Supporting Array Dependence Testing for an Optimizing/Parallelizing C Compiler

Justiani and Laurie J. Hendren

McGill University, Montréal, Québec, Canada

Abstract. The effectiveness of parallelizing and optimizing compilers depends on the ability to do accurate dependence analysis. In the case of programs that use arrays, array dependence analysis methods are critical, and powerful methods for dependence testing have been widely established. In order to collect the input required to actually apply the dependence tester, one must first apply the following *support phases*: (1) locate all admissible loop nests, (2) collect the normalized index expressions for each array reference, and (3) determine which pairs of array references must be tested. When implementing a dependence testing framework in C, each of these support phases must deal with complexities such as the presence of pointers and complicated control flow due to complex loop structures. Furthermore, each phase must be performed as accurately as possible so as to maximize the number of admissible loop nests and minimize the number of dependence pairs requiring testing.

This paper describes the design and implementation of the support phases as developed in the dependence testing framework for the McCAT (McGill Compiler Architecture Testbed) optimizing/parallelizing C compiler. By taking advantage of the simplified and structured intermediate representation, and the advanced points-to (alias) and reaching definition analyses available in the McCAT compiler, we provide a unified framework for implementing all of the support analyses required for array dependence testing. As part of this framework we demonstrate scalar backward analysis, generalized induction variable detection, canonical subscript analysis, symbolic manipulation, and demand-driven constant propagation in the presence of complex C features.

1 Introduction and Motivation

Accurate and efficient data dependency analysis is an important cornerstone of any parallelizing compiler. For programs using arrays and nested loops, various powerful dependence analysis methods have been widely established [3, 5, 6, 11, 12, 13, 17]. The basic dependence problem is to decide whether two subscripted references to the same array in a loopnest access the same memory location under certain constraints imposed by the boundaries of the loop iteration space. In general, the dependence problem reduces to solving a system of linear diophantine equations subject to a set of linear inequality constraints. This is a two-step process. The first phase is to set up a system of dependence equations and inequalities. In the second phase, a decision algorithm determines if the system has integer solution [17]. The goal of the dependence testing is to disprove dependence of as many array subscripted pairs as possible and as early as possible.

This mathematically well-defined problem of dependence testing requires, as input, a set of equations and equalities derived from the application program under analysis. These equations and equalities must be collected in as precise a manner as possible, even for complicated programs that use complex loops and/or pointer data

structures. Thus, one requires a complete framework of *support analyses* that can be used to collect accurate inputs for dependence testers. Without such a framework, even the most powerful dependence testers are useless.

Unlike typical scientific programs written in Fortran, which often have regular loop nests and simple forms of aliasing (due to call-by-reference), the more general features found in C tend to promote more complex loops and the use of data structure abstractions that often involve pointers. Making overly conservative assumptions about loops and/or aliasing due to pointers can significantly reduce the quality of the dependence testing result. Thus, the development of *support analyses* in the context of C parallelizing compilers must effectively handle these additional complexities.

1.1 Motivating Examples

All dependence testing methods are usually built upon the following assumptions: (1) admissible loopnests have a nice, regular behaviour expressed by the behaviour of loop variables, (2) all the array subscripts and loop limits have been represented in canonical forms or affine functions of loop indices, (3) subscript-pairs are collected from pairs of references to the same array name.

However, real C programs often do not fit nicely into these patterns, and the support analyses must deal with several complications. From the examples in Figure 1, we can demonstrate that the following points need to be addressed before any dependence tests can be applied:

Example 1: This program illustrates the problem of detecting whether or not the loop is admissible. In general, admissible loops should not update the loop variable. The presence of pointer assignment in statement S1 means that an *admissible loopnest detector* should check that the assignment to $*s$ does not update the loop variable i. Without precise alias or points-to analysis, one must conservatively assume that $*s$ might refer to i. In our compiler context, we can use the results of our *points-to* (alias) analysis to decide whether or not s can point-to i [1, 2].

Example 2: This program illustrates the case where the programmer has not provided the array subscripts in terms of the loop variables. Thus, we need automatic methods to convert the array subscripts `ind1` and `ind2` into a canonical form relative to the loop variables. This involves scalar backward analysis, induction variable analysis, demand-driven constant propagation, canonical subscript analysis, and possible involvement of symbolic manipulation of the canonical subscript expressions. Furthermore, these methods must all be able to handle expressions using pointer variables.

Example 3: This program illustrates the problem of determining if two array names refer to the same array. In order to be safe, the dependence testers must be used in all cases where the array names *may* refer to the same array. In this program we need to detect whether or not the array names `v` and `w` refer to the same array.

