUA, AND KESHAV PINGALI

Compiler Research: The Next 50 Years

WE PRESENT A perspective on the past contributions, current status, and future directions of compiler technology and make four main recommendations in support of a vibrant compiler field in the years to come. These recommendations were drawn from discussions among presenters and attendees at a U.S. National Science Foundation-sponsored Workshop on Future Directions for Compiler Research and Education in 2007. As 2007 was the 50th anniversary of IBM's release of the first optimizing compiler, it was a particularly appropriate year to take stock of the status of compiler technology and discuss its future over the next 50 years. Today, compilers and high-level languages are the foundation of the complex and ubiquitous software infrastructure that undergirds the global economy. The powerful and elegant technology in compilers has also been invaluable in other domains (such as hardware synthesis). It is no exaggeration to say that compilers and high-level languages are as central to the information age as semiconductor technology.

In the coming decade, 2010 to 2020, compiler research will play a critical role in addressing two of the major challenges facing the overall computer field:

Cost of programming multicore processors. While machine power will continue to grow impressively, increased parallelism, rather than clock rate, will be the driving force in computing in the foreseeable future. This ongoing shift toward parallel architectural paradigms is one of the greatest challenges for the microprocessor and software industries. In 2005, Justin Rattner, chief technology officer of Intel Corporation, said, "We are at the cusp of a transition to multicore, multithreaded architectures, and we still have not demonstrated the ease of programming the move will require..."[3]

Security and reliability of complex software systems. Software systems are increasingly complex, making the need to address defects and security attacks more urgent. The profound economic impact of program defects was discussed in a 2002 study commissioned by the U.S. Department of Commerce National Institute of Standards and Technology (NIST), concluding that program defects "are so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually, or about 0.6% of the gross domestic product." The 2005 U.S. President's Information Technology Advisory Committee (PITAC) report *Cyber Security: A Crisis of Prioritization* included secure software engineering and software assurance among its top 10 research priorities, concluding with: "Commonly used software engineering practices permit dangerous errors, such as improper handling of buffer overflows, which enable hundreds of attack programs to compromise millions of computers every year. In the future, the Nation [the U.S.] may face even more challenging problems as adversaries—both foreign and do-

mestic—become increasingly sophisticated in their ability to insert malicious code into critical software..."

Cultural Shift

To address these challenges, the compiler community must change its current research model, which emphasizes small-scale individual investigator activities on one-off infrastructures. Complete compiler infrastructures are just too complex to develop and maintain in the academic research environment. However, integration of new compiler research into established infrastructures is required to ensure the migration of research into practice. This conundrum can be solved only through a new partnership between academia and industry to produce shared open source infrastructure, representative benchmarks, and reproducible experiments. If successful, this new model will affect both commercial applications and scientific capabilities. Another 2005 PITAC report *Computational Science: Ensuring America's Competitiveness* highlighted the need for research in enabling software technologies, including programming models and their compilers, to maintain U.S. national competitiveness in computational science (see the sidebar "Agenda for the Compiler Community"). The PITAC report said, "Because the Nation's [the U.S.] research infrastructure has not kept pace with changing technologies, today's computational science ecosystem is unbalanced, with a software base that is inadequate for keeping pace with and supporting evolving hardware and application needs. By starving research in enabling software and applications, the imbalance forces researchers [in the U.S.] to build atop inadequate and crumbling foundations rather than on a modern, high-quality software base. The result is greatly diminished productivity for both researchers and computing systems."

Accomplishments

When the field of compiling began in the late 1950s, its focus was limited to the translation of high-level language programs into machine code and to the optimization of space and time requirements of programs. The field has since produced a vast body of knowledge about program analysis and transformations, automatic code generation, and runtime services. Compiler algorithms and techniques are now used to facilitate software and hardware development, improve application performance, and detect and prevent software defects and malware. The compiler field is increasingly intertwined with other disciplines, including computer architecture, programming languages, formal methods, software engineering, and computer security. Indeed, the term "compiler" has associations in the computer science community that are too narrow to reflect the current scope of the research in the area.

