Improvement of Data Dependence Parser Based on Tree-SSA Optimization Framework

Yang Xia
School of Information, Hunan Vocational College of Technology, Changsha 410118, Hunan, China

Abstract
Tree-SSA optimization framework in GCC provides a powerful program analytical framework. The enhanced data dependence analysis information allows the compiler to transform an algorithm so as to improve the performance. This paper analyzed the data dependence parser of GCC. In terms of the deficiency in the analysis of the linear array access of Fortran program, two improvement were given. Firstly, a non-affine array subscript dependency analysis algorithm was implemented. Secondly, we proposed and implemented a data dependency analysis method for the affine array of the split recursive chain. The experimental results indicate that these two improvements enhance the data dependency analysis ability of GCC, and provide more accurate data dependence information for loop transformation like loop switching.

Key words: Data dependence analysis; Linear array access; Cyclic transformation; Tree-SS.

1. INTRODUCTION

The Tree-SSA optimization framework in GCC provides a powerful framework for program analysis. The enhanced data dependency analysis allows the compiler to transform an algorithm so as to achieve greater locality and improve resource utilization. As a result, the handling capacity can be enhanced, and the performance can be improved. The nested loop optimizer of GCC combines the powerful nested loop analyzer and matrix transformation engine, providing a scalable cyclic transformation optimizer, which can perform unimodular transformation and scaling operations. Data correlation analyzer traces and concluded variables based on a new algorithm without the limitation to a particular pattern. The matrix transformation function utilizes a design similar to building block, and it can implement many general-purpose optimization tool programs. (Berlin, Eelsohn and Sebastian, 2004)

Advanced loop transformation and vectorization are two important optimization architectures. Program transformation must follow the constraint of dependency in the program, which ensures that data become data dependence in accordance with the constraint of correct reading and writing order. Data dependency is caused by reading and writing the same data. Data dependence analysis is the basis of advanced loop transformation and vectorization, and it provides the foundation for the correct implementation of these two optimizations. However, the data dependence parser in GCC cannot analyze and deal with linear array access completely and correctly, which leads to the loss of many optimization opportunities (Pop, Cohen and Silber, 2004).

2. DATA DEPENDENCE ANALYSIS

After genericization and simplification, the compiled language loop structure becomes the same low-level structure as the imperative language: Three-address assignment statement, goto statement, and labeled statement. To retrieve the traditional loop representation from the GIMPLE representation of a program, it is necessary to detect the natural loop. Based on the analysis of statements contained in the loops, loop index and loop boundary are detected. By analyzing the update attributes of scalar variables in the loop, the iteration number of the loop and the representation of the variable value quickly calculating any given iteration number can be extracted. On this basis, it is possible to eliminate the redundant detection after expanding the replication constants to the extended loop. We can also analyze as follows: the representation of the relationship between the reading and writing of the storage space through array access is extracted for traditional data correlation analysis.

The information extracted by the analyzer is recorded through a recursive chain. This representation utilizes the Newton interpolation formula, allowing quickly calculating the value of a function at a given integral point. The primary attribute represented by a recursive chain is the iteration execution of a program on the storage location. Each storage point contains an entry point of a loop. During the execution of the loop, the stored value changes with the operation of the update statement. The description of the update expression is embedded in the representation of the recursive chain, so that the representation of the original program can be

49
retrieved from the representation of the recursive chain. In other words, only scalar attributes of interest are selected, and the scalar attributes that cannot be determined are described as unknown elements.

2.1. Recursive Chain

In Figure 1, r1, r2, r3 in registers are initialized to the coefficients of the recursive chain. The values of these registers are then updated in the specified index loop of the recursive chain: In the loop in Figure 1, the value of register r2 is updated in the cycle, and its evolution is described through \( \{1,⁎,2\} \) recursive chain. r1 is accumulated with the successor value of r2 from its initial value 0, and it can be described through \( \{0,+\{1,⁎,2\}1\} \) recursive chain.

![Figure 1. Recursive chain 1](image1)

![Figure 2. Recursive chain 2](image2)

Register r6 in Figure 3 can be described by a multivariate scalar evolution \( \{7,\{8,+,9\}2\} \). The value of R6 in loop 2 accumulates the value of r5 contained in loop 1.

