Index-Association Based Dependence Analysis and Its Application in Automatic Parallelization

Yonghong Song Xiangyun Kong

Sun Microsystems, Inc.
{yonghong.song,xiangyun.kong}@sun.com

Abstract. In this paper, we present a technique to perform dependence analysis on more complex array subscripts than the linear form of the enclosing loop indices. For such complex array subscripts, we decouple the original iteration space and the dependence test iteration space and link them through *index-association* functions. Dependence analysis is performed in the dependence test iteration space to determine whether the dependence exists in the original iteration space. The dependence distance in the original iteration space is determined by the distance in the dependence test iteration space and the property of index-association functions. For certain non-linear expressions, we show how to equivalently transform them to a set of linear expressions. The latter can be used in traditional dependence analysis techniques targeting subscripts which are linear forms of enclosing loop indices. We also show how our advanced dependence analysis technique can help parallelize some otherwise hard-to-parallelize loops.

1 Introduction

Multiprocessor and multi-core microprocessor machines demand good automatic parallelization to utilize precious machine resources. Accurate dependence analysis is the essential for effective automatic parallelization.

Traditional dependence analysis only considers array subscripts which are linear functions of the enclosing loop indices [6, 8, 13]. Various techniques, from a simple one like the GCD test to a complex one like the Fourier-Motzkin test, are applied to determine whether two array references could access the same memory location. For more complex subscripts, these techniques often consider them too complex and will give up with the assumption that a dependence exists. Figure 1(a) shows a simple example, where these traditional techniques are not able to parallelize it because they make the worst assumption. (In this paper, the program is written in Fortran format.)

This paper tries to conquer this conservativity. We apply a decoupled approach where a new *dependence test* iteration space is constructed for dependence test purpose. The *original* iteration space is linked to the dependence test iteration space by the mapping through *index-association* functions. We call our approach *index-association based dependence analysis*. Dependence analysis

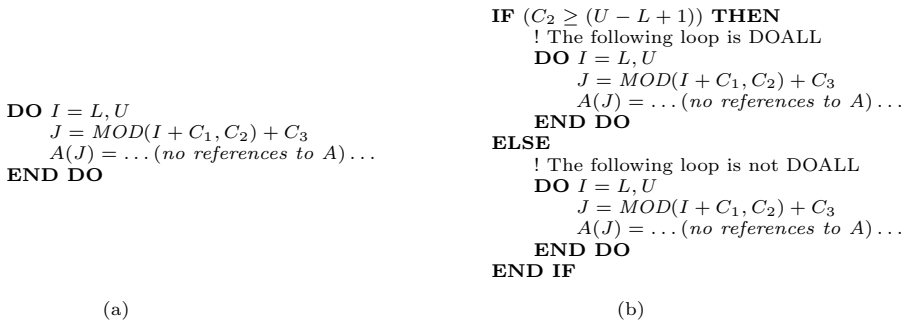


Fig. 1. Example 1

is performed under the dependence test iteration space. Whether the dependence exists in the original iteration space is determined by whether the dependence exists in the dependence test iteration space. If the dependence exists, the dependence distance in the original iteration space is determined by the dependence distance in the dependence test iteration space and the property of index-association functions.

We also present a general approach to equivalently transform a non-linear expression, involving *plus*, *minus*, *multiplication* and *division*, to a set of linear expressions. The latter can be used in dependence testing with traditional techniques.

When performing traditional dependence analysis and analyzing the index-association functions, our dependence analysis framework is also able to generate certain conditions under which cross-iteration dependence does not exist in the original iteration space. Such a condition can often be used as a run-time test for parallelization vs. serialization of the target loop. With the combination of index-association based dependence analysis and such *two-version code* parallelization, the code in Figure 1(a) can be parallelized as the code in Figure 1(b).

We have implemented the index-association based dependence analysis in our production compiler. Before this implementation, our compiler already implemented several dependence tests targeting subscripts which are linear functions of enclosing loop indices, which already enables us to parallelize a lot of loops. With this new implementation, our compiler is able to parallelize some loops which otherwise are not able to be parallelized without it. We select two well-known benchmarks from SPEC CPU2000 suite. With our technique, several important loops inside these two benchmarks can be parallelized successfully.

