A Programmable ANSI C Transformation Engine

Maarten Boekhold, Ireneusz Karkowski, Henk Corporaal and Andrea Cilio

Delft University of Technology
Mekelweg 4, P.O. Box 5031, 2600 GA Delft, The Netherlands
{maartenb,irek,heco,smallpox}@cardit.et.tudelft.nl

Abstract. Source code transformations are a very effective method of parallelizing and improving the efficiency of programs. Unfortunately most compiler systems require implementing separate (sub-)programs for each transformation. This paper describes a different approach. We designed and implemented a fully programmable C code transformation engine, which can be programmed by means of a powerful and easy to use transformation language. Its possible applications range from coarse-grain parallelism exploitation to optimizers for multimedia instruction sets.

1 Introduction

Due to advances in IC technology, multiprocessor systems are becoming ever more affordable these days. While most of these systems are used in either the server market or in scientific research, it can be expected that multiprocessor systems will also show up in embedded systems. Especially, because of the feasibility of single-chip multi-processor implementations.

To be able to use the full computing power that is available in such systems, it is necessary to execute the embedded applications in a parallel mode. Unfortunately most of the existing embedded codes are written in a sequential programming languages. Also the programmers usually feel more at ease in writing sequential programs. (Semi-) automatically transforming sequential programs to their parallel equivalents represents therefore an attractive alternative. Direct parallelization however often does not lead to an efficient implementation. A series of code transformations [6] are necessary to enable efficient parallelization. Since the number of standard transformations (and combinations of them) is large, writing separate (sub-)programs for each of them represents a long-term and tedious task. Even if finished, any new transformation requires each time a substantial effort, due to the very low code reuse. Tools that make writing such programs easier ([3,5]), represent only a partial solution. They make programming faster, but still the transformations have to be coded separately. An alternative is to design a programmable transformation engine, which could be easily configured for most useful code transformations.

This paper presents one such source-to-source code transformation tool itself (called **`ctt`**, **C**ode **T**ransformation **T**ool), targeted at translation of ANSI C programs. It can be configured with new transformations by means of a dedicated *transformation language.* The language has been carefully designed to enable efficient specification of most common code transformations. Its syntax has been derived from ANSI C. Thanks to that the language is easy to learn and powerful to use. Note that the tool does not decide *if* a possible transformation should be applied. This decision is currently left to the user. The design of the program allows its use in any ANSI C translation context. Potential applications include coarse-grain parallelism exploitation [4], ILP enhancement, and optimizations for multi-media instruction sets.

This work has been inspired by a system called MT1 [1], developed at the University of Utrecht and the University of Leiden, both in the Netherlands. MT1 is a system that performs a translation from Fortran77 to Fortran90. Unfortunately it has some drawbacks making it useless in our context. First of all, the research within our laboratory is mainly concerned with embedded applications (which are most often written in C). Secondly, with MT1 it is impossible to apply a number of interesting types of transformations (for example *inter-procedural).* It has no explicit control over variables and it is impossible to create *new* variables in a transformation. Finally, it can only operate in an interactive way, and it is therefore not possible to use MT1 from within other applications. Our work aims not only at being an ANSI C version of MT1, but also at adding the above capabilities.

The remainder of this paper is organized as follows. Section 2 introduces the basics of the transformation language that is used to specify transformations (the interested reader is referred to [2] for more details). Section 3 demonstrates the use of the language for specification of a simple loop transformation, giving feeling about its power and flexibility. The implementation details of the transformation program are briefly discussed in section 4. Finally, section 5 concludes the paper.

2 Transformation language

Writing separate programs for different transformations can be avoided if we properly organize the process of applying a transformation. Our transformation engine uses such an organization. Its translation process is divided into 3 distinct stages:

- **Code selection stage:** In this stage the engine searches for code that has a strictly specified structure (that matches a specified *pattern*). Each fragment that matches this *pattern* is a candidate for the transformation.
- **Conditions checking stage:** Transformations can pose other (non-structural) restrictions on a matched code fragment. These include, but are not limited to, conditions on data dependencies and properties of loop index variables.
- **Transformation stage:** Code fragments that matched the specified structure and additional conditions are replaced by new code, with the same semantics.

The structure of the transformation language closely resembles these steps, and contains three subsections called PATTERN, CONDITIONS and RESULT (see **Figure 1**). As can be deduced, there is a one to one mapping between blocks in the transformation definition and the translation stages.

While a large fraction of the embedded systems are still programmed in assembly language, the ANSI C has become a widely accepted language of choice for this domain. Therefore, we decided to derive our transformation language from the ANSI C. As result, all C language constructs can be used to describe a transformation. Using only them would however be too limiting. The patterns specified in the code selection stage would be too specific, and it would be impossible to use one pattern block to match a wide variety of input codes. Therefore we extended our transformation language with a number of *meta-elements.* Among others the following meta-elements were added and can be used to specify *generic* patterns, i.e. patterns that represent more than one element in the input C sources:

- *Statements:* keyword `STMT` represents any statement.
- *Statement lists*: keyword `STMTLIST` represents a list of statements.
- *Expressions*: keyword `EXPR` represents any expression.
- *Variables:* keyword `VAR` represents any variable (of any type).
- *Procedure calls:* keyword `PROCCALL` represents any procedure, which satisfies specific requirements.

