INTERPROCEDURAL PARALLELIZATION USING MEMORY CLASSIFICATION ANALYSIS

BY

JAY PHILIP HOEFLINGER

B.S., University of Illinois, 1974
M.S., University of Illinois, 1977

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2000

Urbana, Illinois

ABSTRACT

This thesis outlines a way of addressing the goal of precise interprocedural analysis, based on a combination of techniques: for representing memory accesses within interprocedural sections of code, for summarizing dependence information in program contexts, and for testing that dependence. The thesis presents a new technique for summarizing the memory access activity in an arbitrary section of code, called Memory Classification Analysis (MCA), using a precise form for representing memory access patterns, called the Access Region Descriptor (ARD). A new, simple dependence test, the Access Region Test (ART), is also described which uses the summary sets of ARDs produced by MCA. This test is capable of parallelizing loops containing non-affine subscript expressions, such as those found in FFT codes. A unified parallelization framework is described, which combines privatization, reduction and induction analysis. Array references using subscripting arrays, such as are found in sparse codes are precisely representable using ARDs, and can sometimes be parallelized using the parallelization framework. Parallelization conditions are generated at critical points in the analysis when dependence cannot be disproved. These can be used to drive on-demand deeper program analysis. Whatever conditions remain unproven can then be generated as code to be used for runtime dependence testing. Its precise memory access representation makes the ARD useful within algorithms for generating data movement messages.

To

my loving wife Donna and my wonderful teenagers, Chris and Dan, for all the sacrifices you have had to make, just because "I wanna be a doctor". Thanks for the love, for putting up with me, and for dragging me away from my work into the real world at regular intervals.

And To

my parents, Aug and Rose Hoeflinger, whose love and support never wavered, and who taught me that anything is possible through hard work.

I cherish all of you.

ACKNOWLEDGMENTS

I would like to acknowledge the valuable advice and guidance given me by my advisor, Professor David Padua. His sage words helped me steer around many obstacles, and when we disagreed, he was usually right.

I would like to thank Yun Paek for our many conversations and his collaboration on several papers. He always has challenged me with his search for precise thinking and for finding the truths underlying our work.

My thanks and admiration also goes out to the many past and present members of the Polaris group at Illinois. I have the highest regard for the bright and talented people I have had the pleasure of working with during my years here, especially Rudi Eigenmann, Paul Petersen, Bill Blume, Lawrence Rauchwerger, Peng Tu, John Grout, Bill Pottenger, Jaejin Lee, Calin Cascaval, Yuan Lin, Peng Wu and Gheorghe Almasi.

Thanks to Sheila Clark for her excellent support of the work in our office, for being there to share stressed-out stories, and for being a friend.

And a final thank you to my committee members, Professors Sanjay Kale, Michael Heath, and Wen-mei Hwu for committing their time and energies to the consideration of my work.

TABLE OF CONTENTS

Chapter

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.1 Interprocedural Analysis

Precise interprocedural analysis is an increasingly important component of a modern compiler. Parallelizing compilers, especially, have a need for precise global information because of the hardware advances made in the last decade. Modern processors can take advantage of instruction-level parallelism, reducing the amount of parallelism available for use between processors. Also, as processor speed increases relative to memory speed, more instructions are needed between memory fetches to keep the processors busy. To accommodate these changes, people have attempted to parallelize loops further out in loop nests and to batch expensive data accesses together. The greater the compiler's ability to parallelize loops containing procedure calls, and to represent data accesses precisely, the greater its chances for success in these tasks.

But, interprocedural analysis adds complexity to the analysis problems faced by a compiler writer because each analysis technique must be either augmented to cross procedure boundaries, or must be able to accept a summary of the subroutine's activity. In either case, the compiler must model the parameter passing mechanism at call sites. This is especially difficult when a parameter passing regime is used (as it is in Fortran) such that an actual array can be passed to a formal array which is declared differently in the subroutine (this is referred to as the **array reshaping problem**). This is a widely-exploited feature of Fortran, but can be quite difficult for a compiler to handle. Complicated algorithms have been devised to accomplish this modeling accurately in some cases, although in other cases they are forced to approximate.

Three keys to successful interprocedural analysis are

- a consistent access summarization technique, usable by the various compiler components,
- a precise memory access representation, and
- a solution to the *array reshaping problem*.

Procedure inlining is a partial solution to the interprocedural analysis problem. The goal is to remove procedure boundaries entirely, by incorporating a procedure's code at each call

site, thus avoiding interprocedural analysis and its problems. Several problems result from this, however. First, when an array is reshaped across procedure boundaries it is sometimes very difficult to generate correct code for it at the call site. Second, because code is copied, the size of the source code increases, sometimes becoming very large. For these reasons, inlining has not been a practical option in general.

Recently, as computers became faster, main memory and disk larger, inlining was investigated again [19], but much the same conclusions were reached. Even for moderately-sized programs, the memory and compile time requirements can grow too large. Yet even if we could overcome the compile time and space limitations with future machines, the resulting fully-inlined program can be very difficult for the compiler to optimize because complex array subscripting expressions often result and the natural code locality produced by having the program divided into separate routines is lost.

1.2 Dependence Analysis

A critical part of precise interprocedural analysis is the dependence test. Many dependence tests have been developed over the years, and their power has increased sufficiently that modern parallelizers are able to parallelize a large percentage of the loops which they encounter in programs.

Dependence analysis was turned into an equation-solving activity by researchers such as David Kuck, Yochi Muraoka, Ross Towle, and Utpal Banerjee [6] at Illinois in the early 1970s. They proposed equating the subscript expressions of two array reference sites, solving the Diophantine equations which result, and then applying the constraints derived from the loop bounds to determine whether the solutions were feasible. *Banerjee's Inequalities* grew out of this effort as a way to check whether the Diophantine equations have no solutions within the iteration bounds of a particular loop. This method of dependence testing has proven to be very successful, but it cannot be applied where the subscript expressions are not linear in form.

More recently, dependence testing has been generalized to fit into a framework built around a linear-constraint solver. The Omega Solver was developed as an optimized form of the Fourier-Motzkin linear constraint-solving technique. Linear constraints can be derived from a program for many different kinds of analysis. Interprocedural analysis can be done by combining the

constraints derived from a subroutine with the constraints which are active at a call site. A solution to a given constraint system can be projected onto an arbitrary set of variables, making it possible to use the technique to generate run-time tests involving unknown variables, and produce symbolic dependence and distance vectors, plus many other types of information useful to a compiler. However, linear constraint solvers, like the Banerjee Test, are generally not usable in the presence of non-affine expressions.

Three important categories of array subscript expression components which still prevent parallelization of the loops where they appear are

1. unknown terms (such as values read from an input file),
2. terms in array subscript expressions about which too little is known, due to insufficient or imperfect program analysis, and
3. loops which use non-affine subscripts, such as appear in programs implementing a Fast Fourier Transform.

In many compilers, the appearance of any of these components in a loop causes the whole loop to be serialized. Some people attack the first category by compiling a program in the presence of its input files. This makes the compiled program usable only with that set of input files. Others try to speculatively parallelize such loops, but at execution time must *roll back* any changes made to memory if a dependence is discovered, then run the loop serially. The second category can include any term which the compiler doesn't understand or cannot process, such as `MAX(1, N)`. The compiler writer can theoretically reduce the size of this category by improving the compiler's analysis capabilities. The third category is related to the first two in that the compiler cannot process non-affine subscript expressions, but it is placed in a third category because it is the *form* of the expression, not any of its terms, which makes it unanalyzable. Prior to this thesis, only Blume's Range Test [10] could handle non-affine subscript expressions.

Subscripted-subscript expressions are well-known as being difficult for a compiler to analyze. They fit in the above classification in category two, unless the subscripting array is read from an input file. This means that the program often contains sufficient information for allowing a compiler to understand the access pattern caused by the subscripted-subscript access, but advanced analysis techniques are required to enable the parallelization of loops containing them.

Very few compilers can parallelize loops which contain array references whose subscript expressions fall into any of these three categories.

1.3 Compiler Organization

Compilers have traditionally been developed using a pass-oriented approach. A pass was defined as a traversal of the source program, or its machine representation. The first Fortran compiler [3] was developed at IBM in this way in the mid-1950s. At that time it was largely unknown what techniques would be necessary to develop the compiler, so they broke up the task to make it manageable. Each of the six "sections" of that compiler was developed by an autonomous group which developed its own input and output specifications.

Parallelizing compilers adopted this approach for similar reasons - it was not known what analysis would become important, or in what order to do the various analyses. Over the years, research and experience has led the field to a general consensus about the types of analysis which are important, but has not clarified the question of how to order them. *Should induction be run before or after reduction analysis? Should we run dependence analysis first or last? How can we have separate passes while avoiding having to re-do some basic analysis?* These are some typical questions which a compiler writer must face, and for which there have been no authoritative answers.

While the pass structure is useful for software engineering reasons, it has drawbacks because the various passes often need to do very similar analyses. This can lead to two different routines to do the same job, or duplicated code which is run repeatedly for each pass.

Some attempts have been made to take an "artificial intelligence" approach to combining compiler analyses [18], allowing the form of the program to direct the application of the compiler's techniques. In the end, these efforts have largely been abandoned because they have grown to be too complex.

1.4 The Focus of this Thesis

This thesis addresses the issues mentioned here by integrating several analyses into a single framework. Each technique has its natural place in the framework. This avoids the recomputation of analysis information for each technique.

This thesis will propose a simple analysis technique, called Memory Classification Analysis (MCA), based on a precise memory access representation called the Access Region Descriptor (ARD), which can support an interprocedural data dependence test called the Access Region Test (ART). Simplification operations are defined on the ARD. The precision of the ARD representation and its simplification operations, together make the ARD an ideal vehicle for generating messages for data movement between processors. The ARD is not tied to the declared dimensionality of an array and therefore the array reshaping problem is eliminated. This enables precise translation of memory access summaries across procedure boundaries.

The ARD can also handle non-affine subscript expressions, such as those which occur in FFT codes, enabling parallelization in situations that no other dependence test can handle. The ART does interprocedural privatization, reduction and induction analysis as a part of its normal operation. MCA is able to summarize the important information about the memory accesses in an arbitrary section of code in its *summary sets*. The summary sets are then used by the ART to parallelize the code.

When the techniques described in this thesis are unsure about their results, they are designed to gather conditions which, if proven true, could enable parallelization. These conditions can drive on-demand deeper analysis of the program, and whatever conditions can still not be proven can be formed into a test to be executed at runtime to choose between the serial and parallel versions of a loop. The conditions can also be used to assist in generating conditional prefetch commands.

2 EXISTING COMPILER ANALYSIS TECHNIQUES

Much research has focused on the question of what techniques are important to implement in a compiler. One of the first such studies [17] was done at Illinois as part of the Cedar project, by a group of people of which the author was a member. Similar studies have been done at Stanford [21]. Researchers at Stanford [22], and Minnesota [20], plus the PIPS group at École des mines de Paris [13, 14] and the Parafrase-2 group at Illinois [31], have implemented compilers including the same basic transformations found to be important in the Cedar Project. The results have been similar enough to form a general consensus as to which analysis techniques are important to provide in a parallelizing compiler. The Illinois results will be presented here as representative of this consensus.

2.1 Parallelization Studies at Illinois

The Polaris parallelizing compiler grew out of the Cedar project at the Center for Supercomputing Research and Development (CSRD), located on the campus of the University of Illinois at Urbana-Champaign. CSRD was formed in late 1984 for the purpose of building the Cedar supercomputer.

Once the Cedar machine was up and running, the Cedar Fortran parallelizer was used to compile the Perfect Benchmark codes. The codes were run on the Cedar machine and the speedup for them was measured. When the speedup proved to be much less than hoped for, an effort was launched to determine first, whether parallelism existed in the codes, and second, what analysis techniques implemented inside a compiler could achieve a speedup on the Cedar machine. The Perfect Benchmark codes were hand-parallelized in a mechanical manner, using only information and techniques which might realistically be available to a compiler.

The results of this study, presented in [17], were that the transformation techniques with the most overall impact across all codes were

- array privatization
- parallel reductions

- generalized induction variable (GIV) analysis, and
- transformations which map loops to the machine.

The most important analysis techniques were found to be

- interprocedural symbolic analysis
- improved data dependence analysis, and
- run-time analysis techniques.

It was decided to implement, in a new compiler, the most important techniques: array privatization, parallel reductions, and GIVs; along with an improved dependence test, a powerful symbolic infrastructure, interprocedural value propagation, and subroutine inlining. This new compiler was called Polaris.

After the Polaris compiler was built, studies of the performance resulting from its parallelization [17] confirmed what the hand analysis had predicted – significant speedups were possible from automatic analysis alone.