In the rest of the paper, we describe the previous work in Section 2. We present a program model in Section 3. We then describe our index-association based dependence analysis in Section 4. We present how to transform a non-linear expression to a set of linear expressions in Section 5. We show how our advanced dependence analysis helps automatic parallelization in Section 6. We present experimental results in Section 7. Finally, a conclusion is drawn in Section 8.

```
DO I_1 = L_1, U_1, S_1
    DO I_2 = L_2, U_2, S_2
        ...
            DO I_n = L_n, U_n, S_n
                J_1 = f_1(I_1, ..., I_n)
                J_2 = f_2(I_1, ..., I_n)
                ...
                J_m = f_m(I_1, ..., I_n)
                using linear form of (J_1, ..., J_m) in array subscripts
            END DO
        ...
    END DO
END DO
```

Fig. 2. Program Model

2 Previous Work

Dependence analysis has been studied extensively. Maydan *et al.* use a series of special case dependence tests with the hope that they can catch the majority of cases in practice [6]. They use an expensive integer programming method as the backup in case all these special tests fail to determine whether the dependence exists or not. Goff *et al.* present a practical dependence testing by classifying subscripts into different categories, where different dependence tests are used in different categories [3]. Pugh presents an integer programming method for exact dependence analysis with the worst exponential complexity in terms of loop levels and the number of array dimensions [8]. Feautrier analyzes dependences using parametric integer programming [2]. His technique takes statement context into consideration so that the dependence test result is more accurate. All the above techniques focus on array subscripts which are linear functions of enclosing loop indices.

Dependence analysis with array subscripts which are not linear functions of enclosing loop indices has also been studied. Blume and Eigenmann propose range test, where the range of symbolic expressions is evaluated against the loop index value [1]. The loop can be parallelized if the range of elements accessed in one iteration does not overlap with the range of the other elements in other iterations. Haghighat and Polychronopoulos handle non-linear subscripts by using their mathematical properties [4]. They use symbolic analysis and constraint propagation to help achieve a mathematically easy-to-compare form for the subscripts. Hoeflinger and Paek present an access region dependence test [5]. They perform array region analysis and determine dependence based on whether array regions overlap with each other or not. All these works are complementary to our work and can be used in our work as our dependence test iteration space can be extended to include more complex subscripts.

3 Program Model

Figure 2 illustrates our program model. Our target loop nest is an n-level perfect nest where $n \geq 1$. The loop lower bound $L_k (1 \leq k \leq n)$ and loop upper bound

U_k are linear functions of the enclosing loop indices $I_p(1 \leq p \leq k-1)$. The loop steps $S_k(1 \leq k \leq n)$ are loop nest invariants.

In the beginning of the innermost loop body, we have $m(m \geq 1)$ functions which maps a set of values $(I_1, \ldots, I_n)$ to a new set of values $(J_1, \ldots, J_m)$. In the rest of loop body, linear combinations of $(J_1, \ldots, J_m)$ are used in array subscripts.

We call the iteration space defined by all possible values of $(I_1, \ldots, I_n)$ as the *original* iteration space. We call the iteration space defined by all possible values of $(J_1, \ldots, J_m)$ as the *dependence test* iteration space. We call such a mapping from the original iteration space to the dependence test iteration space as *index association* and functions $f_k(1 \leq k \leq m)$ as *index-association function*.

In modern compilers, symbolic analysis is often applied before data dependence analysis to compute f_k. Traditional dependence analysis techniques are able to handle index-association functions f_k that are linear functions. For such cases, the function can be forward substituted into the subscript (to replace $J_k(k = 1, \ldots, m)$) and traditional techniques apply. However, if any f_k is a non-linear function (e.g., the tiny example in Figure 1), traditional techniques often consider the subscript too complex and assume the worst dependence conservatively. In the next section, we present details of our index-association based dependence analysis, which tries to conquer such conservativity.

4 Dependence Analysis with Index Association

The index-association based dependence analysis can be partitioned into three steps. First, the dependence test iteration space is constructed. Second, dependence analysis is conducted in the dependence test iteration space. Finally, the dependence relation in the original iteration space is determined by the result in the dependence test iteration space and the property of index-association functions. We elaborate the details below.

4.1 Constructing Dependence Test Iteration Space

The original iteration space can be viewed as a n-dimensional space. For dimension $k(1 \leq k \leq n)$, we have a constrain (L_k, U_k, S_k), where L_k is the lower bound, U_k is the upper bound and S_k is the step value.

To construct the dependence test iteration space, the compiler needs to analyze the index-association functions. Currently, our compiler requires the index-association function f_k have the following two properties:

- Each index function f_k only takes one original loop index variable as the argument (Note that different f_k can take the same loop index variable as the argument.) For example, our compiler can handle the index-association functions like $J_1 = DIV(I_1, 2)$, while it is not able to handle $J_1 = DIV(I_1 + I_2, 2)$, where both I_1 and I_2 are outer loop index variables.

