Unified Interprocedural Parallelism Detection *

J. Hoeflinger (hoefling@uiuc.edu)
Center for Simulation of Advanced Rockets
University of Illinois at Urbana-Champaign

Y. Paek (ypaek@kaist.ac.kr) and K. Yi (kwang@kaist.ac.kr)
Department of Electrical Engineering & Computer Science
Korea Advanced Institute of Science & Technology

Abstract. In this paper, we outline a way of detecting parallelism interprocedurally within a program. Our method goes beyond mere dependence testing, to embrace methods of removing dependences as well, namely privatization, induction recognition and reduction recognition. This method is based on a combination of techniques: a "universal" form for representing memory accesses within a section of code (the Access Region Descriptor), a technique for classifying memory locations based on the accesses made to them by the code (Memory Classification Analysis), and a dependence test (the Access Region Test). The analysis done with Access Region Descriptors is based on an intersection operation, for which we present an algorithm. Access Region Descriptors are independent of any declarations which may exist in a program, so they are subroutine- and language-independent. This makes them ideal for use in interprocedural analysis. Our experiments indicate that this test is highly effective for parallelizing loops containing very complex subscript expressions.

Keywords: Parallelization, Dependence Analysis, Compiler, Privatization

1. Introduction

Modern computer architectures, with ever-faster processors, make it increasingly important for parallelizing compilers to do their analysis interprocedurally. A compiler that parallelizes only intraprocedurally is confined to parallelizing loops in the leaf nodes of a call graph. There are, quite often, not enough operations in the leaf nodes to make parallelization pay off. For loop parallelization within the shared memory model, the compiler should parallelize at the highest level in the call graph where parallelization is possible, to overcome parallel loop overhead costs. In addition, interprocedural dependence analysis is essential for producing SPMD message passing code from a serial program.

* This work is supported in part by Army contract DABT63-95-C-0097; Army contract N66001-97-C-8532; NSF contract MIP-9619351; and a Partnership Award from IBM and the KAIST seed grant program. This work is not necessarily representative of the positions of the Army or the Government.

Traditional dependence testing has been developed without regard to its applicability across procedure boundaries. All pairs of memory references which may access the same memory location within a loop are compared. These memory references occur at discrete points within the loop, thus we say that these methods are *point-to-point* dependence tests. Point-to-point tests require $O(n^2)$ comparisons where n is the number of memory references to a particular array within the loop. Obviously, as the position within the call graph gets further from the leaves, the number n can grow, and this growth can cause interprocedural point-to-point dependence testing to get unwieldy.

These considerations have motivated us to take a different approach to dependence testing: the memory accesses within a program section are summarized, then the summaries are intersected to determine dependence between the sections. Through a series of experiments, we have found that this approach not only reduces the number of comparisons for dependence testing, but also allows us to handle very complex array subscript expressions.

This paper is organized as follows. After discussing previous related work in Section 2, we will continue our discussion in Section 3 by showing how we summarize the memory access activity of an arbitrary program section. Then, in Section 4 , we will describe a novel notation which is practical for summarizing various complex array accesses encountered in many scientific programs, and in Section 5, we show how to use access summaries stored in the this notation to perform multiple-subscript, interprocedural, summary-based dependence testing. To evaluate the effectiveness of our dependence test, we implemented it in the Polaris [4] compiler and experimented with actual codes from Perfect, SPEC, and NASA benchmark suites. The experimental results presented in Section 6 show that our test holds promise for better detection of parallelism in actual codes than other tests.

2. Previous Work

2.1. Intraprocedural Dependence Testing

Most point-to-point dependence testing methods rely on an equation-solving paradigm, where the pair of subscript expressions for two array reference sites being checked for dependence are equated. Then an attempt is made to determine whether the equation can have a solution, subject to constraints on the values of variables in the program, such as loop indices. In the general case, a system of linear relations is built and a solution is attempted with a linear system solver to determine if the same memory location can be accessed in different loop iterations.

Two of the earliest point-to-point dependence tests were the GCD Test and the Banerjee Test [2]. In practice, these were simple, efficient, and successful at determining dependence, since most subscript expressions occurring in real scientific programs are very simple. However, the simplicity of these tests results in some limitations. For instance, they are not effective for determining dependence for multidimensional arrays with *coupled* subscripts, as stated in [9]. Several *multiple-subscript* tests have been developed to overcome this limitation: the multidimensional GCD Test [2], the λ-test [12], the Power Test [21], and the Delta Test [9].

The above tests are exact in commonly occurring special cases, but in some cases are still too conservative. The Omega Test [16] provides a more general method, based on sets of linear constraints, capable of handling dependence problems as integer programming problems.

All of the just-mentioned tests have the common problem that they cannot handle subscript expressions which are non-affine. Non-affine subscript expressions occur in irregular codes (subscripted-subscript access), in FFT codes (subscripts frequently involving 2^I), and as a result of compiler transformations (induction variable closed forms and inline expansion of subroutines). To solve this problem, Pugh, et al.[17] enhanced the Omega test with techniques for replacing the non-affine terms in array subscripts with symbolic variables. This technique does not work in all situations, however. The Range Test [3, 4] was built to provide a better solution to this problem. It handles non-affine subscript expressions without losing accuracy. Overall, the Range Test is almost as effective as the Omega Test and sometimes out-performs it, due mainly to its accuracy for non-affine expressions [3]. One critical drawback of the Range Test is that it is not a multiple-subscript test, and so is not effective for handling coupled-subscripts.

2.2. Interprocedural Summarization Techniques

Interprocedural dependence testing demands new capabilities from dependence tests. Point-to-point testing becomes unwieldy across procedure boundaries, and so has given way to dependence testing using summaries of the accesses made in subroutines. The idea of using access summaries for dependence analysis was previously proposed by several researchers such as Balasundaram, et al.[1] and Tang [18]. Also, the Range Test, though it is a point-to-point test, uses summarized range information for variables, obtained through abstract interpretation of the program.

To perform accurate dependence analysis with access summaries, the compiler needs some standard notation in which the information about array accesses is summarized and stored for its dependence analyzer. Several notations have been developed and used for dependence analysis techniques. Most notable are *triplet notation* [3, 10, 19] and sets of linear constraints [1, 7, 18]. However, as indicated in [11], existing dependence analysis techniques have deficiencies directly traceable to the notations they used for access summarization. Triplet notation is simple to work with, but not rich enough to store all possible access patterns. Linear constraints are more general, but can not precisely represent the access patterns due to non-affine subscript expressions, and require much more complex operations.

So, clearly there is room for a new dependence test and a new memory access representation, to overcome the limitations of existing techniques.

2.3. Parallelism Detection

While dependence testing has been studied exhaustively, a topic which has not been adequately addressed is a unified method of **parallelism detection**, which not only finds dependences, but categorizes them for easy removal with important compiler transformations.

Eigenmann, et al [8] studied a set of benchmark programs and determined that the most important compiler analyses needed to parallelize them were array privatization, reduction and induction (idiom) analysis, and dependence analysis for non-affine subscript expressions, and that all of those must be done in the presence of strong symbolic interprocedural analysis.

The need for improved analysis and representational techniques prompts us to go back to first principles, rethink what data dependence means and ask whether dependence analysis can be done **with compiler transformations in mind**.

The key contribution of this paper is the description of a *general interprocedural parallelism detection* technique. It includes a general dependence analysis technique, described in Section 5, called the *Access Region Test* (ART). The ART is a multiple-subscript, interprocedural, summary-based dependence test, combining privatization and idiom recognition. It represents memory locations in a novel form called an *Access Region Descriptor* (ARD)[11], described in Section 4, based on the Linear Memory Access Descriptor of [14].

3. Memory Classification Analysis

In this section, we formulate data dependence analysis in terms of a scheme of classifying memory locations, called Memory Classification Analysis (MCA), based on the order and type of the accesses within a section of code. The method of classifying memory locations is a general one, based on abstract interpretation[5, 6] of a program, and may be used for purposes other than dependence analysis.