ILLUSTRATION BY SIMON GEILFUS

The most remarkable accomplishment by far of the compiler field is the widespread use of high-level languages. From banking and enterprise-management software to high-performance computing and the Web, most software today is written in high-level languages compiled either statically or dynamically. When object-oriented and data abstraction languages were first proposed back in the late 1960s and early 1970s, their potential for vastly improving programmer productivity was recognized despite serious doubts about whether they could be implemented efficiently. Static and dynamic optimizations invented by the compiler community for this purpose put these fears to rest. More recently, particularly with the introduction of Java in the mid-1990s, managed runtime systems, including garbage collection and just-in-time compilation, have improved programmer productivity by eliminating memory leaks.

Compiler algorithms for parsing, type checking and inference, dataflow analysis, loop transformations based on data-dependence analysis, register allocation based on graph coloring, and software pipelining are among the more elegant creations of computer science. They have profoundly affected the practice of computing because they are

incorporated into powerful yet widely used tools. The increasing sophistication of the algorithms is evident when today's algorithms are compared with those implemented in earlier compilers. Two examples illustrate these advances. Early compilers used ad hoc techniques to parse programs. Today's parsing techniques are based on formal languages and automata theory, enabling the systematic development of compiler front ends. Likewise, early work on restructuring compilers used ad hoc techniques for dependence analysis and loop transformation. Today, this aspect of compilers has been revolutionized by powerful algorithms based on integer linear programming.

Code optimizations are part of most commercial compilers, pursuing a range of objectives (such as avoiding redundant computations, allocating registers, enhancing locality, and taking advantage of instruction-level parallelism). Compiler optimization usually delivers a good level of performance, and, in some cases, the performance of compiler-generated code is close to the peak performance of the target machine. Achieving a similar result with manual tuning, especially for large codes, is extraordinarily difficult, expensive, and error prone. Self-tuning program generators for linear algebra and signal processing are particularly effective in this regard.

Tools for identifying program defects and security risks are increasingly popular and used regularly by the largest software developers. They are notably effective in identifying some of the most frequent bugs or defects (such as improper memory allocations and deallocations, race conditions, and buffer overruns). One indication of the increasing importance of program-analysis algorithms in reliability and security is the growth of the software tools industry that incorporate such algorithms.

One manifestation of the intense intellectual activity in compilers is that the main conferences in the area, including the ACM Symposium on Programming Language Design and Implementation (PLDI), ACM Symposium on Principles of Programming Languages (POPL), ACM Symposium on Principles and Practice of Parallel Programming (PPoPP), and ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), are among the most influential, respected, and selective in computer science.[a] Another indication of the field's influence is that the phrases "programming languages" and "compilers" occur in the citations of no less than seven Turing award winners, including Peter Naur in 2005 and Fran Allen in 2006.

Compiler Challenges

The growing complexity of machines and software, introduction of multicores, and concern over security are among the more serious problems that must be addressed today. Here, we describe the role compiler technology plays in addressing them:

Program optimization. We live in the era of multicore processors; from now on, clock frequencies will rise slowly if at all, but the number of cores on processor chips is likely to double every couple of years. Therefore, by 2020, microprocessors are likely to have hundreds or even thousands of cores, heterogeneous and possibly specialized for different functionalities. Exploiting large-scale parallel hardware will be essential for improving an application's performance or its capabilities in terms of execution speed and power consumption. The challenge for compiler research is how to enable the exploitation of the power of the target machine, including its parallelism, without undue programmer effort.

David Kuck, an Intel Fellow, emphasized in a private communication the importance of compiler research for addressing the multicore challenge. He said that the challenge of optimal compilation lies in its combinatorial complexity. Languages expand as computer use reaches new application domains and new architectural features arise. Architectural complexity (uni- and multicore) grows to support performance, and compiler optimization must bridge this widening gap. Compiler fundamentals are well understood now, but where to apply what optimization has become increasingly difficult over the past few decades. Compilers today are set to operate with a fixed strategy (such as on a single function in a particular data context) but have trouble shifting gears when different

a An indication of the influence of compiler and programming language research is the citation rate rank of the field's major conferences relative to other computer science conferences and journals worldwide as reported by Citeseer (citeseer.ist.psu.edu/impact.html). Their rank on Citeseer as of December 2008 is PLDI (3rd), POPL (13th), PPoPP (14th), and OOPSLA (28th) out of a total of 1,221 computer science conferences and journals.

