# LANCE: A C Compiler Platform for Embedded Processors

Dr. Rainer Leupers

University of Dortmund
Computer Science 12
Embedded Systems Group
44221 Dortmund, Germany
email: leupers@LS12.cs.uni-dortmund.de

**Abstract**

This paper describes LANCE, a software system for development of C compilers for embedded processors. LANCE comprises an ANSI C frontend, a collection of machine-independent code optimization tools, a C++ library for accessing and manipulating the intermediate program representation, as well as a backend interface for assembly code generators. The backend interface is compatible to standard code generator generator tools and therefore allows for compiler development for application-specific embedded processors at a relatively low implementation effort. LANCE is mainly intended to facilitate C compiler design for embedded processors, so as to eliminate the need for time-consuming assembly programming. Embedded processors for which LANCE based C compilers have been successfully built include both RISCs and DSPs. Initially designed for research purposes only, LANCE is now also being used for production-quality compiler development. Due to its clear tool structure, simple intermediate program representation, and machine independence, LANCE is particularly suitable for fast compiler development for new application-specific [1] processors.

## 1 Introduction

In today's embedded system design flows, there is a trend towards system specification in the C programming language. This can be observed both in the area of hardware design and simulation (see e.g. the SystemC language and tools by Synopsys or C Level Design Inc.), but in particular in software development, where C is step by step replacing assembly-level programming. This is due to the numerous advantages of C over other specification formalisms: C is largely machine-independent, it is a widespread and well-tried high-level programming language, and it offers very high simulation speed as compared to VHDL

[1] Publication: Embedded Systems/Embedded Intelligence, Nürnberg (Germany), Feb 2001

or Verilog. In addition, C offers enough flexibility to specify low-level, hardware-oriented operations, which is usually a necessity in embedded software development.

As embedded software is getting more and more complex, high-level language programming of embedded processors in C is becoming common. In particular this holds for modern DSP processors, whose architectural features make assembly programming increasingly difficult and time-consuming. This creates a need for efficient C compilers for embedded processors. For standard "off-the-shelf" embedded processors, C compilers are usually available from the processors vendors or from third party providers. However, also an increasing number of low-volume, application-specific processors (frequently DSPs) are being designed by system and intellectual property companies, for which in-house compiler development know-how and resources are usually not available. Therefore, application-specific processors mostly still have to be programmed in assembly. In fact, the need for assembly-level programming of embedded processors is frequently a very significant bottleneck in the design of embedded hardware/software systems.

In order to overcome this problem, the Embedded Systems Group at the University of Dortmund has been concentrating on compiler development for application-specific embedded processors. Focus is on effective code optimization techniques for processors with irregular data path architectures and VLIW-like instruction-level parallelism. As a common platform for the development of C compilers for new processors and research on code optimization techniques, the LANCE system has been developed. LANCE mainly covers the machine-independent parts of a C compiler in order to allow for a maximum degree of reuse. In comparison to similar compiler platforms (e.g. GNU gcc, lcc, SUIF, or Trimaran) LANCE is relatively easy to use, it already performs a number of code optimizations, and it is not restricted to specific processor classes. The LANCE system comprises the following components:

**ANSI C frontend:** The C frontend analyzes the C source code and generates a low-level intermediate representation. In case of syntax or semantical errors in the C code, error messages similar to those of GNU gcc's messages are emitted. The intermediate representation is largely machine-independent, only the bit width of the C data types and their memory alignment have to be specified in the form of a configuration file.

**Intermediate representation (IR):** The IR consist of so-called three address code, i.e. code with at most three operands per statement. This simple format strongly facilitates the implementation of tools operating on the IR, e.g. machine-independent optimizations. A special feature of the LANCE IR is the fact that the IR itself is kept in a low-level, assembly-like C syntax. Therefore, the IR can be compiled just like the original C program. This feature makes the IR easy to understand for new developers and it strongly supports validation of the C frontend and new IR optimization tools: By compiling both the C source and the corresponding IR code, and executing the resulting programs for test input data, their behavioral equivalence can be easily checked. Naturally, there is no proof of correctness, but by using this pragmatic validation methodology, LANCE has reached a high degree of stability.

