Compiler Techniques for Fast Migration of Embedded Applications

Thilo S. GAUL[1] and Günter SCHUMACHER[2]
*[1]Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, Zirkel 2
76131 Karlsruhe, Germany; Tel: +49 721 608-7398; Fax: +49 721 30047
Email: gaul@ipd.info.uni-karlsruhe.de*
*[2]Universität Karlsruhe, Institut für Angewandte Mathematik,
Postfach 6980, 76128 Karlsruhe, Germany; Tel: +49 721 608-2841;
Fax: +49 721 6087669; Email: guenter.schumacher@math.uni-karlsruhe.de*

Abstract: With a number of microprocessor architectures in use today, the flexibility to change from one target platform to another, in respond to market or customer demands, is decisive for competitiveness for application developers. In order to increase this flexibility, the Architecture Neutral Distribution Format (ANDF) has been developed within OMI (Open Microprocessor Systems Initiative). After some demonstrating applications, it turned out, that the availability of respective back-ends (installers) becomes the most crucial part of this technology.

During the OMI/SAFE project, an adequate solution has been found to this problem. By means of special compiler generating tools developed at the University of Karlsruhe, an installer for a specific platform can be provided with much less effort than before. The approach also allows to build configurable installers which is of great importance for families of microprocessors and for DSPs.

1. Introduction

The development of software technologies to improve portability has been identified as one of the key strategies in the software area within OMI. This is because the increasing importance of software reuse in all kind of microelectronics developments. For example, a number of silicon vendors in OMI have recently stated that they expect the number of software engineers in their development teams to grow from 15% to 50% in the next 2 years due to the increased supply of software and mixed software/hardware functions. Another fact is that adding programmable features to a chip requires libraries or a new compiler, or both. And last but not least, developers have to face the fact that IP design reuse could decrease product turnaround time to 6 months or less.

Consequently, many efforts have been spent to develop software reuse techniques, among which the Architecture Neutral Distribution Format (ANDF), a pure European development, is going to become a world-wide recognised standard for exchanging software components among different platforms. As the standard not yet established and other assumed alternatives such as Java coming up, potentially interested industry was going to drop ANDF. This was mainly due to the lack of practical experience how ANDF favours embedded system development and how it fits into traditional development environments.

In the meantime, two European funded projects (OMI/ANTI-CRASH and OMI/SAFE) have demonstrated the superior nature of ANDF as a technology. It turns out that through ANDF not only the software functionality can be ported to other platforms but also the quality aspects. It also features properties which are predestined for the special needs of safety critical applications. Nevertheless, platform independence always implies higher intelligence of the corresponding compiling tools which normally means higher complexity and higher costs as well. Whilst higher compiler costs have to be compared to the reduced costs of software porting, the increasing complexity of such components have still to be considered.

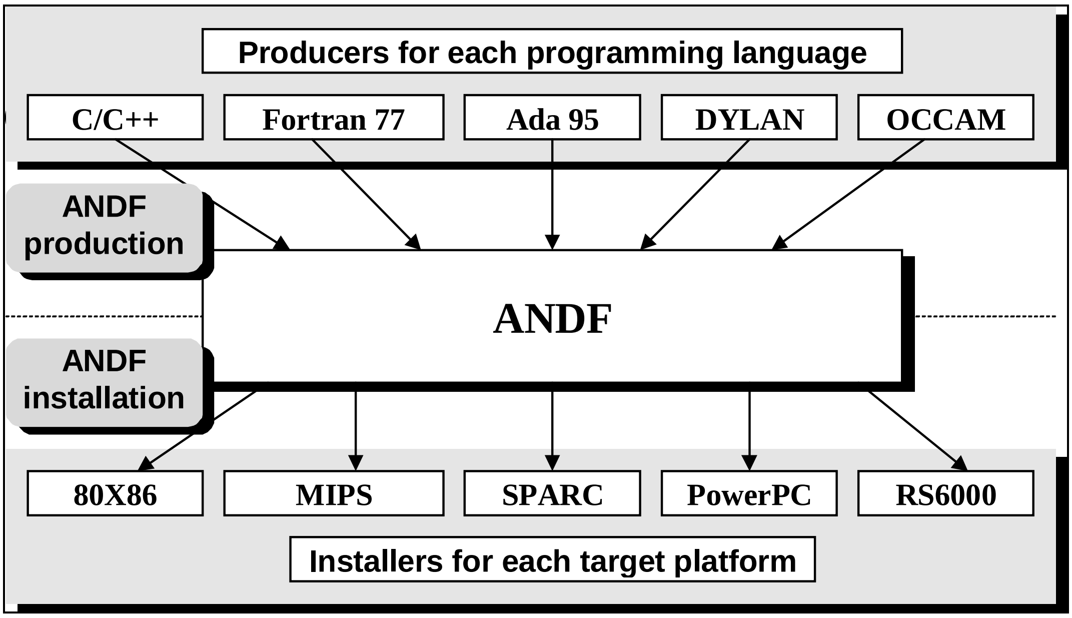
This was the starting point for an investigation towards automatic generation of compiler components within the OMI/SAFE project. The goal was to evaluate the feasibility of using compiler-generating tools with respect to