2.2 The Transformation Techniques

The following subsections describe the transformation techniques, implemented in Polaris, which are relevant to this thesis.

2.2.1 Array Privatization

The term *privatization* refers to the transformation of certain variables from being globally accessible to being only locally accessible to each processor. This requires that separate memory space be reserved for the variables in each processor's private memory. Privatization is possible in a situation such as:

```
do I = 1, N
   A = . . .
     = . . . A . . .
end do
```

This use of `A` within the loop represents a data dependence since the same memory location is used by all iterations. But privatizing `A` removes the dependence, potentially allowing the parallelization to occur.

If the above loop is to be parallelized, since `A` is defined before it is used within each iteration, it may be allocated private memory space within each processor's memory. If such a variable is used outside the loop, then we have to make sure that what would have been the last value of the variable in the serial loop gets copied out to the proper memory location for use outside the loop. This copy is called a *last value assignment*.

The same idea may be applied to arrays [37], but the subscripting expressions, used to assign array elements, complicate the analysis. It must be determined that some part (or all) of the array is assigned within each iteration before being used.

Flow-sensitive array privatization adds further difficulties. For instance, if an array is assigned under one condition and used under a different one, it can still be privatized as long as the compiler can prove that the condition guarding the use *implies* that the condition guarding the assignment was true. For example,

```
do I = 1, N
   if ( P ) then
      do J = 1, M
         A(J) = . . .
      end do
   endif
   if ( Q ) then
      do J = 1, M
         . . . = . . . A(J) . . .
      end do
   endif
end do
```

In this case `A[1:M]` can be privatized if we can prove that `Q` $\Rightarrow$ `P`. If so, then we know for sure that the assignment always precedes the use.

2.2.2 Parallel Reductions

Reductions also appear to the compiler as a data dependence, since the same memory location is used in multiple iterations. The pattern is as follows:

```
do I = 1, N
   A = A + . . .
end do
```

A memory location is read, then some arbitrary expression is evaluated and combined with it using a reduction operator (such as addition in the example), and the result is stored back in the original location. Arrays can be used as a reduction location in the same way, as long as the array element being read is the same as the one being written. Any number of these reduction statements can be used within a loop nest. For example:

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

Subscripted-subscripts can even be involved in this same general form:

```
do I = 1, N
   do J = 1, M
      A(B(J)) = A(B(J)) + . . .
   end do
   do J = 1, M
      A(B(J)) = A(B(J)) + . . .
   end do
end do
```

A reduction can be parallelized [32] in a number of ways, all of them potentially allowing the original operations to be done in a different order from that of the original loop. Because of this, roundoff error can accumulate differently in the parallel loop than in the original serial loop, and the results of the computation could be different from those of the serial loop. For this reason, the user must explicitly consent to the parallelization of reductions.

2.2.3 Generalized Induction Variable Analysis

Inductions are closely related to reductions. The pattern of use for the induction variable is the same as that for reduction variables, but the expression being added or multiplied with the induction variable is an integer constant instead of an arbitrary expression. Just as for reductions, any number of induction assignments can appear in a loop. For example,

```
do I = 1, N
   do K = I, N
      J = J + 1
      A(J) =  . . .
      J = J + 1
      A(J) + . . .
   end do
end do
```

The induction variable, like a privatizable variable and a reduction variable would appear to be a data dependence. The dependence is removed by the compiler computing a *closed form* for the variable, calculated in each iteration separately. The loops surrounding the induction assignments can be triangular[1] as well as rectangular. Triangular loops cause a more complicated closed form expression for the induction variable [32].

[1] an inner loop uses the loop index of an outer loop in one of its bounds expressions

2.3 The Symbolic Analysis Techniques

Symbolic analysis takes several forms inside Polaris. The interprocedural component is split into interprocedural value propagation and subroutine inlining. This is complemented by intraprocedural range propagation and powerful expression simplification and comparison routines.

2.3.1 Intraprocedural Symbolic Analysis

The intraprocedural range propagation algorithm [9] uses a data-flow based technique to discover lower and upper bounds on the values of program variables through the use of widening and narrowing operators. The variable ranges are stored in *range dictionaries*, which relate the value ranges to statements in the program. The range dictionaries then supply range information to the various analysis passes.

Within the Polaris privatization pass, the program is converted to Gated Single Assignment form [38], which is then used to do symbolic flow-sensitive analysis. Some primitive condition-proving is carried out using the def-use chain links inherent in the GSA form.

All Polaris expression manipulation is supported by powerful expression simplification routines. These routines convert expressions to a standard sum-of-products form, using constant folding and structural simplification, which facilitates expression comparison. Sophisticated comparison routines make use of the range dictionaries to determine whether one expression is less than, equal to, or greater than another expression.

2.3.2 Interprocedural Symbolic Analysis

Interprocedural value propagation [8] is used within Polaris. It is a version of a standard constant propagation algorithm in which the value being propagated is a symbolic expression. At join nodes in the interprocedural flow graph, if all expressions for the same variable entering the node are structurally identical, we can assign that expression to the variable at the node.

The propagation algorithm clones (specializes) subroutines where different sets of values flow into its parameters at different `call` sites. Whenever the values for formal parameters are determined for a particular subroutine, assignment statements are inserted inside the routine, at the subroutine entry point, assigning those values to the formal parameters.

2.3.3 Subroutine Inlining

Subroutine inlining [19] is the general technique of copying the body of a subroutine to its call sites, translating references to formal parameters into references to its actual parameters. There are several advantages to doing this:

- the subroutine boundary is eliminated, thereby allowing an intraprocedural analysis or transformation technique to essentially work interprocedurally.
- the subroutine jump, parameter storing and loading, and register saving and restoring overheads are eliminated at runtime.
- for small routines, register allocation can be made more efficient, because it can be done within two routines at once.
- any array reshaping which exists is eliminated.

Unfortunately, there are also disadvantages involved with subroutine inlining:

- the source code size can grow exponentially.
- compile time increases, in proportion to the source code growth, and the extra compile time is spent repeatedly compiling the same routines.
- complicated subscript expressions can result from array reshaping situations, sometimes complicated enough to foil dependence analysis.
- the declared dimensionality of an array which is a formal parameter is lost.

An example of the latter disadvantage is the following situation:

```
real A(P,Q,R)
call X ( A, N )
  . . .
subroutine X ( A, N )
real A(N,N)
do I = . . .
   do J = . . .
     A(I,J) . . .
   end do
end do
end
```

The access to `A` in subroutine `X` is obviously parallel. But when `X` is inlined, the mis-matched array declarations forces the inliner to linearize accesses to the array and the clear parallelism of the original subroutine is obscured:

```
real A(P*Q*R)
do I = . . .
   do J = . . .
      A(I-1 + (J-1)*N) . . .
   end do
end do
end
```

The information can be recovered by clever use of range information for the program variables, but the code is definitely not as easy to parallelize in its inlined form.

2.3.4 Improved Data Dependence Analysis

A data dependence test called the Range Test [8] was implemented in Polaris. It uses the symbolic expression manipulation package and the symbolic range information heavily.

The Range Test tests a pair of array references within a loop nest to determine whether they access the same memory locations. It uses two tests to determine this. In the first test,

the loop bounds are used to calculate the maximum and minimum array offsets produced for each array reference. If it can prove that the maximum offset of one reference is less than the minimum offset of the other, then it can prove the independence of the two references.

If the first test does not succeed in proving independence, then the second is used. In evaluating the independence due to a particular loop, the Range Test computes the maximum and minimum offsets for the two references symbolically, *in terms of the loop index* for the loop being tested. Then, it symbolically adds the stride of that loop to the minimum expression of the first reference and checks whether it is greater than the maximum of the second reference. If that is true, then it reverses the roles of the two references and symbolically adds the stride of the loop to the minimum expression for the second reference and checks whether that is greater than the maximum of the first reference. If both those checks are true, then the Range Test reports independence between the two references. If both checks are not true, the loops in the loop nest are interchanged (virtually, not physically), and the test is retried, until no more interchanges are left to be tried.

For sake of efficiency, the data dependence test pass in Polaris consists of a sequence of dependence filters, which eliminate dependences prior to the Range Test:

VarSpaceFilter - eliminates potential dependences between different variables (for example between A and B) which are not equivalenced together,

InputDepFilter - eliminates input dependences,

LoopNestFilter - eliminates dependences between references which have no loops in common,

ReadOnlyCallArgFilter - eliminates dependences due to a subroutine call whose arguments are read-only,

GcdFilter - the GCD Test,

SimpleSubscriptTestFilter - eliminates dependences due to identical subscript expressions,

RangeTestFilter - the Range Test,

SelfEqualFilter - eliminates self dependences with an = direction,

SelfOutputFilter - eliminates dependences with a backward (illegal) direction,

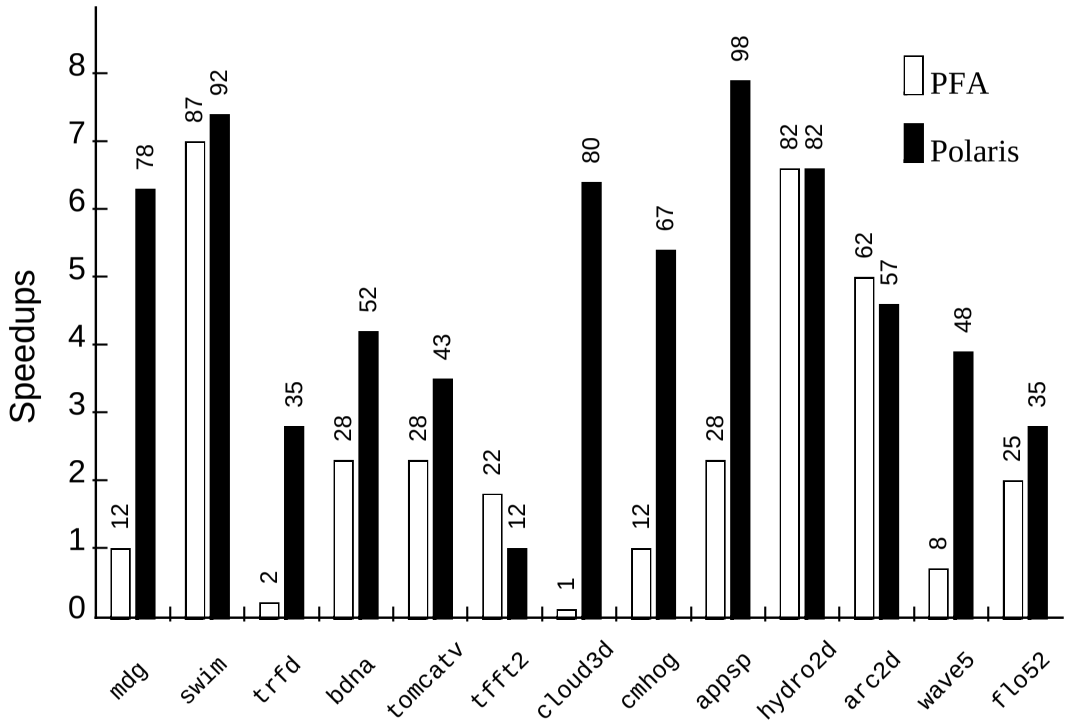

Figure 2.1: Speedups obtained from executing codes on a Silicon Graphics Challenge computer with 8 *MIPS R4400* processors. The codes were compiled with Polaris and with PFA, a commercial parallelizing compiler. The parallel execution time is compared with the serial execution time for each code. The numbers on top of the bars represent parallel efficiency.

PrivateFilter - eliminates dependences due to privatized variables, and

ReductionFilter - eliminates dependences due to reduction variables.

The use of this combination of filters was found by Blume to be essentially as powerful, on a set of benchmark codes, as was the same filters with the Omega Test substituted for the Range Test.

2.4 Polaris Results

The results (shown in Figure 2.1) of studies [7] using the Polaris compiler has proven that the combination of transformation and analysis techniques implemented in it are sufficient to get good speedups on a set of benchmark codes. This is used as evidence that those techniques form a good basis for a parallelizing compiler. These techniques will be combined in the compiler framework proposed in Chapter 6.

3 ARRAY REGION REPRESENTATION TECHNIQUES

A compiler issue which has drawn little attention in the past is how the form used for representing memory accesses within arrays affects the accuracy of compiler analysis. This thesis takes the view that the accuracy of the representation is critically important to the accuracy of the analysis. This chapter examines prior work which has addressed the issue of representing array accesses.

The prior work which has been done on representing the regions within an array which are accessed by a section of code has proceeded in three general directions: linear constraint-based forms, reference-list forms, and triplet-notation based forms.