The traditional notion of data dependence is based on classifying the relationship between two accesses to a single memory location. The operation done (Read or Write), and the order of the accesses determines the type of the dependence. A data dependence arc is a directed arc from an earlier instruction (the *source*) to a later instruction (the *sink*), both of which access a single memory location in a program. The four types of arcs are determined as shown in Table I.

Table I. Traditional data dependence definition.

Dependence Type	Input	Flow	Anti	Output
Earlier access	Read	Write	Read	Write
Later access	Read	Read	Write	Write

Input dependences can be safely ignored when doing parallelization. Anti and output dependences (also called *memory-related* dependences) can be removed by using more memory, usually by privatizing the memory location involved. Flow dependences (also called true dependences) can sometimes be removed by transforming the original code through techniques such as induction variable analysis and reduction analysis [20].

A generalized notion of data dependence between arbitrary sections of code can be built by returning to first principles. Instead of considering a single instruction as a memory reference point, we can consider an arbitrary sequence of instructions as an indivisible memory referencing unit. The only thing we require is that the memory referencing unit be executed entirely on a single processor. We refer to this memory referencing unit as a *dependence grain.*

DEFINITION 1. *A section of code representing an indivisible, sequentially executed unit, serving as the source or sink of a dependence arc in a program, will be called a* **dependence grain**.

This definition of dependence grain corresponds to the terms coarse- and fine-grained analysis, which refer to using large and small dependence grains, respectively.

If we want to know whether two dependence grains may be executed in parallel, then we must do dependence analysis between the grains. Since a single grain may access the same memory address many times, we must summarize the accesses in some useful way and relate the type and order of the summaries to produce a *representative dependence arc* between the two grains.

DEFINITION 2. *A* **representative dependence arc** *is a single dependence arc showing the order in which two dependence grains must be executed to preserve the sequential semantics of the program. A single representative dependence arc summarizes the information which would be contained in multiple traditional dependence arcs between single instructions.*

For medium- and coarse-grain parallelization, there can be many accesses to a single memory location within each dependence grain. Instead of keeping track of the dependences between all possible pairs of references which have a reference site in each grain (as in point-to-point testing), it is desired to represent the dependence relationship between the two grains, for an individual memory location, with a single representative dependence arc.

There are many possible ways to summarize memory accesses. The needs of the analysis and the desired precision determine which way is best. To illustrate this idea, the next two sections show two ways of summarizing accesses: the simple, but low-precision *read-only summary* scheme and the more useful *write-order summary* scheme.

3.1. The Read-only Summary Scheme

It is possible to define a representative dependence such that it carries all of the dependence information needed for the potential parallelization of the two grains. When no dependence exists between any pair of memory references in the two grains, neither should a representative dependence exist. When two or more accesses to a memory location exist in a grain, we must simply find a way to assign an aggregate access type to the group, so that we can determine the representative dependence in a way which retains the information we need for making parallelization decisions.

Consider two grains which execute in the serial form of a program, one before the other. One consistent way to summarize dependence (for a single memory location) between the two grains is to determine whether the accesses are read-only in each grain, and define dependence as in Table II. We call this the *read-only summary* scheme.

Table II. One possible representative dependence definition - the read-only summary scheme.

Dependence Type:	Input	Flow	Anti	Output
earlier Read-Only?	Yes	No	Yes	No
later Read-Only?	Yes	Yes	No	No

Figure 1 illustrates dependence summarization with the read-only summary scheme. When an input dependence exists between two grains, it can be ignored. When a flow dependence exists between grains, in general the grains must be serialized.

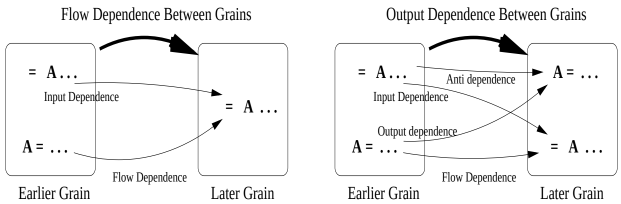


Figure 1. Dependence between grains depends on whether the two grains are read-only. The situation on the right shows a case where `A` can be privatized in the later grain, eliminating the output dependence.

When an anti dependence exists between grains, it means that only reads happen in one grain, followed by at least one write in the other. An output dependence means that at least one write occurs in both grains. In both anti and output dependence situations, if a write to the location is the first access in the later dependence grain, then it would be possible to run the grains in parallel by privatizing the variable in the later grain.

However, in the read-only summary scheme we don't keep enough information in the summary to determine whether a write happened first in the later grain or not, so we would miss the opportunity to parallelize by privatization. This shows that while read-only summarizing can detect dependences, it does not classify the dependences clearly enough to allow us to eliminate the dependence by compiler transformations. We will derive a better scheme in the next section.

3.2. The Write-order Summary Scheme

When the dependence grains are loop iterations, there exists a special case of the more general problem in that a single section of code represents all dependence grains. This fact can be used to simplify the dependence analysis task.

If we were still using read-only summarization and doing loop-based dependence testing, there would no longer be four cases, just two. The iteration is either read-only or it is not. However, to be able to differentiate between the anti and output dependences which can be removed by privatization and those which cannot, the case where it is not read-only can also be divided into two cases: one where a write is the first access to the location (WriteFirst) and one where a read is the first access (ReadWrite). This gives three overall classes, shown in Table III.

Table III. Loop-based representative dependence table.

Access	ReadOnly	ReadWrite	WriteFirst
Dependence Type	Input	Flow	Anti/Output

When an iteration only reads the location, dependence can be characterized as an Input dependence (and ignored). When the iteration reads the location, then writes it, the variable cannot be privatized. This results in a dependence which cannot be ignored and cannot be removed by privatization, so it will be called a Flow dependence. When an iteration writes the location first, any value in the location when the iteration starts is immediately over-written, so the variable can be privatized. Since these dependences can be removed by privatization, they will be called memory-related dependences.

Since privatization can be done in the memory-related dependence case, and that case is signaled when a write is the first access, all we need to do to identify these cases is to keep track of the case when a location is written first. The input and flow dependence cases are characterized by a read happening first, and differentiated by whether a write occurs later or not. We call this the *write-order summary* scheme.

It makes sense to use the write-order summarization scheme for the general case as well as for loops. Any locations which are read-only in both grains would correspond to an input dependence, those which are write-first in the later grain would correspond to a memory related dependence (since it is written first in the later grain, the later grain need not wait for any value from the earlier grain), and all others would correspond to a flow dependence. This is illustrated in Table IV.

Table IV. A more effective way to classify dependences between two arbitrary dependence grains, using the classes ReadOnly, WriteFirst and ReadWrite - **the write-order summary scheme**.

	later ReadOnly	later WriteFirst	later ReadWrite
earlier ReadOnly	Input	Anti/Output	Flow
earlier WriteFirst	Flow	Anti/Output	Flow
earlier ReadWrite	Flow	Anti/Output	Flow

So, the read-only summary scheme could serve as a dependence test, while the write-order summary scheme can detect dependence as well as provide the additional information necessary to remove dependences by a privatization transformation. As we will see in Section 5.4, a few simple tests added to the write-order summary scheme can collect enough information to allow some dependences to be removed by induction and reduction transformations.

3.3. Establishing an Order Among Accesses

Knowing the order of accesses is crucial to the write-order summarization scheme, so we must establish an ordering of the accesses within the program. If a program contained only straight-line code, establishing an ordering between accesses would be trivial. One could simply sweep through the program in "execution-order", keeping track of when the accesses happen. But branching statements and unknown variables make it more difficult to show that one particular access happens before another.