**IR optimization tools:** LANCE contains a library of common machine-independent code optimizations, such as constant folding, dead code elimination, as well as jump and

loop optimizations. Dependent on the required optimization level, the optimization tools can be called separately or can be executed via a shell script.

**Flow analysis:** Control and data flow of a C program can be analyzed and visualized by means of a graph display tool. This helps in understanding the structure of a C program and facilitates the design of backends for assembly code generation.

**Backend interface:** The backend interface transforms the three address code IR into so-called data flow trees (DFTs). Each DFT represents of piece of computation in the C code and comprises arguments, operations, storage locations, as well as data dependencies between those. The DFT format generated by LANCE is fully compatible to widespread backend design tools like IBURG and OLIVE. In this way, operational backends for generating machine-specific assembly code can be designed quickly.

The remainder of this paper describes further details of the different LANCE components and outlines their application in research and industrial compiler design.

# 2 ANSI C frontend

The compilation of C source code into assembly code is usually organized into different phases (fig. 1). After an optional source-level optimization phase, an analysis of the source code is performed by the C frontend.

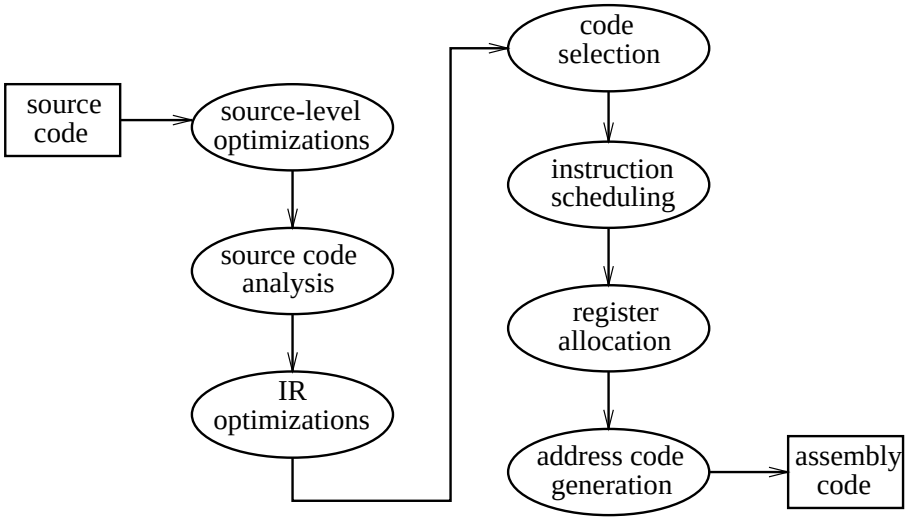


Figure 1: *Phase organization in a compiler*

The frontend performs syntactical and semantical correctness checks on the C source code and emits error messages if required. Otherwise, the C code is translated to a low-level intermediate representation (IR).

In LANCE, parsing of the C code is performed by means of an *attribute grammar*. It is well-known that ”almost” context-free programming languages like C can be parsed according to a context-free grammar plus an additional semantical analysis. Parsers for context-free

languages can be constructed by means of the widespread ”lex” and ”yacc” tools, which generate parser source code in C for given grammar specifications. The main problem is that the semantical analysis (which e.g. checks for correct declaration of identifiers and operand types in expressions) as well as the IR generation cannot be easily integrated into a yacc-based parser, but require a additional passes over the source program.

In contrast, using an attribute grammar for parsing allows to integrate syntactical and semantical analysis and also IR generation virtually into a single pass. This makes the implementation of the frontend very clean and modular. This feature is particularly important whenever C language extensions are required, which is frequently the case for DSP compilers.

The basic idea is that arbitrary information (such as variable types or declaration scopes) can be attached to the grammar symbols (terminals and nonterminals) in the form of *attributes*. The attribute values can be used to perform semantical analysis as a ”by-product” of syntax analysis, i.e. each time a grammar rule is applied during parsing all corresponding semantical checks can be executed at the same time. For instance, when parsing a variable name within an arithmetic expression, it can be checked whether the variable has been declared at an earlier program point.