- development time for back-end components
- architectural flexibility through parameterisation
- output quality of the generated component (code size, performance)

In this paper, we describe the first promising results that are expected to give new stimulation for promoting ANDF as an international standard.

2. The Architecture Neutral Distribution Format

In its original meaning, Architecture Neutral Distribution Format means just the idea of a platform independent format. In 1989 the OSF issued a Request for Technology for an implementation of an architecture-neutral format, an intermediate language to support application portability. Among 15 qualifying submissions, OSF announced in 1991 that they had selected a subset of TDF developed by the Defence Evaluation and Research Agency (U.K.) to be the core technology of ANDF. Therefore, TDF (TenDRA Distribution Format) is now *the* ANDF and no distinction is done between these two terms.



Besides the definition of an intermediate language, the term *ANDF Technology* means the rigorous separation of the front-end (taking the source program) and the back-end (generating the binary) of a compiler. Any such front-end (called *producer*) is intended and designed to be target independent whilst the back-end (called *installer*) is language independent.

Any API (for any source language) therefore has a source level definition and a platform specific definition (provided together with the installer). APIs are closely related to standard programming libraries like ANSI C, X11 or POSIX. Because a producer only uses abstractions of APIs, no further assumptions about the architecture on which the program is supposed to be run are necessary.

Since the first release of TDF (as ANDF), several activities have been established to provide components for the ANDF technology, i.e., producers, installers, validations suites, etc. A reasonable part of these activities have been funded under the ESPRIT programme. Therefore, the ANDF technology must be considered as a real European development. At times when standards become more and more important, a successful European standard for – generally spoken – real-time interfaces would bring greater focus on European providers of respective technology. Although this is rather ”psychological” since ANDF is open for anyone, examples like Java demonstrate the existence of this effect.

The lessons from Java have brought up another interesting aspect. Java is currently considered as something like an “ANDF”, even in the real-time area – although it is quite different from the ”real” ANDF. In fact, Java is restricted to C-like constructs while it also features C-like uncertainties. On the other hand, ANDF (both as a language and as technology) features language-independence while it enables also safe programming, a fact which has been proven by the OMI/ANTI-CRASH project. Nevertheless, many companies have started to focus on Java while ANDF is technically still the better alternative.

Therefore, a European driven ANDF standardisation (as already started at the ISO level) is mandatory and urgently recommended. To overcome the infamous “chicken-and-egg” paradigm (no standards without industrial interest, no industrial interest without standards), more demonstrating actions have to be established as the one proposed by this project. The project consortium will also closely co-operate within the newly established ANDF-Club, a special interest group that acts as a forum for all common ANDF activities.

ANDF was always said to be too big and complicated, too much parameterisation. This is true in a compiler environment where only one language is translated to a small set of target architectures; in this case the intermediate representation can be driven by the features of the target machine. But the more programming languages have to be integrated into this simple framework, the more general the intermediate language has to be. ANDF was designed to be a most general exchange platform, architecture neutral in the sense that it provides a real superset of most intermediate operators and is widely parameterisable in most architecture dependent language features. This allows building a compiler system for a lot of different source languages and target machines, which always uses the same compiler infrastructure. ANDF as an *m* to *n* interface between the various combinations of *m* front-ends and *n* back-ends assures, that a lot of code can be reused, especially transformations and optimisations on intermediate language level.