For example, take the loop in Figure 2. The write to `A(I)` happens before the read of `A(I)` only if both `P` and `Q` are true. But if `Q` is true and `P` is false, then the read happens without the write having happened first. If `P` and `Q` have values which are unrelated, then the compiler has no way of knowing the ordering of the accesses to `A` in this loop. On the other hand, if the compiler can show that `P` and `Q` are related and that in fact `Q` being true *implies* that `P` must have also been true, the compiler can know that the write happened first. So, for code involving conditional branches, the major tool the compiler has in determining the ordering of the accesses is *logical implication*.

To facilitate the use of logical implication to establish execution order, the representation of each memory reference must potentially

```
for I = 1 to N {
   if (P) {
      A(I) = ···
   }
   if (Q) {
      ··· = A(I)
   }
}
```

Figure 2. Only through logical implication can the compiler determine the ordering of accesses to array `A` in the `I`-loop.

have an *execution predicate* attached to it. In fact, the access in Figure 2 could be classified as ReadOnly with the condition $\{\neg P \wedge Q\}$, WriteFirst with condition $\{P\}$ and ReadWrite otherwise.

DEFINITION 3. *The* **execution predicate** *is a boolean-valued expression, attached to the representation of a memory reference, which specifies the condition under which the reference actually takes place. An execution predicate* P *will be denoted as* $\{P\}$.

3.4. Using Summary Sets to Store Memory Locations

We can classify a set of memory locations according to their access type by adding a symbolic representation of them to the appropriate *summary set.*

DEFINITION 4. *A* **summary set** *is a symbolic description of a set of memory locations.*

We have chosen to use Access Region Descriptors (ARDs), described in Section 4, to represent memory accesses within a summary set. To represent memory accesses for use in the write-order summary scheme, according to Table III, requires three summary sets for each dependence grain: ReadOnly (RO), ReadWrite (RW), and WriteFirst (WF).

3.5. Classification of Memory References

Each memory location referred to in the program must be entered into one of these summary sets, in a process called *classification.* A program is assumed to be a series of nested elementary contexts*: *procedures,*

* If the programming language does not force this through its structure, then the program will have to be transformed into that form through a *normalization* process.

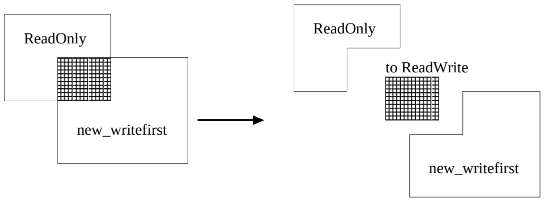


Figure 3. The intersection of earlier ReadOnly accesses with later WriteFirst accesses - the result is a ReadWrite set.

simple statements, `if` *statements*, *loops*, and `call` *statements*. Thus, at every point in the program, there will be an enclosing context and an enclosed context.

The contexts are traversed in "execution order". The summary sets of the enclosing context are built by (recursively) calculating the summary sets for each enclosed context and distributing them into the summary sets of the enclosing context. We can determine memory locations in common between summary sets by an intersection operation, as illustrated in Figure 3.5.

Classification takes as input the current state of the three summary sets for the enclosing context (RO, WF, and RW) and the three new summary sets for the last enclosed context which was processed (RO_n, WF_n, and RW_n) and produces updated summary sets for the enclosing context. The sets for the enclosed context are absorbed in a way which maintains proper classification for each memory location. For example, a memory location which was RO in the enclosing context (up to this point) and is WF or RW in the newly-calculated enclosed context becomes RW in the updated enclosing context. The steps of classification can be expressed in set notation, as shown in Figure 4.

3.5.1. *Program Context Classification*

Simple statements are classified in the obvious way, according to the order of their reads and writes of memory. All statements within an `if` context are classified in the ordinary way, except that the *if-condition* P is applied as an execution predicate to the statements in the `if` block and $\neg P$ is applied to the statements in the `else` block. Descriptors for the `if` and `else` blocks are then intersected and their execution predicates are *or*'ed together, to produce the result for the whole `if` context.

Classifying the memory accesses in a loop is a two-step process. First, the summary sets for a single iteration of the loop must be collected by a scan through the loop body in execution order. They contain the symbolic form of the accesses, possibly parameterized by the index of the loop. Next, the summary sets must be *expanded* by

$$
\begin{array}{lll@{\qquad}lll@{\qquad}lll}
t_{WF} & \leftarrow & WF_n - WF & & & & t_{RW} & \leftarrow & RW_n - WF \\
t_{WF} & \leftarrow & t_{WF} - RW & t_{RO} & \leftarrow & RO_n - RO & t_{RW} & \leftarrow & t_{RW} - RW \\
RW & \leftarrow & RW \cup (t_{WF} \cap RO) & t_{RO} & \leftarrow & t_{RO} - WF & RW & \leftarrow & RW \cup (t_{RW} \cap RO) \\
RO & \leftarrow & RO - (t_{WF} \cap RO) & t_{RO} & \leftarrow & t_{RO} - RW & RO & \leftarrow & RO - (t_{RW} \cap RO) \\
t_{WF} & \leftarrow & t_{WF} - (t_{WF} \cap RO) & RO & \leftarrow & RO \cup t_{RO} & t_{RW} & \leftarrow & t_{RW} - (t_{RW} \cap RO) \\
WF & \leftarrow & WF \cup t_{WF} & & & & RW & \leftarrow & RW \cup t_{RW}
\end{array}
$$

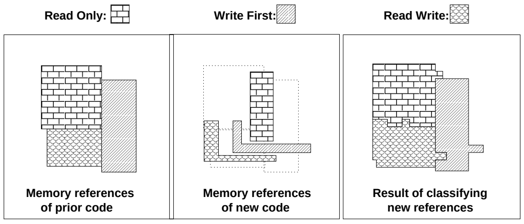


Figure 4. Classification of new summary sets RO_n, WF_n, and RW_n into the existing summary sets RO, WF, and RW, and a pictorial example of adding new summary sets to existing summary sets.

the loop index so that at the end of the process, the sets represent the locations accessed during the entire execution of the loop.

The expansion process can be illustrated by the following loop:

```
do I = 1, 100
   A(I) =  . . .
end do
```

For a single iteration of the surrounding loop, the location `A(I)` is classified WriteFirst. When `A(I)` is expanded by the loop index `I`, the representation `A(1:100)` results. Summary sets for `while` loops can be expanded similarly, but we must use a basic induction variable as a loop index and represent the number of iterations as "unknown". This expansion process makes it possible to avoid traversing the back-edges of loops for classification.

Classification for a `call` statement involves first the calculation of the access representation for the text of the `call` statement itself, calculation of the summary sets for the procedure being called, matching formal with actual parameters, and finally translating the summary sets involved from the called context to the calling context (described further for ARDs in Section 4.3).

4. The Access Region Descriptor

To manipulate the array access summaries for dependence analysis, we needed a notation which could precisely represent a collection of memory accesses. As briefly mentioned in Section 2, our previous study [11] gave us a clear picture of the strengths and weaknesses of existing notations. It also gave us the requirements the notation should meet to support efficient array access summarization.

- Complex array subscripts should be represented accurately. In particular, non-affine expressions should be handled because time-critical loops in real programs often contain array references with non-affine subscripts.
- The notation should have simplification operations defined for it, so that complex accesses could be changed to a simpler form.
- To facilitate fast and accurate translation of access summaries across procedure boundaries, non-trivial *array reshaping* at a procedure boundary should be handled efficiently and accurately.
- The notation should provide a *uniform* means for representing accesses to memory, regardless of the declared shape of the data structures in the source code.

To meet these requirements, we introduced a new notation, called the *Access Region Descriptor*, which is detailed in the previous literature [11]. The ARD is derived from the *linear memory access descriptor* introduced in [13] and [14]. To avoid repetition, this section will only briefly discuss a few basics of the ARD necessary to describe our dependence analysis technique in Section 5.

