XGILF – A Conceptual Frame for Compiling and Linking Generic Libraries

Roland Weiss and Uwe Kreppel

Arbeitsbereich Computeralgebra, Wilhelm-Schickard-Institut für Informatik,
Universität Tübingen
Sand 13, 72076 Tübingen, Germany
{weissr, kreppel}@informatik.uni-tuebingen.de

Abstract The classical approach to compiler construction impedes development of languages that support the generic programming paradigm, e.g. C++ and Ada95. We will document the problems and describe an conceptual implementation frame that better serves this purpose. It is built around an XML based intermediate representation of the code. The key point in our approach is to defer code generation to link- or runtime. Moreover, we will show how the characteristics of our frame enable new high level optimizations, like runtime algorithm selection for generic functions.

Keywords: generic programming, algorithm selection, compiler construction, code generation, XML

1 Introduction

A typical compiler is divided into two major parts. First, the source program is lexically and semantically analyzed and transformed into an intermediate representation (IR) by the front-end. Code generation, the back-end's task, operates on this representation and outputs object code, which is executable, target specific machine code. The operating system usually provides linker[1] and loader to get these files into memory and to execute them. Figure 1 depicts this infrastructure, it follows [3].

But the two parts of a compiler, the front-end and the back-end, are perceived as one component by the developer, they were introduced primarily to enable a modular compiler implementation. Therefore, they are normally combined in one program. After creating the executable or library, the traditional compiler's work is done. No part of the compiler executes at runtime[2].

Today's C++ compilers adhere to this design, because they need to support the operating system's library and executable formats. This fact has a deep impact on their support for templates, C++'s generic programming facility. When

[1] In most cases, object files are stored in a proprietary, compiler specific format. Therefore, a linker from the compiler vendor turns them into libraries and executables conforming to the operating system.

[2] We speak of runtime from the very moment an application program is started by the user.

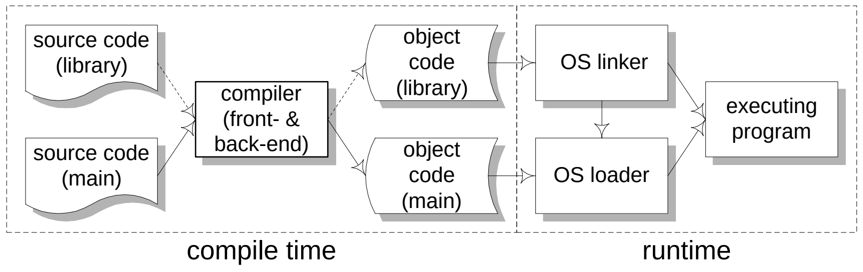

Figure1. Traditional compiler and operating system infrastructure.

one compiles a program that uses a generic library like the STL[3] (see [35] and [32]), all generic concepts, i.e. template classes and functions, must be instantiated with the application's concrete types.

One major problem here is that such libraries cannot be precompiled, as libraries are collections of object files, which contain only executable machine code. At this stage, all language specific and user defined types have to be resolved to machine level data types. Therefore, it is not possible to produce machine code for a generic concept, because the concrete type is not known at the library's compilation time[4]. This forces the C++ compiler to recompile generic concepts over and over again. The compiler implementors came up with different solutions for this problem. They vary from postponing compilation until instantiation time to storing instantiation templates in proprietary formats, which are not accessible to the user.

Furthermore, as C++ uses implicit instantiation, a good strategy to find all required instantiations is needed. Two strategies are common practice, the first one is based on interaction between the linker and the compiler, and the second one relies on a so-called repository (Stroustrup gives a good survey on this topic in [36], p. 365ff). Note that this is not true for Ada95, which forces the user to explicitly state every instantiation used in the subsequent code.

The conclusion we draw from these observations is obvious. The dynamic and static libraries used in most contemporary compiler and operating systems are not appropriate for holding generic libraries. In the rest of this paper we will present our alternative approach, which aims at overcoming this situation.