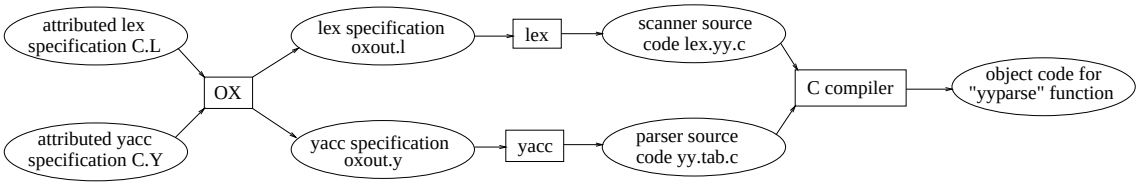


Figure 2: *Generation of C frontend source code using OX*

The largest part of the frontend source code in LANCE is automatically generated from an attribute grammar specification of ANSI C using the attribute grammar compiling system OX (fig. 2). OX reads two specification files, one for lexical and one for syntactical and semantical analysis. The files are essentially usual lex and yacc specifications, augmented by the required attribute definitions. From this specification, OX generates real lex and yacc specifications, which in turn are transformed into C source code by lex and yacc. Finally, a C compiler such as GNU gcc can be used to generate the object code for the frontend.

The use of generator technology in the form of OX, lex, and yacc results in a large source code reduction for the frontend, which in turn strongly improves reliability and maintainability of the code. In case of LANCE, the OX frontend specification consists of about 7400 lines of code, which are automatically expanded into more than 23000 lines of real C code.

The LANCE ANSI C frontend is almost completely machine-independent and thus can be reused in C compilers for arbitrary target processors. Only a few numerical parameters about the target machine have to be specified in a configuration file. These are the bit width of C types (e.g. short, int, long, float) as well as their alignment in memory. As the C language standard does not specify the exact bit width of types, the corresponding values are usually chosen by the compiler designer based on the characteristics of the target processor.

# 3 Intermediate representation

LANCE uses a widespread format for the intermediate representation (IR): *three-address code*. This format consists of a sequence of statements, each of which references at most three variables: two arguments and one result. Examples are *assignments* with arithmetical or logical operators such as `a = b + c`, or copy operations like `a = b`. In addition, three address code may comprise control flow statements.

The main motivation for using three address code is its simple structure. All high-level C constructs, such as for and while-loops, nested if-then-else-statements, switch-statements, complex arithmetic expressions and conditionals, and implicit address arithmetic for array and structure access are no longer present in the IR, but are broken down into sequences of simple statements. Additionally, all implicit type conversions in the original code are translated into explicit cast operations, and code for initialization of local variables is automatically inserted. This strongly facilitates the implementation of tools for processing C programs, such as IR optimization passes or compiler backends.

The IR file generated by the LANCE C frontend is structured into *symbol tables* and *functions*. The IR functions directly correspond to the functions in the original C code, i.e., there is one IR function for each C function. Each function is a list of *IR statements*, which exist in five types:

**Assignments:** An assignment is a three address code C statement with a destination and at most one operation or a function call on its right hand side.

**Jumps:** A jump is a C ”goto” statement with a target label.

**Branches:** A branch is a C if-statement of the general form `if (c) goto label`, where the condition `c` is a variable or a constant.

**Labels:** Labels are directly represented as C labels.

**Return statements:** A return from a function call is either a ”void” return statement, or a return with a value of the form `return x;` for some variable or a constant `x`.

For each rule of the C language grammar there is one function that translates a certain C construct into an equivalent sequence of three address code statements. The general concept is simple: complex statements are split into three-address code by insertion of auxiliary variables and appropriate ”goto” constructs. However, care must be taken in the ordering of IR statements and translation of implicit address arithmetic. We exemplify IR generation for the following piece of C code:

```
void f()
{
  int i,A[10];
  i = A[2]++ > 1 ? 2 : 3;
}
```

For the IR for function `f`, we need 8 auxiliary variables, denoted as `t1` to `t8`. These are declared in the local symbol table of `f`, together with the original local variables `A` and `i`. The symbol table in C syntax looks as follows:

```
int A[10];
char *t1,*t3;
int i,t2,t5,t6,t7,t8;
int *t4;
```