The recursive chain \( \{C0,+,...,CK\} \) is computed at the given integer point X by formula \( \{C0,+,...,CK\}(x) = \sum_{i=0}^{k} C_i \left( \frac{x}{i} \right) \) (and its coefficients are integral coefficients ((C0,...,CK)). In the special case of linear recurrence chains, this formula becomes \{base,+,step\}(x)+base+step*x, where base and step are two integer constants. Recursive chains can handle symbolic coefficients. However, the above formulas are not always valid in this case. (Robert A van Engelen, Johnnie Birch, Kyle A Gallivan., 2004; Müller M S, Baron J, Brantley W C, et al 2012)

2.2. Peeled Recursive Chain

Peeled recursive chain is another extension of the traditional recursive chain representation in order to represent the variables, the initial value of which will be rewritten at the first iteration. Due to the occurrence of the peeled recursive chain, we chose syntax similar to the SSA Phi node. The symbol for the peeled recursive chains is the loop Phi node itself, and the semantics of the peeled chain is as follows:

\[
(a,b)_k = \begin{cases} 
  a, & \text{when loop } K \text{ iterates for the first time} \\
  b, & \text{otherwise}
\end{cases}
\]

a and b are two recursive chains that may be symbolic forms. As long as the loop Phi node does not define a strongly connected component in the SSA graph, a peeled recursive chain can be constructed.

The data dependency parser of GCC uses a scalar evolution parser to obtain the array access subscript information in the Tree-SSA intermediate code of the program. The data dependency parser of GCC does not handle non-affine array subscript access, but making conservative conclusions and believing that there is an unknown dependence relationship. As a result, all program transformation that requires data dependence information as the condition will be hindered. This leads to the loss of a lot of program transformation opportunities that can improve program performance. (Yanqing, H., Baruch, K., Reuven,C. 2011)

3. IMPROVEMENT OF DATA DEPENDENCY ANALYSIS IN GCC

3.1. Data Dependence Analysis of Non Affine Array Indexed Access

The general data dependence analysis of non affine array subscript access is very complex. However, the special situation of subscript access in the same array is easier to deal with because of the same subscript access function. The analysis and test results show that the same array subscript access occurs frequently in the same loop nesting. Accurate analysis of special non-affine array subscript access can provide accurate data
dependence information for loop transformation and other optimization so as to increase the chance of optimization.

Here is a loop nesting in fast Fourier transform:

```plaintext
dimension a(nx, ny),...
do 10 m=1, nx
do 10 n=1, ny
   b(m, n)=b(m, n)*ist
10 continue
```

The recursive functions obtained by GCC scalar evolution parser in both accesses to b are
\[ \{0,+,1\}_1,+,\text{ish:817}_98\}_2 \] . Since there is an unknown compiled variable \text{ish:817}_98 in the recursive chain, this is a non affine array subscript access. GCC dependency parser conservatively maintains that there is an unknown data dependency. In fact, the dependency distance vector between the two accesses to the array b is zero vector. If the loop nesting is conducted with loop interchange, the data locality and be significantly improved so as to shorten the running time. When the two subscript access functions are exactly the same, the dependency distance vector is zero vector. Based on this property, we have preliminarily implemented a non affine array subscript dependency testing algorithm. The framework of the algorithm is as follows:

1. There is no dependency if the arrays accessed in the two array subscript accesses are not the same.
2. Otherwise:
   1. If the subscript access functions in the two array subscript accesses are exactly the same, the conflict iteration is set to zero;
   2. Otherwise there is an unknown dependency.

Through the non affine array subscript dependency testing algorithm preliminarily implemented, it can be aimed the conflict iteration between the two accesses of array b in the above loop is zero, and there is no data dependency between loops. Accordingly, the data locality of the program is improved through loop exchange.

### 3.2 Data Dependence Analysis of Affine Array Subscript Access Based on Split Recursive Chain

Considering the following small program example1.f:

```plaintext
real *8 a (N, N)
do 100 i=1, N
do 100 j=1, N-1
100 a(i, j+1)=a(i, j)+100
```

The subscript access recursive chain obtained through the scalar evolution analyzer is
\[ (i, j):\{0,+,1\}_1,+,1024\}_2 \] . Its dependence equation is
\[ -1024+i+1024*j'-j' -1024*j' = 0 \] , which is a Quadratic integer equation. The greatest common factor of variable coefficient is 1, and the GCD test cannot rule out the possibility of dependency. It is very complex to solve this dependence equation. The data dependence parser of GCC makes a conservative judgment for the example program, and believes that there is data dependence with unknown relationship.