1.2 Our Support Analysis Framework

In this paper we present a framework for *support analyses* to support array dependence testing that has been designed and implemented for the McCAT optimizing/parallelizing C compiler. [1] This framework has been designed to handle the full

[1] The McGill Compiler/Architecture Testbed is being developed in order to study the interaction between compiler techniques and advanced architectural features [8].

```
Example 1 :
main()
{ int  w[100];
  int i,p;
  int *s;
  s = &i;
  ...
  for (i=1;i<=99;++i)
  {
S1 :    *s= 2 * p;
        w[i] = w[i+1];
  }
}
```

```
Example 2 :
main()
{ int  w[10];
  int i,p,q,a,ind1,ind2;
  int *s,*r;
  s = &i;
  r = &q;
  q = 2;
  for(i=1; i<=9; i++)
  {  p  = q + 3;
     *r = p - 4;
     a  = p - i;
     ind1=a-2*((*s)+3*i);
     ind2=p+2*i;
     w[ind1]=w[ind2];
  }
}
```

```
Example 3 :
main()
{ int  v[10];
  int i,j;
  int *w;
  w=v;
  ...
  for(i=1;i<10;++i)
  {
  S1 : v[i-2] = w[i+1];
  }
}
```

Fig. 1. Motivating Example C Programs

complexities of the C language that affect array dependence analysis, while at the same time supporting complete information and compact representation for applying various dependence testing methods.

The rest of this paper is organized as follows. First we discuss the overall structure and foundations of our approach in section 2. Section 3 presents the introduction of our *support analysis* framework, while sections 4, 5, 6 and 7 provides the details of the framework and illustrate them with some simple examples. Finally, we cover related work in section 8 and draw conclusions in section 9.

2 Foundations

Although powerful features of C language make the *support analyses* nontrivial, the McCAT compiler provides an environment which provides the necessary foundations: (1) SIMPLE, a *compositional* structured intermediate representation that was designed to handle various complications in C in a standard, simple and structured manner, (2) precise interprocedural alias information as computed by *points-to* analysis, and (3) reaching definition analysis that includes reaching definitions for pointer variables.

Figure 2 illustrates these important parts of the McCAT environment. Note that the first phase takes a collection of C program files, and produces a simplified and compositional structured representation called SIMPLE. Then, points-to analysis and reaching-definition analysis take as input the simplified representation and provides as output SIMPLE decorated with points-to information and reaching-definition information. The important points relevant for our topic are briefly described in Sections 2.1 and 2.2. A complete presentation of the points-to analysis and reaching-definition analysis is discussed elsewhere [1, 2, 9, 14]. Given the decorated SIMPLE representation, the next step is the support analysis for dependence testing, which is the main topic of this paper. This support analysis takes advantage of *the simplification phase*, *the reaching-definition information* and *the points-to information*.

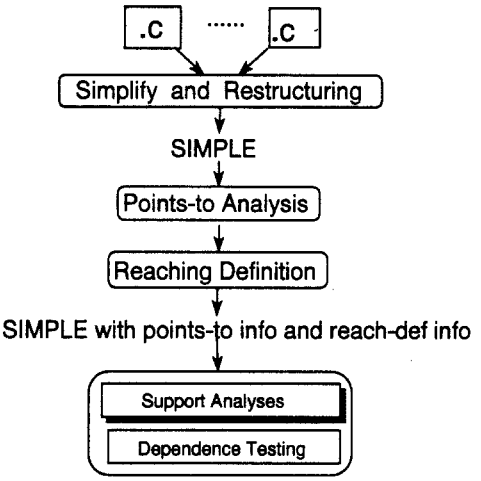

Fig. 2. Overview of the McCAT Environment

2.1 SIMPLE Intermediate Representation

The first foundation of our approach is that McCAT provides a compositional structured intermediate representation called SIMPLE that can represent all the complexities of C in a simple, standard and structured fashion. The *simplify* phase includes: breaking down complex statements into a series of basic statements, simplifying complex expressions and control structures into simple ones, simplifying function arguments to constants or variables, and moving initializations in the declarations into statements in the body. Even though simplified, SIMPLE retains the array and structure references for various analyses that need high-level variable references and type information. Therefore, this representation has been designed to be most suitable for compositional (structured) analysis framework [8].

In Figure 3, we present a simplification of an example program that we use throughout this paper. On the left is the original program, and on the right is the SIMPLE version.

2.2 Points-to and Reaching Definition Analysis

The compositional nature of SIMPLE allows regular and explicit control flow in the program representation that is suitable for compositional analyses. One of the most important analyses is *points-to* analysis which is an interprocedural analysis that computes all possible *points-to* relationships for each program point. We say that `p` *definitely points-to* `x` if `p` definitely contains the address of `x`. Similarly, we say that `p` *possibly points-to* `x` if `p` possibly contains the address of `x`. Unlike traditional alias analysis which computes alias pairs of the form `(p,*x)` and `(*p,**y)`, our points-to analysis gives every important stack location a name, and encodes only the relationships between these names. In most cases the stack location names are just the variable names. The only special case is that we generate special names for stack locations corresponding to function parameters with pointer types. For example, a parameter `a` with type `**int` would have three stack locations `a`, `a(1)`, and `a(2)` where the names `a(1)` and `a(2)` are abstract names corresponding to locations accessible via `*a` and `**a`. This naming scheme and associated points-

```
main()
{ int  w[11];
  int i,k,p,q,a;
  int *r,*y;
  r = &q;
  if(1) y = &p;
  else y = &i;
  p = 4;
  for (i = 1; i <= 10; ++i)
  { q = p + 2;
    k = q + 2;
    a = w[p-(*r+*y)];
    p = *r - 5;
    w[k] = p;
  }
}
```

```
main()
{ int w[11],i,k,p,q,a;
  int *r,*y;
  int temp_3,temp_2,temp_1,temp_0,temp_4;
  r = &q;
  if (1) y = &p; else y = &i;
  p = 4;
  for (i=1;i<=10;i=i+1)
  { q = p + 2;
    k = q + 2;
    temp_2 = *r;
    temp_3 = *y;
    temp_1 = temp_2+temp_3;
    temp_0 = p - temp_1;
    a = w[temp_0];
    temp_4 = *r;
    p = temp_4 - 5;
    w[k] = p;
  }
}
```

