Maciej POLIWODA, Piotr BŁASZYŃSKI
WEST POMERANIAN UNIVERSITY OF TECHNOLOGY, FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY,
Żołnierska 49, 71-210 Szczecin

A conversion of "while" and "do-while" loops to allow dependency analysis by existing tools

Ph.D. eng. Maciej POLIWODA

He graduated from the Faculty of Computer Science of Szczecin University of Technology, he received his PhD in 2002 and works in the Department of Software Engineering. His research interests are parallel programming, compilation techniques.

e-mail: mpoliwoda@wi.zut.edu.pl

Ph.D. eng. Piotr BŁASZYŃSKI

He graduated from the Faculty of Computer Science of Szczecin University of Technology. He received his PhD in 2004 and works in the Department of Software Engineering. His research interests are compilation techniques, programming languages.

e-mail: pblaszynski@wi.ps.pl

Abstract

In this paper there are presented methods for conversion of "while" and "do-while" loops to the form of "for" loops. The aim of this conversion is to allow dependence analysis by many existing tools. The dependence analysis allows determining which parts of the analyzed program code must be executed sequentially, and which can be executed independently. Such an analysis is used to reduce the use of resources in embedded systems. Automating the analysis also allows specifying types of classes of algorithms and their implementation which can be the subject of optimization. The nature of optimization can also be determined.

Keywords: compiler, loop conversion, dependency analysis.

Konwersja pętli „while" i „do–while" w celu analizy zależności przez istniejące narzędzia

Streszczenie

Większość narzędzi analizujących zależności "LooPo", "Clan" i "Petit" posiada możliwość analizy pętli typu for, w których wykonuje się operacje na tablicach za pomocą operatorów indeksowania. W chwili obecnej pętle while są analizowane przez analizator "LooPo". Analizatory zależności "clan" i "petit" analizują jedynie pętle for. Zasadną więc jest koncepcja konwersji konstrukcji typu "while", "do–while" do postaci pętli "for", gdyż po wykonaniu konwersji możliwe staje się użycie dowolnego narzędzia do analizy zależności. Użycie różnych narzędzi do analizy zależności daje możliwość uzyskania wyników analizy w postaci odpowiedniej dla zaimplementowanych algorytmów zrównoleglających, co pozwala na uzyskanie większego zbioru propozycji zrównoleglenia kodu pętli. Wykonanie konwersji umożliwi również zastosowanie szeroko implementowanego popularnego standardu OpenMP w celu podziału przestrzeni iteracji między niezależne wątki. Docelowa postać pętli spełnia wymagania dotyczące pętli for przedstawione w specyfikacji OpenMP v3.0. Zastosowanie zgodności z tym standardem umożliwia analizę zależności w pętli przez większość analizatorów pętli w języku C i zapisanie zrównoleglonej postaci pętli zgodnie ze standardem. Algorytm opisany w tym artykule jest zrealizowany kompilatorze "Stepson", który jest aktualnie rozwijany. Informacje o skompilowanym programie są przechowywane w postaci drzewa. Opisany algorytm pozwala zwiększyć liczbę pętli, które można automatycznie analizować.

Słowa kluczowe: kompilator, konwersja pętli, analiza zależności.

1. Introduction

Most tools for analyzing dependencies like "LooPo"[3], "Clan"[4] and "Petit"[5, 6, 9] have the ability to study a "for" loop which performs operations on tables using indexing operators. In these days "while" loops are analyzed only by the analyzer "LooPo". The dependence analyzers "Clan" and "Petit" analyze only "for" loops. It seems a reasonable idea to convert the structure of "while", "do-while" to the form of a "for" loop, because after the conversion it is possible to use any tools to analyze dependencies.

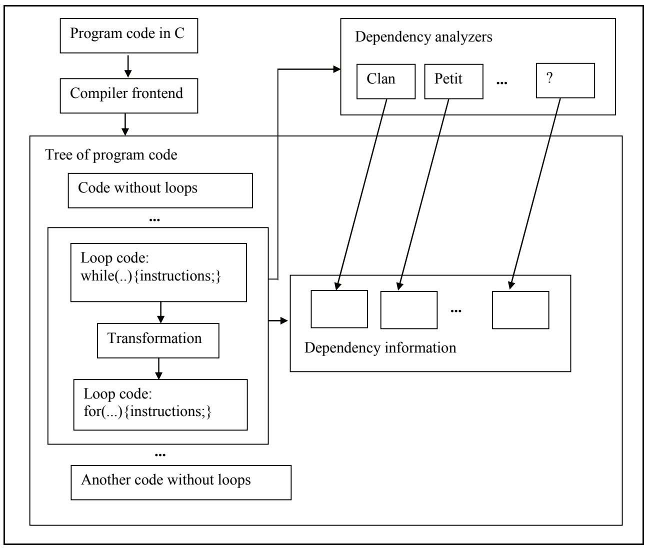

Fig. 1. "while" loop transformation
Rys. 1. Transformacja pętli „while"

Use of various tools for analysis of the relationship makes it possible to obtain the analysis results in a form suitable for different implemented parallelizing algorithms, which allows for a larger set of parallelization proposals of loops code.