Table 1. Iteration space mapping

operator	expression	iteration space
plus	$f(I)+c$	$(l+c, u+c, s)$
minus	$f(I)-c$	$(l-c, u-c, s)$
mult	$cf(I)$	(cl, cu, cs)
division	$f(I)/c$	$(l/c, u/c, (\lfloor s/c \rfloor, \lceil s/c \rceil))$
modulo	$MOD(f(I), c)$	$(MOD(l, c), MOD(u, c), (s, \_others\_))$

It is possible to relax such a requirement for index-association functions, in order to cover more cases. For certain cases, we can transform the index-association function to make it conform to the requirement. For example, for the function $f_k(I_1, I_2) = DIV(I_1, 2)+2*I_2$, we can have $f_{k1}(I_1) = DIV(I_1, 2)$, $f_{k2}(I_2) = 2 * I_2$ and $f_k(I_1, I_2) = f_{k1}(I_1) + f_{k2}(I_2)$. If we propagate $f_k(I_1, I_2)$ into the subscripts, index-association functions f_{k1} and f_{k2} will satisfy the requirement. For more general cases, however, it is much more difficult to compute the dependence test iteration space. We leave such extension for our future work.

– The operators in f_k must be *plus*, *minus*, *multiplication*, *division* or *modulo*. The f_k can be composed using the permitted operators recursively. For example, our compiler is able to handle $J_1 = DIV(2I_1 + 3, 4)$ where I_1 is an outer loop index.

Given the original iteration space, our compiler tries to construct the corresponding dependence test iteration space for $J_k = f_k(I_p)(1 \le k \le m, 1 \le p \le n)$ with a form (l_k, u_k, s_k), where l_k is the lower bound, u_k is the upper bound and s_k is the step. Supposing the loop I_p has a lower bound L_p and an upper bound U_p, we have $l_k = f_k(L_p)$ and $u_k = f_k(U_p)$. The step s_k represents the difference between two J_k values mapped from two adjacent I_p values. Note that s_k could be a sequence of values, including 0.

Suppose that in Figure 2 there exists a dependence from iteration $(i_{11}, i_{21}, \ldots, i_{n1})$ to iteration $(i_{12}, i_{22}, \ldots, i_{n2})$. We say the corresponding dependence distance is $(i_{12} - i_{11}, i_{22} - i_{21}, \ldots, i_{n2} - i_{n1})$ in the original iteration space. Suppose that $(j_{11}, j_{21}, \ldots, j_{m1})$ are the corresponding J values for $(i_{11}, i_{21}, \ldots, i_{n1})$, and $(j_{12}, j_{22}, \ldots, j_{m2})$ for $(i_{12}, i_{22}, \ldots, i_{n2})$. The dependence distance in the dependence test iteration space is $(j_{12} - j_{11}, j_{22} - j_{21}, \ldots, j_{n2} - j_{n1})$.

Table 1 illustrates our basic iteration space mapping from original iteration space to dependence test iteration space, assuming the iteration space for $f(I)$ is (l, u, s). For *division*, two different steps may result. For *modulo*, because of the wrap-around nature of the function, some negative steps may appear which are represented by *_others_* in the table. The dependence test iteration space is computed by recursively computing the iteration space for sub-expressions of $f_k(I_p)(1 \le p \le n)$, starting with I_p and ending with $f_k(I_p)$.

Here, we want to specially mention the following two scenarios:

– Because it may potentially generate many negative step values for a *modulo* operator, a condition is often generated considering the relation between $u - l + 1$ and c (in Table 1), in order to limit the number of negative steps.

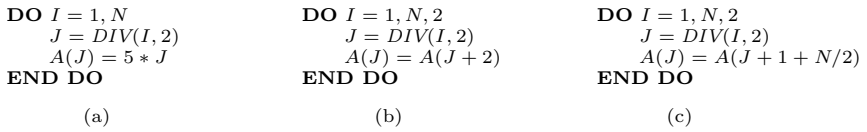


Fig. 3. Example 2

– It is possible to have different J_k associated with the same I_p such as $J_{k_1} = f_{k_1}(I_p)$ and $J_{k_2} = f_{k_2}(I_p)$. The coupling relation of J_{k_1} and J_{k_2} will be lost in the dependence test iteration space, which will cause difficulties when the dependence distance in the dependence test iteration space is mapped back to the original iteration space. For such cases, if functions f_{k_1} and f_{k_2} are both linear forms, we will perform forward substitution for these functions and have a single $J_k = I_p$ as the index-association function. Otherwise, we can still perform dependence analysis in the dependence test iteration space. However, we are not able to compute the dependence distance in the original iteration space precisely.

Figure 3 shows three examples. For Figure 3(a), the original iteration space is $(1, N, 1)$. The dependence test iteration space is $(0, N/2, s)$, where the step s is variant with a value of 0 or 1. For Figures 3(b) and (c), the original iteration space is $(1, N, 2)$. The dependence test iteration space is $(0, N/2, 1)$.

4.2 Dependence Analysis in The Dependence Test Iteration Space

After the dependence test iteration space is constructed, dependence analysis can be done in the dependence test iteration space, where traditional techniques, which target the linear form of the enclosing loop indices, are applied.

However, note that the dependence test iteration space could have multiple step values in certain dimension. For such cases, traditional techniques have to assume a step value which is greatest common divisor of all possible non-zero step values. If the step value could be 0, we also assume a step value of 0 during dependence analysis. With such assumptions, we may get conservative results. In Section 5, we describe a technique which can potentially give us better results for such cases.