We propose to store generic libraries in an intermediate representation and defer actual code generation to link- or runtime. Of course, this proposal raises several important questions, because it influences the compiler and operating system design. We have to explain what kind of IR is employed, how the compiler is structured and the level of support the operating system has to provide.

[3] The STL is now part of the official C++ standard library.

[4] A uniform object layout can be assumed to circumvent this problem, but this is not possible in C++ with its built-in types, and it also results in bad runtime performance.

We want to make clear that we do not intend to present a completely implemented and runnig system. Rather, a solution to the instantiation problem is given in sections 2 and 3. Building on this proposal, we will try to identify all interdependencies among affected components and outline the remaining open problems and possible answers, reflected in sections 4 and 5.

2 The Intermediate Representation XGILF

The choice of the IR depends on several parameters. The higher the abstraction level, the more options for code generation are available because less information is lost, source languages are more easily mapped to it and one achieves good target machine independence. But it takes more time to produce code from a high level IR and data-flow analysis is more complex.

Moreover, there exist several designs for IRs, the most common ones are abstract stack machines, e.g. the Java Virtual Machine [29], register transfer lists, which are used in the gcc project, abstract assemblers (see [33]) and all flavors of tree languages.

We favored an IR that is closer to the source language than to the target machine in order to be able to perform high level transformations. This is important for generic programming, as depending on a generic function's input parameters, which can be the instantiation's types and their actual values, we may want to switch between different algorithm implementations of the same function. This application will be discussed in sections 4 and 5.

Another decisive motivation was the availability of tools that can handle our representation. Recently, a lot of work is being invested into XML [5], which is a textual tree representation at its core. So we adopted XML as our representation format. All XML capable browsers can be used to display it and the wide range of available software, e.g. parsers, class libraries and converters, is waiting to be exploited. The linking capabilities of XML make it an ideal choice for a library format, different parts of the library's components can be referenced inside the very file, inside the file system and even on the world wide web. Our IR is named XGILF, an acronym for XML-based generic intermediate link format. An IR formulated in XML has one more powerful feature, we get platform independence at the representation level.

XML, combined with a DTD, supports typing of its elements only in a rudimentary form, however we are interested in a strictly typed IR so that we can type safe instantiate generic concepts stored in the library. Therefore the concept's symbol table is also encoded in an XGILF file, together with its syntax tree.

3 The Infrastructure of a System Based on XGILF

We've just explained the content of an XGILF library or program file. The fact that no machine code is stored inside it necessitates drastic changes to the system's infrastructure, compared to the traditional one shown in figure 1.

In contrast to a traditional compiler, in our system its two parts are independent programs that operate at different times. The front-end performs the usual syntactical and semantical analysis and outputs XGILF. It is still called by the developer and runs at compile time. Instead of feeding this IR directly to the code generating back-end, we use it as the on-disk representation of libraries and executables. The abstract representation still contains template code which has to be instantiated. The code instantiations will be subject to traditional code generation[5] This code generation will take place at a later stage. There exist two natural possibilities: we can produce machine code at link time or at program startup time. In the case of dynamically linked libraries these two choices are virtually the same. We have exercised the startup option for several reasons. The most important ones are that we wanted the ability to modify an executable even at runtime (as we already mentioned) and to use dynamic libraries.

Thus, the operating system loader's task of reading an executable into memory and starting its execution is augmented by code generation. The code generator becomes a runtime component and turns into an operating system service. For proprietary systems, like Windows NT and various Unix flavors, one does not have the power to extend the system loader. In order to avoid this problem, we create two files for a main program. One is the XGILF file, holding its semantic information. The second one is a system conforming executable that starts our code generator. The task of the linker is quite simple, it has to extract requested segments out of an XGILF library file and turn them over to the code generator.