The implementation of the conversion will also allow the use of the widely popular and implemented the OpenMP standard for allocation of iteration space among independent threads.

2. Conversion conditions

The converted "while" loop and the target "for" loop must meet certain conditions. The code of the "for" loop has to be accepted by a dependence analyzer. If the criteria of the target "for" loop are known, there can be specified the criteria for the "while" loop to enable to convert this loop to the corresponding "for" loop.

2.1. The target form of the converted loop

The target figure of the loop meets the requirements set out in the OpenMP [1] loop specification v3.0. The application of compliance with the OpenMP standard provides the ability to analyze dependencies in the loop for most analyzers for the C language. It also provides the possibility to write a parallel version of a loop in the OpenMP standard.

The requirements are presented below:

for (init-expr; test-expr; incr-expr) structured-block

init-expr One of the following:
var = lb
integer-type var = lb
random-access-iterator-type var = lb
pointer-type var = lb
test-expr One of the following:
var relational-op b
b relational-op var
incr-expr One of the following:
++var, var++, --var, var—, var += incr, var -= incr,
var = var + incr, var = incr + var, var = var - incr
var is One of the following:
A variable of a signed or unsigned integer type.
relational-op One of the following:
<, <=, >, >=
lb and b Loop invariant expressions of a type compatible with the type of var.
incr a loop invariant integer expression.

It is not permissible to change variable values lb, b or incr during the execution of a loop inside a structured-block.

2.2. Conditions for the converted “while” loop

Below there are presented the conditions for converting the “while” loop:

while(test-expressions) structured-block

indexes incr-expressions:
Loop code snippet in which changing of one or more index variables occurs. Change of the value of a single variable index is the same as “incr-expr” in the OpenMP standard.

test-expressions:
Index values of all variables changing value in the “Indexes incr-expressions.” are tested. Testing of the variable index is the same as in the “test-expr” in the OpenMP standard. Logical operators & & and | | are acceptable between checks “test-expr”.

structured-block:
Includes “indexes incr-expressions” and “statements” in any number and order of occurrence.

statements:
Looping the same way as in the “structured-block” in the OpenMP standard. The restrictions on the instructions in the loop body are contained in the documentation of various analyzers.

2.3. Example

In Fig. 2 there is shown a simple example of a converted “while” loop. In this example ‘i’ and ‘j’ variables are index variables. Other variables are present only in the statements. This “while” loop can be converted to “for” loop because index variables change is constant inside the loop body (and among all iterations).

```
i=si;
j=sj;
while(i<ie && j>je){
        i += si1;        //”incr-expr”   ”indexes incr-expressions”

        k += 4;          //”statement”
        a[i] = a[j];     //”statement”   ”statements”

        i+ = si2;        //”incr-expr”
        j+ = sj1;        //”incr-expr”   ”indexes incr-expressions”

        b[i] = [j];      //”statement”   ”statements”
}
```

Fig. 2. Example of a simple “while” loop with OpenMP standard description
Rys. 2. Przykład prostej pętli „while” z opisem w standardzie OpenMP

3. Conversion algorithm

The target “for” loop performs iterations on a single index variable whose value is the number of steps of the “while” loop. Change of the loop index variables values of the “while” loop in the “for” loop is moved outside the loop. The values of index expressions are dependent on the initial value of the “while” loop index variables, the total sum of the step size of indexing variables and the actual number of loop iterations.

3.1. Determination of the number of loop iterations

The number of iterations in the “while” loop is determined by the “test-expressions” condition consisting of one or more “test-expr”, where boundaries are being tested for a single variable index.

- When the number of iterations is determined by “test-expressions” and to test the limits of two index variables, the logical operator “&&” is used (for example test-expr-1 && test-expr-2) equal to the minimum number of iterations that meets the “test-expr-1”, and the number of iteration satisfying “test-expr-2”.

- When the number of iterations determined by the “test-expressions” and to find the limits of two index variables, the logical operator “| |” is used (for example test-expr-1 || test-expr-2) equal to the maximum number of iterations that meets the “test-expr-1” and the number of iterations that meets the test-expr-2.

The number of iterations for index variable “test-expr” when the index variable changes value in “indexes incr-expressions” according to “incr-expr”, is described:

- For the relations '<' and '>':
  ((end – start)/(sum of step values))

- For the relations '<=' and '>=':
  ((end – start)/(sum of step values)) + 1

where: end – value, which is compared to the index value; start – index value before the first iteration; sum of steps - the sum of the values that changes the index value in the individual “incr-expr”.