In order to solve the above problems, the anti linearization technique is generally adopted to decompose the above dependence equation into an equation set composed of two equations. However, this method needs to use the loop iteration as the basis for decomposition. When the loop iteration range is a compiled unknown variable, and the array size is a compiled constant, the constant and the variable cannot be compared. In this case, this method is not applicable. Taking into account the characteristics of the recursive chain, the right side of the recursive chain is the cyclic increment, and the left side is the subscript offset after the linearization of array. In this paper, the recursive chain is divided based on the method recovering logic data structure proposed by M. Cierniak and W. Li in order to recover the multidimensional subscript information. Typically, each one-dimensional subscript expression contains at least one loop variable, and each loop variable in this array access is most likely to occur in some one-dimensional subscript expression, which is a simple subscript condition. It is assumed that the handled array access satisfies the simple subscript condition. After obtaining the offset of each dimension, the current result will be verified whether satisfy the condition. The algorithm will be finished, and the recover fails when the condition is not satisfied. (V Maslov. Delinearization., 1992; M Cierniak, W Li., 1995)

The subscript vector S is composed of subscript function of array access. S can be expressed as \( S = Au + \delta \), where A is the access matrix, \( \delta \) is the offset vector, and u is the cyclic index vector. For the sake of convenient
description, define \( \overline{S} = Au \), and \( S = \overline{S} + \delta \). Take the split recursive chain \( \{0,+1\}_{-1}, {+1,1024}_{-2} \) as an example to describe the algorithm splitting recursive chain as follows:

(1) Increment of the subscript variation in each loop is the step size of the subscript in the loop. The step vector we obtained is \((1,1024)^T\).

(2) Mapping vector \( m \). For a legal Fortran array access, its mapping vector \( m = (m_1, ..., m_d)^T \) satisfies three conditions: \( m_i = 1 \); \( \forall i = 2, ..., d, m_i \geq m_{i-1} \); \( \forall i = 2, ..., d, m_i \mod m_{i-1} = 0 \).

If the array access satisfies a simple subscript condition, a permutation of the non-zero element of the step vector is replaced as a mapping vector. Take the absolute value of the elements in the step vector and sort it from small to large to obtain a new vector. Each element in the mapping vector is the corresponding element in the new vector. If the first element of the mapping vector is greater than 1, it will be set as 1. The mapping vector of the sample program is \((1,1024)^T\).

(3) Calculate subscript range vector \( w \). For array access of Fortran programs, the mapping vector and the subscript range vector are closely related, and the following relationship is established:

\[
(\forall i = 2, ..., d) m_i = \prod_{j=1}^{i-1} W_j
\]

According to formula (1), all elements of the subscript range vector except for the last element can be obtained by using the obtained mapping vector, and set the last element as the largest integer. The subscript range vector in the sample is \((1024, 2147483647)^T\).

(4) Calculate the access matrix \( A \). Utilize the step size vector and mapping vector to calculate array access matrix \( A \). According to the assumption that the processed array satisfies the simple subscript condition, the access matrix \( A \) is constructed column by column. For the \( i \)th column of the access matrix, the following expression is established:

\[
v_{i} = \sum_{j=1}^{d} A_{ij} m_j
\]

Where, \( v_i \) is the \( i \)th element of the step vector, \( m_j \) is the \( j \)th element of the mapping vector. The access matrix of the sample program is:

\[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\]