Fig. 3. Example of SIMPLE

to abstraction provides a compact representation that can be used directly in our support analyses.

Based on the points-to analysis, the reaching-definition analysis provides a list of all definitions for uses at each program point. For the case of a use of the form `x`, the reaching definitions for `x` include all direct definitions of the form `x = a op b` as well as any definite or possible indirect definitions of the form `*p = a op b` where `p` points-to `x`. In the case of a use of the form `*p`, we include all reaching definitions (both direct and indirect) for all variables pointed to by `p`.

3 Overview of the Support Analysis Framework

The goal of the support analysis is to provide a suitable environment for the application of dependence testing methods. Such a support analysis framework consists of three phases as given below and each of these phases are described more fully in subsequent sections.

Admissible loopnest detection: The first phase is a mechanism to filter only certain types of loopnests that are amenable for analysis with dependence testers. This filter should find as many admissible loopnests as possible.

Subscript Normalization: Once the admissible loopnests have been detected, each array subscript must be expressed in a normalized form (most often in terms of the enclosing loop variables and loop bounds). In general this requires a variety of subscript normalizations such as: scalar backward analysis, generalized induction variable detection, canonical analysis, demand-driven constant propagation, and symbolic manipulation.

Array-pair collection: Given that each array subscript is expressed in a canonical form, the next phase must determine which array reference pairs must be tested. In general, two array references `a[exp1]` and `b[exp2]` must be tested if `a` might refer to the same array as `b`.

4 Admissible Loopnest Detection

Loopnest detection selects certain loopnests that are amenable to be analyzed. The idea behind this selection is to guarantee that the loopnest behaviour can be expressed by the regular behaviour of the loop indices. In dealing with loops, one must address the following potential problems with loopnests that make them inadmissible. Note that some of the problems are made more complex by the general form of loops in C, and the presence of pointers.

Problem 1: Each loop should be defined with the initialization, increment and test on exactly one loop variable. If a pointer variable of the form `*p` is used as the loop variable, then `p` should point-to exactly one variable.

Problem 2: Many dependence testers assume that the increment to the loop variable is 1. If the loop provided by the programmer has a different increment, then loop normalization must be applied [3, 5, 13, 19].

Problem 3: The body of the loop should be free from irregular control flow such as `break` and `continue`. The loop body should not call functions that update the loop variables.

Problem 4: The loopnest body should not modify the loop variables (either directly, or indirectly through a pointer).

Problem 5: The loopnest body should not modify the value of the increment or loop bound. There may exist some programs where there is an assignment to the increment or loop bound, but this assignment does not change the value.

Problems 1, 2 and 3 are very obvious. We choose to illustrate problems 4 and 5 using the two examples in Figure 4. In example 1, at program point S1, the points-to information says that s *definitely points-to* i. Since $*s$, which is exactly i, is modified in S1, we can not express any array subscripts in term of linear function of loop variable i anymore, because i is not regularly incremented by 1. Thus, the loopnest in example 1 is not admissible. Suppose if in S1 we have information saying that s *definitely points-to* some variable other than i, then the loopnest is admissible. Therefore, in an environment with pointers, without any points-to information, in order to be *safe*, the analysis may result in the worst assumption. In example 2, at program point S1, loop-bound variable *tem*0 is being modified in the loopnest scope. Yet, the redefinition does not really change the value of *tem*0 as recognized in the loop header since the modifiers are both defined outside the loopnest scope. Thus, this loopnest can be categorized as admissible.

```
Example 1 : Bad-loop
main()
{ int  w[100],i,p,*s;
  s = &i;
  for (i = 1; i <= 99; ++i)
  {
S1 :     *s= 2 * p;
         w[i] = w[i+1];
  }
}
```

```
Example 2:
m = 5; n = 9;
for(i=0;i<m;i=i+1)
{    tem1 = n + m;
     tem0 = tem1 + n;
     for(j=0;j<tem0;j=j+1)
     {  c[i][j]=c[i][j];
        tem1 = n + m;
   S1: tem0 = tem1 + n;
     }
}
```

Fig. 4. Example Loopnests

5 Subscript Normalization

Having filtered all admissible loopnests in the program, the next step is subscript normalization for every array reference appearing in the admissible loopnests. When designing and implementing the subscript analysis for C, the following three major problems may arise:

Problem 1: The first problem is that programmers often reduce the number of redundant computations using a scalar variable as temporary variable to store the value of a common subexpression [19]. Then this temporary scalar variable is used as the array subscripts such that the array indices are not in the form of linear function of loop variables. In the case of our SIMPLE intermediate representation, each complex array reference is broken down into a series of three address statements. This causes a similar problem.

