Transforming and Parallelizing ANSI C Programs Using Pattern Recognition

Maarten Boekhold, Ireneusz Karkowski and Henk Corporaal

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

Abstract. 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 transformation engine. It can be programmed by means of a transformation language. This language was especially designed to be easy to use and flexible enough to express most of the common and more advanced transformations. Its possible applications range from coarse-grain parallelism exploitation to optimizers for multimedia instruction sets. It has been successfully tested on various applications.

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 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 [15] 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 tedious task. Even if finished, any extension to the already implemented set of parallelizations requires each time a substantial effort, due to the very low code reuse. Tools, which make writing such programs easier ([8,14]), 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, 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.

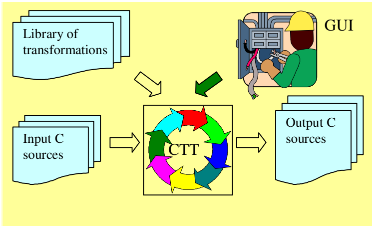

Figure 1 Schematic overview of the transformation program.

A top view on our transformation system is presented in Figure 1. The block in the middle represents the transformation engine itself (called `ctt`, **C**ode **T**ransformation **T**ool). Transformations, which are specified in the *transformation language,* form a library of transformations. Input files are translated using a subset of transformations from this library. Finally output C source files are generated. The design of the program allows its use in any ANSI C translation context. Potential applications include coarse-grain parallelism exploitation [11], ILP enhancement (for example for data locality/low power [16]), and optimizations for multi-media instruction sets [5].

The remainder of this paper is organized as follows. Section 2 describes related work, and in particular the MT1 system [2,3], which has been an inspiration for us. Section 3 introduces the transformation language that is used to specify transformations. The implementation details of the transformation program are briefly discussed in section 4. Section 5 presents the results of experiments with the system, confirming the power and flexibility of the transformation language. Finally, section 6 concludes the paper.

2 Related work

The traditional approach to code transformations has been different than ours. For each transformation, separate program code had to be written. Although there exist a myriad of tools that try to make this task as easy as possible (Suif [14] and Sage++ [8] for example), the amount of coding required remains significant. Such tools represent only a partial solution. They make programming faster, but still every transformation has to be implemented separately.

This work has been inspired by a system called MT1 [2,3], from 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. Its name should be pronounced as 'empty', which describes its basic state: it is a compiler, which contains almost no intelligence. In this state, it does an identity transformation of its input. All additional knowledge about transformations should be added to MT1 at a later stage.

Unfortunately MT1 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:

- MT1 cannot apply *inter-procedural* transformations.
- It has no explicit control over variables and it is impossible to create *new* variables in a transformation.
- 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.

3 Transformation language

This section will introduce our transformation language. Section 3.1 shortly introduces the reader to the topic of code transformations. The general overview of the syntax of our transformation language is given in section 3.2. The remaining subsections will present features specific to different stages of a transformation. Because of the space limitations we present only the most important elements of the transformation language. The interested reader is referred to [4] for more details.

3.1 Preliminaries

As a short introduction to the world of the transformations, let us start with two examples. For the educational purposes relatively simple loop transformations were selected. We will use them in the remainder of the paper to explain both the way our system works, and the transformation language itself (see [4] for much more elaborate examples).

Example 1 Loop interchange *transformation exchanges the loops within a set of tightly nested loops. Consider a loop nest shown in upper part of Figure 2. When applied to these 2 tightly nested loops, the transformation exchanges the inner and outer loops. As the result code in the lower part of Figure 2 is obtained. Before the transformation however may be applied, the program should check if there exist no* `(<,>)` *dependencies [15] within the loop body. Only if this condition is met, the inner and outer loop may be exchanged.*

```
for ( i = 0 ; i < 200 ; i++ ) {
        for (j = 0 ; j <= 100 ; j++ ) {
                  A[i][j] = A[i-1][j] + B[i][j];
        }
}
```

```
for ( j = 0 ; j <= 100 ; j++ ) {
        for ( i =0 ; i < 200 ; i++ ) {
                  A[i][j] = A[i-1][j] + B[i][j];
        }
}
```

Figure 2 Code before and after loop interchange transformation.

Some transformations may be applied to the code, which is spread among different procedures. Inter-procedural transformations form a very important class, and are fully supported by our transformation engine. The following example presents an example transformation belonging to this category.

```
....
for (i=0;i<100;i++){
     do_something();
}
....

void do_something() {
     procedure body
}
```

```
....
do_something2();
....

void do_something2() {
     int i;

   for (i=0;i<100;i++){
        procedure body
   }
}
```

Figure 3 Code before and after loop embedding transformation.