3. ANDF Based Compiler Construction

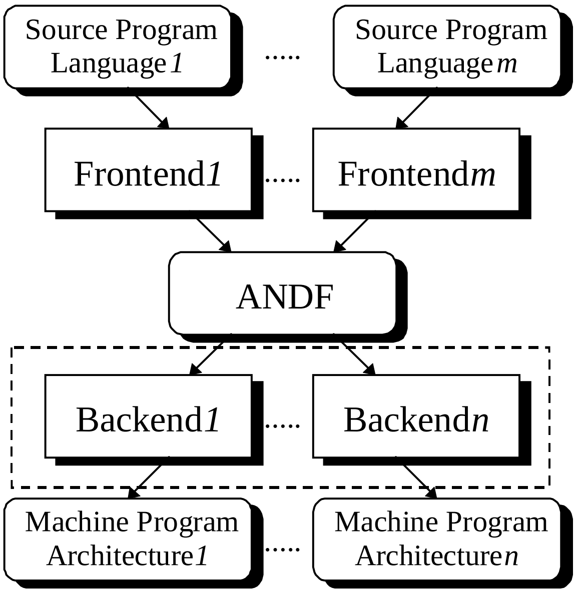
The last decades of compiler construction research have produced a lot of fancy techniques for the construction of fast, safety or highly optimising compilers but the very few have come to an industrial relevance. The best chance for such a technique to be used in practice is to be integrated into a generator tool. The best known examples of such techniques are deterministic finite automatons for lexical analysis and stack automatons for the analysis of context-free languages. Nowadays everyone who deals with language translations knows the corresponding tools LEX and YACC (and their derivatives) that use these techniques.

The main aspect is, that the mentioned techniques found their way into generator tools, which generate concrete parts of a compiler from easy to maintain and extendible specifications. Nowadays every programming language description comes with a specification in EBNF, from which a YACC specification can be derived easily. This is not the case for other parts of the compiler and most of industrial relevant compiler systems are still hand-written.

The tool approach we present in this paper shows a further step in the automation of compiler construction.

3.1 The Compiler Framework

The compiler framework developed in OMI/SAFE was designed to maximise reuse and reliability. This „developers best friend“ goal is achieved by:


Source Program Language *1*
.....
Source Program Language *m*
Frontend*1*
.....
Frontend*m*
ANDF
Backend*1*
.....
Backend*n*
Machine Program Architecture*1*
.....
Machine Program Architecture*n*

- dividing the compiler into well manageable phases
- dividing phases into language and architecture dependent and independent parts
- generating compiler parts from specifications

First the compiler is divided classically into a front-end and a back-end where ANDF serves as the intermediate language. This is not only a conceptual subdivision, but this is a concrete interface where different front-ends and back-ends can be exchanged - even dynamically. In a concrete development framework this reduces the amount of combinations of front-ends with back-ends from *m*n* to *m+n* and thus reduces the costs for porting the compiler to new architectures or languages. ANDF programs produced by the front-end can also be saved as binary files, which can be distributed and translated further with any ANDF-back-end, without any knowledge about the language they were produced from. The feature of being able to distribute binary coded intermediate programs is similar to Java-Byte-Code, with the difference that the latter is neither independent of the source language nor architecture neutral.

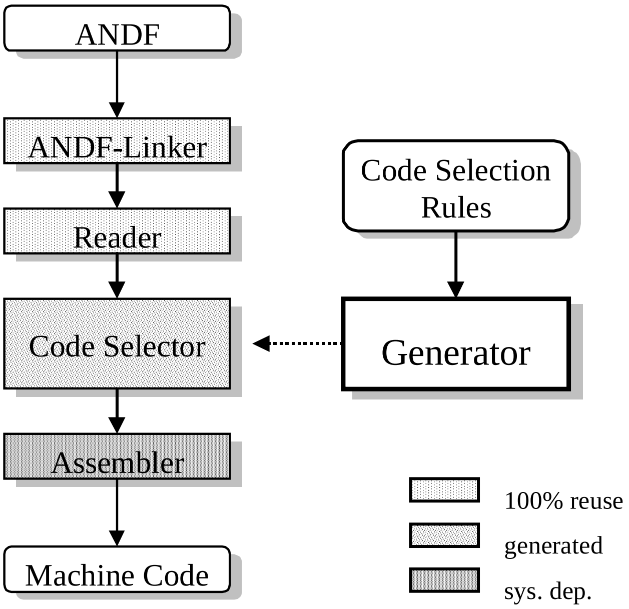
In the rest of this paper we will concentrate us on the back-end part (*installer*) of such a compiler and the generator techniques used here.

