Journal of Universal Computer Science, vol. 7, no. 5 (2001), 410-419
submitted: 14/4/01, accepted: 18/5/01, appeared: 28/5/01 © Springer Pub. Co.

Issues in Compiling

Gerhard Goos
(Fakultät für Informatik
Universität Karlsruhe, Germany
ggoos@informatik.uni-karlsruhe.de)

Hermann Maurer on occasion of his 60th birthday

Abstract: We consider the state of the art in compiler construction and where to go from here. Main topics are improved exploitation of present (and future) hardware features, the interaction between compiling techniques and processor design, and the use of compiling techniques in application areas such as component-based software engineering and software reengineering.

Keywords: Compilers, Software/Program Verification, Reusable Software, Program Transformation, Program Synthesis.

Categories: D.3.4, I.2.2

1 The Past

Compiler construction is the oldest part of software engineering. It delivered its first products for wide-spread use, FORTRAN-, COBOL- and ALGOL-compilers, etc. in the fifties and early sixties of the last century.

From its very beginning compiler writers tried to base their solutions on theoretically well-founded concepts such as finite automata, context-free grammars, stack automata, etc. Or to say it the other way around: the needs of the compiler writers were the starting point for much theoretical research which indeed lead to significant progress also on the practical side. With the use of attribute grammars for semantic analysis, abstract interpretation and data flow analysis as basis for code optimization, and the use of bottom-up term rewriting systems for code selection, this close interaction between theory and practice is continuing up to now.

Compiler construction also developed mastery in a number of techniques useful in many other areas of software construction. The most prominent example is the use of generators, e. g. parser generators, for constructing pieces of code from high-level specifications; all current automatic coding systems for generating code from decision tables, UML- or Statecharts-specifications or for designing user interfaces arose from this construction paradigm.

On a high level, compiler architecture for imperative programming languages has been fairly stable over the last decades, cf. fig. 1: source programs are analyzed and represented as parse trees which then are attributed for exhibiting

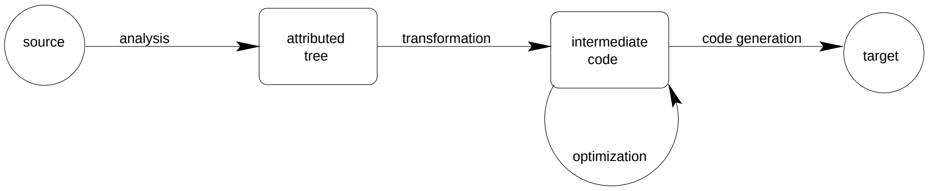

Figure 1: High level compiler architecture

semantic properties; from such an attributed tree an intermediate representation is generated which exhibits control and data flow; this representation is the basis for global optimization transformations and for code selection leading to the final target code. In real code the structure may look quite differently: E. g. one-pass compilers discard the tree and the intermediate representation as explicit data structures; nevertheless these data structures are present as sequences of arguments and results respectively of certain procedure calls. The methodology of systematically transforming a modular high level architecture into an implementation architecture adapted to the particular language and environment conditions is another major contribution of compiler construction to software engineering.

Compiler construction has been declared dead as a research topic on several occasions. The author experienced this first in 1965 when it was declared that compiler construction is now a topic for the software industry since everything which is of scientific interest is already known. The early nineties saw a similar situation: compiler construction was viewed by many as a highly specialized area which is not of general interest except for a few specialists in compiler construction companies.

This paper shows that on the one side there is a still growing number of research topics about compiling issues which are of high scientific and economic interest. It also shows that the poor mastery of compiling techniques has led to deplorable system designs and delays in problem solutions in a range of application areas. We mostly take a pragmatic view in which problems visible outside the reasearch community proper are valued higher. For this reason we mostly neglect the progress made in designing and processing functional and logical programming languages.

Instead we concentrate on the lessons to be learned from processing imperative (including object-oriented) languages and discuss the following issues:

- How could we arrive at dependable compilers, in particular for safety-critical systems?

- How to improve the efficiency and other qualities of the generated code?
- Interaction between compiler and processor design;
- The use of compiler technology in other application areas, in particular for program analysis.

2 Verifying Compilers

Despite all advances in quality assurance dependability of compiled code is still not satisfactory, cf. [Ste00, Sun00, Bor00]. Especially, programs often show different behavior with and without optimization. The generated code should exactly behave like the source program; this is of utmost importance especially in safety-critical applications.