Problem 2: The second problem is that there may exist some *induction variables* whose values are systematically incremented or decremented by a certain value in a loop. The use of such induction variables may *hide* the linearity and admissibility conditions of array subscripts. However, often the induction variables may be rewritten relative to the loop variables, thus making more subscripts admissible.

Problem 3: The final major problem is that it is unlikely that all array subscript expressions are written in the standard canonical form $a_0 + a_1 * i_1 + \cdots + a_n * i_n$, where $a_0, a_1, \cdots, a_n$ are integer coefficients and $i_1, i_2, \cdots, i_n$ are loop variables, such that the dependence testing methods can not be applied directly. Thus, each index expression must be rewritten to conform to the standard canonical form.

The goal of the subscript normalization is essentially to capture the canonical forms of array subscripts in order to apply dependence testing. This normalization is achieved through a three-step process: (1) build a *general expression tree* which captures all possible index expressions and induction variables (uses points-to and reaching definition analysis), (2) given the general expression tree, build a list of possible *subscript expression trees*, and (3) express each subscript expression in canonical form.

5.1 Phase 1 : Building a General Expression Tree

In our SIMPLE intermediate form, each complex array subscript expression has been simplified to either a constant or a variable name. In order to collect all possible expressions corresponding to a variable name index, we perform a backward demand-driven analysis at the loopnest level using reaching definition and points-to information to build a general expression tree.

To illustrate our approach, consider the previous example in Figure 5(a). Consider that we want to build a general expression tree for index `tem0` of array `w` in statement `S1` as shown in Figure 5(b). The basic idea is that we trace back through all reaching definitions of `tem0` building a general expression tree. In this case the index `tem0` has definition `tem0 = p-tem1` that reaches `S1`, so we build the expression `p-tem1` under the index `tem0`. Similarly, we continue this process for `p` and `tem1` recursively. When an indirect reference is reached, we use the points-to information to expand the indirection with all variables that the indirect reference points-to. For example, `*r` is expanded to `q` and `*y` is expanded to `p` and `i`. Moreover, there is a possibility that *an induction pattern* is detected during the backward process. This is when the same variables with the same reaching-definitions are repeated in the

backward process, such as variable p inside the dashed box that shows induction pattern in the example in Figure 5(b). If this is the case, after the induction pattern tree is completely built, an *induction processing function* is called to calculate the induction formula for all the induction variables existing in the pattern. From our example, the induction calculation is done for variables p, tem4 and q. If later there is a *use* of an induction variable that already has a calculated formula, such as variables p and q under *r and *y in Figure 5(b), the backward analysis stops for that particular path, and keeps the pointer to the calculated formula for later use.

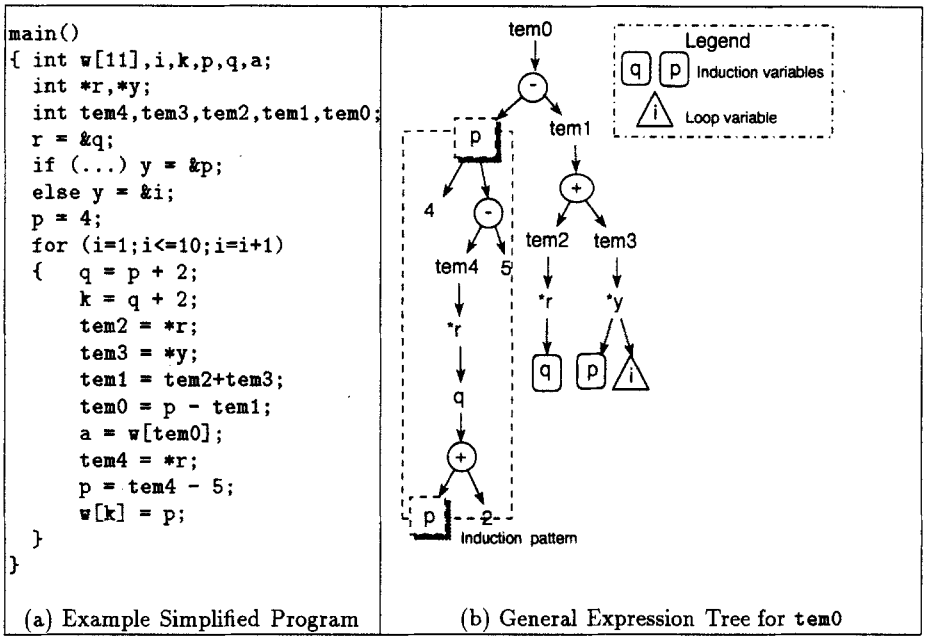

Fig. 5. Building General Expression Tree

To summarize, a general expression tree is represented with the original variable at the root, and leaves of the following form: (a) a constant, (b) a loop variable, (c) an induction variable, or (d) a loop invariant variable (i.e. a variable with a definition outside of the loopnest). Interior nodes in the expression tree are of the following form: (a) a variable (has children representing all binary or unary expressions that could define the variable), or (b) an indirect reference (has children representing all variables pointed to by the reference).