3.1 Linear Constraint-based Techniques

Using linear constraint-based techniques to represent array accesses was first proposed by Triolet, et al [35]. Their representation was called a *region* and the overall parallelization technique was called *direct parallelization*. The array regions accessed in a subroutine were attached to a **call** statement for the subroutine in the form of a set of linear constraints constructed from the subscript expressions used to address array elements in the subroutine. This was done for the purpose of dependence analysis. They also used a set of constraints to represent the possible values of variables, and called this an *execution context*.

When a potential dependence between two array references was being tested, the linear inequalities associated with the two references were combined to form a linear system and the feasibility of the system was tested using Fourier-Motzkin elimination [16] techniques.

The other principle technique for doing interprocedural analysis at that time was inline-expansion of subroutines (subroutine inlining) [25]. Direct parallelization was deemed superior to subroutine inlining by Triolet because it made the resulting program more readable and limited the size of the routine which must be compiled, but direct parallelization suffered from a number of drawbacks:

- non-affine expressions cannot be used within Fourier-Motzkin techniques[1],
- Fourier-Motzkin reports real-number solutions, not just integer solutions,
- Fourier-Motzkin requires that the linear inequalities form a convex hull, forcing a loss of accuracy when some regions must be widened to make them convex,
- non-trivial array reshaping at a procedure boundary prompted the assumption that the whole array was accessed in the subroutine, and
- its theoretical complexity is exponential.

Work on constraint-based techniques since 1986 has attempted to address all of these drawbacks and make the representation useful for analyses other than just dependence analysis.

The advent of Pugh's Omega Solver [33] made the use of Fourier-Motzkin techniques practical. The algorithms built-in to the Omega Solver optimized it so as to avoid exponential running times for the constraints derived from most practical program situations. It also eliminates the requirement that the linear constraints form a convex hull, avoiding one source of precision loss.

The PIPS project at Ècole des mines de Paris [13, 14] has added an indicator of the accuracy of the representation to the representation itself. When linear system manipulations must approximate, sometimes it is appropriate to under-approximate (**MUST** analysis), while at other times it is appropriate to over-approximate (**MAY** analysis). But by making both MUST and MAY approximations at each step, it can be determined when a result is *exact* (when the MUST and MAY regions are identical). The PIPS compiler also uses an algorithm to handle some of the simpler array reshaping situations.

Work on the SUIF system at Stanford uses a very similar representation [2, 23, 28] to that of PIPS, and has a more sophisticated algorithm to handle the general problem of array reshaping. It employs *flow-sensitive interval analysis* and *selective procedure cloning* when a routine is called from two or more contexts, providing the benefits of full inline expansion without the large code growth. When using Fourier-Motzkin elimination to find solutions for a linear system, if such solutions are found, SUIF uses a branch-and-bound technique to determine

[1]When non-affine expressions occurred, constraints involving them were removed, causing the representation to lose accuracy.

whether any of the solutions are integers. The SUIF system uses a set of optimized operations for doing intersection, union, subtraction, etc, with systems of linear inequalities.

Both the PIPS and SUIF projects use the linear constraint-based representation as the basis for doing operations such as array privatization and locality analysis, in addition to dependence analysis.

Balasundaram and Kennedy [4] proposed a technique, involving Data Access Descriptors (DADs), which uses a set of linear constraints applied to an array reference, but only allows certain forms for the constraints. Constraints can be in one of two forms:

$$\text{lower bound} \ \leq \ \text{variable} \ \leq \ \text{upper bound}$$

$$\text{lower bound} \ \leq \ \text{variable } \mathit{op} \text{ variable} \ \leq \ \text{upper bound}$$

where *op* represents one of the operators $\{+, -\}$.

For example, a DAD could be written:

$$\left\{ A(x_1, x_2) \;\middle|\; \begin{array}{c} l_1 \leq x_1 \leq u_1 \\ l_2 \leq x_2 \leq u_2 \\ l_3 \leq x_1 + x_2 \leq l_3 \\ l_4 \leq x_1 - x_2 \leq l_4 \end{array} \right\}$$

DADs are limited in that they cannot express the whole range of constraints found in programs, but they can represent many of the commonly occurring forms, such as a whole array, a single row or column, a diagonal, or a triangular section. The typical set operations (intersection, union, etc) are defined on DADs.

3.2 Reference List-based Techniques

Li and Yew proposed Atom Images [26] and Burke and Cytron proposed the conversion of all arrays to a single declared dimension and all subscript expressions to a linearized form [11]. The reference-list techniques rely on making a list of each individual array reference in the code section. These methods retain all the precision of the original program because they keep all relevant information about each access, but they were not designed to summarize this

information. Linearization solves the array-reshaping problem since every array is represented in one-dimensional form.

3.3 Triplet Notation-based Techniques

Triplet notation is a simple representation for a set of integer values for each dimension which start at a lower bound and proceed to the upper bound via a stride. For example, the triplet notation for a two-dimensional array X is denoted by

$$X(lowerbound_1 : upperbound_1 : stride_1, lowerbound_2 : upperbound_2 : stride_2)$$

Triplet notation represents array accesses exactly when the subscript expressions are simple, with one loop index per array dimension. Triplet-based techniques have attempted to capitalize on the observation that most array subscript expressions are simple. When one loop index appears in the subscript expression of more than one array dimension of an array reference, the subscripts are said to be *coupled*. According to the study of Shen et al [34], 80% of subscript expressions are non-coupled. This observation implies that most references in a program are simple, and for these references the representation is exact. When the representation is exact, a compiler can use simple set operations (union, intersection, etc) without losing any precision.

Researchers at Rice University [12, 24] have devised several variants of *regular section descriptors* (RSDs), with operations defined on a lattice. RSDs are of the form $I + \alpha$, where I is a loop index and α is a loop invariant. Restricted RSDs were devised to handle diagonal access, and then Bounded RSDs were devised to express triplet notation with full symbolic expressions. The Polaris project at the University of Illinois has used full symbolic triplet notation to represent array accesses within its array privatization analysis [36] and its dependence analysis [10]. Researchers at the University of Minnesota have used Guarded Array Regions [20] which are equivalent to Bounded RSDs, with an additional predicate (guard). More information can be added to the guard to sharpen the accuracy in a given situation.

4 THE ACCURACY OF TRIPLET NOTATION

Descriptors representing memory accesses are the raw material used by compiler algorithms. Inaccuracies in these descriptors are bound to make the results of the compiler inaccurate. This is especially true for primitive languages like Fortran77, which use only predefined data types. Programmers must somehow map the data structures used in their algorithms onto the simple Fortran data types. Fortran77 also lacks a dynamic memory allocation mechanism, forcing programmers to do their own memory allocation. Programmers typically handle these deficiencies in Fortran by carefully dividing large arrays into sections, requiring complex subscripting functions to access the appropriate sections.

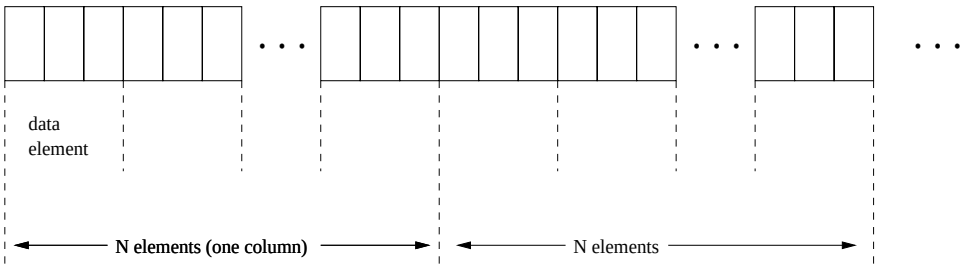

Figure 4.1: Programmer-built composite data structure in a one-dimensional array.

For example, a two-dimensional NxN array of data elements which each consist of three floating point values might be implemented inside a one-dimensional floating point array, by allocating the first $3N^2$ elements of the array. Refer to Figure 4.1 for a diagram of the layout of such an array.

The code for accessing the whole NxN array might look like this:

```
      real MEM(1000000)
      do I = 1, N
         do J = 1, N
R:          MEM((I-1) + (J-1)*3*N + 1) =  . . .
S:          MEM((I-1) + (J-1)*3*N + 2) = . . .
T:          MEM((I-1) + (J-1)*3*N + 3) = . . .
         end do
      end do
```

An inaccurate representation of the memory accessed by each reference site may force the compiler to conservatively assume that the references overlap, when in fact they don't because they actually refer to totally different logical entities which happen to reside in the same array.

In the case of the example, the triplet notation for the access to `MEM` in statement `R` would be $[1 : 3N^2]$. The triplet notation for the access in statement `S` would be $[2 : 3N^2 + 1]$. The triplet notation for statement `T` would be $[3 : 3N^2 + 2]$. These three ranges overlap, causing any dependence test based on triplet summarizations to declare a data dependence.

In this chapter, the problem with the accuracy of triplet notation for representing array accesses in Fortran77 will be discussed. Then, in Section 5.5, it will be shown how a new descriptor solves some of these problems and enables interprocedural analysis. Finally, in Sections 5.7 and 8.2, it will be shown how the simplification of these new descriptors enables more efficient analysis.

4.1 The Sources of Inaccuracy

As discussed in Chapter 3, the two major forms used for representing memory accesses are constraint-based notation and triplet notation. Triplet notation is used for its simplicity and constraint-based notation is used for its accuracy.

Both triplet notation and constraint-based notation use a form which has the same number of dimensions as the number declared in the program. This can cause problems for triplet notation in situations, as in the example above, where two loop indices are used in the same dimension, because a single triplet cannot express the access patterns caused by two independent indices. The other ways in which triplet notation loses accuracy are documented below.

Triplet notation loses accuracy when one of the following situations occurs for an array reference:

- an array reference is used in a subscripting expression (subscripted-subscripts).
- non-affine subscript expressions are used.
- the array reference occurs inside a triangular loop.
- a loop index appears in more than one subscript position (coupled subscripts [27]).
- more than one loop index appears in a single subscript position (multi-index subscripts).

4.1.1 Subscripted-subscripts

```
do I = 1, N
   . . . A(B(I)) . . .
end do
```

In most cases, it is unknown what values are in the array `B`, so the best way to make use of triplet notation is to make the descriptor:

```
A[min(B): max(B)]
```

Of course, this is still inaccurate since the values in array `B` may not be consecutive. If the values in `B` are not consecutive, this descriptor will include memory locations which are not actually accessed by the original loop.

4.1.2 Non-affine Expressions

Non-affine expressions often arise when the closed form for an induction variable is computed. If the induction takes place inside a triangular loop, there will be a term in the closed form involving `I**2`. If that induction variable is used in an array subscript expression, the expression will be non-affine.

Non-affine expressions can also arise in the subscript expressions of FFT codes. The use of the term `2**I` in array subscripts is typical for these codes.

When these forms happen, the "step" of the triplet notation would have to include the loop index variable itself. This is not allowed in triplet notation.

4.1.3 Triangular Access

```
do I = 1, N
    do J = I, N
       . . . A(I, J) . . .
    end do
end do
```

The triplet notation for the access to `A` above would be

```
A[ 1:N, 1:N ]
```

This obviously includes more memory locations than are actually accessed in the loop.

4.1.4 Coupled Subscript Access

When an array reference carries a loop index in more than one subscript position, the result is similar to triangular access, in that triplet notation must report the full range of that loop index, even though not all combinations of those values occur. For example in the loop:

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

the triplet notation form reports $\mathtt{N}^2$ accesses:

```
A[ 1:N, 1:N ]
```

although only `N` locations are actually accessed by the program.

4.1.5 Multi-index Subscripts

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

The triplet notation for this loop will definitely lose some of the information which exists in the original program. The access in the program moves in steps of one according to the `J` index, and steps of `N` according to the `I` index. This is impossible to represent in triplet notation. The best triplet notation can muster is

```
A[ 2:N*N+M ]
```

If `N` is greater than `M`, there are memory locations inside this region which are not accessed. If `N` is less than `M`, there are locations inside the region which are accessed multiple times. Both situations are useful to know about and neither can be gleaned from the triplet form.

4.2 Summary

The program situations described above represent all the ways in which a Fortran77 program can be written and not accurately represented with triplet notation. This can be easily seen by checking the forms which are allowed after eliminating all of the above forms from subscript expressions.

By eliminating all arrays from subscript expressions, we are left with only scalars. By eliminating non-affine expressions, we are left with subscripts of the form:

$$k_0 * I_0 + k_1 * I_1 + \cdots + k_{n-1} * I_{n-1} + k_n,$$

where k_i represents some constant expression and I_i represents a loop index. By eliminating triangular loops, we force all loops in the loop nest surrounding the array reference to be rectangular[1]. By eliminating multi-index subscripts, we are left with subscripts of the form:

$$k_i * I_i + c_i,$$

for each subscript position, where I_i is the loop index for some loop in the nest, and the two expressions k_i and c_i are constant expressions. By eliminating coupled-subscripts, we insure that each subscript expression contains a different loop index.