Let us ponder about the code generator some more. Michael Franz sketched a similar system in his dissertation [15], which was later implemented and publicized [16]. His incentive was to provide the foundation for a component based operating system, he was not concerned with the instantiation problem of generic concepts. He advocates a fast, consequently unsophisticated code generator, because the user will not accept perceivably prolonged startup times. Of course, code quality suffers in this approach upon the first launch of a program. We cope with this problem by introducing a code cache, which holds the latest version of a translated XGILF file. Now we can afford a long compilation run, which applies the whole range of traditional optimizations. For a main program, we can fill the cache immediately after the compiler front-end emitted its output[6]. This corresponds to code generation at link time. Figure 2 depicts our system[7].

The infrastructure just presented gives us the power to store precompiled generic libraries and create efficient instantiations of its generic concepts: At startup time all concrete types of the generic concepts are known and completly

[5] Note that instantiation itself cannot be called an *optimization*, real optimizations are only possible on the instantiated code. But we will use the term in this context anyway, because it is common practice to use the term optimization for many code transformations in the field of compiler construction.

[6] This policy makes only sense if the application is run on the same machine on which it is developed and compiled.

[7] In this and the following figures we use standard flowchart shapes: A rectangle denotes an active process, a rectangle with a waved lower line denotes a document or textual data and rectangle with bended left and right lines denotes stored data.

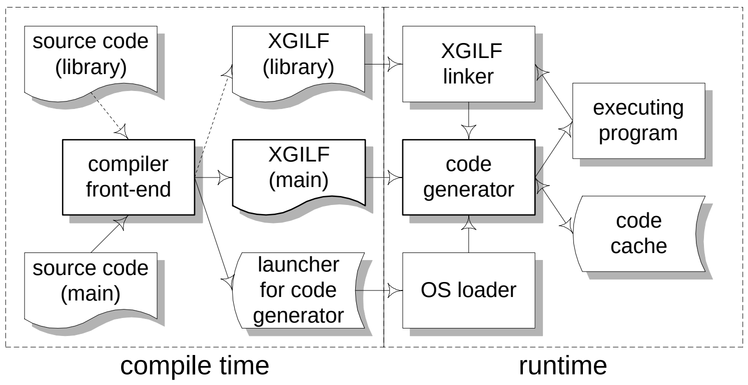

Figure2. Compiler and operating system infrastructure using XGILF.

resolved machine code can be produced from the XGILF representation, without using boxed representations of built-in types. This comes at the price of prolonged response times whenever XGILF code is requested whose architecture specific representation does not already reside in the code cache. Possible cases are startup time and whenever modules are dynamically linked. But owing to the late time of code generation, additional optimization opportunities arise.

4 New Optimization Opportunities

A target independent IR allows the system to emit code tailored for the processor architecture the program is actually running on. Nowadays this is important even for the same architecture, because different members have heavily varying characteristics. E.g., the family of processors compatible to the Intel Architecture evolved from a pure CISC architecture to a typical RISC architecture internally. Furthermore, most manufacturers of such processors are right now on their way to a 64-bit architecture, which will be needing other optimizations (alignment, cache locality etc). Also, new features are added constantly with every new processor generation, like MMX, Internet Streaming SIMD Extensions (see [22] for both) and 3DNow! [1]. Our approach can handle such varying system configurations in a clean way by compiling only those features into the executable code that are really available at the platform the program is launched on.

Another noteworthy point is the ability for intermodular optimizations. At runtime all active modules are known and data-flow analysis is not limited to their boundaries. The optimizer knows the context of a called algorithm and can perform individual manipulations like inlining, code movement and register allocation tailored to the situation.

Let us concentrate our main efforts on transformations specific to generic programming now. First, we will clarify the terminology. A *generic* function is a function where some or all of the input and output parameters' types can be chosen from a set of types. Thus, it describes a family of functions. A function *specialization* restricts these types to a special subset. Specializations can also be declared for special values of its input parameters, projections in a recursion-theoretical sense. The specialized functions still compute the same generic function, but for a limited set of parameters. Every function must have at least one *algorithm implementation*, i.e. an intrinsic algorithmic solution. Producing code for a fixed set of parameter types and values of a generic function is called *instantiation*. *Algorithm selection* is the process of selecting one algorithm implementation among possible candidates. We refer to instantiation combined with algorithm selection as *runtime instantiation*.