3.2 Back-end Architecture

The main aspects at the construction of compiler back-ends are retargebility and reliability. Efficiency of the generated code is also an import point for embedded systems, but unlike to code generation for high-performance workstations memory considerations are often more important. The generator approach used in OMI/SAFE allows to optimise code generators for both runtime efficiency and memory consumption.

Following ANDF mechanisms to divide an installer into architecture neutral and architecture dependent parts, our approach performs a stepwise transformation from „high-level“ ANDF to low-level machine code:

1. Read and link the architecture neutral ANDF code together with machine dependent ANDF libraries and application programming routines (*Linked-ANDF*)
2. Select target machine code for *Linked-ANDF* programs (code selection)
3. Assemble and bind produced code to executable programs



The latter is a standard job for a system specific assembler/linker tool chain and is normally provided by the target machine manufacturer. Implementations of task 1 (ANDF-Linker and Reader) can be reused at 100%, because they do not depend on the target and are implemented architecture neutral. Several C-implementations are available, one as a result of the OMI/SAFE project. The code selection phase (task 2) performs the mapping of data types and operations to the target machine while trying to use target resources optimally. Obviously this is the most tedious task to implement and tool support is urgently required.

3.3 The Generator Technology

There exist a variety of techniques that address the problem of matching machine code to intermediate languages. Common methodology is to specify source and target language terms, which are related by code selection rules annotated with costs. From those specifications a cost controlled rewrite system is generated, that implements the code selector. The mechanism assures that always the cost minimal code - memory consumption or execution time - is selected. Efficient tree transducers or bottom up rewrite systems achieve practicability. The user of the generator does not have to bother with the generated transducer system, he just has to assure, that the specified rule set is complete w.r.t. to the input language (ANDF) and of course, that the single rules are locally correct.

Most powerful machine instructions can be used not only to implement one node of the program tree but several nodes at the same time. In order to take full advantage of this instruction set property the declarative specification of the code generator describes machine instructions by tree patterns. This is done by defining rules. Each rule describes a node pattern and the corresponding sequence of processor instructions, which will be the output for this pattern. In order to produce code for the entire expression tree, the code generator picks out a suitable set of rules so all nodes are covered once. Now the tree is traversed in postfix order and for each rule of the set the corresponding machine instructions are emitted.

Many processors have an ample instruction set, which may lead to a lot of different, possible covers. These covers are all correct, but the results may have not the same code quality. In order to select the best cover, each rule has the above mentioned cost statement. The code generator computes the total cost of each possible cover by adding the costs of all rules belonging to the cover. Then the cover of minimal cost is chosen and for this the code is produced.

Several generators with industrial relevance have been built in the recent years and are now included in compiler toolboxes (BURG, IBURG/MBURG, PAGODE, BEG). The back-end generator BEG is the tool with most user support and is complete in the sense, that it is possible to specify the whole code generation process. BEG produces highly efficient code generators, includes several register allocators and also generates instruction schedulers from specifications.

BEG was developed and used in ESPRIT-project COMPARE and is now maintained and sold by H.E.I.-Informationstechnik, Germany. The commercial version comes with full support, a public domain version with less features is also available. The practicability has shown up in several compiler projects (COMPARE, MOCKA, Sather-K, Java-Byte-Code) where code generators for different processors (VAX, 68k, Transputer-T800, MIPS, Sparc, PowerPC, Pentium) were produced.

4. First results

This paper also reports first results and experiences of implementing ANDF back-ends with the new generator approach. We will give an overview on human resources and technical results of the first phase of the installer part within the OMI/SAFE project.

A code generator consists of intermediate language specific and target machine dependent parts. The language part models the input representation and performs optimisations on ANDF-terms. This part can be reused 100% for a new compiler and will be available in the public domain and also commercially as a result of the OMI/SAFE project. The machine dependent part has of course to be adapted for a new architecture, but the specification mechanism allows to concentrate on the target machine facilities. The compiler writer does not have to bother with the transformation process itself, but he can concentrate on single aspects and local transformations.

Table 1 gives an overview on the usage of human resources needed related to lines of specification and C-code produced.

		% of total man-power	lines of code-gen.-spec.	lines of C-code (generated)
Reader		20%	---	17.000
ANDF-specific code-gen. part		40%	1.500	10.000
Target-specific code-gen. part	Architecture dependent	40%	800	25.000
	Processor dependent		400	8.000

Tabelle 1 Human Resources

Architecture dependent means, that this part only depends on the target architecture or family, not on the concrete processor.