In some respects, this general expression tree form is similar to SSA-form [4, 18] in the sense that it captures an induction pattern. The major difference is that our technique builds a structure specific for particular indices. Furthermore, our structure captures all possible expressions due to both control flow (multiple possible reaching expressions) and indirect pointer references (multiple possible pointed-to variables).

5.2 Phase 2 : Building a List of Expression Trees

The next step of subscript normalization is the process of building a list of *alternative expression trees* for an array index from the general expression tree built in the previous phase.

We use the previous example to illustrate this phase. If we look at the root and the leaves of the general expression tree in figure 5, we can derive that there are two possible canonical forms for array index $tem0$ that comes from expression $tem0 = p - (q + p)$ or the expression $tem0 = p - (q + i)$. Furthermore, p and q are induction variables where p is equivalent to $-3 * i + 7$, and q is equivalent to $-3 * i + 9$. Therefore, the array index $tem0$ has two possible canonical values, which are $3 * i - 9$ and $-1 * i - 2$. The list of subscript expression trees for $tem0$ built in this phase is shown in Figure 6 (left). The list of expression trees is built by traversing the general expression tree bottom up, building a list of possible expression trees for each sub-tree. In addition, all induction variables are expanded to their appropriate expressions.

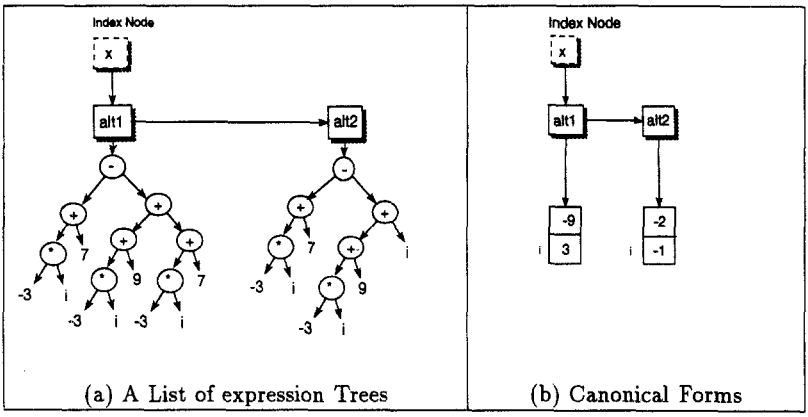

Fig. 6. Canonical Analysis

5.3 Phase 3 : Canonical Analysis

After creating the list of expression trees of indices, the next step is canonical analysis. This phase is a recursive traversal on the subscript expression tree of each index which applies rewriting rules to get the canonical form $a_0 + a_1 * i_1 + \cdots + a_n * i_n$ where $i_1, i_2, \cdots, i_n$ are loop indices, and $a_0, a_1, \cdots, a_n$ can be either integer coefficients or unresolved symbolic coefficients. For example, expression trees for index $tem0$ in Figure 6(a) will generate arrays of integer coefficients (canonical form) as shown in Figure 6(b). The following rules are incorporated to get the canonical form :

Rule 1 Constant Distribution: This is the case when we have an expression of the form $c*(a_0+a_1*i_1+\cdots+a_n*i_n)$ where $a_0, a_1, \cdots, a_n$ are integer coefficients and c is integer constant, then we transform it into $(c*a_0)+(c*a_1)*i_1+\cdots+(c*a_n)*i_n$.

Rule 2 Negative Propagation: This is the case when there are some *additive terms* scoped by a negative sign, we distribute the negative sign down to the coefficient level. For example expression $a_0-(a_1*i_1+a_2*i_2)+\cdots+a_n*i_n$ will be transformed into $a_0 + (-a_1) * i_1 + (-a_2) * i_2 + \cdots + a_n * i_n$.

Rule 3 Coefficient Grouping: This is the case when there is more than one coefficient of the same index expressed in some *additive terms*, such as $a_0 + a_1 * i_1 + \cdots + a_n * i_n + p * i_1 - q * i_n$. We transform it into $a_0 + (a_1 + p) * i_1 + \cdots + (a_n - q) * i_n$.

Rule 4 Symbolic Coefficient Resuming: This is the case where some coefficients are not integer. The above three rules are applied, but the remaining symbolic coefficients are expressed in *symbolic expression subtrees.*

When the above rules do not resolve the canonical forms totally (all integer coefficients are formed) or partially (some symbolic coefficients left), we then apply partial symbolic elimination/execution on the expression trees, which can result in canonical forms.

6 Collecting Array-Pairs

The final phase of the support analysis is to determine which array reference pairs must be tested. When dealing with C this can be relatively complicated since the arrays are often referenced via pointers. Thus, we must use the points-to information to determine when two array references may refer to the same actual array.