In C++ and Ada95, a generic function can have only one implementation. But out of performance considerations, a generic function can have different specializations. We want to go one step further. For a given function specification, our system accepts more than one intrinsic algorithmic solution[8]. Whenever the user calls such a function, the system's task consists of choosing the fastest algorithm out of a pool of algorithms, considering the current context. This duty is left to the programmer in current languages. Since we have all the information at hand provided by the XGILF file and runtime profiling data, we perform runtime instantiation, one of the main benefits of our approach. This leads to a system composed of a huge set of more or less specialized functions, providing on the one hand the possibility of having a fast version of the function instantiated. On the other hand, the algorithm selection is totally transparent to the library user.

It is also of great importance that the characteristics at the machine level have an effect on the algorithm implementation. Our field of special interest is computer algebra, where algorithms can run for days and switching between one function's different algorithm implementations at the appropriate trade-off points can have a deep impact on its runtime, e.g. using machine addition for numbers that fit into a few machine words and addition for numbers of arbitrary precision (see [39] and [18]) otherwise.

But let us have a closer look at the well known sorting algorithms. Depending on the type of a container whose elements should be sorted, it is reasonable to choose a different sorting algorithm, like quicksort for arrays and mergesort for linked lists and files. The generic sort function in the STL requires random access iterators and thus cannot be used with lists. This is a very clumsy way to handle this algorithm selection problem. Code that uses the sort function cannot deal with lists, because they have to be sorted with a member function, which has different syntax. Our system will select the appropriate algorithm based on realtime profiling data, which means an effective resolving of the genericity. This can happen at runtime, resulting in *runtime instantiation* of generic functions.

Not only the type of the container is a viable criterion for the algorithm selection, also the container's attributes like its elements' values and its size can help to choose a better algorithm. Since quicksort's complexity degenerates to $O(n^2)$ for certain input sequences, it would be better to apply another sorting

[8] Of course, also a specialization can have different implementations.

algorithm right from the start, if a frequent pattern in the sequence of elements stored inside a container could be determined. Of course, not only containers, but all other types, their values and additional parameters, like the length of a list or the degree of a polynomial, can expose significant information for algorithm selection. Even some well known findings from static complexity analysis or some previously developed heuristics can be useful. David Musser introduced introsort [31], a sorting algorithm based on quicksort that switches to heapsort if a certain recursion depth is reached while partitioning. This effectively protects against running into worst case. He uses a depth limit of $O(\log\ n)$, where n is the length of the input. Knowing this limit while implementing quicksort should enable us to switch to another sorting algorithm without having to implement introsort explicitly. Therefore, we need the help of a runtime profiler. It counts the recursion depth whenever an online optimizer requests this service. If the counter reaches its limit, the optimizer selects another sorting algorithm with guaranteed $O(n\ \log\ n)$ complexity to replace quicksort at runtime.

5 Enabling Effective Algorithm Selection

Obviously, the algorithm selection unit (ASU) itself, which comprises the selection algorithm, is one of the most important components in our system. The ASU retrieves data from the algorithmic database (ADB) and interpolates it. Compared to an offline algorithm selection procedure by the library desinger or user, it has to perform definitely better. To gain usable predictions we just cannot rely on static analysis, even if it is combined with runtime data. This situation is much alike the one analyzed in [38] where David Wall shows that estimated profiles are significantly worse than using real profiles to predict program behavior. To meet this observation, we use a runtime profiler to measure the exact executing time of the selected algorithm and store it in a database.

The algorithmic database is designed to be another central component. For each function specification, it holds the realizing algorithm names and their characteristics, which we need for making the selection. These are precomputed execution time[9], data from the runtime profiler, and known worst case indicators, to mention just a few.