These first results show, that on the one hand the specification mechanism is very powerfull – relation from lines of spec. to lines of C-code is at least 10 to 1 – and on the other hand that the biggest part can be reused for a new architecture or processor family.

5. Conclusions

First results in the OMI/SAFE project show, that a code generator technique like BEG is especially well suited for complex intermediate languages like ANDF and for embedded system processors with complex instruction sets, register files and addressing modes.

Compared to hand written compiler back-ends, the usage of code generator tools decreases the retargeting time to a new architecture significantly as well as it improves reliability.

The main benefits of using a compiler generator for embedded systems are retargebility and reliability. Comparing hand written back-end implementation and writing a declarative description the latter has several advantages:

- the description text is distinctly shorter than the source text of a conventional hand made back-end (the current version is generated from approx. 2700 lines of specification)
- specifying the code generation at a higher declarative level is more convenient for a programmer, because it improves understanding and communicativeness; even complicated details can be described by rather simple rules
- writing a description takes less time and effort because it is shorter and clearer; the programmer can concentrate himself to local aspects, the global mechanism is generated and assures correct compilation
- the division of the code generation process into several phases (rewriting, covering, code emission) allows an easy debugging of the code generator itself
- the BEG performs consistency checks on the description; this improves completeness, correctness and reliability of the produced code.
- It supports cost controlled code generation, which allows to produce locally optimal code (w.r.t. time economy or resource economy) and to come close to the program global optimum.
- adding new correct rules can not cause any worsening; extensions introduced to improve code quality of a particular coding problem won't suppress any other minimal cover found correctly before the extension
- the correctness of the produced code can be approved easily by simply proving local correctness of single rules
- because BEG generates automatically a register allocator, it is no longer necessary to design and to implement one, that saves additional time.

Especially the correctness aspect is very important for reliability of safety critical applications and a lot of work has been done on this area, too. For example in [5] we showed how to prove the whole code generation process correct on the basis of local correctness. The whole code generation specification can be verified against the semantics of source and target language, which results in an most reliable compiler specification. Such a compiling specification can then be implemented correctly by techniques described in [6] and others. The whole generator can be proven correct, especially with the back-end techniques described in this paper.

References

[1] Helmut Emmelmann, Code selection by regularly controlled term rewriting. In R. Giegerich and S.L. Graham, editors, *Code Generation - Concepts, Tools, Techniques*, Workshops in Computing. Springer-Verlag, 1992, S. 3-29

[2] H. Emmelmann, F.W. Schroer, R. Landwehr: BEG - a Generator for Efficient Back-Ends, Proceedings of the Sigplan’89 Conference on Programming Language Design and Implementation. Portland, Orgeon, June 21-23, 1989, Sigplan Notices, Vol. 24, Number 7, July 1989

[3] Albert Nymer and Joost-Pieter Katoen. Code Generation based on formal BURS theory and heuristic search. Technical report inf 95-42, University of Twente, 1996

[4] Todd A Proebsting. BURS automata generation. ACM Transactions on Programming Languages and Systems, 17(3):461-486, May 1995

[5] Wolf Zimmermann and Thilo Gaul. On the Construction of Correct Compiler Back-Ends: An ASM Approach. *Journal of Universal Computer Science (JUCS*), 3(5):504-567, 1997

[6] Wolfgang Goerigk and Axel Dold and Thilo Gaul and Gerhard Goos and Andreas Heberle and F. W. von Henke and Ulrich Hoffmann and Hans Langmaack and Holger Pfeifer and Harald Ruess and Wolf Zimmermann. Compiler Correctness and Implementation Verification: The VERIFIX Approach, *International Conference on Compiler Construction*, *1996*, Linkoeping, Sweden.

[7] H.S. Jansohn: Automated Generation of Optimized Code. GMD-Bericht Nr. 154, R.Oldenbourg Verlag, 1985

[8] A.V. Aho, M. Ganapathi, S.W. Tjiang: Code Generation Using Tree Matching and Dynamic Programming. 1987

[9] A. Balachandran, D.M. Dhamdhere, S.Biswas: Efficient Retargetable Code Generation Using Bottom-up Tree Pattern Matching, Computer Languages, 15(3), 1990, S. 127-140

[10] R.S. Glanville: A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers, PhD Thesis, University of California, Berkeley, 1978