4.1. Representing the Array Accesses in a Loop Nest

If an array is declared as an m-dimensional array:

$$\mathtt{A}(L_1 : U_1, L_2 : U_2, \cdots, L_m : U_m),$$

then referenced in the program with an array name followed by a list of *subscripting expressions* in a nested loop, as in Figure 5,

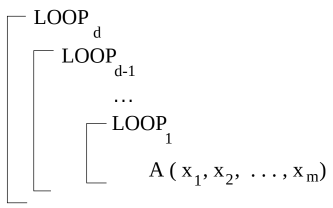


Figure 5. An m-dimensional array reference in a d-loop nest.

then implicit in this notation is an *array subscripting function* F_m which translates the array reference into a set of offsets from a base address in memory:

$$F_m(x_1, x_2, \cdots, x_m) = \\ (x_1(\mathbf{i}) - L_1)\lambda_1 + (x_2(\mathbf{i}) - L_2)\lambda_2 + \cdots + (x_m(\mathbf{i}) - L_m)\lambda_m$$

where $\mathbf{i}$ refers to the set of loop indices for the surrounding nested loops, $\mathbf{i} = (i_1, i_2, \cdots, i_m)$ and λ_k refers to a set of constants determined by the rules of the programming language.

As the nested loop executes, each loop index i_k moves through its set of values, and the subscripting function F_m generates a sequence of offsets from the base address, which we call the **subscripting offset sequence**:

$$F_m(x_1, x_2, \cdots, x_m)|_{i_d, i_{d-1}, \cdots, i_1} = S_1, S_2, \cdots, S_n$$

The isolated effect of a single loop index on F_m is a sequence of offsets which can **always** be **precisely represented** in terms of

- its starting value,
- the expression representing the difference between two successive values, and
- the total number of values in the sequence.

For example, consider even a non-affine subscript expression:

```
real A(0:*)
do I = 1, N
     A(2**I)
end do
```

The subscripting offset sequence is:

$$F_1(2^I)\big|_I = 2, 4, 8, 16, \cdots$$

The difference between two successive values can be easily expressed. To be clear, the difference is defined to be *the expression to be added to the Ith member of the sequence to produce the $I+1$th member of the sequence*:

$$S_{I+1} - S_I = 2^I.$$

There are N members of the subscripting offset sequence, they start at 2, and the difference between successive members is 2^I.

4.2. Components of an ARD

We refer to the subscripting offset sequence generated by an array reference, due to a single loop index, as a *dimension* of the access. We call this a *dimension* of an ARD.

DEFINITION 5. *A* **dimension** *of an ARD is a representation of the subscripting offset sequence for a set of memory references. It contains*

- *a starting value, called the* **base offset**
- *a difference expression, called the* **stride***, and*
- *the number of values in the sequence, represented as a* **dimension index***, taking on all integer values between 0 and a* **dimension-index bound** *value.*

Notice that the access produced by an array reference in a nested loop has as many dimensions as there are loops in the nest. Also, the dimension index of each dimension may be thought of as a *normalized* form of the actual loop index occuring in the program when the ARD is originally constructed by the compiler from the program text.

In addition to the three expressions described above for an ARD dimension, a *span* expression is maintained, where possible, for each dimension. The span is defined as the difference between the offsets of the last and first elements in the dimension. The span is useful for doing certain operations and simplifications on the ARD (for instance detecting *internal overlap*, as described in Section 4.4), however it is only accurate when the subscript expressions for the array access are *monotonic*.

A single base offset is stored for the whole access. An example of an array access, its access pattern in memory, and its LMAD may be seen in Figure 6.

The ARD for the array access in Figure 5 is written as

$$\mathcal{A}^{\delta_{1[i'_1 \leq u_1]}, \delta_{2[i'_2 \leq u_2]}, \cdots, \delta_{d[i'_d \leq u_d]}}_{\sigma_1, \quad \sigma_2, \quad \cdots, \sigma_d} + \tau,$$

with a series of d comma-separated strides $(\delta_1 \cdots \delta_d)$ as superscripts to the variable name and a series of d comma-separated spans $(\sigma_1 \cdots \sigma_d)$ as subscripts to the variable name, with a base offset (τ) written to the right of the descriptor. The dimension index is only included in the written form of the LMAD if it is needed for clarity. In that case,

$$[\textit{index} \leq \textit{dimension-bound}]$$

is written as a subscript to the appropriate stride.

4.3. Interprocedural Translation of an ARD

A useful property of the ARD is the ease with which it may be translated across procedure boundaries. Translation of array access information across procedure boundaries can be difficult if the declaration of a formal array parameter differs from the declaration of its corresponding actual parameter. Array access representations which depend

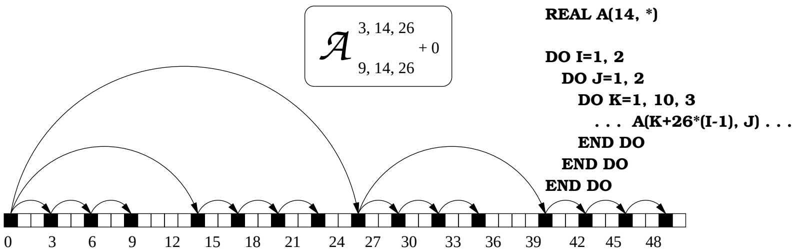


Figure 6. A memory access diagram for the array `A` in a nested loop and the Access Region Descriptor which represents it.

on the declared dimensionality of an array (as most do) are faced with converting the representation from one dimensionality to another when array reshaping occurs. This is not always possible without introducing complicated mapping functions. This is the *array reshaping problem.* Table V indicates that significant array reshaping occurs in many scientific applications, as published in [11].

Table V. The figures in each entry indicate percentages of `calls` doing reshaping in various benchmark programs from Perfect, SPEC and NASA. They were computed from a static examination of the programs mentioned.

`trfd`	`arc2d`	`tfft2`	`flo52`	`turb3d`	`ocean`	`mdg`	`bdna`	`tomcatv`	`swim`
66.7	50.8	46.2	40.8	16.1	6.6	3.2	0.0	0.0	0.0

We refer to a memory access representation that is independent of the declared dimensionality of an array as a "universal" representation, because it becomes procedure independent and even language independent. A universal representation eliminates the array reshaping problem because it need not be translated to a new form (a potentially different dimensionality) when moving to a different execution context. The ARD is an example of a universal representation.

When a subroutine is called by reference, the base address of a formal array parameter is set to be whatever address is passed in the actual argument list. Any memory accesses which occur in the subroutine would be represented in the calling routine in their ARD form, relative to that base address. Whenever it is desired to translate the ARD for a formal argument into the caller's context, we simply translate the formal argument's variable name into the actual argument's name, and add the base offset of the actual parameter to that of ARD for the formal parameter. For example, if the actual argument in a Fortran code is an indexed array, such as

```
call X( A(I*2+1) ) ! Real A(1:100)
```

then the offset from the beginning of the array `A` for the actual argument is $2I$. Suppose that the matching formal parameter in the subroutine `X` is `Z` and the LMAD for the access to `Z` in `X` is

$$\mathcal{Z}\,{}^{10,200}_{100,400} + 10.$$

When the access to `Z`, in subroutine `X`, is translated into the calling routine, then the LMAD would be represented in terms of variable `A` as follows:

$$\mathcal{A}\,{}^{10,200}_{100,400} + 10 + 2I,$$

which results from simply adding the offset after the renaming of `Z` to `A`. Notice that `A` now has a two-dimensional access even though it is declared to be one-dimensional.

4.4. Properties of ARDs

This subsection briefly describes several basic properties of ARDs that are useful for our dependence analysis based on access summary sets.