Example 2 *Consider the piece of code on the left in Figure 3. The loop embedding inter-procedural transformation moves the* `for` *loop to the body of the procedure* `do_something`. *As a result the code on the right in Figure 3 is generated. The transformation is always legal, and may be used to obtain more efficient code* *(smaller procedure call overhead), or to enable parallelism exploitation in the loop (data dependencies in the transformed code can be more easily analyzed). Note that we still may need the old version of the procedure* `do_something` *if it is called from more than one place.*

Both above transformations differ quite substantially, and therefore, using standard approaches, two separate programs would have to be written for each of them. This can be avoided if we properly organize the process of applying a transformation. Our engine uses such an organization. The translation 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 restrictions include, but are not limited to, conditions on data dependencies and properties of index variables.
- **Transformation stage:** Code fragments that matched the specified structure and additional conditions are replaced by new code, which has the same semantics as the original code.

The structure of the transformation language closely resembles these steps, and is shown in Figure 4. As can be deduced, there is a one to one mapping between blocks in the transformation definition and the translation stages.

```
PATTERN {
    description of the code selection stage
}

CONDITIONS {
    additional constraints
}

RESULT {
    description of the new code
}
```

Figure 4 Structure of a transformation description.

3.2 Language overview

The choice of the transformation language has a very large impact on the applicability of the tool. The following requirements should be considered:

1. The language should allow the specification of most useful transformations.
2. It should be powerful enough to express the functionality of typical embedded applications (It must be possible to add sections of code in the result code).
3. The language should be easy to learn and efficient.
4. It should be closely related to one of the languages used in the embedded systems community.

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, to satisfy the above requirements 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 (and not satisfy the first requirement above). 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.

Due to implementation details, a few variants of these meta-elements exist. For example there exist separate meta-elements for upper bound and step expressions in `for` loops (keywords `BOUND, STEP_EXPR`). As can be seen, we included also meta-elements for the variables. This 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 sections we will describe the use of transformation language constructs in each sub-block of the transformation definition (recall Figure 4).

3.3 The Pattern Block

The pattern block contains a description of the code selection stage. It describes the structure of code (a *pattern*) that may be a candidate for a transformation. The transformation engine uses a *pattern-matching* algorithm to find code fragments in the input C source that match it.

The block may be specified using all introduced (meta-)elements. On the first occurrence of a meta-element in the pattern block, the code at the corresponding location in the input is *bound* to this meta-element's number. If a meta-element is used more than once with the same number (for variables the same meta-variable name), the corresponding input code on subsequent occurrences must be <u>identical</u> to the input code that was bound to this meta-element at its first occurrence. The examples below illustrate that.

Example 3 *A pattern block that describes the code selection stage of* the loop interchange *transformation is shown in Figure 5. 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. Clearly this* pattern *matches the loop nest from Figure 2.*

```
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);
    }
  }
}
```

Figure 5 Pattern block for the loop interchange transformation.

Example 4 *Consider again the loop embedding transformation. The pattern block for it can be specified using the* PROCCALL *meta-element (see Figure 6). The presence of this meta-element means that at that particular location in the input code, a procedure call is expected, where the called procedure must satisfy a specified pattern* p_pat. *The whole pattern means: "Look for a* `for` *loop which contains a* *single call to a procedure (which takes no real arguments). The called procedure may contain any list of statements as its body". Of course more complex procedure patterns (with arbitrary number of procedure arguments) are possible.*

```
PATTERN {
     VAR x;

     for (x=EXPR(1);BOUND(1,x);STEP_EXPR(2,x)) {
          PROCCALL(1, p_pat);
     }
}

void p_pat() {
     STMTLIST(1);
}
```

Figure 6 Pattern block for the loop embedding transformation.

As could be seen in Example 4 the pattern for the called procedure must be specified outside of any block in Figure 4 (C does not allow procedure definitions inside any compound statement). Also other definitions, which may be necessary to define the behavior of the transformation stages, must be defined before the transformation definition blocks. Typically there is a single file for each transformation. It starts with inclusion of standard header files, which is followed by local definitions and the definitions of the transformation stages. For all of them the ANSI C scope rules apply.