6.1 Points-to Analysis for Arrays

When calculating points-to information, the most straight-forward approach is to approximate an entire array with one stack location. Thus, the information that `a` definitely points to `b` means that `a` points to the first location of array `b`.

Example 1 in Figure 7 gives an example of points-to information collected for a simple program. After the statement `c = a` we have the information the `c` definitely points-to `a`. This means that direct references to array `a` and indirect references to array `c` might interfere, and they must be tested.[2] After the conditional we have the information that `d` possibly points-to `a` or `d` possibly points to `b`.

We may also get points-to relationships via parameters that point to arrays. In Example 2, Figure 7, we see that in the function `g`, we have the information that parameter `x` definitely points-to an invisible location `x(1)`, while `y` definitely points-to a *different* invisible location `y(1)`.[3] This means that indirect references to array `x` and array `y` are guaranteed to be distinct, and there is no need to do dependence tests on such pairs. However, in function `f` we have the situation where `x` definitely points-to `x(1)` and `y` definitely points-to `x(1)` as well. Thus, the indirect references to `x` and `y` in function `f` may interfere, and the dependence tester must be applied to such pairs.

If we also consider the possibility of pointer arithmetic and the ability to capture the address of interior elements of arrays, a better approach for points-to analysis is to abstract the entire array as two stack locations: one stack location stands for the location of the first element of the array, and the other stack location stands for the rest of the array. Thus, for each array `a`, we have information about `a_head` and `a_tail`. In Example 3, Figure 7, after the statement `b = a` we have the information that `b` definitely points-to `a_head`, while after the statement `c = &a[exp]` we have

[2] Note that in C one must look at the *type* of the array to determine if it is a direct reference or an indirect reference. In Example 1 there will be *direct* references to `a` and `b`, and *indirect* references to `c` and `d`.

[3] These invisible location names are generated by the points-to analysis, and are used as anonymous names for variables that are not visible in the scope of the procedure under analysis.

the information that c possibly points-to a_head or c possibly points-to a_tail (assuming that exp could be any valid index into a, including 0.). The final statement, d = b++ illustrates that after incrementing b, d definitely points-to a_tail.

```
Example 1
main()
{  int a[100], b[100], *c, *d;
   c = a;  /* c = &a[0] */
   /* (c -> a) */
   ...
   if (exp)
     d = a;
   else
     d = b;
  /* (d -> a)?  (d -> b)? */
   ...
}
```

```
Example 2
main()
{  int a[100], b[100];
   g(a,b);
   f(a,a);
}
g(int *x, int *y)
{  /* (x -> x(1))
         (y -> y(1)) */
   ...
   x[i] = y[i+1];
   ...
}
f(int *x, int *y)
{  /* (x -> x(1))
         (y -> x(1)) */
   ...
   x[i] = y[i+1];
   ...
}
```

```
Example 3
main()
{  int a[100], *b, *c, *d;
   b = a; /* &a[0] */
   /* (b -> a_head) */
   ...
   c = &a[exp];
   /* (c -> a_head)?
      (c -> a_tail)? */
   ...
   d = b++;
   /* (d -> a_tail) */
}
```

Fig. 7. Examples of Points-to analysis for arrays

6.2 Using Points-to Analysis

Given the points-to information as outlined in the previous section, the problem of collecting array pairs for dependence testing is vastly simplified. If one array reference is a write, and the other array reference is a read or write, then dependence testing must be performed if it is possible that the array references refer to the same actual array. Given points-to analysis, there are three ways in which two array references refer to the same array:

1. if both are *direct* references, and they refer to the same array name; *or*
2. if one is a *direct* reference to some array a, and the other is an *indirect* reference to some array b, and b points-to a; *or*
3. if both are *indirect* references via some names a and b, and there exists a third abstract stack name c such that a points-to c and b points-to c.

Thus, to collect all array pairs to be tested, one just considers all read/write and write/write pairs, and determines if one of these three cases applies. If so, the pair must be tested.

7 Advanced Features

7.1 Symbolic Manipulation

The symbolic manipulation and execution of expressions is extremely useful for solving symbolic dependence analysis. As a result of our support analysis, the canonical forms for index expressions are expressed as an array of either integer coefficients or symbolic expression subtrees. This representation allows us to perform several symbolic manipulations in a straightforward manner.

First, when a pair of subscript expressions contain the same symbolic term, we have to make sure that the same definitions reach the pair of variables involved in both expressions. This is to guarantee that there is no update to the variables involved in the symbolic comparison along the control flow paths between the two references. If the above condition is satisfied, then we can apply symbolic inequality and symbolic elimination when forming the dependence equation for the dependence testers.