DEFINITION 6. *Given an ARD with a set of stride/span pairs, we call the sum of the spans of the first k dimensions the* **k-dimensional width** *of the ARD, defined by*

$$\mathrm{width}_k(\mathcal{D}) = \sum_{j=1}^{k} \sigma_{i_j}.$$

4.4.0.1. *Internal Overlap of an ARD* The process of expanding an ARD by a loop can cause *overlap* in the descriptor. For example, in the following Fortran `do`-loop

```
do I=1,10
   do J=1,5
      ...  A(I*4+J) ...
   end do
end do
```

the ARD for `A` in the inner loop is $\mathcal{A}^1_4 + I * 4 + 1$. When the ARD is expanded for the outer loop, it becomes $\mathcal{A}^{1,4}_{4,36} + 4$ which exhibits an overlap due to the outer loop. This is because the access due to the outer loop does not stride far enough to get beyond the array elements already touched by the inner loop. This property may be determined by noticing that the stride of the n-th dimension is not greater than the $n-1$ dimensional width of the ARD.

4.4.1. *Zero-span dimensions*

A dimension whose span is zero adds no data elements to an access pattern. This implies that whenever such a dimension appears in an ARD (possibly through manipulation of the ARD), it may be safely eliminated without changing the access pattern represented. Likewise, it implies that at any time a new dimension may be introduced with any desired stride and a zero-span, without changing the access pattern.

Simplification operations exist for eliminating dimensions within an ARD, for eliminating ARDs which are found to be covered by other ARDs, and for creating a single ARD which represents the accesses of several ARDs. Since these operations are not needed for the exposition of this paper, they will not be described here, but the reader is referred to [11, 14, 13].

5. The Access Region Test

In this section, we first describe the general dependence analysis method, based on intersecting ARDs. The general method can detect data dependence between any two arbitrary sections of code. Then, we show a simplification of the general method, the *Access Region Test*, which works for loop parallelization, and show a multi-dimensional, recursive intersection algorithm for ARDs.

5.1. General Dependence Testing with Summary Sets

Given the symbolic summary sets RO_1, WF_1, and RW_1 (as discussed in Section 3.4), representing the memory accesses for an earlier (in the sequential execution of the program) dependence grain, and the sets RO_2, WF_2, and RW_2 for a later grain, it can be discovered whether any locations are accessed in both grains by finding the intersection of the earlier and later sets, and by consulting Table IV. Any non-empty intersection represents a dependence between grains. However, some of those dependences may be removed by compiler transformations.

The intersections that must be done for each variable are:

$$
\begin{array}{c}
\mathrm{RO}_1 \cap \mathrm{WF}_2 \\
\mathrm{WF}_1 \cap \mathrm{WF}_2 \\
\mathrm{RW}_1 \cap \mathrm{WF}_2 \\
\mathrm{RO}_1 \cap \mathrm{RW}_2 \\
\mathrm{WF}_1 \cap \mathrm{RO}_2 \\
\mathrm{WF}_1 \cap \mathrm{RW}_2 \\
\mathrm{RW}_1 \cap \mathrm{RO}_2 \\
\mathrm{RW}_1 \cap \mathrm{RW}_2
\end{array}
$$

If all of these intersections are empty for all variables, then no cross-iteration dependences exist between the two dependence grains. If any of the following are non-empty: $\mathrm{RO}_1 \cap \mathrm{WF}_2$, $\mathrm{WF}_1 \cap \mathrm{WF}_2$, $\mathrm{RW}_1 \cap \mathrm{WF}_2$, or $\mathrm{RO}_1 \cap \mathrm{RW}_2$, then they represent dependences which can be removed by privatizing the intersecting regions.

If $\mathrm{RW}_1 \cap \mathrm{RW}_2$ is non-empty, and all references involved are in either induction form or reduction form, then the dependence may be removed by induction or reduction transformations. This will be discussed in more detail in Section 5.4.

If any of the other intersections: $\mathrm{WF}_1 \cap \mathrm{RO}_2$, $\mathrm{WF}_1 \cap \mathrm{RW}_2$, or $\mathrm{RW}_1 \cap \mathrm{RO}_2$ are non-empty, then they represent non-removable dependences.

5.2. Loop Dependence Testing with the ART

The *Access Region Test* (ART) is used within the general framework of Memory Classification Analysis, doing write-order summarization. This means that the entire program is traversed in execution order, using abstract interpretation, with summary sets being computed for the nested contexts of the program and stored in ARDs. ARDs are used as the semantic elements for the abstract interpretion. The interpretation rules are exactly those rules described for the various program contexts in Section 3.5.1. Whenever loops are encountered, the ART is applied to the ARDs to determine whether the loops are parallel, or parallelizable by removing dependences through compiler transformations.

As stated in Section 3.2, dependence testing between loop iterations is a special case of general dependence testing, described in the last section. Loop-based dependence testing considers a loop iteration to be a dependence grain, meaning all dependence grains have the same summary sets.

Once we expand the summary sets by the loop index (Section 3.5.1), cross-iteration dependence can be noticed in three ways: within one LMAD, between two LMADs of one summary set, or between two of the summary sets.

5.2.1. *Overlap within a Single ARD*

Internal overlap due to expansion by a loop index is described in Section 4.4. When overlap occurs, it indicates a cross-iteration dependence. This condition can be easily checked during expansion and flagged in the ARD, so no other operation is required to detect this.

5.2.2. *Intersection of ARDs within a Summary Set*

Even though two LMADs in the same summary set do not intersect initially, expansion by a loop index could cause them to intersect. Such an intersection would represent a cross-iteration dependence. Such an intersection within RO would be an input dependence, so this summary set need not be checked.

Internal intersections for both WF and RW must be done, however. In Figure 7, for example, when the two writes to array `A` are first assigned to a summary set, they do not overlap. The two write-first ARDs are initially $\mathcal{A}_0^0 + I - 1$ and $\mathcal{A}_0^0 + 4 - I$. Since the base offsets are different, the intersection is assumed to not overlap (the conservative assumption). This causes them to be separately assigned to the WF set. After expansion for I, (and creation of the dimension index I'), the normalized ARDs both become $\mathcal{A}_3^1 + 0$, which *do* intersect, indicating a dependence. This intersection would be found by attempting to intersect the ARDs within WF.

```
do I = 1, 4
   A(I) = ···
   A(5-I) = ···
end do
```

Figure 7. Example illustrating the need for internal intersection of the summary sets.

5.2.3. *Intersection of Two Summary Sets*

There are only three summary sets to consider in loop dependence testing, instead of six (because there is only one dependence grain), so there are only three intersections to try, instead of the eight required in Section 5.1. After expansion by the loop index, the following intersections must be done:

$$\begin{gathered} \mathrm{RO} \cap \mathrm{WF} \\ \mathrm{RO} \cap \mathrm{RW} \\ \mathrm{WF} \cap \mathrm{RW} \end{gathered}$$

An intersection between any pair of the sets RO, WF, and RW involves at least one read and one write operation, implying a dependence.

5.3. The Loop-based Access Region Test Algorithm

For each loop L in the program, and its summary sets RO, WF, and RW, the ART does the following:

- Expand the ARDs in all summary sets by the loop index of L.
- Check for internal overlap, due to the loop index of L, of any ARD in WF or RW. Any found within WF can be removed by privatization. Any found in RW is removed if all references involved are in either induction or reduction form. Once overlap for an ARD is noted, its overlap flag is reset.
- Check for non-empty intersection between any pair of ARDs in WF (removed by privatization) or RW (possibly removed by induction or reduction).
- For all possible pairs of summary sets, from the group RO, WF, and RW, check for any non-empty intersection between two ARDs, each pair containing ARDs from different sets. Any intersection found here is noted as a dependence and moved to RW.

If no non-removable dependences are found, the loop may be declared parallel. Wherever uncertainty occurs in this process, *demand-driven deeper analysis* can be triggered in an attempt to remove the uncertainty, or run-time tests can be generated.