For the ASU, initial data has to be provided to base the selection on. There are several methods considered how to collect these data. The ideal method would be to have for each algorithm an analytic computing time function of all its input complexity parameters. Such a function would be completely determined by a small set of constants depending on the environment, i.e. platform and compiler. These constants would have to be measured for each new environment while the analytic computing time functions would be the same. The other extreme would be, having no analytical time complexity term at all. In this

[9] While building an XGILF library, the execution times for some representative input parameters will be precomputed on user demand. So we get a first insight into the location of trade-off points and we get first data to use for runtime prediction through interpolation.

case test sets are needed for the complete range of expected applications of the algorithms. For each new environment these test sets must be used to determine the computing time functions empirically.

Between these two extreme cases – analytic and empircal computing time functions – intermediate ways of retrieving computing time functions are possible. For example, one could have analytical knowledge about the asymptotic computing times and empirical found knowledge for *smaller* input data. In this case one could try to regain the constants suppressed in the asymptotic analysis and supply the low order terms in order to get an analytic function for the whole range of inputs. For example, the asymptotic term $O(n \log n)$ would be expanded to $(a_1 n + a_0)(b_1 \log n + b_0) = a_1 b_1 \; n \log n + a_1 b_0 \; n + a_0 b_1 \log n + a_0 b_0$, and one can then apply the analytic method.

In other cases this approach is of little help because the computing time may depend on properties of the input which are hard to formalize. For example, sorting permutations of a fixed length n, the time may depend on the actual permutation of the elements or how close it is to be sorted. In these cases we compute the average execution time or use moments of higher order to base our decisions on. Since the prediction of execution times based on measurements is only really reliable if the input data does not change, we need some techniques of interpolation and extrapolation or other assumptions about the computing time function to cope with varying input data.

After using our system intensivly for some time, the ADB contains a lot of execution time data. As more data becomes available, the ASU can predict more exactly which algorithm will perform best. Hence, our profile-driven optimizing architecture improves the more it is used. Passing over to another architecture need not result in loosing all data stored in the ADB so far, if we transform it according to the methods discussed.

The whole architecture is based upon components. Due to the modular design, we can adapt the whole system to conform to the underlying hardware as well as to special needs of the running aplication. For example, we can replace a component in case of having found some better fitting selection criteria (replacing part of the selection unit) or when hardware changes so that other hardware performance counters will be provided (replacing part of the profiler). A new component may be added if a useful automatic complexity analyser will be available [9].

Concluding this section, we summarize the interaction of the components building our optimization system. We have a clear distinction between the static and the dynamic part of the optimizing process. In the static part, we record the complexity analysis of algorithms and their empirically found worst case indicators. This data and the source code are the input of the compiler front-end. On demand, timings of the new algorithms are precomputed after generating the IR.

The three major components of the dynamic part are the ADB, the ASU (part of the online optimizer) and the runtime profiler. The code generator queries the ASU for algorithms while compiling a function inside an XGILF file.

Then the ASU interacts with the ADB to perform several checks on the requested function (availability, uniqueness), keeps the ADB up to date and finally gets a list of all available algorithms realizing the function, together with their timings. Based on this results, the ASU selects the appropriate algorithm and sends its result back to the code generator. It also instructs the code generator where and what kind of profiling code should be inserted.

The profiler is constantly fed with live data from the executing program in order to detect time critical sections. If a worst case indicator is available and the profiler is instructed to watch this counter, it sends a message to the optimizer on overrun. The optimizer tries to enhance the critical sections. At the highest abstraction level, this means to select the most efficient algorithm instantiation for the current and maybe future parameters, i.e. the ASU selects the best fall-back algorithm and orders the code generator to exchange the algorithms. After an alogrithm has finished, the profiler sends its data to the ASU which updates its interpolation process and sends the new data to the ADB. Figure 3 shows the complete XGILF runtime system, including the components discussed in this section.