To illustrate this, consider the examples in Figure 8. In example 1, after applying the subscript normalization phase, we get the canonical form $(1)i+(2)j+(x-y)$ as the index of array w in statement S1 and $(1)i+(1)j+(2+x-y)$ as the canonical index of statement S2. Since there are no updates to variables x and y along the control flow paths between statements S1 and S2, *symbolic elimination* can be applied in order to get the subscript dependence equation $i_1+2j_1-i_2-j_2-2=0$. Similarly, in example 2, the symbolic elimination on the indirect addressing can be applied, since there are no updates of variable x or array b along the control flow path between S1 and S2.

```
Example 1 :
main()
{ int  w[100];
  int i,j,x,y;
  x = ...;
  y = ...;
  ...
  for (i=1;i<=9;++i)
   for (j=1;j<=9;++j)
   {
S1 : w[i+2j+x-y] = ...;
S2 :          ... = w[i+j+2+x-y];
   }
}
```

```
Example 2 :
main()
{ int  w[100];
  int  b[10];
  int i,j,k,x;
  x = ...;
  ...
  for (i=1;i<=9;++i)
   for (j=1;j<=9;++j)
   {
S1 :  w[i+2j+x-b[k]]= ...;
S2 :             ... =w[i+j+2+x-b[k]-2];
   }
}
```

Fig. 8. Symbolic Elimination

7.2 Extended Backward Analysis

Currently, the backward analysis for subscript normalization is done in the scope of loopnests, since it is used to reveal the subscript expressions in terms of loop variables. The method used in this backward analysis is a solid foundation for more general uses of backward demand-driven analysis. For example, we can extend the backward analysis to handle complete function bodies, or even to handle interprocedural reaching definitions. The extension of the backward analysis to the function level would be useful for *loop invariant* variables and it can improve the power of symbolic manipulation and constant propagation.

Figure 9 shows us some examples of how the extended backward analysis gives some advantages in symbolic manipulation and constant propagation. In example 1, if we apply backward analysis at the loopnest level, we get the canonical pair $(1)i+(2)j+(y-6)$ and $(1)i+(1)j+(x+2)$. Whereas, the extended backward analysis can capture the definition for y in terms of x so that the canonical pair will be $(1)i+(2)j+(x-11)$ and $(1)i+(1)j+(x+2)$. Applying symbolic elimination to this pair will result in $i_1+2j_1-i_2-j_2-13=0$ for the dependence equation. In

example 2, the extended backward analysis captures the intraprocedural constant propagation. The backward analysis at the loopnest level produces the canonical pair $(1)i + (2)j + (y-6)$ and $(1)i + (1)j + (x+2)$. Whereas, the extended backward analysis can capture definitions for $y = x - 4$ and $x = 8$ such that the canonical pair will be $(1)i + (2)j + (-2)$ and $(1)i + (1)j + (10)$.

```
Example 1 :
main()
{ int  w[100];
  int i,j,x,y;
  x = ...;
  y = x-5;
  ...
  for (i=1;i<=9;++i)
   for (j=1;j<=9;++j)
   {
s1 :  w[i+2j+y-6] = ...;
S2 :          ... = w[i+j+2+x];
   }
}
```

```
Example 2 :
main()
{ int  w[100];
  int i,j,x,y;
  x = 8;
  y = x-4;
  ...
  for (i=1;i<=9;++i)
   for (j=1;j<=9;++j)
   {
S1 : w[i+2j+y-6] = ...;
S2 :          ... = w[i+j+2+x];
   }
}
```

Fig. 9. Extended Backward Analysis

8 Related Work

The traditional treatment of support analyses usually consists of several phases such as scalar forward substitution, induction variable substitution, canonical transformation and constant propagation, where each phase is a transformation of program code segments [19]. Since subscript normalization is the inverse of *redundant expression elimination*, applying the sequences of subscript transformations will result in *some new code segments* which require *dead code elimination* and *redundant expression elimination* to be reapplied, which is inefficient.

Recently, an implementation based on SSA using use-def chains has been described [15]. Instead of using def-use chains for each definition, as in [10, 16], which supports forward-flow analysis due to consistency of direction between flow analysis and def-use chains, the SSA-based method offers a demand-driven data-flow analysis which typically requests information at a program point from its data-flow predecessors. In some respects, the SSA-based approach is similar to our approach - our reaching definition information is basically use-def chains which enable the demand-driven analysis. The key difference is that we base our analysis on tree-based compositional intermediate representation and structured analysis, while the SSA-based approach is built on graph-based analysis.

Detecting induction variables using the SSA form has also been discussed [4, 18]. The proposed technique is based on SSA graphs and a modified Tarjan's algorithm for recognizing SCRs (Strongly Connected Regions) of CFG. This technique is then expanded to recognize other types of IVs such as Wrap-Around Variables (WAV), Flip-Flop and Periodic variables (FPV), Non-Linear Induction Variables (NLIV) and Monotonic Variables (MV). In addition, the algorithm can also identify Nested Induction Variables (NIV) in the presence of determinable trip counts of the loop. Currently, our induction processing (based on the patterns collected during the backwards demand-driven analysis) can handle basic, multiply-defined,

mutually-updated and nested induction variables properly, and can detect periodic and monotonic induction variables, but does not calculate the formulas. Non-linear and geometric induction variables are currently being incorporated without any serious problems.