5.4. Detecting Reduction and Induction Patterns

As stated in Section 2.3, idiom recognition is very important for parallelizing programs. Inductions and reductions both involve an assignment with a *linear recurrence* structure:

```
<variable> = <variable> <op> <expression>
```

The forms differ slightly, as shown in the following (`I` represents an integer variable and `R` represents a floating point variable):

$$\begin{array}{ll} \mathtt{I} = \mathtt{I} < \text{op} > \ < \text{integer constant} > & : \text{Induction} \\ \mathtt{R} = \mathtt{R} < \text{op} > \ < \text{floating point expression} > & : \text{Reduction} \end{array}$$

Each of these patterns originally presents itself as a dependence, but compiler transformations [15, 4] can remove the dependence.

Three levels of tests, done within the write-order summary scheme structure, can be used to positively identify reductions and inductions.

5.4.0.1. *Level 1* The first test is for the *linear recurrence* structure of the assignment statement. When the pattern is found, the ARD for the statement is marked as passing the Level 1 test, plus the operator (<op>) and the type of the <expression> (integer constant or floating point expression) are stored. The ARD is marked with an *idiom type* of *possible induction* if it is an integer variable and the expression is an integer constant. Otherwise it is marked as a *possible reduction*.

5.4.0.2. *Level 2* During intersection of the ARDs of a particular variable within the ReadWrite summary set (as part of the ART, described in Section 5.2.2), if one is marked as having passed Level 1, then if any other ARD in RW for the variable did not pass Level 1, with the same idiom type and operator, it fails Level 2. If there are any ARDs for the variable in either ReadOnly or WriteFirst and the ARD is a possible reduction, then it fails Level 2. If the ARD is a possible induction and any ARDs exist for the variable in WriteFirst, then it fails Level 2. Otherwise the ARD passes Level 2.

5.4.0.3. *Level 3* To pass Level 3, an ARD marked as passing Level 2 must be marked as having internal overlap due to expansion by the loop index of an outer loop. This means that there is a dependence due to the access, carried by the outer loop.

An ARD marked as having passed Level 3 can be considered an idiom of the stored type, and appropriate code can be generated for it. This three-level process will find inductions and reductions interprocedurally, because of the interprocedural nature of the ART.

5.5. Generality of the ART

The Access Region Test is a general, conservative dependence test. By design, it can discern three types of dependence: input, flow, and memory-related. It cannot distinguish between anti and output dependence, but that is because for our purposes it was considered unimportant - both types of dependence can be removed by privatization transformations. For other purposes, the general MCA mechanism can be used to formulate a mechanism, with the appropriate summary sets, to produce the required information, much as data flow analysis can be formulated to solve various data flow problems. In Sections 3.1 and 3.2, we showed two different formulations of MCA for doing dependence analysis. The read-only summary scheme is simply a dependence test, while the write-order summary scheme provides enough information to test for dependence, and also remove some dependences through compiler transformations.

5.6. Loop-carried Dependence Handled by the ART

Any dependence within an inner loop is essentially ignored with respect to an outer loop because of the fact that after expansion by a loop index, any intersecting portions of two ARDs are represented as a single ARD and moved to the RW summary set. If there are intersecting portions, they are counted as cross-iteration dependences for that loop, but because they are reduced to a single ARD, no longer will be found to intersect for outer loops. Intersections due to outer loops will be solely due to expansions for outer loop indices. This process is illustrated in Figure 8.

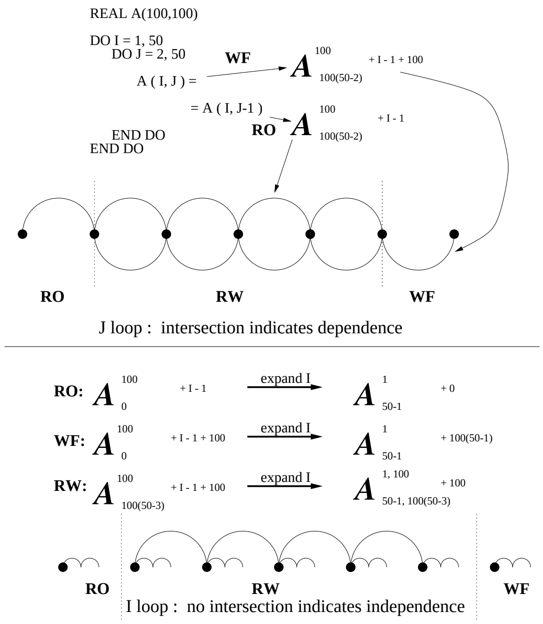


Figure 8. How the ART handles loop-carried dependence.

5.7. A Multi-Dimensional Recursive Intersection Algorithm

Intersecting two arbitrary ARDs is very complex and probably intractable. But if the two ARDs to be compared have the same strides (we call these *stride-equivalent*), or the strides of one are a subset of the strides of the other (we call these *semi-stride-equivalent*), which has been quite often true in our experiments, then they are similar enough to make the intersection algorithm tractable. We present Algorithm Intersect in Figure 9.

Input: Two ARDs, with properly nested, sorted dimensions :
$\mathrm{ARD}_{\mathrm{left}} = \mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma_1,\sigma_2,\cdots,\sigma_d} + \tau$, $\mathrm{ARD}_{\mathrm{right}} = \mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma'_1,\sigma'_2,\cdots,\sigma'_d} + \tau'$ (such that $\tau \leq \tau'$),
k: the number of the dimension to work on $(0 \leq k \leq d)$
Output:List of ARDs

intersect($\mathrm{ARD}_{\mathrm{left}}$, $\mathrm{ARD}_{\mathrm{right}}$, k) **returns** ARD_*List*
  $D \leftarrow \tau' - \tau$
  `if` ($k == 0$) `then`   // scalar intersection
    `if` ($D == 0$) `then`
      $\mathrm{ARD}_{\mathrm{rlist1}} \leftarrow$ **ARD**_*scalar*(τ)
      add_to_list(ARD_*List*, $\mathrm{ARD}_{\mathrm{rlist1}}$)
    `endif`
    `return` ARD_*List*
  `endif`
  `if` ($D \leq \sigma_k$) `then`
    $R \leftarrow \mathtt{mod}(D, \delta_k)$
    $m \leftarrow \lfloor \frac{D}{\delta_k} \rfloor$
    // periodic intersection on the left
    $\mathrm{ARD}_{\mathrm{rlist1}} \leftarrow$ **intersect**(**remove_dim**($\mathrm{ARD}_{\mathrm{left}}, k, \tau + m\delta_k$),
        remove_dim($\mathrm{ARD}_{\mathrm{right}}, k, \tau'$), $k - 1$)
    $\mathrm{ARD}_{\mathrm{rlist1}} \leftarrow$ **add_dim**($\mathrm{ARD}_{\mathrm{rlist1}}$, **dim**($\delta_k$, **min**($\sigma_k - m\delta_k, \sigma'_k$)), τ')
    add_to_list(ARD_*List*, $\mathrm{ARD}_{\mathrm{rlist1}}$)
    `if` (($k > 1$) **and** ($R + \sigma'_{k-1} \geq \delta_k$)) `then`   // periodic intersection on the right
      $\mathrm{ARD}_{\mathrm{rlist2}} \leftarrow$ **intersect**(**remove_dim**($\mathrm{ARD}_{\mathrm{right}}, k, \tau'$),
          remove_dim($\mathrm{ARD}_{\mathrm{left}}, k, \tau + (m+1)\delta_k$), $k - 1$)
      $\mathrm{ARD}_{\mathrm{rlist2}} \leftarrow$ **add_dim**($\mathrm{ARD}_{\mathrm{rlist2}}$, **dim**($\delta_k$, **min**($\sigma_k - m\delta_k, \sigma'_k$)), $\tau + (m+1)\delta_k$)
      add_to_list(ARD_List, $\mathrm{ARD}_{\mathrm{rlist2}}$)
    `endif`
  `else`
    // intersection at the end
    $\mathrm{ARD}_{\mathrm{rlist1}} \leftarrow$ **intersect**(**remove_dim**($\mathrm{ARD}_{\mathrm{left}}, k, \tau + \sigma_k$),
        remove_dim($\mathrm{ARD}_{\mathrm{right}}, k, \tau'$), $k - 1$)
    add_to_list(ARD_*List*, $\mathrm{ARD}_{\mathrm{rlist1}}$)
  `endif`
  `return` ARD_*List*
`end` **intersect**