Given a pair of references, there are three possible results from the dependence test in the dependence test iteration space.

– If there exists no dependence in the dependence test iteration space, then there will be no dependence in the original iteration space.
– If there exists a dependence with a distance d in the dependence test iteration space, then we compute the dependence distance in the original space based on d and the property of index-association functions. This will be further explored in the next subsection.
– If there exists a dependence with an unknown distance in the dependence test iteration space, we simply regard that there exists an unknown distance dependence in the original iteration space.

Table 2. Dependence distance mapping

operator	org expr	org dist	new expr	new dist
plus	$f(I)+c$	d	$f(I)$	d
minus	$f(I)-c$	d	$f(I)$	d
mult	$cf(I)$	d	$f(I)$	d/c if $MOD(d,c)=0$, no dependence otherwise
division	$f(I)/c$	d	$f(I)$	$(dc-c+1,\ldots,dc+c-1)$
modulo	$MOD(f(I),c)$	d	$f(I)$	d

In Figure 3(a), because the step can have a value of 0, the dependence distance from $A(J)$ to itself could be 0 in the dependence test iteration space. In Figures 3(b) and (c), however, there exists no dependence from $A(J)$ to itself in the dependence test iteration space. In Figure 3(b), there exists a dependence from $A(J+2)$ to $A(J)$ with distance 2 in the dependence test iteration space. In Figure 3(c), because the dependence test iteration space for J is $(0, N/2, 1)$, we can easily get that there exist no dependence between $A(J)$ and $A(J+1+N/2)$ in the dependence test iteration space.

4.3 Computing Dependence Distance in Original Iteration Space

Given a dependence distance in the dependence test iteration space, we need to analyze the property of index-association functions in order to get the proper dependence distance in the original iteration space. Table 2 illustrates how we compute the dependence distance in the original iteration space based on index-association functions, where "org expr" and "org dist" represents the original expression and its associated distance, and "new expr" and "new dist" represents the sub-expression in the original expression and its associated distance. The dependence distance in the original iteration space is computed by recursively computing the distance for the sub-expression of $J_k = f_k(I_p)(1 \leq k \leq m, 1 \leq p \leq n)$, starting with $f_k(I_p)$ and ending with I_p.

In Table 2, we want to particularly mention the dependence distance calculation of $f(I)$ for *multiplication* and *division*. Let us assume that iterations i_1 and i_2 have a dependence. For *multiplication*, we have $cf(i_1) - cf(i_2) = c(f(i_1) - f(i_2)) = d$. We can derive $f(i_1) - f(i_2) = d/c$ if $MOD(d,c) = 0$. Otherwise, there will be no dependence between $f(i_1)$ and $f(i_2)$. For *division*, we have $f(i_1)/c - f(i_2)/c = d$. We want to find the range of $f(i_1) - f(i_2)$. Through mathematical manipulation, we can find $dc - c + 1 \leq f(i_1) - f(i_2) \leq dc + c - 1$ for general cases, as illustrated in Table 2. For certain cases, however, we can get more precise result. For example, if $MOD(f(i), c)$ is always equal to 0, the distance for $f(I)$ would be solely $(f(i_1) - f(i_2))/c$.

In Figure 3(a), there exists a dependence from $A(J)$ to itself with a distance 0 in the dependence test iteration space. Because of index-association function $DIV(I, 2)$, it is easy to see that the corresponding distance in the original iteration space is 0 or 1. (The -1 is an illegal distance and is ignored.)

In Figure 3(b), there exists a dependence from $A(J + 2)$ to $A(J)$ with a distance 2 in the dependence test iteration space. Because of index-association

```
Input: A perfect loop nest conforming to Figure 2.
Output: Dependence relations between references inside the loop nest.
Procedure:
    Analyze f_k(k = 1, ..., m) and try to construct the dependence test iteration space.
    if (the dependence test iteration space cannot be constructed successfully) then
        Assuming a worst-case dependence test iteration space.
    end if
    for (each pair of references r_1 and r_2)
        if (there exists no dependence in the dependence test iteration space) then
            There exists no dependence in the original space.
        else if (the dependence distance is d in the dependence test iteration space) then
            Compute the distance in the original space based on d and f_k.
        else
            There exists dependence in the original space with unknown distance.
        end if
    end for
End procedure
```

Fig. 4. Top algorithm for index-association based dependence analysis

```
DO I = 1, 100, 3
  J = 5 * I/4
  A(J + 9) = A(J) + 1
END DO
```

Fig. 5. Example 3

function $DIV(I, 2)$, the corresponding distance in the original iteration space would be 3 or 4 or 5.

