Advanced Generator Techniques for Embedded Compilers

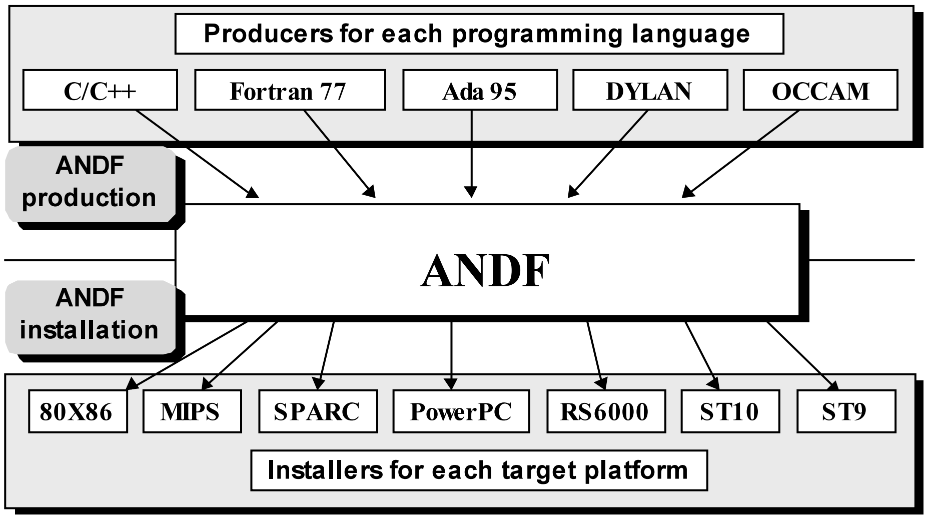
Thilo S. GAUL[1] and Günter SCHUMACHER[2]
[1]*Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, Zirkel 2 76131 Karlsruhe, Germany; Tel: +49 721 608-7398; Fax: +49 721 608-9095 Email: gaul@ipd.info.uni-karlsruhe.de*
[2]*Universität Karlsruhe, Institut für Angewandte Mathematik, Postfach 6980, 76128 Karlsruhe, Germany; Tel: +49 721 608-2841; Fax: +49 721 6087669; Email: guenter.schumacher@math.uni-karlsruhe.de*

Abstract: As regards competitiveness, flexibility to change from one target platform to another is decisive for application developers, especially in the area of embedded systems. The Architecture Neutral Distribution Format (ANDF), developed and evaluated within OMI (Open Microprocessor Systems Initiative) has turned out to be a key technology to improve this flexibility. The basic idea is to break compilers into front-ends (for specific languages) and back-ends (for specific microprocessors) where both pieces easily could be replaced by a "plug-and-play" compatible component.
Recently, the ANDF technology has been applied to standard embedded application domains and even to safety critical applications in several ESPRIT-projects with clearly visible benefits for developing time, costs and code reliance. The availability of compiler back-ends (installers) turned out to be the most crucial part of this technology. During the OMI/SAFE and OMI/FAME projects, a new generation of compiler generator tools has been applied, which address the generation of compiler back-ends. With such a back-end generator tool - developed at the University of Karlsruhe - an installer for a specific platform can be provided with much less effort than before. This approach also allows to build configurable installers which is of great importance for families of microprocessors and for DSPs.
In this paper we will show first practical results of the OMI/SAFE and OMI/FAME projects, with emphasis on measurements of human resources on the one hand and efficiency of the produced code on the other hand, compared to standard compilers.