Therefore, after eliminating all of the situations listed in this chapter as causing inaccuracy in triplet notation, we are left with a rectangular loop nest surrounding array references containing subscript expressions in which each subscript position uses a single loop index, which appears in no other subscript position. Such array references are precisely representable in triplet notation.

[1]Rectangular loops have no loop indices in the loop bound expressions for that loop.

5 THE ACCESS REGION DESCRIPTOR

A hybrid form of memory access representation, combining a generalized triplet notation with constraints, was first described in [29]. It was further refined in [30] and called a **Linear Memory Access Descriptor** (LMAD). This thesis refines it yet again.

The Linear Memory Access Descriptor contains the basic information that is needed to describe a memory access pattern. However, to use the LMAD for flow-sensitive program analysis, additional information must be attached to it. The LMAD with its attachments will be referred to as an **Access Region Descriptor** (ARD), which contains an LMAD to describe the memory access pattern.

This chapter will fully describe the LMAD and show how to construct it from a program. It will describe the operations used to manipulate it and simplify it, some of its properties, and how it can be used to represent various access patterns which cannot be represented in triplet notation. Finally, the additions to the LMAD, needed to make it into an ARD, will also be described.

5.1 Basic Definitions

A **memory location** within a computer is a unit of memory with a hardware address, which can be **accessed** (read or written) by a hardware instruction. The **memory space** of a computer is the linear sequence of memory locations which make up all the memory usable by a program running on the computer.

A program is given access to the memory space of the computer through the computer language in which the program is written. The language allows the user to supply a **variable name** for a set of one or more memory locations. The language also defines a set of basic **data types** which refer to a set of rules about how to interpret the contents of the memory locations which make up a variable. Some languages allow a user to define new data types. Most languages allow the user to define collections of variables, called **arrays**.

Arrays are defined as having one or more **declared dimensions**. Each declared dimension has a declared lower bound and upper bound. One particular member of the array is called an

array element and referenced within a program at an **array reference site** which uses the array name and a list of **subscripting expressions**, one per declared dimension of the array. In Fortran, it is written as:

$$\mathrm{A}(expr,\ expr,\ \cdots,\ expr),$$

while in C or C++, it is written as:

$$\mathrm{A}[expr][expr]\ \cdots\ [expr].$$

A program typically uses loops nested around array references. Each loop typically has a *basic induction variable* [1] associated with it, which from now on will be called the **loop index**. It is safe to assume a single loop index for each loop, because if there are multiple induction variables being incremented within a loop, one can always be designated as the basic induction variable, and then used to express the values of all other induction variables in the loop [1]. If no induction variable exists within a loop in the original program, one can always be added by assigning a value of zero to it immediately before the loop entry point and incrementing it by 1 immediately after the loop entry point (inside the loop).

Each array reference selects an element from the array by use of an **array subscripting function**. The subscripting function is a function which maps a series of subscript expressions, one per declared dimension of the array, into an array element offset. This offset allows us to select any of the array elements within the array.

Let us assume that an array `A` is declared with m dimensions, using the lower and upper bounds ($lower : upper$) of the following declaration form:

$$\mathtt{A}(L_1 : U_1, L_2 : U_2, \cdots, L_m : U_m) \tag{5.1}$$

An array reference to the m-dimensional array `A`, which might look like:

$$\mathtt{A}(x_1, x_2, \cdots x_m)$$

is assumed to invoke a **subscripting function** F_m, which is defined as follows:

$$F_m(x_1, x_2, \cdots, x_m) \quad = \quad (x_1(\mathbf{i}) - L_1)\lambda_1 + (x_2(\mathbf{i}) - L_2)\lambda_2 + \cdots + (x_m(\mathbf{i}) - L_m)\lambda_m \tag{5.2}$$

where $\mathbf{i}$ refers to the set of loop indices for the surrounding 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 language. If the language stores arrays in column-major order, the λ_k is defined as:

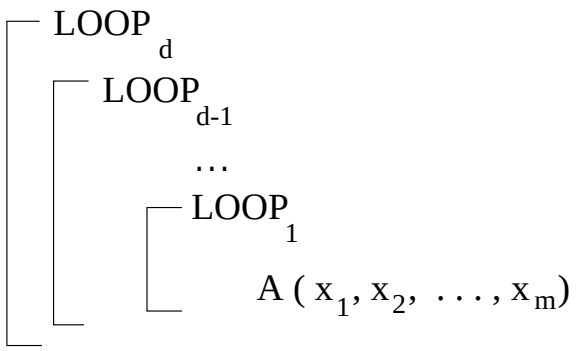

Figure 5.1: An m-dimensional array reference appearing in a d-loop nest.

$$\lambda_1 = 1$$
$$\lambda_k = \lambda_{k-1} \cdot (U_{k-1} - L_{k-1} + 1) \quad \text{for } k \neq 1.$$

If the language stores arrays in row-major order, the λ_k is defined as:

$$\lambda_k = \lambda_{k+1} \cdot (U_{k+1} - L_{k+1} + 1) \quad \text{for } k \neq m$$
$$\lambda_m = 1.$$

The Inbounds Assumption *All array subscript expressions will be assumed to be within the bounds of the associated declared dimension. If they are not, then the program is not a standards-conforming program and a compiler cannot be expected to produce correct code.*

So, in terms of declaration form 5.1, it is assumed that

$$L_k \leq x_k \leq U_k.$$

5.2 Representing the Array Accesses in a Loop Nest

When an m-dimensional array reference appears in a d-loop nest, as in Figure 5.1, the succession of values for each index produces a succession of values from the subscripting function, the **subscripting offset sequence**:

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

Even though the subscript expressions for the array reference may seem very complex, if the affect of a single loop index is isolated, the sequence of values almost always forms a simple, easily-describable sequence of integers.

For example, consider the following array reference:

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

Even though this array reference has coupled subscripts, which was established in Section 4.1.4 to not be representable in triplet notation, examining the output of the subscripting function, the sequence of integers can be described quite simply:

$$F_2(I, 2I)|_I = \ M+2,\ 3M+5,\ 5M+8,\ 7M+11, \cdots$$

The difference between two successive members of the sequence is constant: $2M+3$, there are N members in the sequence, and they start at $M+2$.

Consider another "complex" case:

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

The subscripting offset sequence is:

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

Again, 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*:

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

Again, there are N members of the subscripting offset sequence and they start at 2.

The subscripting function, F_m, establishes a relationship between the subscript expressions in an array reference and memory locations. This relationship is crucial to establishing the validity of a representation of the memory access.

Theorem 1 *A precise representation of the subscripting offset sequence provides a precise representation of the memory access.*

PROOF: Let's assume a precise representation of the subscripting offset sequence. This means that we can compute the offset in units of array elements of any element of the subscripting offset sequence. To determine the memory location of a particular subscripting offset sequence element, we merely must determine the size of an array element in memory units, multiply by the element offset, and add that to the memory address of the first element of the array to get the address of the starting memory location for that array element. This provides a one-to-one correspondence between subscripting offset sequence elements and memory locations. ■

The term which will be used to refer to a set of memory locations accessed by a set of memory references in a program will be the **access region**.

Definition 1 *Given an array A, and a program region $\mathcal{P}$ (such as an individual array reference, a statement, a loop, or a subroutine), the* **access region** *of A due to $\mathcal{P}$ is the set of all memory locations represented by elements of A accessed during execution of $\mathcal{P}$. The order of the access is unimportant.*

5.2.1 The Linear Memory Access Descriptor

The Linear Memory Access Descriptor makes use of Theorem 1 by providing a way to accurately describe the subscripting offset sequence produced by an array reference within a section of code, such as a loop. The difference between successive values of the subscripting offset sequence is called the **stride**, the number of elements in the subscripting offset sequence for a particular loop index is determined from the loop bounds of the index, and the difference between the first and last values in the sequence is called the **span**. A stride/span pair is called an **access**

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

Figure 5.2: The LMAD Form.

dimension. One access dimension will be computed for each surrounding loop. The span is meant to represent the full array element distance moved during the execution of the loop.

The effect of each surrounding loop can be calculated separately, giving a stride/span pair (access dimension) for each. Together, the collection of dimensions for a memory access represents the **pattern** of the access due to the whole loop nest. For a particular memory access, the pattern begins at a particular point in memory, which is called the **base offset**, since it represents an offset from the first element of the array.

Together, the dimensions of an access and the base offset make up what is called a **Linear Memory Access Descriptor**, or LMAD. It is written as shown in Figure 5.2, with the name of the program variable on the left, a comma-separated list of strides (δ_i) as superscripts to the variable name, a comma-separated list of spans (σ_i) as subscripts to the name, and the base offset (τ) separated from the rest of the descriptor by a plus sign. The stride and span for a particular dimension are at the same relative position in the two lists. A **dimension-index** (explained below) and its upper bound is associated with each dimension, and is optionally written as

$$[\text{variable name} \le \text{upper bound}],$$

subscripting the stride of its dimension. The notation is shown in Figure 5.2.

Refer to Figure 5.3 for an example memory access, and the LMAD which may be constructed for it. The memory access diagram in the figure shows the elements of an array `A` which are accessed by a loop. The nested loop causes three different strides through memory – a stride of three caused by the inner-most loop (the `K` loop), a stride of 14 caused by the `J` loop, and

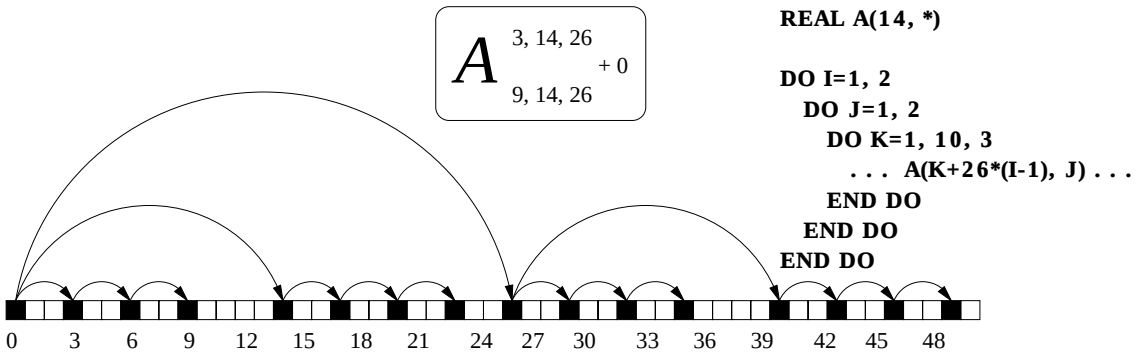

Figure 5.3: A memory access diagram for the array `A` in a nested loop and the Access Region Descriptor which represents it.

a stride of 26 caused by the `I` loop. Notice that constructing the LMAD does not require knowledge of the size of the second declared dimension of `A`. Also notice that the loop structure of the program causes a three-dimensional access to the array even though the array is declared with two dimensions. This is a common programming practice, and it is very important that an array access representation be able to represent such a pattern accurately.

The **dimension-index** associated with each dimension of an LMAD is a variable that takes on a set of values, much as the loop index in a program does. The dimension-indices act as the loop indices of a set of nested loops, generating the subscripting offset sequence for the array reference. The original dimension-indices represent a normalized form for the loop indices in the program. *Loop normalization* [5] is a well-known technique used to produce a new loop index which starts at 0 and proceeds to an upper bound in steps of 1.

For the original loop

```
do I = P, Q, R
   . . .  A(I) . . .
end do
```

a dimension-index `I'` is produced whose lower bound is 0, and whose upper bound is $(\mathtt{Q}-\mathtt{P})/\mathtt{R}$. Every appearance of the original loop bound in the offset expression of the LMAD is replaced by the expression $\mathtt{I}'\cdot\mathtt{R}+\mathtt{P}$. This produces an LMAD which represents the same memory locations as if the original loop were written in the normalized form:

```
do I' = 0, (Q-P)/R, 1
   . . .  A(I'*R + P) . . .
end do
```

The dimension-indices provide a means to determine how many elements of the subscripting offset sequence are produced by that dimension. Since the lower bound and stride are always 0 and 1, respectively, the only value which needs to be stored with the LMAD is the upper bound. This is called the **dimension-index bound**, represented as u_i for dimension i. The number of elements of the subscripting offset sequence due to dimension i is $(u_i + 1)$.

For instance, the dimension-index for the `I` loop in Figure 5.4 would have the bounds:

$$0 \leq I' \leq 99,$$

so the dimension-index bound would be 99, and the number of elements in the subscripting offset sequence would be 100.

The dimension-index for a dimension would be written in the LMAD notation as a subscript on the stride expression for that dimension, but it need not appear there when it is "understood". It must appear when the dimension-index itself is used in some stride or span expression on the LMAD.