The decision of including meta-elements for the variables was motivated by the desire of avoiding direct relationship between variable names in the transformation definition and the variable names in the input C sources. Meta-element must be assigned a number (except for variables), which should be included in braces behind their keyword (`STMT(1)` and `STMT(2)` represent then different C statements). Some meta-elements take also additional arguments (for example `BOUND` and `STEP_EXPR` take also as argument the name of the loop index variable). In the following section we will present a simple example of using the language constructs in each sub-block of the transformation definition.

3 Example

Loop interchange transformation, when applied to 2 tightly nested loops, exchanges the inner and outer loops. A pattern block that describes the code selection stage of *the loop interchange* transformation is shown on the top left of Figure 1. This pattern has the following meaning: "Look for 2 tightly nested loops, of which the inner loop can contain any statement list". The expression 1 has been used twice (in both loops) and therefore the lower bounds in both loops must be the same. Before the transformation however may be applied, the program should check if there exist no `(<,>)` dependencies [6] within the loop body. In addition, the loop-body should not contain `break` or `continue` statements. Only if these conditions are met, the inner and outer loop may be exchanged. They may be specified as shown on the bottom of Figure 1. The '*' in the second statement denotes any dependence. The result block is shown on the right. This description says: "Replace the matched code with 2 tightly nested loops, where the body of the new inner loop is the same as the body of the inner loop of the original code".

```
PATTERN {
  VAR x, y;

  for(x=EXPR(1);BOUND(1,x);STEP_EXPR(2,x)){
    for(y=EXPR(1);BOUND(2,y);STEP_EXPR(3,y)){
            STMTLIST(1);
    }
  }
}
```

```
RESULT {
  VAR x, y;

  for(y=EXPR(1);BOUND(2,y);STEP_EXPR(3,y)){
    for(x=EXPR(1);BOUND(1,x);STEP_EXPR(2,x)){
      STMTLIST(1);
    }
  }
}
```

```
CONDITIONS {
  stmtlist_has_no_unsafe_jumps(1);
  not(dep("* direction=(<,>)
      between stmtlist 1 and stmtlist 1"));
}
```

Figure 1 Specification of the loop interchange transformation.

4 Implementation

The transformation program has been written in C++, using the SUIF compiler toolkit. The decision of using SUIF was dictated mainly by the existence of a good ANSI C front end, a convenient internal representation (IR) with C++ interface, and by the existence of an IR to ANSI C conversion utility (s2c). Thanks to that we could concentrate more on the design of the transformation engine itself, which works entirely on the IR level only. The whole transformation trajectory is presented in Figure 2. Both input source and transformation definitions are compiled to the IR by the front-end. After linking them the translation process takes place. In current implementation, statements of the program are visited in the *depth first search* order, and at each of them all transformations are sequentially tried. The user decides which possible transformations are applied. Once the transformations have been applied, the s2c program is used to convert the IR back to ANSI C. All functionality needed to perform a transformation (i.e. code selection, conditions checking and transforming) has been implemented as a collection of C++ classes, which can be accessed through a single C++ class interface. This makes it easy to embed the full functionality within other C++ programs.

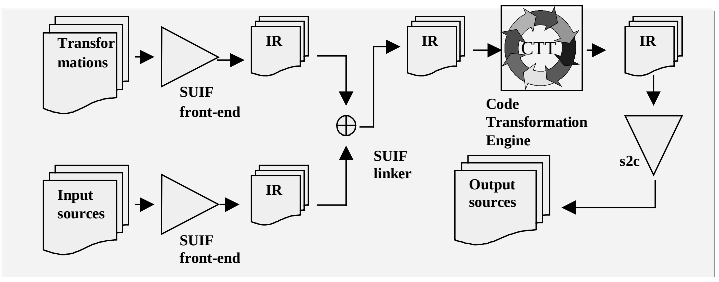

Figure 2 The transformation trajectory.

We provide a GUI, which allows users to experiment with different sets of transformations and provides an easy interface to each of the 3 transformation stages. While the translation process may proceed completely automatically, an interactive mode allows the user to *override* the decision made in the conditions checking stage (e.g. it is possible to apply a transformation even though the conditions checking stage says that this would be illegal).

5 Conclusions

In this paper we presented a programmable engine for code transformations on ANSI C programs. The knowledge about the transformations is added by means of a convenient and efficient *transformation language*. Using this language to specify new transformations is much easier and faster than having to write separate (sub-) programs for each of them. A very large subset of possible transformations is supported, including the inter-procedural ones. All of them (and their combinations) have been successfully specified using the transformation language, thereby proving the viability of its concept.

References

1. Aart J.C. Bik. *A Prototype Restructuring Compiler*. Technical Report INF/SCR-92-11, Utrecht University, Utrecht, the Netherlands, November 1994.
2. Maarten Boekhold, Ireneusz Karkowski and Henk Corporaal. *A Programmable ANSI C Code Transformation Engine.* Technical Report no. 1-68340-44(1998)-08, Delft University of Technology, Delft, The Netherlands, August 1998.
3. Dennis Gannon et al. *Sage.* http://www.extreme.indiana.edu/sage/, 1995.
4. Ireneusz Karkowski and Henk Corporaal. *Design Space Exploration Algorithm for Heterogeneous Multi-processor Embedded System Design.* In 35th Design Automation Conference Proceedings, June 1998, San Francisco, USA.
5. Robert Wilson, Robert Franch, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shin-Wei Liao, Chau-Wen Tseng, Mary Hall, Monica Lam, and John Hennessy. *An Overview of the SUIF Compiler System* http://suif.stanford.edu/suif/suif.html, 1995.
6. Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.