1. Introduction

Although the term "ANDF" spells as "Architecture Neutral Distribution Format", a platform independent exchange format for program code, it summarises under its title a collection of high-quality development tools and, in fact, a whole development technology. The split into front-end and back-end compilation derives new possibilities of code management at the intermediate level, thus improving both code quality and portability.
With these abilities, ANDF turns out to be the key for embedded software development. While there are only few languages in use there exists a large number of different microcontrollers. At least in principle, it is possible to simply plug in a new backend (installer) into an existent tool chain in order to use a new target platform. It has been demonstrated by several industrial projects that the full theoretical benefit can actually be achieved. These evaluations have made available realistic performance figures for ANDF components along with indications on the possible financial benefits when using them.
Having identified the backend as the critical component, the use of compiler generating tools seems to be mandatory. In fact, it has been demonstrated for several targets that

automatic generation can reduce both the development costs and time for a backend by a factor of 3 to 4. In addition, this technique improves long term maintenance, supports "generic" backends and it allows compiler verification.
Backend generators that provide the necessary level of quality for the embedded market



(measured in code size and performance of the compiled code of the generated backends) have just entered the market. University of Karlsruhe and its industrial partners are currently owing one of the most mature generators, called BEG [1,2]. While this tool has already been successfully used to generate installers, a new generation of backend generators is under development which uses graph rewriting techniques rather than tree rewriting. With these novel approach additional features are possible such as globally optimising code generators or further support for object-oriented languages.
In the following we describe the latest technical achievements and performance figures when using BEG in practice to generate ANDF backends.

2. ANDF Based Compiler Construction

Since the first release of 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.

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.

This most general approach usually implies a hard to maintain compiler framework which results in huge costs. In this paper we will show, that it is nevertheless possible to handle a compiler framework with such a general intermediate representation by using modern generator technology, that eases maintenance and reduces development costs.

2.1 Generator Technology

Compiler construction is one of the best exploited areas of computer science and a lot of techniques and methods have been developed for the construction of fast, safe and optimising compilers. To transfer the theoretical results to the practical software engineering, it is necessary to integrate them into tools, to make it even possible to use them. Well known examples of such transfers to practical needs are deterministic finite automatons for lexical analysis and stack automatons for the analysis of context-free languages. Everyone who deals with language translations knows the corresponding tools LEX and YACC (and 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 compiler systems are still hand-written.

In this paper we demonstrate a further step in the automation of compiler construction. Our focus lies on the construction of code-generators, in the case of an ANDF-framework this is the backend of the ANDF-tool-chain, the installer. In the current ESPRIT-projects OMI/SAFE and OMI/FAME we show, how new generator tools for code selection can be efficiently used to generate the whole code-selection phase of compiler backends.

2.2 The Compiler Framework

The compiler framework developed in OMI/SAFE was designed to maximise reuse and reliability. We achieve this goal by:

- the compiler is divided into well manageable phases
- the phases of the compiler are divided into language and architecture dependent and independent parts
- architecture depended and optimisation critical parts of the compiler are generated from specifications

According to the general ANDF approach, 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 number of front-end/back-end combinations 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.