The present state of the art, cf. [GZ99, GZ00], allows for using traditional compiler architecture and generators for lexers, parsers, etc. within a framework for verifying compilers. But the cumbersome work of verifying compilers for the currently widely used programming languages is still before us.

Verifying the correctness of compilers starts from *generally agreed* formal specifications of the programming language and the target machine under consideration. Here is another area of future research; in particular, both specifications should use the same formal method for avoiding the need of translating between different methodologies. For realistic results only the externally observable behavior must be specified and the influence of ressource limitations on the target machine must be considered.

It is often overlooked that a program text, like any other text, is a meaningless sequence of characters and nothing more. Meaning (or semantics) can only be associated after tokens and the underlying phrase structure is detected, i. e. after the parse tree is constructed. Whether the original program text and the parse tree carry the same semantics is a meaningless question.

This problem can be dealt with by help of a technique known as *program checking*, cf. [BK89, BK95, GGZ98]: We verify (for each program separately) that the inverse mapping parse tree → program text reconstructs the original text and, thus, that both are in a one-to-one correspondence. This technique of program checking can also be applied, e. g. to the optimization and code generation phase of compilers, and in general to many other transformation tasks, cf. [PSS98].

3 Optimization and Code Generation

The target code produced by todays' compilers for widely used programming languages is exploiting less than 50% of the available performance of the given processor-main storage system. Superfluous instruction sequences and cache misses, and inadequate exploitation of instruction-level parallelism and pipelines are the main culprits. With the arrival of new processor architectures and the widening gap between processor and storage speed the situation is dramatically worsening. The major causes for this sad state of affairs are a combination of unsatisfactory treatment of particular problems in optimization and code generation, and general deficiencies of properly engineering compiler back-ends.

Lexical analysis, parsing, semantic analysis and a simple-minded form of code generation, e. g. for a stack machine, as traditionally taught in compiler courses are all well understood problems. For them there exist theoretically highly sophisticated methods which in practice deliver highly efficient compiler modules. Breakthroughs in these areas can only be expected if radically different types of programming languages would be introduced. This does not say that innovative ideas about these problems, e. g. [PQ95], are impossible; but they could only extend an already highly competitive set of methods.

On the other hand these topics are only covering less than 40% of the code and execution time of a practically useful compiler. Some highly optimizing compilers spend alone 60% of their execution time in register allocation.

From an engineering point of view it is useful to subdivide the path from an attributed structure tree to the target code as in fig. 1 by inserting an intermediate representation IR which encodes the original program in terms of target machine semantics, i. e. the data types, basic operations and control flow are expressed in terms of storage words, operations on them and conditional jumps. The transition from an attributed tree to such an IR takes $O(n)$ on the average and $O(n^2)$ in the worst case.

Nearly every decision problem afterwards is however NP-complete or worse; in practice its solution can only be approximated. NP-completeness shows different faces: Some problems are linear on the average and difficult only in pathological cases; some are inherently difficult such as register allocation; some require additional profiling information for distinguishing frequent execution paths.

In the area of global optimization on the level of the IR there is a wealth of algorithms for achieving certain goals, cf. [Muc97, Mor98]; the most important ones are certainly constant folding, value numbering, partial redundancy elimination, cf. [MR79, DS88, DS93, KRS94, KCL$^+$99], and strength reduction. Value numbering is automatically taken care of when static single assignment form (SSA) is used as intermediate representation, cf. [CFR$^+$89, CFR$^+$91, CC95, Tra99]. Strength reduction is perhaps the oldest loop optimization and is available in several forms.

But what about all the other data flow driven optimizations published over the decades? And what about interprocedural optimizations? Some of them may be quite useful. Some are not worth to be considered separately; their effects are taken care of by a combination of other optimizations. The real problem is: a taxonomy of all this work is missing. This becomes particularly evident in the field of interprocedural optimizations: Many ideas and methods have been invented, demonstrated and their effects measured. But most of this work has been done in separation from other optimizations and mostly in an artificial compiling environment which does not resemble the standard compiler architectures and the intermediate representations used therein. As a result, the technical problems caused by storage explosion at compile time for large programs have largely been ignored. There are also only a few papers which consider the negative impact of separate compilation on interprocedural optimization.

These problems become particularly relevant when compiling object-oriented languages. For them moving objects from the heap into the stack based on points-to analysis, replacing polymorphic method calls by monomorphic calls, inlining of method bodies and interprocedural optimizations in cooperation with other optimizations may improve performance of the target program by factors of 10, i. e. it is possible to get from Java programs translated into native code similar performance as from C, cf. [Tra99, Exc99, Ins98]. But just-in-time compiling or any other scheme dealing with each separately compilable unit separately have no chance of arriving at such performance.