remove_dim($\mathrm{ARD}_{in}, k, \tau_{\mathrm{new}}$) {
  // Construct and return a new ARD equivalent to ARD_{in},
  // except without access dimension k and with τ_{new} as the new base offset.
}

dim(δ, σ) {
  // Construct and return a new access dimension with stride δ and span σ.
}

add_dim($\mathrm{ARD}_{in}, \mathrm{dim}_{\mathrm{new}}, \tau_{\mathrm{new}}$) {
  // Construct and return a new ARD equivalent to ARD_{in},
  // except with new dimension $\mathrm{dim}_{\mathrm{new}}$ and with τ_{new} as the new base offset.
}

add_to_list(ARD_list, ARD) {
  // Add ARD to the list of ARDs ARD_list.
}

Figure 9. The algorithm for finding the intersection of multi-dimensional ARDs. For clarity, the removal of the intersection from the two input ARDs, the use of the conservative direction flag, and the use of the execution predicates are not shown, although all these things can be added to the algorithm in a straightforward way.

The algorithm accepts two stride-equivalent ARDs. If the two ARDs are semi-stride-equivalent, then zero-span dimension(s) can be safely inserted into the ARD with fewer dimensions (as discussed in Section 4.4.1) to make them stride-equivalent. The algorithm is also passed a *conservative direction* flag. The flag has two possible values: **under-estimation** and **over-estimation**, which tells the algorithm what to do when the result is imprecise. For over-estimation, the result is enlarged to its maximum value and likewise, under-estimation causes the result to be reduced to its minimum value. If two ARDs are to be intersected and they are not stride-equivalent, then the result is formed, based on the conservative direction.

The algorithm takes as input two ARDs which have all dimensions precisely sorted, $\mathrm{ARD}_{\mathrm{left}} = \mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma_1,\sigma_2,\cdots,\sigma_d} + \tau$ and $\mathrm{ARD}_{\mathrm{right}} = \mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma'_1,\sigma'_2,\cdots,\sigma'_d} + \tau'$, and the number of the dimension, d, to work on. $\mathrm{ARD}_{\mathrm{left}}$ has a base offset which is less than that of $\mathrm{ARD}_{\mathrm{right}}$.

The algorithm compares the overall extent of dimension d for each ARD, as shown in Figure 10(A). If the extents do not overlap in any way, it can safely report that the intersection is empty. If they do overlap, then the algorithm calls itself recursively, specifying the next inner dimension, as shown in Figure 10(B).

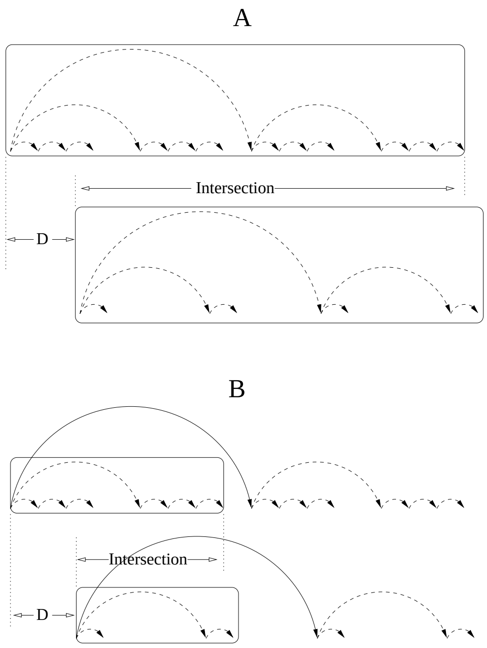


Figure 10. The multi-dimensional recursive intersection algorithm, considering the whole extent of the two access patterns (A), then recursing inside to consider the next inner dimension (B).

This process continues until it can either be determined that no overlap occurs, or until the inner-most dimension is reached, as shown in Figure 11, where it can make the final determination as to whether there is an intersection between the two, considering only one-dimensional accesses. The resulting ARD for the intersection is returned, and as each recursion returns, a dimension is added to the resulting ARD.

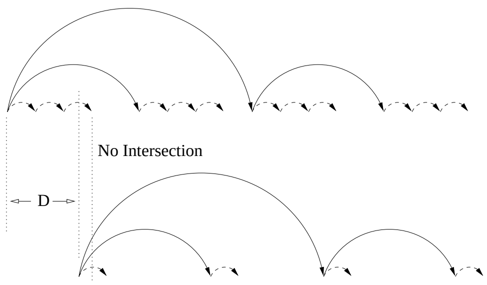


Figure 11. The multi-dimensional recursive intersection algorithm, considering the inner-most dimension, finding no intersection.

For simplicity, in this description, it is assumed that the two ARDs have dimensions which are fully sorted, so that dimension i of one ARD corresponds to dimension i of the other, and that $\delta_d > \delta_{d-1} > \cdots > \delta_1$.

6. Experiments

The Access Region Test has an advantage over other tests discussed in Section 1, in three ways:

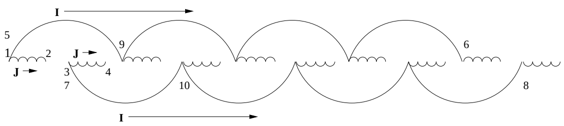
- Reducing dependence analysis to an intersection operation does not restrict the ART from handling certain types of subscripting expressions, such as coupled subscripts, which are a problem for the Range Test, and non-affine expressions, which are a problem for most other tests.
- Use of the ARD provides precise access summaries for all array subscripting expressions.
- The test is implicitly interprocedural since ARDs may be translated precisely across procedure boundaries.

To separate the value of the ART from the value of the ARD, it is instructive to consider the question of whether other dependence tests might be as powerful as the ART if they represented memory accesses with the ARD notation. The answer to this question is "no".

Take as an example the Omega Test. The mechanism of the Omega Test is only defined for affine expressions. The user of the Omega Test must extract the linear coefficients of the loop-variant values (loop indices, etc), plus provide a set of constraints on the loop-variants. The ARDs partially fill the role of the constraints, but if non-affine expressions are used, there is no way to extract linear coefficients for the non-affine parts. A technique for replacing the non-affine parts of an expression with *uninterpreted function symbols* has been developed [17], but it is not general enough to work in all situations. So even using the ARD, the Omega Test could not handle non-affine subscript expressions because its mechanism is simply not well-defined for such expressions.

Likewise, if the Range Test were to use the ARD to represent value ranges for variables, that still would not change its basic mechanism, which makes it a single-subscript test, unable to handle coupled-subscripts. The mechanism of the Range Test forces it to consider the behavior of the subscript expression due to a single subscript at a time, whereas the ART compares access patterns instead of subscript expressions. A simple example in Figure 12 shows the advantage of comparing the patterns. It shows two loop nests which display identical access patterns, yet different subscripting expressions. The top accesses can be determined independent by the Range Test, but the bottom accesses cannot.

```
do I = 1, N
   do J=1, M
      A(10*I + J) . . .
       A(10*I + J + L) . . .
   end do
end do
```




```
do I = 1, N
   do J=1, M
      A(10*I + J) . . .
       A(10*J + I + L) . . .
   end do
end do
```

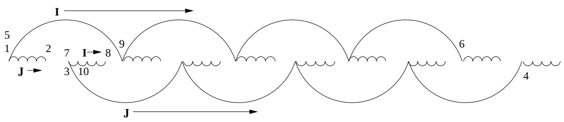


Figure 12. The Range Test can determine the accesses of the top loop to be independent, but not those of the bottom loop. The ART can find both independent, since it deals with access patterns instead of just subscript expressions.