4.4 Overall Structure

Figure 4 shows our overall algorithm for index-association based dependence analysis. The first step of our index-association based dependence analysis is to construct the dependence test iteration space. If the dependence test space cannot be constructed due to complex index-association functions, we have to assume a worst-case dependence test iteration space, i.e., for each J_k with iteration space (l_k, u_k, s_k), we have $l_k = -\infty$, $u_k = +\infty$ and s_k could be any integer value.

As stated previously, if there exists multiple steps for certain dimension in the dependence test iteration space, dependence analysis must assume a conservative step, often the greater common divisor of all possible steps, in order to compute correct dependence relation. The resultant dependence relation, however, might be conservative. For example, for the loop in Figure 5, the steps for J values can be either 3 or 4. So our index-association based approach has to take the conservative step of 1 in the dependence test iteration space. This will assume array references $A(J + 9)$ and $A(J)$ have cross-iteration dependences. Hence, the original loop I cannot be parallelized. In the next section, we present a technique to handle certain index-association functions with *division*, which can be equivalently transformed to a set of linear expressions. The latter can be used to compute the dependence relation, including dependence distances, more precisely than with traditional techniques.

5 Accurate Dependence Analysis with *Division*

The basic idea here is to replace the non-linear expression with a set of linear expressions and then use these linear expressions during dependence testing with traditional techniques. Specifically, we want to find a set of linear expressions which are equivalent to $J = f(I)$, where the index I has the iteration space (L, U, S) and the function f contains operations such as *plus*, *minus*, *multiplication* and *division*.

Without losing generality, we assume $U \geq L$ and $S > 0$. Let t be the loop trip count for loop I, and we have $t = \lfloor \frac{U-L+S}{S} \rfloor$. Let i_1, i_2, ..., i_t represent the t loop index I values, from the smallest one to the largest one. Let $j_p = f(i_p), 1 \leq p \leq t$, be the corresponding J index values.

First, let us take the loop in Figure 5 as an example. We want to express $J = 5 * I/4$ as a set of linear expressions. For the I value sequence $(1, 4, 7, 10, 13, 16, 19, 22, \ldots, 97, 100)$, the corresponding J value sequence is $(1, 5, 8, 12, 16, 20, 23, 27, \ldots, 121, 125)$. Clearly, the J value sequence is not a linear sequence because the difference between adjacent values vary. However, note that the difference between every pth and $(p+4)$th J values $(1 \leq p \leq t-4)$ is a constant of 15. Therefore, the original J value sequence can be represented as 4 linear sequences, each with a step of 15 and initial value, 1, 5, 8 and 12 respectively.

To generalize the above observation, for a sequence of J values $j_p (1 \leq p \leq t)$, we want to find τ, the number of linear expressions needed to represent j_p, and σ, the step value for each individual linear expression.

The difference between the J values in the J value sequence can be expressed as

$$\begin{aligned} js_1 = j_2 - j_1 = f(i_2) - f(i_1), \\ js_2 = j_3 - j_2 = f(i_3) - f(i_2), \\ \cdots \\ js_{t-1} = j_t - j_{t-1} = f(i_t) - f(i_{t-1}). \end{aligned}$$

With the semantics of τ, we have $js_p = js_{p+\tau}, \forall 1 \leq p, p+\tau, \leq t-1$, holds. This is equivalent to

$$f(i_{p+1}) - f(i_p) = f(i_{p+\tau+1}) - f(i_{p+\tau}), \forall 1 \leq p, p+\tau \leq t-1. \quad (1)$$

Different index-association functions f may require different complexities to compute τ. Conservative methods can also be applied if the compiler is not able to do sophisticated analysis and manipulation. The compiler has to make the worst assumption if it can not find a compiler-time known constant τ, e.g., using the dependence analysis technique in Section 4.

Now suppose τ is available, for each linear expression, we can easily compute the corresponding step as

$$\sigma = f(i_{p+\tau}) - f(i_p), 1 \leq p, p+\tau \leq t-1. \quad (2)$$

In this paper, we do not try to construct the trip count for different linear expressions and rather conservatively assume a trip count which equals to that for the linear expression with the initial value of $f(L)$, which also has the maximum trip count over all τ linear expressions.

With τ and σ available, the $J = f(I)$ can be expressed as

$$J = \tau * I' + r' \tag{3}$$

where I' is an integer variable and its iteration space is $(0, \lceil \frac{f(i_t)-f(i_1)}{\sigma} \rceil, 1)$, and r' is a set of τ discrete numbers $\{f(i_p) | 1 \leq p \leq \tau\}$.

Since the set of linear expressions is equivalent to the original non-linear expression, whether a dependence exists with the original non-linear expression can be determined by whether a dependence exists with the transformed set of linear expressions. For any dependence distance value d (regarding loop index I') computed with transformed linear expressions, the dependence distance in the original I iteration space can be computed based on d and the difference between corresponding r'. For example, suppose that we have a dependence between $j_1 = f(i_1) = \tau * i'_1 + r'_1$ and $j_2 = f(i_2) = \tau * i'_2 + r'_2$, with a dependence distance $i'_2 - i'_1 = d$. We have $f(i_2) - f(i_1) = \tau * d + r'_2 - r'_1$, from which we can further estimate $i_2 - i_1$, maybe conservatively.