(5) Calculate the offset vector \( \delta_i \). The formula for calculating the \( i \) dimension offset \( \delta_i \) in array subscript access function is \( \delta_i = t_i - t_{i-1} w_i \), satisfying \( 0 \leq \overline{S}_i + \delta_i < w_i \), where \( t_i \) is the leftmost value of the recursive chain. The value of \( t_{i-1} w_i \) will be used to calculate \( S_{i-1} \), \( w_i \) is the \( i \)th element of the subscript range vector, and \( \overline{S}_i \) is the product of the non-zero element in the \( i \)th line and the loop index variable of the \( i \) dimension of the access matrix \( A \). The offset of the last dimension is \( \delta_d = t_d \). If the loop iteration range is known, the assumptions will be ensured to be established by the constraint \( 0 \leq \overline{S}_i + \delta_i < w_i \). In many cases, the maximum number of iterations is unknown, and the value of \( \delta_i \) cannot be determined. When the maximum number of iterations is unknown, we consider the positive and negative characteristics of the \( \overline{S}_i \) when the loop variable takes the minimum value. When the \( \overline{S}_i \) is greater than zero, we will calculate the integer solution \( \delta_i \) and \( t_{i-1} w_i \), satisfying \( 0 \leq \overline{S}_i + \delta_i \) and \( \delta_i + t_{i-1} w_i = t_i \). Otherwise, we will calculate the integer solution \( \delta_i \) and \( t_{i-1} w_i \), satisfying \( \overline{S}_i + \delta_i < w_i \) and \( \delta_i + t_{i-1} w_i = t_i \). In this case, it is impossible to determine whether the handled array access satisfies the simple index condition. This can be achieved by adding a tag to the array that does not satisfy the simple subscript condition in the front-end processing. When this tag is set, the algorithm will not be executed.

The offset of the first dimension \( \delta_1 \) of the sample program is 0, and the offset \( \delta_2 \) of the second dimension is 0. After obtaining the access matrix and the offset vector, we transform the subscript access of each dimension into the recursive chain: \( \{+,+1\}_{-1}, \{1024,+1\}_{-2} \). Similarly, the recursive chain \( \{1024,+1\}_{-1}, \{1024\}_{-2} \), is further divided into two recursive chains \( \{1024,+1\}_{-1} \) and \( \{2048,+1\}_{-2} \). GCC data dependence parser analyzes the two pairs of recursive chains respectively to obtain the dependence distance vector \((0,1)\). The analyzed loop transformation framework is conductive to improving the locality of the data. Since the dependence distance obtained by the dependency analysis is non-negative, we can transform the loop successfully.
4 EXPERIMENTAL RESULT

We implemented the above improved algorithm in GCC, and tested swim.f of example1.f, SPEC2000 in this paper and rapid Fourier transform FORTRAN program dfft.f. The obtained experimental results are as shown in Table 1.

<table>
<thead>
<tr>
<th>Items</th>
<th>dfft.f</th>
<th>example.f</th>
<th>swim.f</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before improvement</td>
<td>After improvement</td>
<td>Before improvement</td>
</tr>
<tr>
<td>Dependent test number</td>
<td>1339</td>
<td>1339</td>
<td>3</td>
</tr>
<tr>
<td>Determined dependency</td>
<td>473</td>
<td>483</td>
<td>2</td>
</tr>
<tr>
<td>Undetermined dependency</td>
<td>234</td>
<td>224</td>
<td>1</td>
</tr>
<tr>
<td>Number of independence</td>
<td>632</td>
<td>632</td>
<td>0</td>
</tr>
<tr>
<td>Compilation overhead</td>
<td>5.098</td>
<td>5.116</td>
<td>0.293</td>
</tr>
<tr>
<td>Swappble loop</td>
<td>0</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Execution time (s)</td>
<td>209.759</td>
<td>167.304</td>
<td>0.101</td>
</tr>
</tbody>
</table>

As can be seen from table 1, after adding the improved algorithm, some data dependencies that cannot be determined were accurately identified as the determined data dependencies. In example 1, an undetermined data dependency was accurately identified as the determined data dependency, which provided the basis for the loop transformation. As a result, the program execution time was reduced by 75.25%, and the compiling time was only increased by 3.75%. Although the execution time of the swim.f has not changed, the original 124 undetermined data dependencies were identified as the determined data dependency with a slight increase. The original 10 data dependencies in dfft.f were accurately identified as the determined data dependencies, which provided basis for the loop transformation. Also, nine loops were successfully transformed, which decreased the execution time by 20.24% and increased the compiling time by 0.35%.

5. CONCLUSIONS

Although the GCC data dependence parser has a good effect on data dependence of array subscript access like C language, the data dependence analysis ability for linear array subscript access of Fortran is very weak. Also, the subscript access function containing unknown parameters is not processed. The data dependency analysis framework of GCC was deeply analyzed. The data dependence analysis of linear array subscript access with known array size is enhanced by splitting the recursive chain functions of linear array subscript access. These improvements provide more accurate data dependency information for loop transformations and other optimizations, which is likely to lead to more opportunities for optimization.

REFERENCES