Knowing the number of iterations “StepCount”, while they are performing the loop, there can be written instructions “for” which will perform the same number of iterations, where the number of iterations is equal to the value of the index variable “StepCounter” as:

```
for(    StepCounter = 1;
        StepCounter <= StepCount;
        StepCounter++
) structured-block
```

In this algorithm it is necessary to find the index variables of each loop. To do this, the algorithm must check a while loop condition. One of variables is a loop index and other may be constant or some kind of loop bounds. To determine which one of those variables is the index variable, a compiler checks the left side of assignments. Only the variables modified in the "while" loop body can be an index. In a more complicated case, when there is two (or more) loops in the analyzed code, the index variables are determined separately for each "while" loop. When a variable is modified outside the inner loop it could not be the index variable of the inner loop[8].

3.2. Elimination of "indexes increxpressions" from "structured-block"

In parts of the code "statements", there is replaced a variable index of the expression:

- id + (StepCounter – 1) * StepSum when the expression occurs before the change of the index variable

- id + StepCounter * StepSum when the expression occurs after the change of the index variable

where: id – identifier of the replaced variable index; StepCounter – "for" loop step counter; StepSum – sum of step values that has "incr-expr" changing the value of the index id, prior to the current wording in the statements.

3.3. Example

In Fig. 3 there is shown a simple example of a converted "while" loop in the form of the "for" loop. Note the 'StepCount' variable, which is the only index variable in the converted "for" loop.

```
i=si;
j=sj;
StepCount = min( (ie - i) / ( si1 + si1), (je - j) / sj1 )
for(StepCounter = 1; StepCounter <= StepCount; StepCounter++){
        k += 4;
        a[i + StepCounter * si1] = a[j + (StepCounter-1) * sj1];
        b[i + StepCounter * (si1 + si2)] = [j + StepCounter * sj1];
};
i = i+ StepCount * (si1 + si2);
j = j + StepCount * sj1;
```

Fig. 3. Elements of the "for" loop after transformation
Rys. 3. Fragmenty pętli "for" po transformacji

4. Benchmark test

The usefulness of the solutions proposed in this paper was tested on the C and C++ code provided by the Collective Benchmark suite (*http://cTuning.org/cbench*). The authors developed a program (some kind of a compiler and shell script to manage many files compilation) to check how many "while" loops can be converted to appropriate "for" loop. This tool with instruction can be obtained from http://www.sfs.zut.edu.pl/ whileConvert . It shows how the effectiveness of parallelization can be enhanced by increasing the number of parallelized loops.

The test code includes 7039 loop, with 4487 "for" loop which is about 64%, 2552 loop "while" or "do-while" representing about 36%. Using the proposed algorithm, the compiler was able to convert to "for" form of the loop about 10% of the "while" or "do-while"loops.

The reasons why it was not possible to convert all "while" or "do-while" loop to the "for" loop are as follows:

- The occurrence of relationships '==' or '!=' expressions in "test-expressions",

- The expression "incr-expr" step value of the loop was not constant for all iterations.

These reasons prevent designation of the number of steps to be performed by the target "for" loop to be compatible with the OpenMP standard.

5. Summary

The algorithm described in this paper is implemented in frontend of parallelizing compiler "Stepson", which is actually under development. Information about a compiled program is stored in the tree structure. In this structure, every loop is marked with a Boolean flag. The code of the "while" loops is transformed in a memory, and a new loop version is stored in place of the old version. After that, on every single loop, the compiler executes dependence analysis with the petit analyzer. The results of analysis are stored in the previously mentioned parsing tree. The algorithm described in this paper allows increasing the number of C code loop to be automatically analyzed.

Obviously, there are other similar tools [2] for parallelizing "while" loops, but they are intended mainly for parallelizing or optimizing[7], not for transforming and dependency analysis. A distinguishing feature of the described solution is the possibility to get dependence information to process by another tool.

6. References

[1] Rauchwerger L. and Padua D.: Parallelizing While Loops for Multiprocessor Systems. Proc. of the 9th Int. Parallel Processing Symposium, April 1995, Santa Barbara, CA, pp. 347–356.
[2] Griebl Martin: The Mechanical Parallelization of Loop Nests Containing while Loops. Dissertation, Universitat Passau pp. 80-85.
[3] C'edric Bastoul: Code Generation in the Polyhedral Model Is Easier Than You Think. PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques, September 2004, pp. 7-16.
[4] Wolfe Michael: The tiny loop restructuring research tool. In Proc of 1991 International Conference on Parallel Processing, pages II-46 - II-53, 1991.
[5] Li Wei and Pingali Keshav: A singular loop transformation framework based on non-singular matrices. In 5th Workshop on Languages and Compilers for Parallel Computing, pages 249-260, Yale University, August 1992.
[6] Allen R., Kennedy K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann, 2001.
[7] Banerjee U.: Loop Transformations for Restructuring Compilers. Kluwer Academic, 1993.
[8] The Omega Project [online] http://www.cs.umd.edu/projects/omega/ [dostęp: 2010]

otrzymano / received: 10.09.2010
przyjęto do druku / accepted: 01.11.2010 *artykuł recenzowany*