As an example, we now show how we compute the τ and σ for the expression $J = f(I) = C * I/D$.

If $C*\tau*S$ is divisible by D, the equation $f(i_{p+1}) - f(i_p) = f(i_{p+\tau+1}) - f(i_{p+\tau})$ will hold. To make $C * \tau * S$ is divisible by D, we can let $\tau = \frac{D}{GCD(C*S,D)}$ where $GCD(C * S, D)$ represents the greatest common divisor of $C * S$ and D.

Now, we show how our technique can determine whether a dependence exists between $A(J + 9)$ and $A(J)$ in Example 3 (Figure 5), i.e., whether there exist any instances of J, say j_1 and j_2, and

$$j_1 + 9 = j_2 \tag{4}$$

has a solution.

With our technique, the non-linear expression $J = 5 * I/4$, where loop I's iteration space is $(1, 100, 3)$, can be represented equivalently by

$$J = 15 * I' + r', r' = (1, 5, 8, 12), I' \textit{ has iteration space } (0, 8, 1) \tag{5}$$

With the linear expression (5), equation (4) is equivalent to

$$15 * i_1 + r_1 + 9 = 15 * i_2 + r_2, \tag{6}$$

where i_1 and r_1 are used for j_1, and i_2 and r_2 for j_2.

To consider whether equation (6) has a solution or not, we have

$$\begin{aligned} 15 * (i_1 - i_2) &= (r_2 - r_1) - 9 \\ &= \{1, 5, 8, 12\} - \{1, 5, 8, 12\} - 9 \\ &= \{-11, -7, -4, 0, 4, 7, 11\} - 9 \\ &= \{-20, -16, -13, -9, -5, -2, 2\} \end{aligned}$$

All possible values on the right-hand side are not divisible by 15, so there exists no solution for (4) and no dependence between $A(J+9)$ and $A(J)$. Therefore, the loop I in Figure 5 can be parallelized successfully.

Our index-association based dependence distance can help both general loop transformations and automatic parallelization because it tries to provide a more accurate dependence test result. In the next section, we particularly illustrate how our technique helps automatic parallelization, i.e., whether a certain level of loop is a DOALL loop or not, and under what condition it is a DOALL loop. We do not explore how our technique helps general loop transformations in this paper.

6 Automatic Parallelization with Index Association

For automatic parallelization, our index-association based dependence analysis can help determine whether a loop, which conforms to our program model in Figure 2 with some non-linear index-association functions f_k, is a DOALL loop or not. For those non-DOALL loops, previous work like [7] generate run-time conditionals under which the loop will be a DOALL loop, to guard the parallelized codes. Our compiler also has the ability to generate proper conditions under which a certain loop I_k is a DOALL loop, such as the example in Figure 1. From Table 1, if the index-association function contains operators *division* and *modulo*, multiple step values may be generated in the dependence test iteration space, which makes dependence analysis conservative. To get more precise dependence analysis results, conditionals are often generated so that we can have fewer step values, often just one, in the dependence test iteration space for one index-assocation function. By combining index-association based dependence analysis and such two-version code parallelization, our compiler is able to parallelize some otherwise hard-to-parallelize loops. For example, our compiler is able to determine that the loops in Figures 3(a) and (b) are not DOALL loops and that the loop in Figure 3(c) is a DOALL loop, based on the dependence analysis in Section 4. We will now work through a more complex example to show how we combine index-association based dependence analysis and two-version code parallelization to successfully parallelize one outer loop.

Figure 6(a) shows the original code where C_2, C_3 and S_2 are all compile-time known constants and C_1 is a loop nest invariant. We also suppose that all right-hand sides of assignments $A(J+k) = \ldots (0 \leq k \leq C_3)$ do not contain references to array A. The original iteration space for loop I_2 is $(I_1C_1, (I_1+1)C_1, S_2)$. With the property of index-association function DIV, we can derive the dependence test iteration space for J (corresponding to the original loop I_2) as $(\lfloor \frac{I_1C_1}{C_2} \rfloor, \lfloor \frac{(I_1+1)C_1}{C_2} \rfloor, (\lfloor \frac{S_2}{C_2} \rfloor, \lceil \frac{S_2}{C_2} \rceil))$, where the step is variant with either $\lfloor \frac{S_2}{C_2} \rfloor$ or $\lceil \frac{S_2}{C_2} \rceil$. Therefore, if the condition $C_3 < \lfloor \frac{S_2}{C_2} \rfloor$ holds, the loop I_2 is parallelizable.