In code selection, register allocation and instruction scheduling there have been major advances over the decades. Especially the use of term rewriting systems, cf. [PLG88, Pro95, ESL89, Emm94], lead to a major advance. Instruction scheduling may be integrated with such schemes but proper integration with register allocation is still unsolved. Moreover, these schemes are mostly bound to consider expression trees within a basic block only; code generation , e. g. generating the MMX-instructions of an Intel-Pentium or certain specialized instructions of DSPs, would cover complete loops; it is quite difficult with such schemes. Another drawback is its unsuitablity of dealing with SSA form as intermediate representation: there we need a code generation scheme for directed acyclic graphs (DAG), not only for trees. Research on how to handle DAGs has stopped in the 70ties in favor of dealing with trees; engineers have continued the work for dealing DSP code selection; but their work has not gone far beyond a collection of "matching tricks", cf. [MG95]. Here a major (theoretical) breakthrough is needed, possibly based on graph rewriting systems. Such progress could probably also improve the code generation for DSPs.

Although instruction scheduling is a theoretically solved problem for present CISC- and RISC-architectures it is rarely applied in practice. The reason is that a different scheduler is needed for each member of a processor family, such as the Intel 80x86, Pentium, Pentium II, ... Dealing with all these processors separately would create a major versioning and maintenance problem; this problem would also lead to a huge versioning problem for all software distributed in binary form, starting from operating systems. Thus instruction scheduling is practically unsolved for economic reasons.

Similar remarks apply to optimizations dealing with cache access. This problem is especially important since good minimization of cache misses may reduce execution time by factors, far more than can be achieved by most of the other optimizations together. There exist a number of good optimization schemes for lowering the cost for array accesses. These are in use when compiling programs for scientific computations. But otherwise cache optimization is mostly neglected since there are no good schemes of dealing with navigational access to data structures such as lists or trees.

4 Compiling Issues and Processor Design

MOORE's law states that processor performance doubles every third year. To this end, processor designers and hardware architects not only explore advances in semiconductor manufacturing but especially continue to develop the internal architecture of processors and storage access systems. Goals of this development are, cf. [Soh01],

- increased clock rate: subdivide processor into fairly independent units;
- increased throughput independent of clock rate;
- instruction level parallelism (ILP);
- simultaneous multithreading (SMT);
- increased memory bandwidth

The development may lead to drastic changes in the internal processor architecture whereas the hardware/software interface as seen by the programmer, i. e. the assembly language, is changing only slowly and incrementally.

But most of the advances in processor design, e. g. ILP and multithreading, EPIC (Explicitly Parallel Instruction Computing) as used in the IA-64, data and execution speculation, etc. do not automatically lead to performance increases on the application level. The potential performance increase depends on the ability of compilers to exploit these processor features.

As an example, consider out-of-order execution on the PENTIUM-II/... and INTELs decisions for the IA-64: The PENTIUM-II does data dependency analysis on the next n instructions, $n \leq 40$, for selecting the instruction; this dynamic instruction scheduling may be supported by instruction scheduling performed by a compiler but it also works without such prior steps. On the IA-64, however, instruction scheduling is completely left to the compiler; the processor is performing quite slowly if the compiler has not done a good job on this issue.

Exploiting the VLIW-properties, predicated instructions, dynamic branch prediction (control speculation), speculative loads (data speculation) of the IA-64 are other examples where processor performance crucially depends on the cleverness of the compiler. With the advent of SMT, i. e. the subdivision of a sequential thread into several threads without overhead for context switch the dependency on compiler decisions will further increase.

These examples show that future performance improvements on the hardware side will crucially depend on further advances in optimization and code generation within compilers. Improvements need close interaction between processor and compiler designers. Wrong predictions about the potential of certain optimizations by compilers will lower the overall performance gain.

A decision about a future processor design thus will include a decision that certain kinds of optimiziations can be successfully performed by compilers for widely used languages such as C. However, such a decision can only be a follow-up to research about potential further code optimizations. Thus, research in compiler optimization will be crucial for maintaining MOORE's law.

Similar remarks apply to the development of DSPs and other embedded processors. There the question of control of power consumption and a potential adaption of execution speed to the current needs of an application will play a major role.