The first step in IR generation is to compute the value of `A[2]`. If we assume that an integer value occupies four memory words, the array index 2 needs to be scaled by 4 and must be added to the base address of array `A` in order to obtain the effective address "`A` + 8" of `A[2]`. When implementing this scheme in three address code, it is important to know that in C all constants added to pointers are implicitly scaled, e.g., adding a constant `c` to some integer pointer `p` actually increments `p` by 4 * `c`. Therefore, we perform all address arithmetic in the IR exclusively on `char` pointers, since characters are guaranteed to occupy only a single memory word. This leads to the following IR code segment:

```
t3 = (char *)A;  // cast base to char*
t2 = 2 * 4;      // compute offset
t1 = t3 + t2;    // compute effective address
t4 = (int *)t1;  // cast back to int*
t5 = *t4;        // load value from memory
```

The post-increment of `A[2]` is implemented as follows. The address stored in `t4` can be reused. Note that auxiliary variable `t5` still contains the original value of `A[2]`. This is necessary, since its value *before* the increment is required in the comparison.

```
t6 = t5 + 1;    // increment
*t4 = t6;       // store back into A[2]
```

Next, the condition `A[2] > 1` is evaluated and the result is stored in another auxiliary variable `t7`. The conditional expression itself is translated by means of a branch and two labels. Depending on the comparison result, auxiliary variable `t8` is loaded with either 2 or 3. When control flow joins (label `L2`), variable `i` finally gets its correct value from `t8`. In total, the IR for this looks as follows:

```
    t7 = t5 > 1;      // compare
    if (t7) goto L1;  // jump if >
    t8 = 3;           // load 3 if <=
    goto L2;          // goto join point
L1: t8 = 2;           // load 2 if >
L2: i = t8;           // move result into i
```

The C to IR translation process is very efficient. On a 600 MHz Linux PC, our C frontend emits up to 10,000 IR statements per CPU second, including file I/O. Due to the insertion of new variables and statements, the size of the generated IR is typically about twice as large as the original C source.

One key point in the LANCE IR is that the C language allows to express all IR constructs directly in C syntax. Therefore, as can be seen from the above example, any valid IR is simultaneously a valid (low-level) C code. This feature can be used for compiler validation (fig. 3).

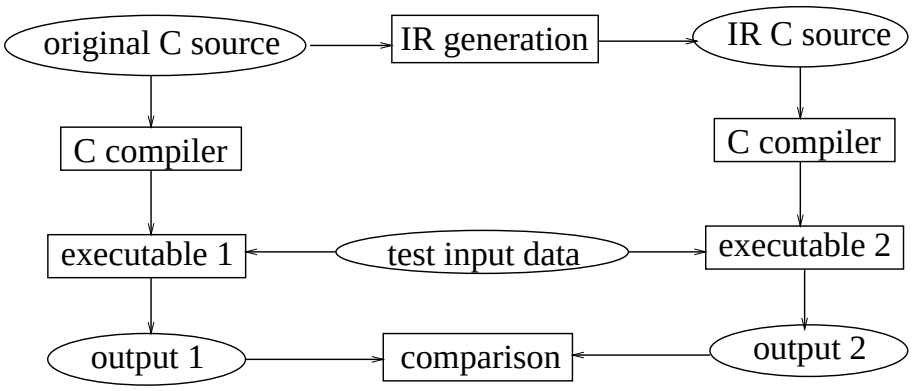


Figure 3: *Validation methodology*

For validation, both the original C program and the IR generated by the frontend are compiled with a usual C compiler on the host machine. The equivalence of the two executables is checked by means of a comparison between their outputs for some test input data. Any difference in the outputs indicates an implementation error. By using a large and representative suite of C programs and test inputs, a sufficient reliability can be ensured. For regression tests, the validation process can also be easily automated.

Also all IR optimization passes mentioned in the following section can be validated with the same methodology. For this purpose, some unoptimized IR is compared to the optimized IR using the same flow as in fig. 3. In this way, a C compiler can be validated down to the level of the optimized IR, which forms the basis for assembly code generation.