A framework to solve the dependence problem in the presence of unknown symbolic expressions has also been introduced [6, 7]. This approach uses forward flow analysis in order to do symbolic constant propagation, partial symbolic execution and approximate semantic analysis in the program, before applying classical data dependence testers. The symbolic constant flow analysis is applied on multi-level linked list representing symbolic expressions. These symbolic manipulations are also available in our support analysis using a different approach. Instead of going forward, our approach is demand-driven backward analysis at the loopnest level, which is currently extended to backward analysis the top of the functions for the loop invariant variables. This extension exposes more opportunities for symbolic.manipulation and will allow for demand-driven constant propagation in the presence of points-to information.

A major difference in our approach is that we fully incorporate points-to information in detecting loopnests, collecting array-pairs and replacing any occurrences of indirect reference and indirect component reference along the backward path in the demand-driven subscript analysis. This allows us to get more precise canonical expressions and thus more precise dependence results.

9 Conclusions and Further Work

This paper has discussed the design and implementation of the support analyses required for precise array dependence testing in our optimizing/parallelizing McCAT C compiler. We have presented this support analysis as three phases: (1) admissible loopnest detection, (2) the collection of normalized index expressions (subscript normalization), and (3) the determination of array-reference pairs that must be tested.

Our approach builds on the structured intermediate representation (SIMPLE), and the results of points-to and reaching-definition analysis. The SIMPLE representation provides a good environment for detecting admissible loop nests, while the points-to analysis and reaching-definition analysis enables our subscript normalization. The points-to analysis is also a key factor in detecting admissible loops and determining the array-pairs that must be tested. Without such analyses, overly conservative assumptions would have to be made, and spurious dependence tests would be required.

We have implemented this support analysis and connected it to a practical dependence testing framework based on a wide variety of dependence testers [5, 11, 12, 17]. We plan to continue this work by incorporating more advanced features in the support analysis, and by experimenting with the effect of precise points-to analysis on the accuracy of the dependence testing.

References

1. M. Emami, L. J. Hendren, and R Ghiya. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation (to appear)*, June 20-24, 1994.
2. Maryam Emami. A Practical Interprocedural Alias Analysis for An Optimizing C Compiler. *Master's thesis, McGill University*, July 1993.

3. Paul Feautrier. Dataflow Analysis of Array and Scalar References. *In the series, Research Monographs in Laboratoire MASI, Universite Paris et Marie Curie, Paris,France*, September 1991.
4. Michael P. Gerlek. Detecting Induction Variables using SSA-form. *OGI-CSE Technical Report 93-014, Oregon Graduate Institute*, June 7, 1993.
5. Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical Dependence Testing. *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 15–29, June 26-28, 1991.
6. M. Haghighat and C. Polychronopoulos. *Symbolic Dependence Analysis for High-Performance Parallelizing Compilers.* Pitman, London and MIT Press, Cambridge, MA, 1991.
7. M. Haghighat and C. Polychronopoulos. Symbolic Analysis: A Basis for Parallelization, Optimization and Scheduling of Programs. *Sixth Annual Workshop on Languages and Compilers for Parallel Computing, Portland, Oregon*, pages dd1–dd23, August 12-14, 1993.
8. L. J. Hendren, C. Donawa, M. Emami, G. Gao, Justiani., and B Sridharan. Designing the McCAT Compiler based on a Family of Structured Intermediate Representations. *Fifth Workshop on Languages and Compilers for Parallel Computing. Also to appear in LNCS*, August 1992.
9. L.J. Hendren, M. Emami, C. Verbrugge, and R. Ghiya. A Practical Context-sensitive Interprocedural Analysis Framework for C Compilers. *ACAPS Technical Memo 72, School of Computer Science, McGill University*, July 1993.
10. Richard Johnson and Keshav Pingali. Dependence-based Program Analysis. *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 78–89, June 23-25, 1993.
11. Xiangyun Kong, David Klappholz, and Kleanthis Psarris. The I Test : An Improved Dependence Test for Automatic Parallelization and Vectorization. *IEEE Transactions on Parallel and Distributed systems*, 2(3):342–349, July 1991.
12. Vadim Maslov. Delinearization : An Efficient Way to Break Multiloop Dependence Equations. *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 152–161, June 17-19, 1992.
13. Dror E. Maydan, J.L.Hennessy, and Monica S. Lam. Efficient and Exact Data Dependence Analysis. *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 1–14, June 26-28, 1991.
14. Bhama Sridharan. An Analysis Framework for the McCat Compiler. *Master's thesis, McGill University*, December 1992.
15. Eric Stoltz, Michael P. Gerlek, and Michael Wolfe. Extended SSA with Factored Use-Def Chains to support Optimization and Parallelization. *OGI-CSE Technical Report 93-013, Oregon Graduate Institute*, June 7, 1993.
16. Mark N. Wegman and F. Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.
17. Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambrigde, Massachusetts, 1989.
18. Michael Wolfe. Beyond Induction Variables. *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 162–174, June 17-19, 1992.
19. Hanz Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, Addison-Wesley Pub. Co., New York, 1990.