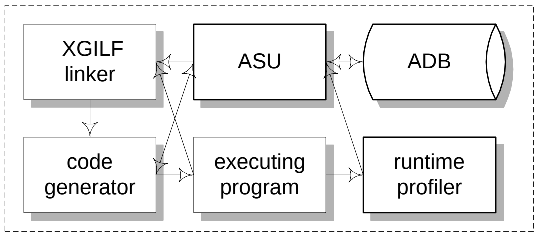

Figure3. The complete XGILF runtime system.

We did not mention two other integral parts of our runtime system, the garbage collector and the debugger. They will be covered separately as they represent independent components which have no direct influence on our conclusions described here. Also, the idea how the ASU is working and interacting with the ADB is only sketched here.

6 Implementation Context and State

Finally, we want to mention the context in which we are applying our framework XGILF. In order to be more than just a kind of type safe macro mechanism, generic languages must provide constructs to constrain the instantiation of generic concepts. This is not the case in a sophisticated way for todays generic languages, but common in the context of specification languages. Therefore SuchThat, our idea of a generic programming language, consists of an imperative and a declarative part, combining both side's advantages. Sibylle

Schupp details the imperative part in her dissertation [34]. The declarative part consists of a revised version of Tecton [30], which was developed mainly by David Musser.

A goal of our project is to implement a generic algebraic library, which takes its main algorithms from the SAC-2 system [7] and the saclib [21]. We want to lift the concrete algorithms to a generic form, e.g. there are 19 summation algorithms in the SAC-2 system.

An implementation of the specification and the type system checker by Rüdiger Loos is in a stable state for a while now. Main work is currently poured into getting the code generator to work. We use the XML parser Xerces C++ from the Apache XML project (see `xml.apache.org`) and produce standard C++ code right now.

7 Related Work

During the evaluation of contemporary programming languages that support the generic paradigm, we concentrated on C++ and Ada95, because they are the languages that have a significant user community and provide sufficient support for generic programming. However, C++ gains extraordinary attention because it was the first language in which Alexander Stepanov could implement his vision of a generic component and algorithm library [35]. See [13] for an implementation of the STL in Ada95 and a good comparison of the generic features present in C++ and Ada95.

C-- [33] and MLRISC [17] are both portable, abstract low level IRs that were introduced in order to provide a common front-end target language that can be easily retargeted for different processors. The National Compiler Infrastructure (NCI) is a major effort to streamline the construction of high quality research compilers. One of its contributions is ASDL [20], a domain-specific language for describing tree data structures. They recently extended their tools to use XML as storage format. SUIF [23] is another part of the NCI, an IR together with an API developed for high level optimizations. It is augmented by the Zephyr Infrastructure [4], which provides a low level optimizer. We consider reusing parts of the tools adaptable for our project. All these formats have in common that they were designed with a traditional compiler in mind. Java Bytecode [29] and Clarity MCode [28] are both developed for on-the-fly compilation and come very close to our needs. Unfortunately, they lack support for generic programming as they assume a fixed set of types.

There are two IRs that are very similar to XGILF in design, namely TDF [10] and SDE [15]. TDF is intended as abstraction of programming languages, in contrast to abstractions from target architectures like C-- and MLRISC. SDE, now integrated into the Oberon-3 system as *slim binaries* [16] by Michael Franz, was one of the main motivators for our system. We differ from their design in that we support generic concepts in our library, we have a textual instead of a binary representation and our optimizations apply at a higher level. Our current focus is on optimizing at the algorithm level, while they try to gain speedups by exploiting intermodular information, by reordering data members in memory and by optimizing instruction scheduling [24].