Parallelizing the outer loop I_1 needs more analysis. Here, by analyzing the loop bounds and steps, our compiler is able to determine that if the condition $MOD(C_1, S_2) = 0$ holds, i.e., C_1 is divisible by S_2, the loops I_1 and I_2 actually can be *collapsed* into one loop. Figure 6(b) shows the code after loop collapsing.

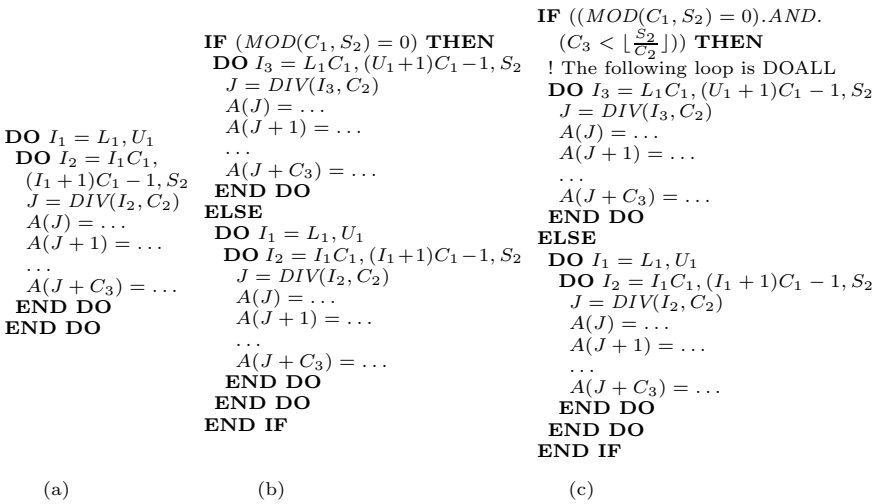


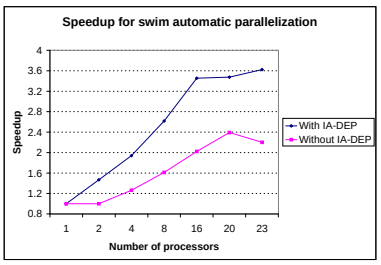
Fig. 6. Example 4

The new loop I_3 in Figure 6(b) can be further parallelized if the condition $C_3 < \lfloor \frac{S_2}{C_2} \rfloor$ holds, as analyzed in the previous paragraph. Figure 6(c) shows the final code where the *collapsed* loop I_3 is parallelized under the condition $MOD(C_1, S_2) = 0$ and $C_3 < \lfloor \frac{S_2}{C_2} \rfloor$. Our compiler is able to successfully parallelize the outer loop I_1 in Figure 6(a).

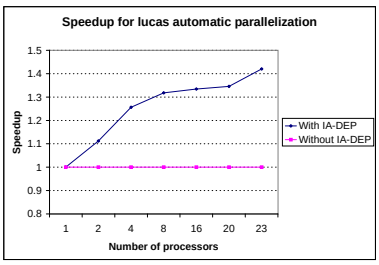
7 Experimental Results

We have implemented our index-association based dependence analysis technique in the Sun ONE Studio [tm] 8 compiler collection [11], which will also be used in our experiments. (We have not implemented the technique presented in Section 5 yet. We plan to evaluate and experiment with it in future releases.) Our compiler has already implemented several dependence analysis techniques for subscripts which are linear forms of enclosing loop indices, such as GCD test, separability test, Banerjee test, etc. Our compiler also implements some sophisticated techniques for array/scalar privatization analysis, symbolic analysis, parallelization-oriented loop transformations including loop distribution/fusion, loop interchange, wavefront transformation [12], etc. Therefore, our compiler can already parallelize a lot of loops in practice. With our new index-association based dependence analysis, we extend our compiler's ability to parallelize more loop nests which otherwise cannot be parallelized.

We choose two programs from the well-known SPEC CPU2000 suite [10], `swim` and `lucas`, which benefit from the technique developed in this paper. In the second quarter of 2003, we submitted automatic parallelization results for SPEC CPU2000 on a Sun Blade [tm] 2000 workstation with 2 1200MHZ UltraSPARC III Cu [tm] processors to SPEC [10], which is the first such submission



(a) `swim`



(b) `lucas`

Fig. 7. Speedup on different number of processors for `swim` and `lucas`

for SPEC CPU2000 on automatic parallelization. Compared to the results on Sun Blade [tm] 2000 with just 1 1200MHZ UltraSPARC III Cu [tm] processor [10], we achieve a speedup of 1.60 for `swim` and a speedup of 1.14 for `lucas`. To evaluate the effectiveness of our technique on more than two processors, we further experimented on a Sun Fire [tm] 6800 server with 24 1200MHZ UltraSPARC III Cu [tm] processors and Solaris [tm] 9 operating system. For each program, we measure the best serial performance as well as the parallel performance with various number of processors up to 23 processors. We did not report the result for 24 processors as in general, due to system activity, it may not bring any speedup over the result for 23 processors.