# 4 IR optimization tools

The IR generated by the frontend can be optimized independent of the target processor. IR optimizations essentially eliminate redundancies from the IR, which are incurred by the human C programmer or by the IR generation process itself. In LANCE, all IR optimizations are implemented as separate tools operating on a common IR format. In this way, new optimization tools can be easily added as ”plug-and-play” components to the system, and the amount and execution order of IR tools can be adjusted by the compiler user, according to the required optimization level. Currently, the LANCE system comprises the following IR optimization tools, all of which rely on standard techniques from compiler construction. Further, more machine-specific, optimizations are currently under development.

**Constant folding:** Arithmetic expressions with a constant result are replaced by the corresponding constants already at compile time.

**Constant propagation:** Variables with a constant value are replaced by the respective constants.

**Copy propagation:** Variables with identical values are unified so as to avoid superfluous copy operations.

**Common subexpression elimination:** Recomputation of already known temporary results is avoided by keeping those results in registers.

**Dead code elimination:** Instructions whose results are never used are removed.

**Induction variable elimination:** Loop variables that are computed as a linear function of another loop variable are simplified, so as to remove costly multiplications from loop bodies.

**Loop invariant code motion:** Computations independent of loop variables are moved outside of the loop body, so as to reduce loop execution time.

**Jump optimization:** Eliminates redundant jumps as well as jump chains.

Since the execution of of of the above optimizations frequently enable new opportunities for other tools, LANCE permits to iterate all (or a subset of) optimizations until no more improvement can be achieved.

# 5 Flow analysis

The LANCE system comprises a C++ library for accessing, analyzing, and manipulating the IR. Among the most important functionalities for code generation and optimization is analysis of control and data flow in a C program. Both types of information can be visualized in the form of graphs (fig. 4).

The control flow graph (CFG) indicates the basic block structure of C functions, while the data flow graph (DFG) represents the mutual dependence of values generated and used by IR statements inside blocks. On one hand, the graph display facilities are very helpful during debugging, e.g. when implementing new IR optimization tools. On the other hand, DFGs form the basic data structure required for assembly code generation. This is due to the fact, that the code generation process can partially be transformed to the problem of covering a given DFG by available instruction patterns. In order to reduce the problem complexity, it is common to first split the (general) DFGs into sets of *data flow trees* (DFTs), i.e. DFGs without common subexpressions. In contrast to DFGs, DFTs can be covered optimally with linear runtime complexity.

# 6 Backend interface

After IR generation and optimization, IR files are normally passed to a processor-specific backend for assembly code generation. In order to bridge the gap between the three-address form of the IR and the DFTs usually required by popular code generator generators such as OLIVE or IBURG, the LANCE system also comprises a backend interface. This interface transforms a three-address code IR into a behaviorally equivalent sequence of DFTs. The operator set of the DFTs is fixed and machine-independent. Therefore, the backend interface can be reused for all different target processors, and the generated data structures are fully compliant with the formats required by OLIVE or IBURG.