Dynamic optimizations and compilation have received some interest in the recent years. Mary Fernández [14] showed how to remove some overhead stemming from Modula's opaque type by deferring code generation to link time. At the University of Washington DyC [19], a C compiler, is being developed that allows dynamic code generation by annotating the C source code. Comparable approaches are being investigated at the MIT in the group around Dawson Engler and Frans Kaashoek and in the group of Charles Consel at INRIA. The MIT group extends C with special constructs for dynamic code generation, the language is called 'C [12]. It now employs a fast code generating back-end called VCODE [11]. Tempo is the compiler from INRIA [8], it also annotates C source code but is based on the GNU C compiler gcc and also does extensive code analysis to automate dynamic code generation. Finally, Mark Leone and Peter Lee worked on dynamic compilation for functional languages like ML [25]. They indicated possible optimization candidates with special functional constructs like curried functions [27]. Their work is continued in the Dynamo project [26]. At Sun, the Java team utilizes similar optimization techniques in the HotSpot engine [37] to improve the performance of Java applications. The Jalapeño group at IBM is building a Java system for SMP server machines that uses a dynamic compiler [6].

8 Conclusions

We presented a conceptual frame for a compiler and operating system infrastructure that efficiently supports generic programming. It overcomes implementation problems idiosyncratic to contemporary generic languages. Furthermore, we sketched a runtime systems which enables transparent runtime instantiation of generic functions. Transparent algorithm selection takes into account the often discrepant level of expertise on the side of the library developer and the library user.

Our main interest now is to gather hard data about our implementation and prove its viability. We also want to provide a binary version of our format and compare the perfomance impact of this solution to our XML version.

References

1. AMD: *3DNow! Technology Manual*, Order Number 21928, Advanced Micro Devices, Inc., 1999.
2. ANSI/ISO Standard: *Programming languages - C++*, ISO/IEC 14882, 1998.
3. Andrew W. Appel: *Modern Compiler Implementation in ML*, Cambridge University Press, 1998.
4. Andrew W. Appel, Jack Davidson, Norman Ramsey: *The Zephyr Compiler Infrastructure*, available at www.cs.virginia.edu/zephyr/papers.html, 1998.

5. Tim Bray, Jean Paoli and C. M. Sperberg-McQueen (editors): *Extensible Markup Language (XML) 1.0*, W3C Recommendation, W3C XML Activity, 10-February-1998.
6. Michael G. Burke, et al: *The Jalapeño Dynamic Optimizing Compilert for Java*, Proceedings of the ACM 1999 Conference on Java Grande: 129-141, 1999.
7. George E. Collins, Rüdiger G. K. Loos: *Specifications and Index of SAC-2 Algorithms*, Technical Report WSI 90-4, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 1990.
8. Charles Consel, François Noël: *A General Approach for Run-Time Specialization and its Application to C*, Proceedings of ACM POPL'96: 145-156, 1996.
9. Vincent Dornic, Pierre Jouvelot, David K. Gifford: *Polymorphic Time Systems for Estimating Program Complexity*, ACM Letters on Programming Languages and Systems 1(1): 22-45, March 1992.
10. DRA: *TDF Specification, Issue 4.0*, Defence Research Agency, Malvern, UK, 1998.
11. Dawson R. Engler: *VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System*, Proceedings of PLDI'96, ACM SIGPLAN Notices 31(5): 160-170, 1996.
12. Dawson R. Engler, Wilson C. Hsieh, Frans Kaashoek: *'C: A Language for High-Level, Efficient, and Machine-Independent Dynamic Code Generation*, Conference Record of the ACM POPL'95: 131-144, 1995.
13. Úlfar Erlingsson, Alexander V. Konstantinou: *Implementing the C++ Standard Template Library in Ada 95*, Technical Report, Computer Science Department, Rensselaer Polytechnic Institute, 1996.
14. Mary F. Fernández: *Simple and Effective Link-Time Optimization of Modula-3 Programs*, Proceedings of PLDI'95, ACM SIGPLAN Notices 30(6): 103-115, 1995.
15. Michael Franz: *Code-Generation On-the-Fly: A Key to Portable Software*, Doctoral Dissertation, Verlag der Fachvereine, Zürich, 1994.
16. Michael Franz, Thomas Kistler: *Slim Binaries*, Department of Information and Computer Science, Technical Report TR 96-24, University of California, Irvine, 1996.
17. Lal George, Allen Leung: *MLRISC – A framework for retargetable and optimizing compiler back ends*, described and available on the WWW at cm.bell-labs.com/cm/cs/what/smlnj/doc/MLRISC/.
18. Torbjörn Granlund: *GNU MP – The GNU Multiple Precision Arithmetic Library*, Edition 2.0.2, Free Software Foundation, 1996.
19. Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, Susan J. Eggers: *DyC: An Expressive Annotation-Directed Dynamic Compiler for C*, Technical Report UW-CSE-97-03-03, Department of Computer Science and Engineering, University of Washington, 1999.
20. David R. Hanson: *Early Experience with ASDL in lcc*, Software – Practice and Experience, 29(5): 417-435, John Wiley & Sons, Inc., 1999.
21. Hoon Hong, Andreas Neubacher, Wolfgang Schreiner: *The Design of the SACLIB/PACLIB Kernels*, Journal of Symbolic Computation 19(1-3): 111-132, 1995.
22. Intel: *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, Order Number 243191, Intel Corporation, 1999.
23. Holger Kienle, Urs Hölzle: *Introduction to the SUIF 2.0 Compiler System*, Technical Report TRCS97-22, Department of Computer Science, University of California, Santa Barbara, 1997.