Collaboration, Research Challenges, Education

Agenda for the Compiler Community

The following agenda for the compiler community demands a broader collaboration between industry and academic institutions, as well as support from government funding agencies, to address the challenges discussed here.

Enablers to facilitate collaborative compiler research:

- **Open and extensible compiler infrastructure with state-of-the-art optimizations;**
- **Collections of benchmarks for evaluating advances in compilers and a strategy for keeping the benchmark collections up to date; and**
- **Methodology for measuring progress and reporting results that encourages all publications to include data in repositories to enable other researchers to reproduce the results.**

Research challenges in optimization:

- **Make parallel programming mainstream;**
- **Write compilers capable of self-improvement; and**
- **Develop performance models to support optimizations for parallel code.**

Research challenges in correctness:

- **Enable development of software as reliable as an airplane;**
- **Enable system software that is secure at all levels; and**
- **Verify the entire software stack.**

Enrich computer science education with compiler technology:

- **Expand compiler courses with examples from new problem domains (such as security); and**
- **Work with experts in other domains to incorporate compiler algorithms into their courses.**

code is encountered in a global context (such as in any whole application).

Kuck also said, "The best hope for the future is adaptive compilation that exploits equivalence classes based on 'codelet' syntax and optimization potential. This is a deep research topic, and details are only beginning to emerge. Success could lead to dramatic performance gains and compiler simplifications while embracing new language and architecture demands."

Make parallel programming mainstream. Although research in parallel programming began more than 30 years ago, parallel programming is the norm in only a few application areas (such as computational science and databases). These server-side applications deal mostly with structured data (such as arrays and relations), and computations can be done concurrently with little synchronization. In contrast, many client-side applications deal with unstructured data (such as sets and graphs) and require much-finer-grain cross-processor synchronization. Programmers have few tools for coding such applications at a high level of abstraction without explicit management of parallelism, locality, communication, load balancing, power and energy requirements, and other dimensions of optimization. Furthermore, as parallelism becomes ubiquitous, performance portability of programs across parallel platforms and processor generations will be essential for developing productive software.

Breakthroughs in compiler technology are essential for making parallel programming mainstream. They will require collaboration with other areas, including tighter integration of compilers with languages, libraries, and runtime environments, to make available the semantic information needed to optimize programs for parallel execution. The historical approach of "whole-program" analysis must be replaced with a hierarchical approach to software development and optimization in which code within a software layer is optimized without analyzing code in lower layers. This abstraction approach requires that each layer have a well-defined API with semantics that describe the information model or ontology of that layer (such as the Google map-reduce programming model). These semantics are used by the compiler to optimize software in higher layers. In the reverse direction, contextual information from higher layers can be used to specialize and optimize code at lower layers.

The most remarkable accomplishment by far of the compiler field is the widespread use of high-level languages.

Because guidance from the programmer is also necessary, interactive or semiautomatic compiler-based tools must also be developed. A noteworthy example of such an approach is a project that used race-detection software to interactively parallelize the Intel IA32 compiler.[1] Interactivity may require incorporation of compilers into integrated development environments, redesigning compiler optimizations to be less dependent on the order in which the optimizations are applied, and redesigning compiler algorithms and frameworks to make them more suitable for an interactive environment.

The advent of just-in-time compilation for languages (such as Java) blurs the distinction between compile time and runtime, opening up new opportunities for program optimization based on dynamically computed program values. As parallel client-side applications emerge, runtime dependence checking and optimization are likely to be essential for optimizing programs that manipulate dynamic data structures (such as graphs).

Develop a rigorous approach to architecture-specific optimization. Compiler front ends benefited greatly from development in the 1960s and 1970s of a systematic theory of lexical analysis and parsing based on automata theory. However, as mentioned in the private communication by David Kuck quoted earlier, there is no systematic approach for performing architecture-specific program optimization, thus hampering construction of parallelizing and optimizing compilers. Developing effective overall optimization strategies requires programmers be able to deal with a vast number of interacting transformations, nonlinear objective functions, and performance prediction, particularly if performance depends on input data. The program optimization challenge is certainly difficult but not insurmountable.