5.2.1.1 Constructing an LMAD

```
      real A(100, 100)
      do I = 1, 100
         do J = 1, M
T:          A( I, J )
         end do
      end do
```

Figure 5.4: An array reference inside a loop nest.

The LMAD for an array reference within a loop nest is constructed from the inside of a loop-nest toward the outside. Consider the array reference depicted in Figure 5.4. First, the LMAD for the access within statement `T` is computed as a scalar access, following the Construct_LMAD algorithm of Figure 5.5. To do this, we ignore the surrounding loops and use the subscripting function F_m to calculate the base offset, while setting the stride and span to zero. Refer to Figure 5.5 for the algorithm.

Then, the LMAD is *expanded* (as described in Section 5.2.1.2) by each loop index in turn, from the inner-most to the outer-most, in order to represent the memory locations accessed by all iterations of each loop.

For example, the statement T in the figure produces the following LMAD for the scalar access, ignoring the loop statements:

$$\text{LMAD}_T \quad = \quad \mathcal{A}_0^0 + J' * 100 + I' \tag{5.3}$$

where J' and I' are dimension-indices, and $0 \leq I' \leq 99$ and $0 \leq J' \leq M - 1$.

Function Construct_LMAD

Input: List of subscript expressions $(x_1, x_2, \cdots, x_m)$
array declaration H$(L_1 : U_1, L_2 : U_2, \cdots, L_m : U_m)$
Output: scalar LMAD for the access

```
Function Construct_LMAD:
    Lmad.τ ← F_m(x_1, x_2, ···, x_m);
    for (loop L = innermost-loop L_1 to outermost-loop L_d) {
        Lmad ← Expand(loop L, Lmad);
    }
  return Lmad;
end Function Construct_LMAD
```

Figure 5.5: A function for constructing an LMAD for an array reference.

5.2.1.2 Expanding an LMAD by a Loop Index

The process of taking the LMAD representing the accesses for a single iteration of a loop and forming a new access descriptor which represents the region accessed when the loop index moves through the whole range of its values is referred to as **expanding** the access by the loop index. Refer to Figure 5.6 for the full algorithm.

The dimensions of the original descriptor are copied intact to the new descriptor. A single new dimension and a new base offset expression are formed, using the original descriptor's base offset expression as input. The operation is detailed in Figure 5.6.

Given a descriptor with $k-1$ (stride, span) pairs (δ_1, σ_1), (δ_2, σ_2), through $(\delta_{k-1}, \sigma_{k-1})$, a loop index i_k, and loop bounds (lower, upper, stride) (b_k, e_k, s_k), a dimension-index i'_k is created with an upper bound u_k of $(e_k - b_k)/s_k$. The original loop index i_k is replaced in the base offset expression with the new expression $i'_k \cdot s_k + b_k$. Then, a temporary stride value

Function Expand

Input: Original LMAD $\mathcal{H}^{\delta_1,\cdots,\delta_{k-1}}_{\sigma_1,\cdots,\sigma_{k-1}}+\tau$,
loop index i_k with loop lower bound b_k, upper bound e_k, and stride s_k
Output: LMAD expanded by the loop index i_k, and new dimension-index i'_k

Note: Here $f[j \leftarrow x]$ means to substitute x for j in function f

Function Expand:
Create dimension-index i'_k with $0 \le i'_k \le u_k$, $u_k = (e_k - b_k)/s_k$
$\tau \leftarrow \tau[i_k \leftarrow i'_k \cdot s_k + b_k]$
$\delta_{temp} \leftarrow \tau[i'_k \leftarrow i'_k + 1] - \tau$;
if $(\delta_{temp} < 0)$ {
$\delta_k \leftarrow -\delta_{temp}$;
if $(u_k \equiv \infty)$ {
$\sigma_k \leftarrow \infty$;
} **else** {
$\sigma_k \leftarrow \tau[i'_k \leftarrow 0] - \tau[i'_k \leftarrow u_k]$;
}
$\tau_{new} \leftarrow \tau[i'_k \leftarrow u_k]$;
} **else** {
$\delta_k \leftarrow \delta_{temp}$;
if $(u_k \equiv \infty)$ {
$\sigma_k \leftarrow \infty$;
} **else** {
$\sigma_k \leftarrow \tau[i'_k \leftarrow u_k] - \tau[i'_k \leftarrow 0]$;
}
$\tau_{new} \leftarrow \tau[i'_k \leftarrow 0]$;
}
Insert new dimension in LMAD, sorted according to δ_k
return $\mathcal{H}^{\delta_1,\cdots,\delta_{k-1},\delta_k}_{\sigma_1,\cdots,\sigma_{k-1},\sigma_k} + \tau_{new}$;
end Function Expand

Figure 5.6: A function for expanding an LMAD by a loop index.

(δ_{temp}) is computed by replacing i'_k wherever it appears in the offset expression, by $i'_k + 1$, then subtracting the original base offset expression.

If the temporary stride is positive, then it becomes the permanent stride. The span is created by subtracting the base offset with zero substituted for i'_k from the base offset with u_k substituted for i'_k, and the new offset expression is produced by replacing i'_k in the offset expression by 0.

If the temporary stride is negative, then the descriptor is *normalized* to have a positive stride. The temporary stride is negated and assigned to the permanent stride. The span is created by subtracting the base offset with u_k substituted for i'_k from the base offset with zero substituted for i'_k. The new dimension is then inserted in "stride-sorted order" into the list of

dimensions on the LMAD. The new base offset expression is produced by replacing i'_k in the offset expression by u_k.

For example, expanding LMAD_T for the J loop produces a new stride expression of

$$[(J' + 1) * 100 + I'] - [J' * 100 + I'] = 100$$

and a new span expression of

$$[(M - 1) * 100 + I'] - [0 * 100 + I'] = (M - 1) * 100.$$

The new offset expression becomes

$$0 * 100 + I' = I'$$

making the expanded descriptor

$$\mathcal{A}^{100}_{(M-1)*100} + I'.$$

5.2.2 Validity of the LMAD Representation

The LMAD provides a valid representation for a sequence of memory references as long as the base offset, the stride expression, and the dimension-index bound are valid. These three expressions provide

- the starting point,
- the difference between successive sequence members, and
- the number of members

which are needed to accurately represent the subscripting offset sequence (as discussed in Section 5.2).

Notice that for a span to represent the full array element distance moved due to a single loop index, the values from the subscripting function must move in the same direction during

the entire range of values for the index. Another way of saying this is that the sequence of values must be **monotonic** over the domain of a loop index for the span to be correct.

Definition 2 *Let $A(\mathbf{x_m}(\mathbf{i}))$ be an array reference, where $\mathbf{x_m}$ stands for the m subscript expressions for the m-dimensional array* `A`*, which are functions of* $\mathbf{i}$*, which stands for the list of the d loop indices $(i_1, \cdots, i_k, \cdots, i_d)$ for loops surrounding the array reference. The subscripting function $F_m(\mathbf{i})$ is* **monotonic** *for index i_k if $F_m(\mathbf{i})$ is either always decreasing or always increasing for the sequence of values taken on by i_k, starting at 0 and proceeding to the dimension-index bound in steps of 1.*

If the subscripting offset sequence is not monotonic over the full domain of the dimension-index, then the span would not necessarily represent the full length of the subscripting offset sequence. The representation can still be valid even if the span is not correct, as long as the base offset, the stride, and the dimension-index bound are correct. Since the span is used in many LMAD operations, however, the operations which can be done on an LMAD with non-monotonic dimensions may be limited.

5.2.2.1 LMADs for `while` Loops

Sometimes the loop bounds are not available, such as for a `while` loop whose exit condition causes a number of iterations which is unknowable at compile-time. Often the stride of the accesses within a `while` loop *is* knowable at compile time, but the number of iterations is not. If either b_k or e_k are not available, then we can use an ∞ value for the span and the dimension-index bound.

5.2.3 Normalizing LMADs

As mentioned in Section 5.2.1.2, expanded descriptors are *normalized* by making the strides positive, wherever possible. Normalizing the access descriptors helps us compare array regions. For example, consider the loop in Figure 5.7. The two references to array `A` proceed in different directions in memory with respect to loop index I_2. Nevertheless, they access exactly the same memory locations during the course of the execution of the outer loop.

If the two descriptors are not normalized, the two descriptors for the references are $\mathcal{A}^{-1,2}_{-2,12}+4$ and $\mathcal{A}^{1,2}_{2,12}+2$. By making the strides positive, the two descriptors both become $\mathcal{A}^{1,2}_{2,12}+2$. In this form, it is trivial to determine that the descriptors both refer to the same memory locations.

```
for I2 = 0 to 6 with step 2 do
    for I1 = 1 to 3 with step 1 do
        A(−I1 + 2 ∗ I2 + 5) = ···
        A(I1 + 2 ∗ I2 + 1) = ···
    end
end
```

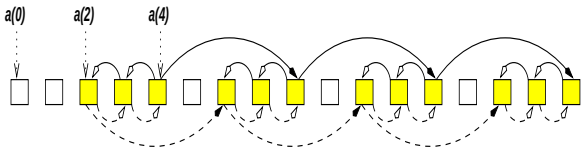

Figure 5.7: Access to array A through two references and their access patterns in memory. Solid lines denote the access for $A(-I_1 + 2 * I_2 + 1)$ and dashed lines the access for $A(I_1 + 2 * I_2 - 3)$. Arrows with white heads and black heads keep track of the access driven by indices I_1 and I_2, respectively.

5.2.4 Definitions for LMADs

Definition 3 *On the assumption that two LMADs* $\mathbf{A}$ *and* $\mathbf{A}'$ *represent the access regions* $\mathcal{R}$ *and* $\mathcal{R}'$*, respectively,*

1. $\mathbf{A} \cup \mathbf{A}'$ *represents the* **aggregated** *LMAD of the two access regions, that is,* $\mathcal{R} \cup \mathcal{R}'$.
2. *If* $\mathcal{R}'$ *is a subregion of* $\mathcal{R}$ *(that is,* $\mathcal{R}' \subset \mathcal{R}$*), then it is written* $\mathbf{A}' \subset \mathbf{A}$.
3. *Let* $\mathbf{A} = \mathcal{A}^{\delta_1,\cdots,\delta_k,\cdots,\delta_d}_{\sigma_1,\cdots,\sigma_k,\cdots,\sigma_d} + \tau$. *Suppose* $\mathbf{A}'$ *is built by eliminating the kth dimension* (δ_k,σ_k) *from* $\mathbf{A}$*, that is,* $\mathcal{A}^{\delta_1,\cdots,\delta_{k-1},\delta_{k+1}\cdots,\delta_d}_{\sigma_1,\cdots,\sigma_{k-1},\sigma_{k+1},\cdots,\sigma_d} + \tau$. $\mathbf{A}'$ *is the* k-**subLMAD** *of* $\mathbf{A}$*. If one or more dimensions of* $\mathcal{A}'$ *is missing from* $\mathcal{A}$*, then* $\mathcal{A}'$ *is simply called a* **subLMAD** *of* $\mathcal{A}$.

Notice that when $\mathbf{A}'$ (with access region $\mathcal{R}'$) is a subLMAD of $\mathbf{A}$ (with access region $\mathcal{R}$), then $\mathcal{R}' \subset \mathcal{R}$.

Definition 4 *Two LMADs* $\mathbf{A}$ *and* $\mathbf{A}'$ *are said to be* **equivalent**, *denoted by* $\mathbf{A} \equiv \mathbf{A}'$*, if they represent the same access region.*

Definition 5 *If* $\mathbf{A}$ *and* $\mathbf{A}'$ *are two LMADs with the same stride/span pairs in the same order (but possibly different base offsets), then* $\mathbf{A}$ *and* $\mathbf{A}'$ *are* **isomorphic**, *denoted by* $\mathbf{A}//\mathbf{A}'$*, meaning the LMADs represent the same access pattern.*

5.2.5 The Dimensional Order for LMADs

The order in which dimensions are written on the LMAD denotes a nesting order for a set of loops which produce the subscripting offset sequence representing the memory access. Dimensions written to the left may be thought of as nested inside dimensions written to the right.

In some cases, such as for array references inside of triangular loops, the stride or span of one dimension may refer to the loop index associated with a different dimension. When this situation occurs, the inner dimension is said to be dependent on the outer dimension. No manipulation of the two dimensions may be done which would cause the original dependent inner dimension to be written to the right of the outer dimension.