24. Thomas Kistler: *Continuous Program Optimization*, Doctoral Dissertation, University of California, Irvine, 1999.
25. Peter Lee, Mark Leone: *Optimizing ML with Run-Time Code Generation*, Proceedings of the ACM SIGPLAN PLDI'97: 137-148, 1997.
26. Mark Leone, R. Kent Dybvig: *Dynamo: A Staged Compiler Architecture for Dynamic Program Optimization*, Technical Report 490, Computer Science Department, Indiana University, 1997.
27. Mark Leone, Peter Lee: *Lightweight Run-Time Code Generation*, Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation: 97-104, 1994.
28. Brian T. Lewis, L. Peter Deutsch, Theodore C. Goldstein: *Clarity MCode: A Retargetable Intermediate Representation for Compilation*, Technical Report TR-95-43, Sun Microsystems Laboratories, Inc., 1995.
29. Tim Lindholm, Frank Yellin: *The Java Virtual Machine Specification, Second Edition*, The Java Series, Addison-Wesley Publishing Company, 1999.
30. D. Kapur, David R. Musser: *Tecton: a framework for specifying and verifying generic system components*, Technical Report 92-20, Computer Science Department, Rensselaer Polytechnic Institute, 1992.
31. David R. Musser: *Introspective Sorting and Selection Algorithms*, Software – Practice and Experience, 27(8): 983-993, John Wiley & Sons, Inc., 1997.
32. David R. Musser, Atul Saini: *STL Tutorial and Reference Guide*, Addison-Wesley Publishing Company, 1996.
33. Simon Peyton Jones, Norman Ramsey, Fermin Reig: *C--: a portable assembly language that supports garbage collection*, Invited Talk, PPDP'99.
34. Sibylle Schupp: *Generic programming — SuchThat one can build an algebraic library*, Ph.D. thesis, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 1996.
35. Alexander Stepanov, Meng Lee: *The Standard Template Library*, Technical Report HPL-94-34, Hewlett-Packard Company, April 1994, revised October 1995.
36. Bjarne Stroustrup: *The Design and Evolution of C++*, Addison-Wesley Publishing Company, 1994.
37. Sun Microsystems: *The Java HotSpot Perfromance Engine Architecture*, Whitepaper, Sun Microsystems, Inc., 1999.
38. David W. Wall, *Predicting Program Behavior Using Real or Estimated Profiles*, ACM SIGPLAN Notices 26(6): 59-70, June 1991.
39. Sebastian Wedeniwski: *Piologie – Eine exakte arithmetische Bibliothek in C++*, Technical Report WSI 96-35, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 1996.