Figure 13 shows another example, from the `tfft2` benchmark code, which neither the Omega Test nor the Range Test can find independent

```
REAL U(1), X(1), Y(1)
DO I=0,2**(M/2)-1
  CALL CFFTZWORK (IS, M-M/2,
    U(1+3*2**(1+M)/2),
    Y(1+I*2**(1+M-M/2)),
    X)
END DO
...
SUBROUTINE CFFTZWORK (IS, M, U, X, Y)
REAL U(1), X(1), Y(1)
DO L0=1, (M+1)/2
  CALL FFTZ2 (IS,2*L0-1, M, U, X, Y)
  CALL FFTZ2 (IS,2*L0 , M, U, Y, X)
END DO
...
```

```
SUBROUTINE FFTZ2 (IS, L, M, U, X, Y)
REAL U(*), X(*),
REAL Y(0:2**(L-1)-1,0:1,0:2**(M-L)-1,0:1)
DO I=0,2**(M-L)-1
  DO K=0,2**(L-1)-1
    ... = X(1+K+I*2**(L-1))
    ... = X(1+K+I*2**(L-1)+2**M)
    ... = X(1+K+I*2**(L-1)+2**(M-1))
    ... = X(1+K+I*2**(L-1)+2**(M-1)+2**M)
    Y(K,0,I,0) = ...
    Y(K,0,I,1) = ...
    Y(K,1,I,0) = ...
    Y(K,1,I,1) = ...
  END DO
END DO
```

Figure 13. A simplified excerpt from the benchmark program tfft2, which the ART can determine to be free of dependences.

due to the apparent complexity of the non-affine expressions involved, yet the ART *can* find them independent interprocedurally at the topmost loop, due to its reliance on the simple intersection operation, its ability to translate ARDs across procedure boundaries, and the powerful ARD simplification operations which expose the simple access patterns hidden inside complex subscript expressions.

As we continued to develop the ART, we needed to evaluate the ART on real programs. Therefore, we implemented a preliminary version of the ART in *Polaris* [4], a parallelizing compiler developed at Illinois, and experimented with ten benchmark codes. In these experiments, it was observed that the ART is highly effective for programs with complex subscripting expressions, such as `ocean`, `bdna`, and `tfft2`. Table VI shows a summary of the experimental results that were obtained at the time we prepared this paper. Careful analytical study confirms that the ART theoretically subsumes the Range Test. This implies that the ART can parallelize all the loops that the Range Test can, even though in this experiment, the ART failed to parallelize a few loops in `flo52` and `arc2d` due to several implementation dependent problems reported in [11].

The numbers of loops additionally parallelized by the ART are small, but some of these loops are time-critical loops which contain complex array subscripting expressions. Our previous experiments reported in [13] also showed that the ART applied by hand increased the parallel speedup for `tfft2` by factor of 7.4 on the Cray T3D 64 processors.

As can be expected, Table VI shows that neither the ART, the Omega Test, nor the Range Test make a difference in the performance for the codes with only simple array subscripting expressions, such as `tomcatv`, `arc2d` and `swim`.

Table VI. A comparison of the number of loops parallelized by a current version of the ART with other techniques. The first line shows the number of loops that the ART could parallelize and the Range Test could not. The second shows the number of loops that the Range Test could parallelize and the Omega Test could not. The third shows the number of loops that the Omega Test could parallelize and the Range Test could not. All other loops in the codes were parallelized identically by all tests. The data in the second and third lines are based on the previous work on Polaris.

tfft2	trfd	mdg	flo52	hydro2d	bdna	arc2d	tomcatv	swim	ocean
7	0	5	0	0	11	0	0	0	8
×	6	6	1	3	0	0	0	0	76
×	0	1	0	0	6	0	0	0	8

Previous techniques based on access summaries did not show experimental results with real programs in their papers [1, 18]. Thus, it is not possible for us to determine how effective their techniques would be for actual programs.

7. Conclusion and Future Work

This paper presents a technique for unifying interprocedural dependence analysis, privatization and idiom recognition in a single framework. This technique eliminates some of the limitations which encumber the loop-based, linear system-solving data dependence paradigm, and expands the notion of a dependence test to include a way of classifying the dependences found, so that a compiler can eliminate them using code transformations.

The framework is built on a general scheme for classifying memory locations (Memory Classification Analysis), based on the order and type of accesses to them. This framework can be reformulated and used for many purposes. The read-only and write-order summarization schemes were presented, but many other schemes are possible for a variety of purposes.

The multi-dimensional, recursive intersection algorithm for ARDs was introduced. It allows us to calculate a precise intersection between two stride-equivalent ARDs. This algorithm forms the core of the dependence analysis calculation. Heuristics can be added to this algorithm to handle cases in which the ARDs are not stride-equivalent. The more precise this intersection algorithm becomes, the more precise data dependence analysis becomes.

We believe that the flexibility and generality afforded by this reformulation of data dependence will make it very useful for many purposes within a compiler. In the future, we intend to use the MCA framework for other analyses, which will automatically extend them interprocedurally. In addition, we intend to formalize our methods by an analysis in terms of the abstract semantic elements and rules within the abstract interpretation framework.

References

1. V. Balasundaram and K. Kennedy. A Technique for Summarizing Data Access and its Use in Parallelism Enhancing Transformations. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1989.
2. U. Banerjee. *Dependence Analysis*. Kluwer Academic Publishers, Norwell, MA, 1997.
3. W. Blume. *Symbolic Analysis Techniques for Effective Automatic Parallelization*. PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, June 1995.
4. W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, W. Pottenger, L. Rauchwerger, and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.
5. Patrick Cousot. Semantic foundations of program analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10. Prentice-Hall, 1981.
6. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languges*, pages 238–252, 1977.
7. B. Creusillet and F. Irigoin. Interprocedural Array Region Analyses. In *Lecture Notes in Computer Science*. Springer Verlag, New York, New York, August 1995.
8. R. Eigenmann, J. Hoeflinger, and D. Padua. On the Automatic Parallelization of the Perfect Benchmarks. *IEEE Transactions on Parallel and Distributed Systems*, pages 5–23, January 1998.
9. G. Goff, K. Kennedy, and C. Tseng. Practical Dependence Testing. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 15–29, June 1991.
10. P. Havlak and K. Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.
11. J. Hoeflinger. *Interprocedural Parallelization Using Memory Classification Analysis*. PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, August, 1998.
12. Z. Li, P. Yew, and C. Zhu. An Efficient Data Dependence Analysis for Parallelizing Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, January 1990.

13. Y. Paek. *Automatic Parallelization for Distributed Memory Machines Based on Access Region Analysis.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, April 1997.
14. Y. Paek, J. Hoeflinger, and D. Padua. Simplification of Array Access Patterns for Compiler Optimizations. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.
15. W. Pottenger. Induction Variable Substitution and Reduction Recognition in the Polaris Parallelizing Compiler. Master's thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, December 1994.
16. W. Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8), August 1992.
17. W. Pugh and D. Wonnacott. Nonlinear Array Dependence Analysis. Technical Report 123, Univ of Maryland at College Park, November 1994.
18. P. Tang. Exact Side Effects for Interprocedural Dependence Analysis. In *1993 ACM International Conference on Supercomputing, Tokyo, Japan*, pages 137–146, July 1993.
19. P. Tu and D. Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 414–423, July 1995.
20. M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Co., New York, 1996.
21. M. Wolfe and C. Tseng. The Power Test for Data Dependence. *IEEE Transactions on Parallel and Distributed Systems*, September 1992.