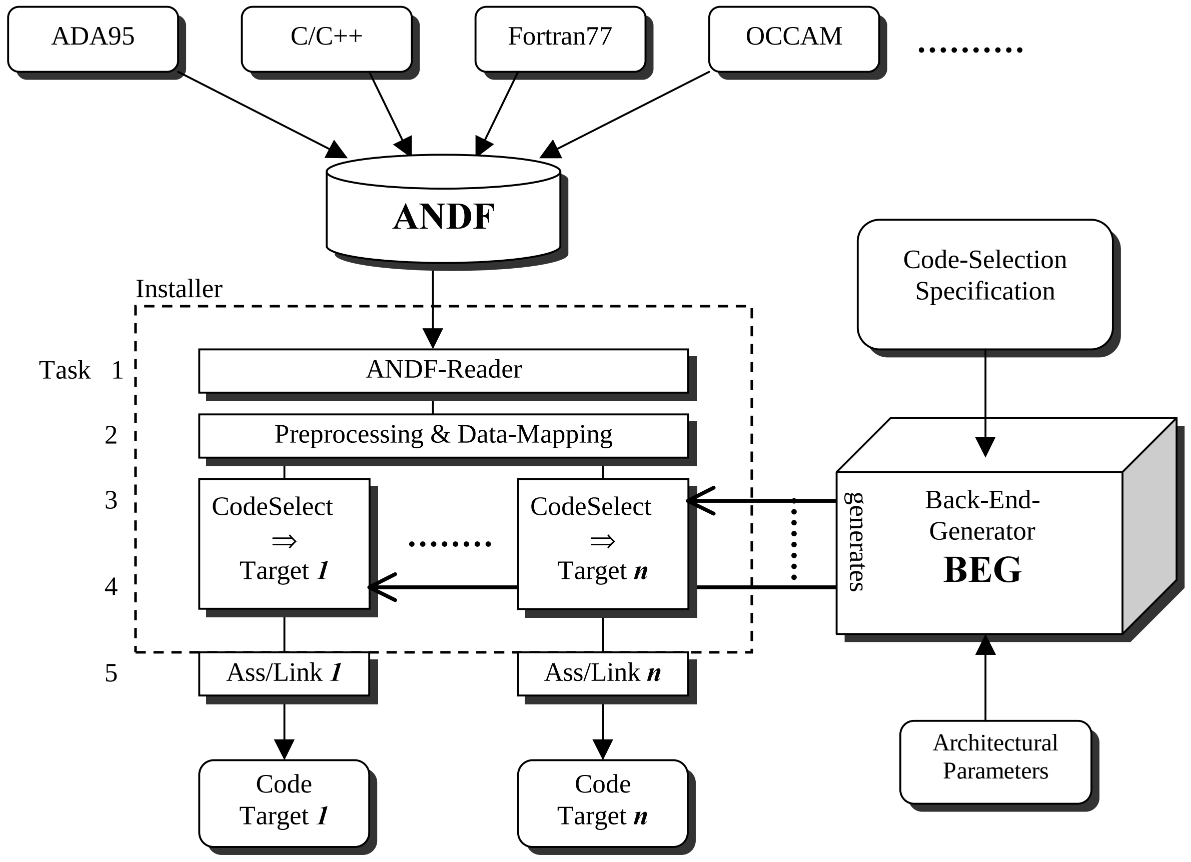
In the rest of this paper we will concentrate us on the back-end part (*installer*) of such a compiler and the generator techniques used here.

2.3 Back-End Architecture

The main aspects at the construction of compiler back-ends for embedded systems are retargebility and reliability [6]. 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.

The ANDF approach defines a strict division into architecture neutral and architecture dependent parts. We refined this approach for back-end purposes to a stepwise transformation from „high-level" ANDF to low-level machine code:

1. Linking architectural neutral ANDF code together with machine and operating system dependent ANDF libraries and application programming routines
2. Mapping types and data structures parameterised with target properties
3. Selecting target machine code
4. Instantiate code with concrete assembler mnemonics
5. Assemble and bind produced code to executable programs



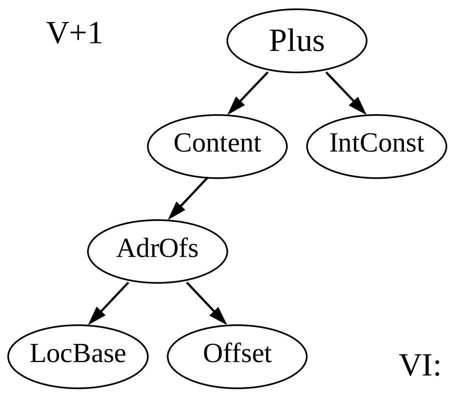
An implementation of task 1 (ANDF-Reader) can be reused at 100%, because it does not depend on the target and is implemented architectural neutral. A very efficient one is available as a result of the OMI/SAFE project. The mapping phase is highly parametrizable with the target properties and is reused as a part of the back-end specification. The code selection phase (task 3) performs the transformation of operations to the target machine while trying to use target resources (time or memory consumption) optimally. Obviously this is – together with task 4 – the most tedious task to implement and tool support is urgently required. Implementations are completely generated with the back-end generator, you can find some measurements and numbers in chapter 3. Task 5 is usually done by tools provided by the chip manufacturer.