7.1 `swim`

The benchmark `swim` is a weather prediction program written in Fortran. It is a memory bandwidth limited program and the tiling technique in [9], which has been implemented in our compiler, can improve data temporal cache locality, thus alleviating the bandwidth problem. For example, in one processor of our target machine, the code without tiling runs in 305 seconds and in 134 seconds with tiling. Tiling improves the performance for a single-processor run with a speedup of 2.28 because of the substantially improved cache locality. After tiling, however, some IF statements and MOD operators are introduced into the loop body because of aggressive loop fusion and circular loop skewing [9], which makes it impossible to reuse the same dependence information derived before tiling. To parallelize such loop nests, our dependence analysis phase correctly analyzes the effect of IF statements and MOD operators, and generates proper conditions to parallelize all four most important loops.

Figure 7(a) shows the speedup for `swim` with different number of processors with and without our index-association based dependence analysis, represented by "With IA-DEP" and "Without IA-DEP" respectively. Without index-association based dependence analysis, the tiled code is not able to be parallelized by our compiler. However, our compiler is still able to parallelize all four

important loop nests if tiling is not applied. We regard the result for such parallelization as "Without IA-DEP" parallelization. For processor number equal to 2, the actual "Without IA-DEP" parallelization performance is worse than the performance of the tiled code on one processor, so we use the result for the tiled code on one processor for "Without IA-DEP" result for two-processor result. From Figure 7(a), it is clear that our index-association based dependence can greatly improve parallel performance for `swim`.

Figure 7(a) also shows that parallelization with IA-DEP scales better than without IA-DEP. This is because `swim` is a memory bandwidth limited benchmark and tiling enables better scaling with most data accessed in L2 cache, which is local to each processor, instead of in main memory. This is true also with large data sizes in OpenMP version of `swim`. In March 2003, Sun submitted the performance results for 8/16/24 threads for SPEC OMPM2001 on Sun File [tm] 6800 server [10]. The results show that without tiling, using OpenMP parallelization directives, the speedup from 8 threads to 16 threads is 1.33. With tiling, turning off OpenMP directive parallelization, however, the speedup is 1.44. The performance of with tiling is also significantly better than without tiling, e.g., SPEC scores 14199 vs. 8351 for 16 threads.

7.2 `lucas`

The benchmark `lucas` tests primality of Mersenne numbers. There are mainly two classes of loop nests in the program. One class is similar to our example 4 in Figure 6, and the other contains indexed array references, i.e., array references appear in the subscripts. Currently, our compiler is not able to parallelize loops in the second class. However, with index-association based dependence analysis, it is able to parallelize all important loops in the first class. Figure 7(b) shows the speedup for `lucas` on different number of processors. Note that no speedup is achieved for multiple processor runs without index-association based dependence analysis since all important loops are not parallelized.

8 Conclusion

In this paper, we have presented a new dependence analysis technique called index-association based dependence analysis. Our technique targets a special class of loop nests and uses a decoupled approach for dependence analysis of complex array subscripts. We also present a technique to transform a non-linear expression to a set of linear expressions and the latter can be used in dependence test with traditional techniques. Experiments show that our technique is able to help parallelize some otherwise hard-to-parallelize loop nests.

Acknowledgements

The authors want to thank the entire compiler optimizer group for their efforts to build and continuously improve the SUN's compilers, on which our work has

relied. The authors also want to thank Partha Tirumalai for his helpful comments which greatly improved this paper.

References

1. William Blume and Rudolf Eigenmann. Non-linear and symbolic data dependence testing. *IEEE Transactions of Parallel and Distributed Systems*, 9(12):1180–1194, December 1998.
2. Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, January 1991.
3. Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 15–29, Toronto, Ontario, Canada, June 1991.
4. Mohammad Haghighat and Constantine Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996.
5. Jay Hoeflinger and Yunheung Paek. The access region test. In *Proceedings of the Workshop on LCPC 1999, also in Lecture Notes in Computer Science, vol. 1863, by Springer*, La Jolla, California, August 1999.
6. Dror Maydan, John Hennessy, and Monica Lam. Efficient and exact data dependence analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, Toronto, Ontario, Canada, June 1991.
7. Sungdo Moon and Mary Hall. Evaluation of predicated array data-flow analysis for automatic parallelization. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 84–95, Atlanta, GA, May 1999.
8. William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.
9. Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 215–228, Atlanta, GA, May 1999.
10. Standard Performance Evaluation Corporation, The SPEC CPU2000 benchmark suite. *http://www.specbench.org*.
11. Sun Microsystems, Inc., Sun ONE Studio 8 Compiler Collection. *http://docs.sun.com*.
12. Michael Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Department of Computer Science, Stanford University, August 1992.
13. Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.