5 Compiler Technology in Application Areas

Compiling must not necessarily produce code in assembly language or code for a higher level abstract machine. It could also produce code on the register transfer level, logical VHDL designs or programs for field programmable gate arrays (FPGA). In hardware-software-codesign also (semi-automatic) partitioning of the FPGA-/software contribution and the dynamic loading of FPGAs can be generated by compilers. This has long been recognized by the hardware engineers; but the design of specification languages for describing FPGA applications, etc. on a high level and the translation of such specifications is still an area for much improvement. It should also be noted that hardware verification and compiler verification have a number of ideas in common.

Generally speaking, any task of text analysis with or without subsequent transformation is an application of techniques as developed for the analysis phase of compilers. In practice, however, the application of such techniques is often severely hampered by lack of knowledge of the technology on the side of the designers of the data or text layout.

This is, for instance, apparent in the design of DTDs (Document Type Definitions) and Schemas for the extensible markup language XML. A DTD or an XML Schema is describing the fully parenthesized structure of a XML document. A “XML-parser” should consider such a DTD or Schema as a grammar and parse the document accordingly. In practice, however, general interpreters read XML documents and validate them against the structure definitions; the generation of specialized parsers is not possible: The main problem for using generators for parsing and processing is that DTDs (or Schemas) are in general non-deterministic. XML documents conforming to a DTD (or Schema) may have different derivations. The DOM (Documents Object Model), the standard syntax tree definition for XML documents, abstracts from the different derivations. However, information computed already during parsing is not captured and must eventually be recomputed for transformations. Transformations are defined by XSLT (eXtensible Stylesheet Language Transformation). This language uses path expression but is mainly imperative and is also interpreted. More descriptive approaches — standard in compiler construction — like pattern matcher or term and graph rewite generators are not used and cannot be used because of the lack of explicite structure and type information in the DOMs. The way how namespaces are dealt with in XML and potential wild cards in grammars further complicate the situation.

That even parsing of context-free languages has not fully arrived on the market place is sometimes dependent on the working environment. When using emacs most so-called syntax-directed editor modes and program analysis tools are unable to properly analyze the context-free structure; they instead search with regular expressions and thus severly limit their capabilities. Commercial tools are mostly much better in this respect.

Program analysis for performance tuning and maintenance tasks, or for reengineering of software is another area heavily based on compiler technology. Here the use of front-end technology is standard. Interesting questions are concerned with the use of data flow analysis and other methods originally developed for code optimization in the context of program analysis.

Software reengineering and the construction of component-based software are two related topics which both may make heavy use of compiler technology. In reengineering, techniques for program analysis are used for rediscovering the structure of software for later adaption. When using components, i. e. independently deployable software-units, this structure is basically given but the interfaces of the components may not fit together. In both cases design patterns, cf. [GHJV94], or similar means may be used for achieving the adapted components. Such patterns may be applied by hand; if the process should be automated then the application of patterns can be considered as a source-to-source transformation based on semantic and data flow information as it is usually detected by compilers.

The use of frameworks, aspect-oriented programming and, more generally, any form of metaprogramming are topics which require similar means as just described. An open problem is the question to what extent the proper synchronization of components in a distributed application can be checked or even generated based on information as generated by a compiler.

6 Conclusions

We have illustrated the potential and needs of further research about compiler issues. On the side of compiling proper the needs are especially in the areas of optimization and code generation. On the application side there is a wealth of already existing knowledge about analyzing and transforming programs and other texts which is waiting to be exploited.

Acknowledgements

I thank Welf Löwe and Thilo Gaul for useful hints and improvements of a draft of this paper.

References

[BK89] M. Blum and S. Kannan, *Program correctness checking ... and the design of programs that check their work*, Proceedings 21st Symposium on Theory of Computing, 1989.

[BK95] Manuel Blum and Sampath Kannan, *Designing programs that check their work*, Journal of the Association for Computing Machinery **42** (1995), no. 1, 269–291.

[Bor00] Borland/Inprise, *Official borland/inprise delphi-5 compiler bug list*, http://www.borland.com/devsupport/delphi/fixes/delphi5/compiler.html, jan 2000, Delphi5 Compiler Bug List.

[CC95] Cliff Click and Keith D. Cooper, *Combining analyses, combining optimizations*, TOPLAS **17** (1995), no. 2, 181–196.

[CFR+89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, *An efficient method of computing static single assignment form*, Symp. on Principles of Programming Languages, ACM, 1989, pp. 25–35.

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadek, *Efficiently computing static single assignment form and the control dependence graph*, ACM Transactions on Programming Languages and Systems **13** (1991), no. 4, 451–490.