3.4 Conditions block

A conditions block contains additional conditions that are imposed on code that was selected in the previous stage. These conditions can be divided into 3 categories:

- **Type properties:** This category includes requirements such as "is this expression a constant?", "does this statement contain a procedure call?" or 'does this expression have site effects?".
- **Structural properties:** In case of some structural properties it is not desirable to have to specify them in the pattern block. Most importantly, if we had to specify the existence or absence of control transfer statements (`break`, `continue` and `return` statements) in the pattern block, we would severely restrict the generality of the descriptions of transformations. Therefore, extra conditions that test for these structural properties are necessary.
- **Data dependencies**: Data dependencies are vital in testing for the legality of code transformations. Depending on the type of transformation we must be able to test for different following properties of data dependencies.

The following example demonstrates these concepts.

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

Figure 7 Conditions block for loop interchange transformation.

Example 5 *Recall* the loop interchange transformation *example. We can see that for this transformation to be legal the following two requirements must be fulfilled. The first one is that the loop-body should not contain* `break` *or* `continue` *statements. Second, that the inner loop-body is not allowed to have a (<,>) data dependencies within itself. These conditions may be specified as shown in Figure 7. The '*' in the second statement denotes any dependence.*

3.5 The Result Block

The result block uses the same part of the transformation language as used in the pattern block. Also a number of the result block specific elements are allowed. These include for example support for creation of new variables in the transformed code. Also strict control over the data type of these newly created variables is possible. The examples below demonstrate the transformation block specification.

Example 6 *Consider the* loop interchange transformation *example. The result block describing it is shown in example in Figure 8. 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".*

Example 7 *To specify the transformation stage of an inter-procedural transformation, we use the* `PROCCALL` *meta-element again. The result block for the* loop embedding *transformation is shown in Figure 9. It means: "Replace the original* `for` *loop and the call with a single new call. The called procedure should contain a for-loop whose body is equal to the body of the original called procedure".*

```
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);
    }
  }
}
```