Recent research at a variety of institutions, including Carnegie Mellon University, the University of California, Berkeley, the University of Illinois, MIT, and University of Tennessee, has dem-

onstrated the potential of offline empirical search to tune the performance of application code to a specific architecture. Often called "autotuning," this approach has produced results competitive with hand tuning in such well-studied domains as linear algebra and signal processing. The basic approach is that either the application programmer expresses or the compiler derives a well-defined search space of alternative implementations for a program that is then explored systematically by compiler and runtime tools so the optimization process is able to achieve results comparable to hand tuning. The challenge is to extend the approach to parallel systems and multicore processors, as well as to integrate the technology into a coherent, easy-to-use system that applies to a large number of complex applications.

Language and compiler technology supporting autotuning will greatly facilitate the construction of libraries implementing numerical and symbolic algorithms for a variety of domains, building on examples from self-tuning linear algebra and signal-processing libraries. In addition to encapsulating efficient algorithms that can be implemented by only a handful of compiler experts and used by a vast number of nonexpert programmers, these libraries can contribute to the design of future high-level languages and abstractions by exposing new and interesting patterns for expressing computation. Furthermore, describing the characteristics of these libraries in a machine-usable format, such that compilers understand the semantic properties and performance characteristics of the library routines, will enable the implementation of interactive tools that analyze and make recommendations about the incorporation of library routines in their codes.

An even more significant challenge is for compiler researchers to extend this approach to online tuning. Evaluating the performance of different versions of a program by running them directly on the native machine is unlikely to scale to large numbers of cores or to large programs, even in an offline setting. One strategy is for compiler researchers to develop tractable machine and program abstractions to permit efficient evaluation of program alternatives, since what's needed is only the relative performance of program alternatives, rather than their absolute performance. The literature has proposed a variety of parallel computing models (such as the Parallel Random Access Machine, or PRAM, model for analyzing parallel algorithms and the LogP model for communication), but they are too abstract to serve as useful targets for performance optimization by compilers. However, through interactions with the theory and architecture communities, new models that more accurately capture today's multicore platforms should be developed. Such models could also be the foundation of a systematic theory of program parallelization and optimization for performance, power, throughput, and other criteria.

Breakthroughs in compiler technology are essential to making parallel programming mainstream.

Correctness and security. The ability to detect coding defects has always been an important mission for compilers. In facilitating programming, high-level languages both simplified the representation of computations and helped identify common program errors, including undeclared variable uses and a range of semantic errors pinpointed through increasingly sophisticated type checking. Much more can and must be done with program analysis to help avoid incorrect results and security vulnerabilities.

Regarding software security, Steve Lipner, senior director of security engineering strategy in Microsoft's Trustworthy Computing Group, said in a private communication: "With the beginning of the Trustworthy Computing initiative in 2002, Microsoft began to place heavy emphasis on improving the security of its software. Program-analysis tools have been key to the successes of these efforts, allowing our engineers to detect and remove security vulnerabilities before products are released. Today, Microsoft's engineering practices for security are formalized in the Security Development Lifecycle, or SDL, which mandates application of program-analysis tools and security-enhancing options. These tools and compiler options are the product of many years of research in program analysis and compilers, which has proven invaluable in addressing the difficult security challenges the industry faces. Microsoft's security teams eagerly look forward to the fruits of continued research in compiler technology and associated improvements in the effective-

ness of the tools that we use to make our products more secure."

Tools for program correctness and security must avoid wasting programmer time with false positives without sacrificing reliability or security and must prioritize, rank, and display the results of the analyses. Furthermore, they cannot negatively affect program performance or ease of use. As with other challenges, these issues are best addressed through a tighter integration of compilers, languages, libraries, and runtime systems. In particular, we anticipate an important role for specialized program-analysis systems that involve techniques relevant to each particular problem domain.

The foremost challenge in this area targets what traditionally is called debugging. The goal is to develop engineering techniques to detect and avoid program defects. The second challenge targets security risks, aiming to develop strategies to detect vulnerabilities to external attacks. The final challenge is to develop automatic program-verification techniques.

Enable development of software as reliable as an airplane. Improving the quality of software by reducing the number of program defects drives much research in computer science and has a profound economic influence on the overall U.S. national economy as indicated by the NIST report mentioned earlier. Compiler technology in the form of static and dynamic program analysis has proved useful in the identification of complex errors, but much remains to be done. Extending this work demands new program-analysis strategies to improve software construction, maintenance, and evolution techniques, bringing the programming process to conform to the highest engineering standards (such as those in automotive, aeronautical, and electronic engineering).