[DS88] K.-H. Drechsler and M. P. Stadel, *A solution to a problem with morel's and renvoise's "global optimization by suppression of partial redundancies"*, ACM Transactions on Programming Languages and Systems **10** (1988), no. 4, 635–640.

[DS93] K.-H. Drechsler and M. P. Stadel, *A variation of knoop, ruthing and steffen's lazy code motion*, SIGPLAN Notices **28** (1993), no. 5, 29–38.

[Emm94] Helmut Emmelmann, *Codeselektion mit regulär gesteuerter termersetzung*, Ph.D. thesis, Universität Karlsruhe, Fakultat fur Informatik, GMD-Bericht 241, Oldenbourg-Verlag, 1994.

[ESL89] H. Emmelmann, F.-W. Schröer, and R. Landwehr, *Beg — a generator for efficient back ends*, Proceedings of the Sigplan '89 Conference on Programming Language Design and Implementation, June 1989.

[Exc99] Excelsior, *The jet compiler*, http://www.excelsior-usa.com/jet.html, 1999.

[GGZ98] W. Goerigk, T.S. Gaul, and W. Zimmermann, *Correct Programs without Proof? On Checker-Based Program Verification*, Proceedings ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification" (Malente), Advances in Computing Science, Springer Verlag, 1998.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: Elements of reusable software components*, Addison-Wesley, 1994.

[GZ99] Gerhard Goos and Wolf Zimmermann, *Verification of compilers*, Correct System Design (Bernhard Steffen and Ernst Rüdiger Olderog, eds.), LNCS, vol. 1710, Springer-Verlag, 1999, pp. 201–230.

[GZ00] Gerhard Goos and Wolf Zimmermann, *Verifying compilers and asms*, Abstract State Machines, Theory and Applications (Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, eds.), LNCS, vol. 1912, Springer-Verlag, 2000, pp. 177–202.

[Ins98] Instantiations, *Super optimizing deployment environment for java*, http://www.instantiations.com/jove/, 1998.

[KCL+99] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow, *Partial redundancy elimination in ssa form*, TOPLAS **21** (1999), no. 3, 627–676.

[KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen, *Optimal code motion: Theory and practice*, ACM Transactions on Programming Languages and Systems **16** (1994), no. 4, 1117–1155.

[MG95] Peter Marwedel and Goosens (eds.), *Code generation for embedded processors*, Kluwer Academic Publishers, 1995.

[Mor98] Robert Morgan, *Building an optimizing compiler*, Butterworth-Heinemann, 1998.

[MR79] E. Morel and C. Renvoise, *Global optimization by suppression of partial redundancies*, Comm. ACM **22** (1979), 129–153.

[Muc97] Steven S. Muchnik, *Advanced compiler design and implementation*, Morgan Kaufmann Publishers, 1997.

[PLG88] E. Pelegri-Llopart and Susan Graham, *Optimal code generation for expression trees: An application of burs theory*, 15th Symposium on Principles of Programming Languages (New York), ACM, 1988, pp. 294–308.

[PQ95] Terence J. Parr and Russell W. Quong, *ANTLR: A predicated-LL(k) parser generator*, Software — Practice and Experience **25** (1995), no. 7, 789–810.

[Pro95] Todd A. Proebsting, *Burs automata generation*, ACM Transactions on Programming Languages and Systems **17** (1995), no. 3, 461–486.

[PSS98] A. Pnueli, O. Shtrichman, and M. Siegel, *Translation validation for synchronous languages*, Lecture Notes in Computer Science **1443** (1998), 235–??

[Soh01] Gurindar S. Sohi, *Microprocessors — 10 Years Back, 10 Years Ahead*, Informatics, 10 Years Back, 10 Years Ahead (Reinhard Wilhelm, ed.), Lecture Notes in Computer Science, vol. 2000, Springer-Verlag, 2001, pp. 209–218.

[Ste00] Reinier Sterkenburg, *Borland pascal compiler bug list*, http://www.dataweb.nl/ r.p.sterkenburg/bugsall.htm, feb 2000.

[Sun00] Sun Microsystems, *Sun official java compiler bug database*, http://java.sun.com/products/jdk/1.2/bugs.html, mar 2000.

[Tra99] Martin Trapp, *Optimierung objektorientierter programme*, Ph.D. thesis, Fakultät für Informatik, Universität Karlsruhe, 1999.

[WG84] William M. Waite and Gerhard Goos, *Compiler construction*, Springer-Verlag, 1984.