2.4 The Back-End 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 [1,2,7,8,10]. 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 [3,4,9]. The user of the generator does not have to bother with the generated transducer system, he/she just has to assure, that the specified rule set is complete with respect to the intermediate language 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 a suitable 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 is annotated with the previously mentioned cost statement. The code generator computes the total costs 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.



I: RULE Content(X) -> Z
{ *loadw Z,(X)* } COST 16

II: RULE AdrOfs(X,Y) -> X
{ *addw X,Y* } COST 10

III: RULE LocBase -> Z
{ *loadw* Z,R_{FP} } COST 10

IV: RULE Plus(X,Y) -> X
{ *addw X,Y* } COST 10

V: RULE Offset.C -> Z
{ *loadw Z,#C* } COST 12

VI: RULE Content(AdrOfs(LocBase,Offset.C)) -> Z
{ *loadw* $Z, \#C(R_{FP})$ } COST 28

The example shows a typical program expression (access to a local variable in an arithmetic expression), and a simple rule system for code selection from (simplified) ANDF expressions to ST9 instructions. It is obvious, that there are several possibilities to cover this simple program tree, the cost optimal strategy is to apply rule VI to the *content* subtree instead of applying the simpler rules I,II,III,V. The situation might be different again, if we add additional rules that i.e. specify cheaper address offset rules, or rules that combine other arithmetic operators at lower costs.

Several generators have been built in the recent years and are now included in compiler toolboxes (BURG, IBURG/MBURG, PAGODE, BEG). Some of them made the step to industrial relevance, from which 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. A lot of work has been done to improve the reliability of compiler backends, especially in the context of optimizing code selectors generated from specifications [5,6].

3. Results and Experiences

To stress retargebility aspects, this paper also reports 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 installer part within the OMI/SAFE and OMI/FAME projects.

A code generator consists of intermediate language specific and target machine dependent parts. The language part models the input representation and performs conditional compilation and optimisations on ANDF-terms. This part can be reused 100% for a new compiler and is 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 BEG-specification and C-code produced. It documents results of the OMI/SAFE project, only specific parts have to be redone for a new installer.

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

	Man power (% of 15MM)	Lines of Specification	**Lines C-code (generated)**
Reader + Pre-Evaluation **100% reuse**	15%	---	17.000
ANDF-specific CG-part **100% reuse**	40%	4800	120.000
Architecture dependent	30%	2200	
Processor dependent	15%	900	

Table 1 Human Resources

These results show, that on the one hand the specification mechanism is very powerful – the relation from lines of spec. to lines of C-code is at least 1 to 15 – and on the other hand that the biggest part can be reused for a new architecture or processor family. I.e. it is possible to obtain a complete new compiler in 2-3 MM if retargeting to a similar processor. Let us stress again, that one obtains the whole family of compilers for the new architecture or processor if the retargeting work is done.

4. Conclusions

Automatic backend generation completes in some sense the ANDF technology in terms of efficiency and quality. Since software has become a key issue even in the embedded market, ANDF has the potential to become a world-wide recognised standard and ANDF based products may become the state-of-the-art, an appreciable situation with only winners:

- The users may buy components from different suppliers. ANDF guarantees the smooth fit into an existing environment.
- Compiler developers may enter into new market segments by providing only parts of a tool chain, e.g. front-ends with graphical user interface.
- In-circuit emulator suppliers may also supply installers for which they can have better integration into their own environments.

This list can easily be extended by advantages for other suppliers of development components such as suppliers for schedulability analysis tools or real-time operating systems. This paper demonstrates that reusability does not counteract the performance of the resulting system. In contrary both goals can be achieved in the proposed ANDF framework.

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