An effective strategy would likely involve analysis techniques, new language features for productivity and reliability, and new software-development paradigms. Nevertheless, at the core of these tools and strategies are advanced compiler and program-analysis techniques that guarantee consistent results while maintaining the overall quality of the code being generated, where metrics for quality might include execution time, power consumption, and code size. Beyond producing more reliable programs, the tools resulting from this research will make the profession of programming more rewarding by enabling developers and testers alike to focus on the more creative aspects of their work.

Enable system software that is secure at all levels. As with software reliability, sophisticated program-analysis and transformation techniques have been applied in recent years to the detection and prevention of software vulnerabilities (such as buffer overflows and dangling pointers) that arise from coding defects. There is some overlap between detection and prevention of software vulnerabilities and the previous challenge of software reliability in that any vulnerability can be considered a program defect. However, these challenges differ in that security strategies must account for the possibility of external attacks in the design of analyses and transformations. Thus, certain techniques (such as system call authentication and protection against SQL injection) are unique to the challenge of ensuring computer security.

Compilers play a critical role in enhancing computer security by reducing the occurrence of vulnerabilities due to coding errors and by providing programmers with tools that automate their identification and prevention. Programming in languages that enforce a strong type discipline is perhaps the most useful risk-reduction strategy. Functional language programs have more transparent semantics than imperative language programs, so language researchers have argued that the best solution to reducing software vulnerabilities is to program in functional languages, possibly extended with transactions for handling mutable state.

Enable automatic verification of the complete software stack. Formally proving that a program conforms to a given specification (program verification) is a powerful strategy for completely avoiding software defects. A manifestation is the longstanding recognition that program verification is one of the great challenges of computer science. More than 40 years ago, John McCarthy, then a professor of computer science at Stanford University, wrote: "Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program."[2]

Although program verification is not traditionally considered a compiler challenge, we include it here due to its potential as a formal solution to the two previous challenges: the interplay between program analysis and automatic verification and growing interest in the verification of compilers.

Verifying compiler code and algorithms would be a good first step toward addressing this challenge for two reasons: First, compilers contain specifications of their own correctness, thus providing clear requirements for the verification process. And second, the verification of the code generated by compilers is a necessary aspect of the verification of software in general. Recent advances in compiler verification anticipate a future when it will be possible to rely on formal and mechanical reasoning at the source level. The ultimate goal is to prove that the compiler is extensionally correct (input-output preserving) and respects time, space, and power constraints.

Although powerful and effective verification tools would make a tremendous contribution to computing practice, the importance of program verification goes beyond its use in improving software quality. As an example of a formal reasoning system, program verification is intellectually important in and of itself. It has been argued by researchers in machine learning that program verification is an ideal subject for the development of the first advanced reasoning system. After all, programming is a subject that computer scientists who study reasoning systems really understand.

Recommendations

To address these challenges, the compiler community needs a vibrant research program with participation by industry, national labs, and universities worldwide. Advances in compiler technology will require the creativity, enthusiasm, and energy of individual researchers, but given the complexity of compiler technology and software systems, long-term projects led by compiler specialists from industry and research institutions is necessary for success. Therefore, we offer four main

Tim Burrell of Microsoft's Secure Windows Initiative describing the Phoenix compiler and automated vulnerability finding at the EUSecWest conference, May 2008, London, U.K.

recommendations:

Enable the creation of compiler research centers. There are few large compiler research groups anywhere in the world today. At universities, such groups typically consist of a senior researcher and a few students, and their projects tend to be short term, usually only as long as a Ph.D. project. Meanwhile, the few industrial groups studying advanced compiler technology tend to focus on near-term solutions, even as the compilers, the programs they translate and analyze, and the target execution environments have increased in complexity. The result is that too much of today's compiler research focuses on narrow problems, ignoring the opportunity to develop revolutionary strategies requiring long-term commitment for their development and evaluation.