Figure 8 Result block for loop interchange transformation.

```
RESULT {
    PROCCALL(1, p_transformation);
}

void p_transformation() {
    VAR x;

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

Figure 9 Result block for loop embedding 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 10. Both input source and transformation definitions are compiled to the IR by the front-end. After linking them the translation process takes place. 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 in other C++ programs.

Although efficient algorithms (using Finite State Machines) exist for finding patterns in tree-structures (``pattern matching" [10,9,1]), these proved to be difficult to use with SUIF. The main problem was that another intermediate representation of the program would have to be used. While it was possible to derive it from the SUIF IR, the possible advantages did not, in our eyes, justify the large effort of implementing it. Therefore, a *straightforward* pattern-matching algorithm has been used [4]. The nodes of program are visited in the DFS order, and at each place all transformations are sequentially tried.

A static data dependency analyzer of SUIF combined with our dynamic data dependence analysis package [12] is used to check the data dependence conditions that are specified in the condition blocks.

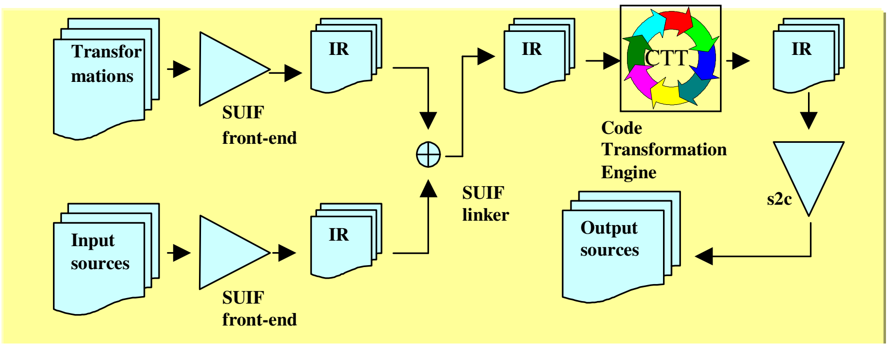

Figure 10 The transformation trajectory.

The Graphical User Interface (GUI) has been written using the Qt toolkit [13]. The GUI 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 Experimental results

In order to test the flexibility of the transformation language, we have specified a large number of transformations (and their combinations). All of them have been tested successfully on typical embedded applications, including 10 DSP programs from [6] and the MPEG-2 encoder from [7].

As example, table in Figure 11 presents results which we obtained using our transformation engine for mapping embedded programs to architectures with multimedia instruction sets. As can be seen, about 85% of the SIMD suitable loops have been automatically recognized and transformed. We must mention that our approach is different than the one described in [5]. Their methodology involved first converting the code to the vector code (working on infinite vectors), and then mapping vector instructions back to parallel loops (using vectors of length 4). In contrast to their approach, we performed the conversion directly.

Benchmark	Description	#loops	#SIMD	SIMD mapped
arfreq	Autoregressive freq. estim.	2	0	0
g722	Adaptive differential PCM	12	7	6
instf	Frequency tracking	9	3	3
interp3	Sample rate convertion	3	0	0
mulaw	Speech compression	1	0	0
music	Music synthesis	4	0	0
radproc	Doppler radar processing	7	1	1
rfast	Fast convolution using FFT	9	5	4
rtpse	Spectrum analysis	10	3	3
mpeg2enc	Video/MPEG-2 encoder	171	26	22
Total		**228**	**45**	**39**

Figure 11 Numbers of loops mapped to SIMD instructions.

Also a large set of well known and general transformations have been tried (see [4] for detailed specifications):

- Loop peeling.
- Index set splitting.
- Loop reversal.
- Loop skewing.
- Loop fusion.
- Wave fronting.
- Inlining.
- Loop fission.
- Strip mining.
- Code sinking.
- Unswitching.
- Loop embedding and extraction.

All the experiments proved the viability of our approach and the strengths of our transformation language. Also the learning time for a C programmer was short. Most specifications of transformations turned out to be simple to write and understand.

Some remarks are in order. There is a class of transformations, which is "by nature" more suitable to be implemented in an algorithmic way; this mainly for efficiency reasons. For example, while specification of inlining transformation is fully supported in our system, it could be implemented much more efficiently using an algorithmic approach (i.e. writing a special program for it). The reason for it is that separate patterns must be written for procedures with different numbers of arguments. Similarly, a single specification of the *wave fronting* transformation does not cover all possible situations in which this transformation could be applied. This is due to the fact that the exact transformation depends on the dependence distances in the loop-body. It is not yet possible to use this information to specify the transformation. Still our system can be used. By "pre-calculating" the transformations for specific dependence distances, patterns for these distances can be separately generated.

6 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.

In the future we plan to further extend the transformation language to obtain an even larger coverage (of a single transformation specification). We believe that, adding the following extra features is feasible:

- ♦ Algorithmic calculation of the expressions and statements used in the transformed code.
- ♦ Support for conditional construction of the code (by adding conditional statements to the result block).
- ♦ Support for iterated construction of the code (by adding loop statements to the result block).

Implementing these elements should bring us even further ahead of any of the existing transformation tools.

References

1. Marianne Baudinet and David MacQueen. *Tree pattern matching for **ML***. Available from David MacQueen, AT&T Bell Laboratories, 600 Mountain Avenue, Murray, Hill, NJ 07974, 1986.
2. Aart J.C. Bik. *A Prototype Restructuring Compiler*. Technical Report INF/SCR-92-11, Utrecht University, Utrecht, the Netherlands, November 1994.
3. Aart J.C. Bik and Harry A.G. Wijshoff. *Mt1: An interactive prototype restructuring compiler*. In Proceedings of the second annual conference of the Advanced School for Computing and Imaging, pages 78--83, June 1996.
4. 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.
5. Gerald Cheong and Monica S. Lam. *An Optimizer for Multimedia Instruction Sets.* In the Proceedings of the Second SUIF Compiler Workshop, Stanford University, USA, August 21-23, 1997.
6. P.M. Embree. *C Language Algorithms for Real-Time DSP*. Prentice-Hall. 1995.
7. MPEG Software Simulation Group. *MPEG-2 Video Codec*. http://www.mpeg.org/.
8. Dennis Gannon et al. *Sage.* http://www.extreme.indiana.edu/sage/, 1995.
9. Josef Grosch. *Transformation of attributed trees using pattern matching*. Lecture Notes in Computer Science, 641:1-15, 1992.
10. Christoph M. Hoffman and Michael J. O'Donnell. *Pattern matching in trees*. Journal of the ACM, 29(1): 68--95, 1982.
11. 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.
12. Ireneusz Karkowski and Henk Corporaal. *Overcoming the limitations of the traditional loop parallelization*. Future Generation Computer Systems, 13(4-5), 1998.
13. Troll Tech. *The Qt too*lkit. http://www.troll.no/products/qt.html, 1998.
14. 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.
15. Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.
16. Nikos D. Zervas, Kostas Masselos and C.E. Goutis. *Code Transformations for Embedded Multimedia Applications: Impact on Power and Performance.* In the Proceedings of the Power Driven Microarchitecture Workshop (in conj. with ISCA'98). June 1998, Barcelona, Spain.