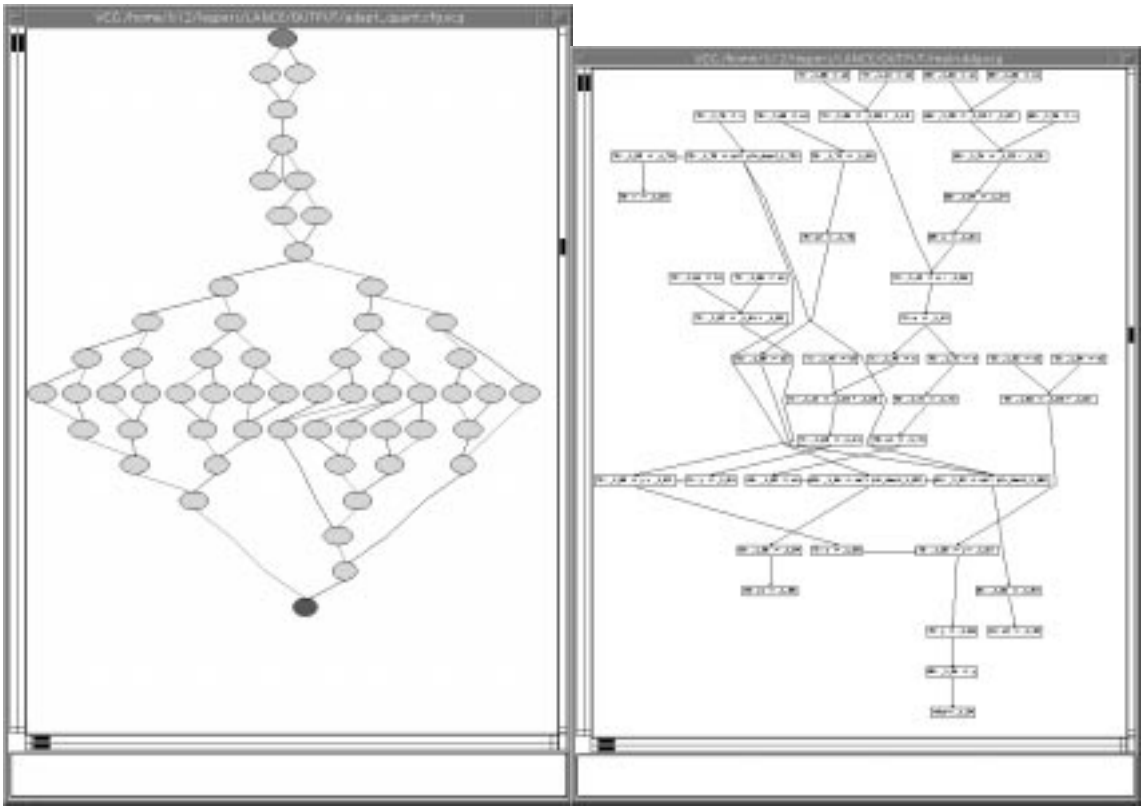

Figure 4: *Control flow graph and data flow graph generated by LANCE*

The basic technique in DFT generation is *substitution*: Under certain conditions, uses of variables can be substituted by their corresponding definitions, which is a simple application of data flow analysis. In this way, expressions are step-by-step enlarged, so as to form DFTs of maximal size. However, care must be taken during determination of DFT boundaries, in order to prevent undesired side effects. For instance, any DFT should contain at most one function call, since function calls may modify global variables. The sequential ordering of DFTs thus has to reflect the original sequence of function calls in the source code and the IR code.

The graphical output capabilities of LANCE facilitate backend design since the compilation process is made transparent. As an example, fig. 5 shows the user interface of the LANCE V1.0 system, which in this case has been retargeted to a TI 'C6201 DSP. The different windows show the C source code, IR code, as well as generated (symbolic and sequential) assembly code. A mouse-click on a line within one window automatically highlights the corresponding line(s) in the other windows.

# 7 Applications and availability

The LANCE system is implemented in C++. The C frontend and the IR optimization tools are available as separate executables, while the functions for IR access and manipulation, as well as the backend interface are available in the form of an object code library

Figure 5: *LANCE V1.0 graphical user interface*

that can be linked to any C or C++ application programs. The LANCE software (current version: V2.0) is available for Solaris and Linux platforms via http://LS12-www.cs.uni-dortmund.de/∼leupers. Licensing is free for research and evaluation purposes.

The LANCE system has been successfully applied in numerous university projects, including C compiler projects for several DSPs (including Texas Instruments 'C2x, 'C5x, and 'C6x) and core processors (including AMS Gepard and ARM7 RISC). By means of the built-in backend interface, LANCE allows for a quick development (typically within a few man-weeks) of working compiler prototypes for new processors. Naturally, more effort has to be spent for development of highly optimizing backends, which typically involve machine-specific code optimizations that are not covered by LANCE.

Besides its use in research projects, LANCE is also the basis for industrial C compiler projects carried out at the ICD (Informatik Centrum Dortmund) technology transfer company. Recently, ICD has delivered a C compiler for an application-specific network RISC processor designed by Infineon Technologies (Munich). Currently, an optimizing C compiler for the OnDSP$^{TM}$ DSP core architecture designed by Systemonic AG (Dresden) is under development.