In those areas of compiler research seeing diminishing returns today from incremental approaches (such as program analysis and optimization), researchers must attempt radical new solutions that are likely to be lengthy and involved. The compiler research community (university and industry) must work together to develop a few large projects or centers where long-term research projects with the support of a stable staff are carried out. Industry and funding agencies must work together to stimulate and create opportunities for the initiation of these centers. Joint industry/government funding must support the ongoing evolution and maintenance of the software.

Create significant university/industry/government partnerships for the development of infrastructure and the gathering of benchmarks while funding individual projects at universities and other research centers. Implementation and experimentation are central to the compiler area. New techniques and tools, as well as new implementations of known approaches, can be meaningfully evaluated only after they are incorporated into industrial-strength compiler infrastructures with state-of-the-art optimizations and code-generation strategies. The absence of widely accepted compiler research platforms has hindered research efforts in compiler technology, design of new programming languages, and development of optimizations for new machine architectures. The development of complete compilers requires an immense effort typically beyond the ability of a single research group. Source code is now available for the GNU compiler and for other compilers developed by industry (such as IBM's Java Jikes compiler, Intel's Open Research Compiler, and Microsoft's Phoenix framework). They may yet evolve into the desired infrastructure, but none currently meets all the needs of the community.

Experimental studies require benchmarks that are representative of the most important applications at the time of the evaluation, meaning the process of gathering representative programs is a permanent process. Numerous efforts have sought to gather benchmarks, but the collections tend to be limited and are often complicated by doubts as to how representative they truly are. A serious difficulty is that many widely used programs are proprietary. In domains where open source applications might represent proprietary software, it would suffice for the purpose of evaluating compilers to make use of open source versions that outperform their proprietary counterparts; representative input data sets must accompany any set of benchmarks. Also desirable is a description of the algorithms and data structures to enable research in new programming languages and extensions that may be better suited for expressing the computation.

Federal agencies and industry must collaborate to establish the necessary funding opportunities and incentives that will move compiler specialists at universities and industry to address the research infrastructure and benchmark challenges. Funding opportunities should also be made available for smaller academic studies and development efforts. The computer science community has achieved notable success in developing novel compiler infrastructures, including: compiler front-end development; program-analysis frameworks for research in compilers; verification tools, security applications, and software-engineering tools; and virtual-machine frameworks for both mainstream and special-purpose languages. Even if the GNU and industry-developed compilers were a useful infrastructure for research, development of new robust, easy-to-use infrastructures by the researchers who need them are critical for future advances. Not only is it important to support such research projects, the research community must recognize their academic merit.

Develop methodologies and repositories that enable the comparison of methods and reproducibility of results. The reproducibility of results is critical for comparing the strategies, machines, languages, and problem domains. Too much of the information available to-

PHOTOGRAPH BY RYO HIROSAWA

day is anecdotal. In many disciplines, reviewers and peer scientists expect papers to include sufficient information so other groups are able to reproduce the results being obtained, but papers do not adequately capture compiler experiments where numerous implementation details determine the final outcome. With the help of open source software, the Web can be used to publish the software and data used in the evaluations being reported. Major conferences and organizations (such as ACM) must provide mechanisms for publishing software, inputs, and experimental data as metadata for the publications that report these experiments. Such repositories are useful for measuring progress and could also serve as a resource to those interested in the history of technology.

Develop curriculum recommendations on compiler technology. Compiler technology is complicated, and advances in the discipline require bright researchers and practitioners. Meanwhile, computer science has grown as a discipline with numerous exciting areas of research and studies to pursue. The compiler community must convey the importance and intellectual beauty of the discipline to each generation of students. Compiler courses must clearly demonstrate to students the extraordinary importance, range of applicability, and internal elegance of what is one of the most fundamental enabling technologies of computer science.

Whereas compiler technology used to be a core course in most undergraduate programs, many institutions now offer their compiler course as an optional upper-level course for computer science and computer engineering students and often include interesting projects that deliver a capstone experience. A good upper-level compiler course combines data structures, algorithms, and tools for students as they build a large piece of software that performs an interesting and practical function. However, these courses are considered difficult by both faculty and students, and students often have other interesting choices. Thus, fewer students are exposed to the foundational ideas in compilers or to compilers as a potential area only for graduate study.

Compiler algorithms are of tremendous educational value for anyone interested in compiler implementation and machine design. Knowledge of the power and limitations of compiler algorithms is valuable to all users of compilers, debuggers, and any tool built using compiler algorithms that encapsulate many, if not most, of the important program-analysis and transformation strategies necessary for performance and correctness. Therefore, learning about compiler algorithms leads to learning about program optimization and typical programming errors in a deep and rigorous manner. For these reasons, programmers with a solid background in compilers tend to excel in their profession.

One approach to promoting knowledge of compiler algorithms involves discussion of specific compiler algorithms throughout the computer science curriculum—in automata theory, programming languages, computer architecture, algorithms, parallel programming, and software engineering. The main challenge is to define the key compiler concepts that all computer science majors must know and to suggest the content that should be included in core computer science courses.

A second complementary approach is to develop advanced courses focusing on compiler-based analyses for software engineering, scientific computing, and security. They could assume the basic concepts taught in core courses and move quickly into new material (such as virus detection based on compiler technology). The challenge is how to design courses that train students in advanced compiler techniques and apply them to areas that are interesting and relevant (such as software reliability and software engineering). They may not be called "compiler courses" but labeled in ways that reflect a particular application area (such as computer security, verification tools, program-understanding tools) or perhaps something more general like program analysis and manipulation.

Conclusion

Although the compiler field has transformed the landscape of computing, important compilation problems remain, even as new challenges (such as multicore programming) have appeared. The unsolved compiler challenges (such as how to raise the abstraction level of parallel programming, develop secure and robust software, and verify the entire software stack) are of great practical importance and rank among the most intellectually challenging problems in computer science today.

To address them, the compiler field must develop the technologies that enable more of the progress the field has experienced over the past 50 years. Computer science educators must attract some of the brightest students to the compiler field by showing them its deep intellectual foundations, highlighting the broad applicability of compiler technology to many areas of computer science. Some challenges facing the field (such as the lack of flexible and powerful compiler infrastructures) can be solved only through communitywide effort. Funding agencies and industry must be made aware of the importance and complexity of the challenges and willing to invest long-term financial and human resources toward finding solutions. C

References

1. Kirkegaard, K.J., Haghighat, M.R., Narayanaswamy, R., Shankar, B., Faiman, N, and Sehr, D.C. Methodology, tools, and techniques to parallelize large-scale applications: A case study. *Intel Technology Journal 11*, 4 (Nov. 2007).
2. McCarthy, J. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, Eds. North-Holland Publishing Company, Amsterdam, The Netherlands, 1963, 33-70.
3. Merritt, R. Computer R&D rocks on. *EE Times* (Nov. 21, 2005); www.eetimes.com/showArticle.jhtml?articleID=174400350.

Acknowledgment

We thank Vikram Adve, Calin Cascaval, Susan Graham, Jim Larus, Wei Li, Greg Morrisett, and David Sehr for their many valuable and thoughtful suggestions. Special thanks to Laurie Hendren for organizing the discussion on education and preparing the text of the recommendation concerning curriculum recommendations on compiler technology. We gratefully acknowledge the support of the U.S. National Science Foundation under Award No. 0605116. The opinions and recommendations are those of the authors and do not necessarily reflect the views of the National Science Foundation. We also acknowledge all workshop participants whose insight and enthusiasm made this article possible, V. Adve, A. Adl-Tabatabatai, S. Amarasinghe, A. Appel, D. Callahan, C. Cascaval, K. Cooper, A. Chtchelkanova, F. Darema, J. Davidson, W. Harrod, J. Hiller, L. Hendren, D. Kuck, M. Lam, J. Larus, W. Li, K. McKinley, G. Morrisett, T. Pinkston, V. Sarkar, D. Sehr, K. Stoodley, D. Tarditi, R. Tatge, and K. Yelick.

Mary Hall (mhall@cs.utah.edu) is an associate professor in the School of Computing at the University of Utah, Salt Lake City, UT.

David Padua (padua@illinois.edu) is the Donald Biggar Willett Professor of Computer Science at the University of Illinois at Urbana-Champaign.

Keshav Pingali (pingali@cs.utexas.edu) is the W.A. "Tex" Moncrief Chair of Grid and Distributed Computing Professor in the Department of Computer Sciences at the University of Texas, Austin, and a professor in the Institute for Computational Engineering and Sciences also at the University of Texas, Austin.