Generate Subscripting Offset Sequence

Input: the LMAD $\mathcal{H}^{\delta_1,\cdots,\delta_d}_{\sigma_1,\cdots,\sigma_d}+\tau$,
with dimension-indices $i'_1, i'_2, \cdots, i'_d$ and
dimension-index bounds $u_1, u_2, \cdots, u_d$,
Output: The Subscripting Offset Sequence $S_1, S_2, S_3, \cdots$

Note: Here $f[j \leftarrow x]$ means to substitute x for j in function f

Function Generate_SOS(LMAD) **returns** Sequence:
$k \leftarrow 0$;
$T_1 \leftarrow 0$;
do $q_1 = 0, u_1, 1$;
$T_2 \leftarrow 0$;
do $q_2 = 0, u_2, 1$;
$\cdots$
$T_d \leftarrow 0$;
do $q_d = 0, u_d, 1$;
$k \leftarrow k + 1$;
$S_k \leftarrow \tau + \sum_{i=1}^{d} T_i$;
$T_d \leftarrow T_d + \delta_d[i'_d \leftarrow q_d]$;
end do
$\cdots$
$T_2 \leftarrow T_2 + \delta_2[i'_2 \leftarrow q_2]$;
end do
$T_1 \leftarrow T_1 + \delta_1[i'_1 \leftarrow q_1]$;
end do
return Sequence S;
end Function Generate_SOS ;

Figure 5.8: Generating the subscripting offset sequence from an LMAD.

The subscripting offset sequence can be generated from an LMAD as shown in the algorithm of Figure 5.8. The sequence elements are produced by the nested loops controlled by the dimension-indices. The very first value in the sequence is the LMAD base offset, τ, since that is the only element for which all dimension-indices have the value 0, causing the $T_1 \equiv T_2 \equiv \cdots \equiv T_d \equiv 0$. Thereafter, the values are determined by the values of the dimension-indices.

Theorem 2 *Let* $\mathbf{A}$ *and* $\mathbf{A}'$ *be LMADs. If* $\mathbf{A}'$ *has the same offset as* $\mathbf{A}$ *and the same stride/span pairs as* $\mathbf{A}$*, none of which are dependent on any other, but written in a different order, then* $\mathbf{A} \equiv \mathbf{A}'$.

PROOF: We can use **Generate_SOS** to prove that the two LMADs are equivalent. Equivalent LMADs produce the same subscripting offset sequence elements, but possibly in a different order. Since the two LMADs have the same offset, the first element of the subscripting offset sequence generated by both will be the same value, since $\sum_{i=1}^{d} T_i = 0$ in both cases. Each unique vector of values for the dimension-indices, $\mathbf{j} = (j'_1, j'_2, \cdots, j'_d)$, where $0 \leq j'_1 \leq u_1, 0 \leq j'_2 \leq u_2, \cdots, 0 \leq j'_d \leq u_d$, generates one of the sequence elements. By nesting the dimension-indices differently, we don't change the number of values each produces, so it is easy to see that a different nesting does not change the number of sequence elements.

Regardless of its position in the loop nest, the T_k variable's value is dependent solely on the current value of the q_k variable. So, a different nesting of dimension-indices, will simply change the order in which the T_i variables take on their values, not the values themselves. Since the sequence results from the $\sum T_i$, the same values will result even when the T_i take on values in a different order. This guarantees that the S_k associated with each unique vector $\mathbf{j}$ will be the same for both the original and the permuted loop nest. We know that each unique vector $\mathbf{j}$ will occur in both the original sequence and the permuted sequence. This means that all the values S_k in the original sequence will have a counterpart somewhere in the permuted sequence. ■

The affect of changing the LMAD dimensional order on the order of memory access is shown in Figure 5.9.

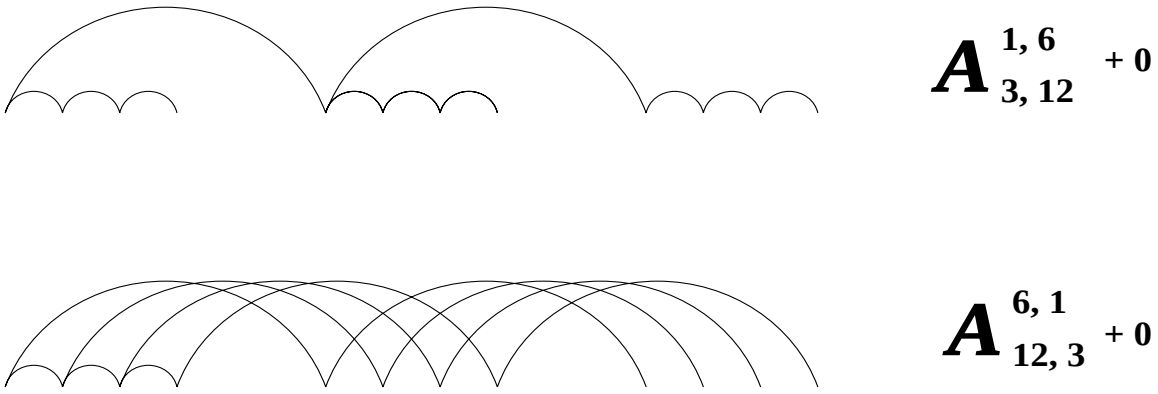

Figure 5.9: Access patterns produced by reversing the order in which dimensions are written.

5.3 Operations on LMADs

5.3.1 Upper and Lower Bounds of an LMAD

$$\text{Let } \mathcal{D} = \mathcal{H}^{\delta_{i_1},\delta_{i_2},\cdots,\delta_{i_d}}_{\sigma_{i_1},\sigma_{i_2},\cdots,\sigma_{i_d}} + \tau.$$

Definition 6 *The* **lower bound** *of an LMAD is the smallest offset represented by the LMAD (the offset of the array element closest to the first element of the array). This is simply the base offset.*

$$\text{lowest}(\mathcal{D}) = \tau \tag{5.4}$$

Definition 7 *The* **LMAD k-dimensional width** *is the sum of the spans of the first k dimensions. The* **LMAD width** *is the overall width of the LMAD. For an n-dimensional LMAD it is the n-dimensional width.*

$$\text{width}_k(\mathcal{D}) = \sum_{j=1}^{k} \sigma_{i_j} \tag{5.5}$$

Definition 8 *The* **upper bound** *of an LMAD is the largest offset represented by the LMAD (the offset of the array element furthest away from the first element of the array).*

$$\text{highest}(\mathcal{D}) = \tau + \text{width}_d(\mathcal{D}) \tag{5.6}$$

5.3.2 The Overlap Test for LMADs

Whenever an LMAD is constructed, it is necessary to test whether any two values in the subscripting offset sequence for a given LMAD are the same. It is possible to make a conservative test to determine that.

Consider the following loop:

```
real A(0:100)

do I=0,N,2
   do J=0,5
      A(3*I+J) = . . .
   end do
end do
```

The LMAD representing the access would be:

$$\mathcal{A}^{1,6}_{5,3N}+0$$

Assuming for simplicity that N is odd, the access region specification above produces the integer sequence:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ... , 3*N-1, 3*N.
```

If the loop had been written slightly differently:

```
real A(0:100)

do I=0,N,2
   do J=0,5
      A(2*I+J) = . . .
   end do
end do
```

the access region form would look slightly differently:

$$\mathcal{A}^{1,4}_{5,2N}+0$$

and the sequence produced would contain some duplicated integers:

```
0, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9, 8, ... , 2*N-1, 2*N
```

When does this duplication occur? A simple test can check for it.

When it is possible to sort the dimensions by their strides, δ_1 being the smallest stride and δ_d being the largest stride, then if

$$\sum_{i=1}^{k-1} \sigma_i < \delta_k \qquad\qquad [\textbf{No overlap test } - \textbf{ dimension } k]$$

holds, then no duplication due to index k can exist in the sequence.

The reason for this is easy to see in terms of the nature of nested loops, and the strides and spans due to each loop. The nature of a nested loop is that during one iteration of a given loop, all the inner nested loops must iterate to completion. For the overall sequence of integers to be unique, the values due to any given loop must "stride over" all the values produced by all loops nested inside it, and the length to be strided over is represented by the sum of the inner spans.

5.3.3 Zero-Span Insertion

If one LMAD is a subLMAD of a second, it is always possible to construct a new dimension in the subLMAD for each missing dimension by using the stride from the second LMAD, a dimension-index bound of 0, and a span of zero. This is useful in the cases where an algorithm is only defined for LMADs with the same number of dimensions.

Theorem 3 *an LMAD* $\mathcal{A}^{\delta_1,\cdots,\delta_d}_{\sigma_1,\cdots,\sigma_d}+\tau$ *can be expanded to form other equivalent descriptors by adding a dimension* $(\delta^*, 0)$ *in any position, such as* $\mathcal{A}^{\delta_1,\cdots,\delta^*,\cdots,\delta_d}_{\sigma_1,\cdots,0,\cdots,\sigma_d}+\tau$*, where* δ^* *can be any integer.*

PROOF: A dimension describes the movement from a lower bound to an upper bound with a stride. The span is defined as the difference between the upper bound and the lower bound. If the span is zero, the upper bound and lower bound are the same, thus describing no movement at all. A dimension involving no movement can have any stride, and still neither adds nor subtracts elements to/from a given access region. ■

5.4 LMAD Matching

When two or more LMADs are involved in some kind of operation, their dimensions must be **matched** to determine how similar the LMADs are. Similarity between LMADs is based on whether the strides, spans and base offsets match.

5.4.1 LMAD Similarity Types

Two expressions are said to match one another when structural transformation according to the rules of arithmetic can transform one into the other. Two dimensions are said to match when both the stride expressions and the span expressions match.

To match LMADs, a **correspondence** must be established between the dimensions of one and the dimensions of the other. A correspondence can be set up between two dimensions when at least the strides of the two dimensions match.

The similarity between two LMADs can be one of several **Similarity Types**, as follows:

equivalent – All dimensions and the base offsets match.

dimension-equivalent – All dimensions match.

stride-equivalent – The LMADs have the same number of dimensions and a dimension-matching can be found where each of the strides of one LMAD matches a stride of the other.

semi-equivalent – A correspondence can be found such that each dimension of one LMAD matches some dimension of the other LMAD, but at least one dimension of the latter remains unmatched, plus the base offsets match.

semi-dimension-equivalent – The base offsets of the two LMADs match, one LMAD has fewer dimensions than the other, and the LMAD with fewer dimensions is a subLMAD of the other.

semi-stride-equivalent – A correspondence can be found such that each dimension of one LMAD has a stride matching that of some dimension of the other LMAD, but at least one dimension of the latter remains unmatched.

The different forms are illustrated in Figures 5.10, 5.11, and 5.12.

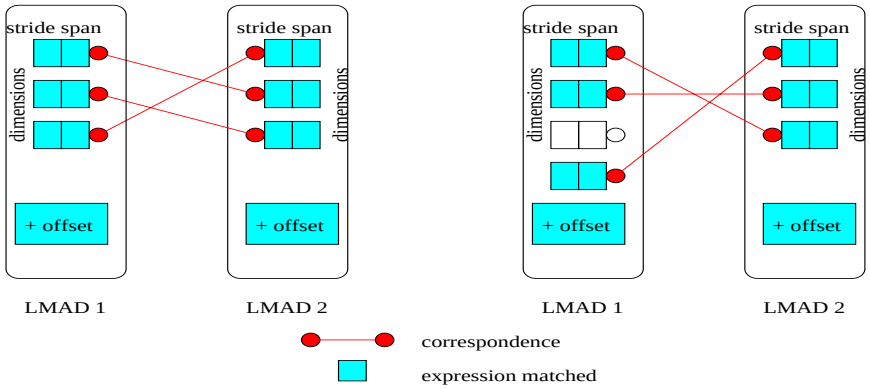

Figure 5.10: Equivalent and Semi-Equivalent LMADs.

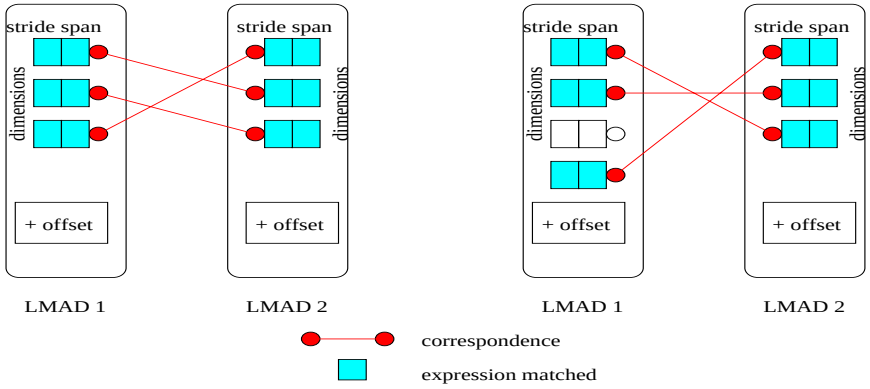

Figure 5.11: Dimension Equivalent and Semi-Dimension Equivalent LMADs.

If at least one dimension from each LMAD remains without a correspondence, then the two LMADs are *not similar*.

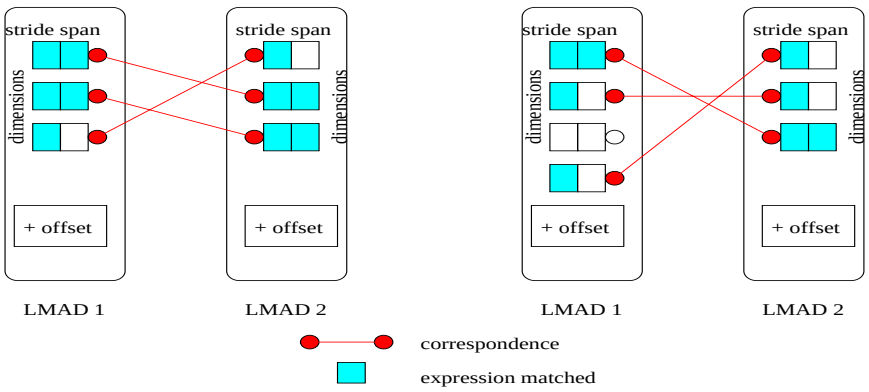

Figure 5.12: Stride Equivalent and Semi-Stride Equivalent LMADs.

5.4.2 Dimension Matching Between LMADs

Many LMAD operations involve comparing two LMADs. Most operations cannot be done unless it can be determined that the LMADs being operated on are similar. Yet there is no guarantee that this is the case. An important advantage of the representational forms which are tied to the declared dimensionality of the variables, unlike the LMAD, is that the descriptors for a particular array will all have the same number and size of dimensions within the same subroutine. Therefore, they all match by default. The array reshaping problem is a manifestation of this dimensionality mismatch-match for those forms.

By eliminating the dependence on the declared dimensionality of the variables, the array reshaping problem for LMADs is eliminated at subroutine boundaries, as will be discussed in Section 5.6, yet the consequence is that a simpler form of it is faced at every operation between LMADs.

One result of this work is that, for the programs tested, the similarity of access patterns for the same variable within a program makes the LMADs comparable in most cases even without a dependence on the declared dimensionality of variables.

In the cases where one of the two LMADs involved in an operation is a subLMAD of the other, the subLMAD may be converted to a stride-equivalent LMAD (according to Theorem 3) by inserting zero-span dimensions with strides matching those of the missing dimensions, as shown in Figure 5.13.

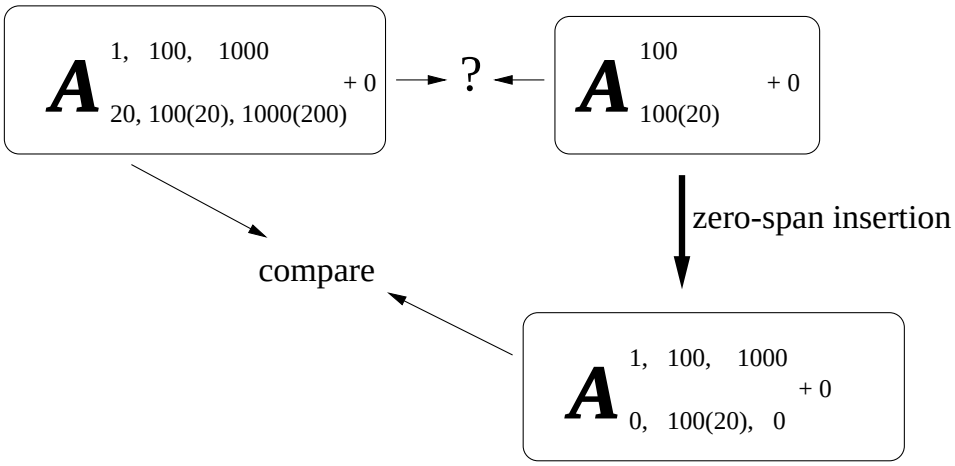

Figure 5.13: Attempting to operate on an LMAD and a subLMAD is made easier by inserting a zero-span dimension.

5.4.2.1 Sorting the LMAD Dimensions

By keeping the dimensions in sorted order when constructing the LMAD, the job of comparing two LMADs is made faster. The value ranges computed by range propagation are used to help with the sorting. Whenever a new dimension is added to a given descriptor, an attempt is made to insert it in the list of dimensions in its proper sorted position, using an insertion sort. Consider the following two loops:

```
          real A(100,100)

          do I=1,N
             do J=1,M
S1:             A(J,I)
             end do
          end do
          do L=1,M
             do K=1,N
S2:             A(L,K)
             end do
          end do
```

The LMAD for statement S1 is constructed as

$$\mathcal{A}^{1,100}_{M-1,100(N-1)}+0$$

By constructing the LMAD for statement S2 from the inner loop to the outer, according to the algorithm in Figure 5.5, the first dimension computed would be the stride-100 dimension. The second dimension computed would have a stride of 1, but it is placed to the left of the stride-100 dimension in the list of dimensions, giving an LMAD which is identical to the one for statement S1.

Without this sorting of the dimensions, attempting to match the dimensions between two LMADs would be an n^2 process since each dimension of one would potentially have to be compared against all dimensions of the other. When the sorting is precise, the matching needs only linear time, since the scan of one LMAD for the match of a particular dimension of the other LMAD can stop when it can be determined that the scan has gone past the stride's proper sorted position, and the scan for the next dimension can start where the last one left off.

5.5 The Accuracy of the LMAD Form

The number of access dimensions which triplet notation can represent accurately is fixed by the declared dimensions of the array. If the array is declared with one dimension, then the maximum number of access dimensions which are accurately representable is one. Since the LMAD form is not related to the declared dimensionality of the array, it has the potential to be perfectly accurate in all five categories of accesses.

In this section, it is shown how the LMAD represents each of the five categories of references where triplet notation is not accurate.

5.5.1 Multiple-index Affine

In the multiple-index affine case, there are more access dimensions than there are declared dimensions of the array. Referring to the code included below, the LMAD itself for this type of access looks no different than it would for an array declared with two dimensions, the left-most dimension having 10 elements.

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

$$\mathcal{A}^{1,10}_{4,10(N-1)} + 10$$

5.5.2 Coupled Subscripts

In the coupled-subscripts case, there are sometimes fewer access dimensions than there are declared dimensions of the array. The code included below is one such case.

```
real A(N,N), B(N)
do I = 1, N
   A( I,I ) =
end do
```

$$\mathcal{A}^{N+1}_{(N+1)(N-1)} + 0$$

5.5.3 Triangular Affine

The clear indicator that there is a triangular loop involved with an access is that a dimension-index occurs in the span of some other dimension. In the code included below, the span of $N(N-I)$ in the J-dimension correctly reflects the fact that the span of accesses decreases as I increases.

```
real A(N,N)
do I = 1, N
   do J = i, N
      A( I,J  ) =  . . .
   end do
end do
```

$$\mathcal{A}^{N_{[J'\leq N-(I'+1)]},N+1_{[I'\leq N-1]}}_{N(N-(I'+1)),(N-1)(N+1)} + 0$$

5.5.4 Non-affine

The clear indicator for a non-affine subscript expression is the appearance of a dimension-index in the stride of its own dimension. For the code included below, this correctly reflects the reality that each step through memory gets longer as I steps through its values. The correct interpretation of the stride expression is that the stride expression indicates how far to stride from the current location to the next location. So, referring to the code below, when $I = 3$, the current memory location is `A(9)` and the distance to travel to get to the next location is determined by the stride $(2I'+1)$ with 3 substituted for I', giving 7, which correctly determines that the next location is `A(9+7)` or `A(16)`.

```
do I = 0, N
   A( I**2 ) =  . . .
end do
```

$$\mathcal{A}_{N^2}^{2I'+1_{[I'\leq N]}} + 0$$

5.5.5 Subscripted-subscript

When a subscripted-subscript is involved, the stride, span, and base offset all will contain references to the subscripting array, and the loop index. In the loop shown below, it can be seen that the correct distance from the current location to the next location (the stride) is simply $C(I+1) - C(I)$. This is true regardless of the monotonicity of the values in the array `C`. The total extent of the movement (the span), however, is $C(N) - C(0)$ only if the array `C` contains monotonically increasing or decreasing values. This information may or may not be derivable from the program text.

```
real C(0:N), B(M)
do I = 0, N
   B( C(I) ) = . . .
end do
```

$$\mathcal{B}_{C(N)-C(0)}^{C(I'+1)-C(I')_{[I'\leq N]}} + C(0) \qquad \{ \text{ span correct if } \texttt{C} \text{ is monotonic } \}$$

Despite the lack of knowledge about the contents of the array C, the LMAD representation is precise by virtue of the stride expression combined with the dimension-index.

Some common subscripted-subscript situations yield to analysis with the LMAD form. In the following, an array B contains the compressed form of a two-dimensional array. The array P is loaded with the starting points of successive rows.

```
do I=1,N
   do J=P(I),P(I+1)-1
      B(J) = . . .
   end do
end do
```

The LMAD form of this access is

$$\mathcal{B}^{1_{[J' \le P(I'+2)-P(I'+1)-1]}, P(I'+2)-P(I'+1)_{[I' \le N-1]}}_{P(I'+2)-1-P(I'+1), P(N)-P(1)} + P(1) - 1$$

which is simplifiable, as will be shown in Section 5.7.3. But the array P must hold monotonically increasing or decreasing values for the resulting LMAD to be valid. Another commonly used pattern, in which a starting point array (START) holds the start of a row and a length array (LEN) holds the length of a row, is as follows:

```
do I=1,N
   do J=0, LEN(I)-1
      B( START(J) + I ) =
   end do
end do
```

The LMAD representation for this is as follows:

$$\mathcal{B}^{1_{[J' \le LEN(I'+1)-1]}, START(I'+2)-START(I'+1)_{[I' \le N-1]}}_{LEN(I'+1)-1, START(N)-START(1)} + START(1) - 1$$

This leads to a natural test for dependence, which is derivable from the definitions of stride and span: LEN(I) $\le$ START(I + 1) $-$ START(I).

5.6 Translating LMADs Across Procedure Boundaries

One of the principle benefits of using LMADs to represent memory accesses is that it is simple to translate them across procedure boundaries, even in the presence of array reshaping.

The array reshaping problem occurs when an array is used as a formal parameter for a subroutine, an array reference is used as an actual parameter, and the declarations for the formal and actual arrays differ in number of dimensions or number of elements of a single dimension.

This causes problems for many compilers when they attempt to translate information about the array across the procedure boundary, because there is sometimes no way to express information that is derived based on one declaration in a form based on the other.

LMADs solve this problem because they are not constrained in any way by the declaration of the array. An access pattern summary for a formal parameter to a subroutine, represented as an LMAD, can be translated to the context of a calling routine by the following steps:

- translate the names of the symbols used in the subroutine into their names used in the calling routine. When no corresponding symbol exists in the calling routine, create a name which does not conflict with any name in the calling routine.
- copy the value range information for each symbol from the subroutine to the calling routine.
- add the base offset of the LMAD for the actual parameter to the base offset of the translated LMAD of the formal parameter.

For example, consider the code fragment in Figure 5.14. The LMAD for the access to the formal parameter `A` in the subroutine is $\mathcal{A}^{D_1,D_2,D_3} + s$ (where D_i stands for a stride/span pair, or dimension). Translating the parameter name and adding the offset produces the LMAD $\mathcal{X}^{D_1,D_2,D_3} + s + 1000(I-1)$, which is a perfectly accurate description of the access in the subroutine.

5.7 Simplification of LMADs

Two or more LMADs may be combined into a single LMAD and two dimensions of a single LMAD may be combined into a single dimension, through simplification operations. This

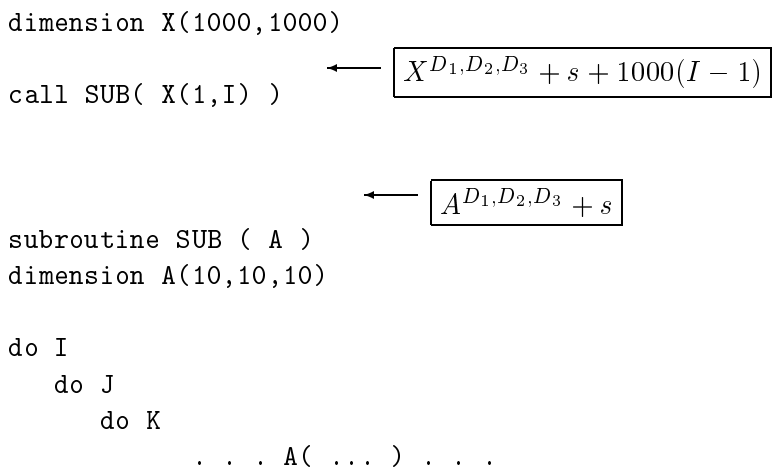

Figure 5.14: Translating LMADs across procedure boundaries.

simplification can translate into increased LMAD processing efficiency. Since some LMAD operations involve processing all possible pairs of LMADs (an n^2 process), reducing the number of LMADs can cause a major reduction in the execution time of those operations. Likewise, reducing the number of dimensions can reduce the execution time for the operations which involve all possible pairs of dimensions.

Simpler descriptors can also be operated on with simpler processing algorithms, which are more likely to avoid a loss of precision. Simpler processing in general means faster processing, as well. So, simplification can have many benefits.

The simplification operations which will be discussed here are *contiguous aggregation*, *coalescing*, and *interleaving*.

5.7.1 Checking for Internal Overlap

Whenever a new access descriptor is created, whether from expanding a descriptor to an outer loop, aggregating two accesses, or coalescing a single access, a "no-overlap test" must be used to determine whether an internal overlap occurs due to *that* operation. Whether earlier operations caused overlap or not is immaterial. For instance, consider the loop nest:

```
real A(N,L)
do I=1,N
   do J=1,M
     do K=1,10
       A(I,K) =
     end do
   end do
end do
```

The access descriptor for array A due to the K loop is $\mathcal{A}^{N}_{9:N}+I'$. The access descriptor for A expanded for the J loop is also $\mathcal{A}^{N}_{9:N}+I'$. There is definitely an overlap due to the J loop, since the same parts of A are accessed on each iteration of the J loop. However, when summarizing that descriptor to the outer I loop, it gets the no-overlap characteristic since different values of I cause no duplication in the access sequence. The resulting access descriptor would be $\mathcal{A}^{1,N}_{N-1,9N}+0$.

5.7.2 Contiguous Aggregation

An operation which combines two LMADs into one is *contiguous aggregation*. Contiguous aggregation is an operation which may be performed on two access descriptors which have a nearly identical pattern, but whose offsets make it possible to combine a dimension from each descriptor into a single contiguous dimension. Two such descriptors are **conjunctive**.

A formal definition of **contiguous** LMADs is as follows:

Definition 9 [Contiguous LMADs] *Two LMADs* $\mathbf{A}$ *and* $\mathbf{A}'$ *are* **contiguous**, *denoted by* $\mathbf{A} \bowtie \mathbf{A}'$, *if* $\mathbf{A} = \mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma_1,\sigma_2,\cdots,\sigma_d} + \tau$ *and* $\mathbf{A}' = \mathcal{A}^{\delta'_1,\delta'_2,\cdots,\delta'_d}_{\sigma'_1,\sigma'_2,\cdots,\sigma'_d} + \tau'$ *(with* $\tau \geq \tau'$*), and for* **selected dimensions** p *from* $\mathbf{A}$ *and* q *from* $\mathbf{A}'$, *there exist stride/span pairs* (σ_p, δ_p) *and* (σ'_q, δ'_q), *satisfying the conditions*

1. $\mathcal{A}^{\cdots,\delta_{p-1},\sigma_{p+1},\cdots,\delta_d}_{\cdots,\sigma_{p-1},\sigma_{p+1},\cdots,\sigma_d}+\tau \;//\; \mathcal{A}^{\cdots,\delta'_{q-1},\sigma'_{q+1},\cdots,\delta'_d}_{\cdots,\sigma'_{q-1},\sigma'_{q+1},\cdots,\sigma'_d}+\tau'$,
2. $\delta_p = \delta'_q$,
3. δ_p *divides* $\tau - \tau'$, *and*
4. $\tau - \tau' \leq \sigma'_q + \delta'_q$

The **aggregation operation** is defined as follows:

Definition 10 [Contiguous Aggregation Operation] *Let* $\mathbf{A}$ *and* $\mathbf{A}'$ *be the contiguous descriptors defined in Definition 9, and let* $t = \tau - \tau'$. *Then,*

$$\mathbf{A} \cup \mathbf{A}' = \begin{cases} \mathcal{A}^{\cdots,\delta_p,\cdots,\delta_d}_{\cdots,\sigma_p+t,\cdots,\sigma_d} + \tau' & \text{for } \sigma_p + t > \sigma'_q \\ \mathbf{A}' & \text{otherwise} \end{cases}$$

As an example of the aggregation operation, consider the following loop nest:

```
real A(100)

do I=1,N,2
   do J=1,3
      A(3*I+J) = . . .
   end do
   do J=4,6
      A(3*I+J) = . . .
   end do
end do
```

The two writes to the array `A` have access descriptors

$$\mathcal{A}^{1,6}_{2,6(N-1)}+4 \text{ and } \mathcal{A}^{1,6}_{2,6(N-1)}+7$$

These access descriptors can be shown to be conjunctive, according to Definition 9 (all strides and spans match; $\delta_{i_p} = \delta'_{i_q} = 1$; 1 divides $7-4$; and $3 \leq 2+1$) , and the aggregation operation produces the following access region descriptor:

$$\mathcal{A}^{1,6}_{2,6(N-1)}+4 \cup \mathcal{A}^{1,6}_{2,6(N-1)}+7 \Rightarrow \mathcal{A}^{1,6}_{5,6(N-1)}+4$$

Definition 11 [Contiguous Aggregation Overlap Condition] *When two regions are determined to be* **conjunctive**, *as described in Definition 9, then the resulting descriptor can be marked with the no-overlap characteristic whenever*

$$\tau - \tau' = \sigma'_{i_q} + \delta'_{i_q} \qquad \textbf{[No overlap test - aggregation]}$$

5.7.3 Coalescing

An array access is called *coalesceable* when it moves with a small stride due to one index and with a larger stride over the accesses of the first stride to the very next element in the sequence.

A formal definition of **coalesceable dimensions** follows:

Definition 12 [Coalesceable Dimensions] *Given the access descriptor:*

$$\mathcal{A}^{\cdots,\delta_j,\cdots,\delta_k,\cdots}_{\cdots,\sigma_j,\cdots,\sigma_k,\cdots} + \tau,$$

if the following conditions hold:

1. δ_j divides δ_k
2. $\sigma_j + \delta_j \geq \delta_k$

then the two dimensions j and k are **coalesceable**, *and the two dimensions can be combined into one by eliminating both dimensions and replacing them with a single dimension with a stride of δ_j and a span formed by evaluating the span due to j at the ending value for the index k, then adding that to the span due to k. The result is the following access descriptor:*

$$\mathcal{A}^{\cdots,\delta_j,\cdots}_{\cdots,\sigma_j[k\leftarrow U_k]+\sigma_k,\cdots} + \tau.$$

This condition holds in the above example, where

Algorithm Coalesce

Input 1: index set $\mathcal{I} = \{i_1, i_2 \cdots, i_d\}$
with constraints $L_k \leq i_k \leq U_k \ \forall k = 1, 2, \cdots, d$
Input 2: LMAD $\mathcal{A}^{\delta_{i_1}, \delta_{i_2}, \cdots, \delta_{i_d}}_{\sigma_{i_1}, \sigma_{i_2}, \cdots, \sigma_{i_d}} + \tau$
Algorithm:
For *all possible combinations of index pairs* (i_j, i_k)
selected from $\mathcal{I}$ do
Select two stride/span pairs $(\delta_{i_j}, \sigma_{i_j})$ and $(\delta_{i_k}, \sigma_{i_k})$
from the descriptor $\mathcal{A}$;
If δ_{i_j} *divides* δ_{i_k} and $\delta_{i_k} \leq \sigma_{i_j} + \delta_{i_j}$ then
$\sigma_{i_j} \leftarrow \sigma_{i_j} + \sigma_{i_k}$;
If σ_{i_j} *is an expression containing* i_k then
Replace i_k in σ_{i_j} with its upper bound U_k;
end
Eliminate $(\delta_{i_k}, \sigma_{i_k})$ from the descriptor $\mathcal{A}$;
$\mathcal{I} \leftarrow \mathcal{I} - \{i_k\}$;
end
end

Figure 5.15: An algorithm that detects coalesceable accesses from an LMAD, by comparing its stride/span pairs.

$$\mathcal{A}^{1,6}_{5,6(N-1)}+4 \quad \text{can be transformed to :} \quad \mathcal{A}^{1}_{6(N-1)+5}+4$$

An algorithm for performing coalescing is shown in Figure 5.15.

Definition 13 [Coalescing Overlap Condition] *When a region can be coalesced, according to Definition 12 above, then it may be further determined that it has the no-overlap characteristic when, for the two dimensions j and k being coalesced,*

$$\sigma_j + \delta_j = \delta_k \qquad \textbf{[No overlap test - coalescing]}.$$

5.7.4 Interleaving

When a set of LMADs has the same pattern and base offsets that differ by a consistent value, the LMADs form an interleaved access pattern. The entire set of LMADs may be combined into a single LMAD. The inverse operation is also possible - a single LMAD may always be broken into a set of interleaved LMADs. The formal definition of interleaving descriptors may be found in Definition 14.

Definition 14 [Interleaving Descriptors] *Given a set of n dimension-equivalent LMADs:* $\mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma_1,\sigma_2,\cdots,\sigma_d} + \tau_1$, $\mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma_1,\sigma_2,\cdots,\sigma_d} + \tau_2$, $\cdots$, $\mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma_1,\sigma_2,\cdots,\sigma_d} + \tau_n$, *if a consistent distance exists between consecutive pairs of LMADs:*

$$\tau_i - \tau_{i+1} = D \mid 1 \leq i \leq n-1$$

then the n LMADs may be combined into a single equivalent LMAD:

$$\mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d,D}_{\sigma_1,\sigma_2,\cdots,\sigma_d,nD} + \tau_1.$$

Definition 15 [n-Interleaved Descriptors] *Let* $\mathbf{A} = \mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma_1,\sigma_2,\cdots,\sigma_d} + \tau$. *For a stride/span pair* (δ_k,σ_k), $1 \leq k \leq d$ *in* $\mathbf{A}$, *where* δ_k *is invariant, and a chosen n,* $1 \leq n \leq (\sigma_k/\delta_k)+1$, *there always exists a set of n descriptors*

$$\{\mathbf{A}_j \mid 1 \leq j \leq n\} = \left\{ \mathcal{A}^{\delta_1,\cdots,\delta'_k,\cdots,\delta_d}_{\sigma_1,\cdots,\sigma'_k,\cdots,\sigma_d} + \tau + (j-1)\delta_k \right\}$$

where $\delta'_k = n\delta_k$ *and* $\sigma'_k = \left\lfloor \dfrac{\sigma_k - (j-1)\delta_k}{\delta'_k} \right\rfloor \delta'_k$, *such that* $\mathbf{A} = \bigcup_{1 \leq j \leq n} \mathbf{A}_j$. *The n descriptors are the n-***interleaved** *descriptors of* $\mathbf{A}$.

There is no way for overlap to occur when constructing the new LMADs described in either definition.

5.8 Components of the Access Region Descriptor

The additional information needed for parallelization with LMADs is

- flags: imprecise-sorting flag, overlap flag, monotonicity flag
- an execution predicate
- a correctness predicate
- a classification predicate
- pointers to the original locations of the array references summarized in a given descriptor
- size in bytes of one data element
- possible reduction flag, possible induction flag, reduction operator

5.8.1 The Flags

5.8.1.1 Imprecise-sorting Flag

As each LMAD dimension is computed, it is inserted into the list of dimensions. An attempt is made to place the new dimension in a list position where all the dimensions with lesser strides precede it and all the dimensions with greater strides follow it. If this is possible, operations which match the dimensions between two LMADs can run more quickly, as discussed in Section 5.4.2.1.

In many cases, this insertion sort can be done precisely, such as when the strides are all integers, or when variable range information can be used to prove a clear relationship between each pair of strides. However, other times the LMAD has strides which are either too complex to compare precisely, or use variables about which too little is known to make a meaningful comparison. When this happens, the **imprecise-sorting** flag is set. Since perfectly sorted dimensions are necessary for an internal overlap check, when the imprecise-sorting flag is set, it may be necessary to perform a run-time sort of the dimensions and a run-time overlap check.

5.8.1.2 Overlap Flag

When an LMAD is expanded for a dimension-index, created by the aggregation of other descriptors, or two dimensions are coalesced into one, the internal overlap check is made, as discussed

in Section 5.7. The **overlap flag** is set according to the outcome of that test. The values of the overlap flag can be *no overlap*, *overlap*, and *overlap unknown*. The *overlap unknown* value may cause the overlap condition to be tested at run-time.

5.8.1.3 Monotonicity Flag

The **monotonicity flag** may be set to indicate the monotonicity of the subscript expression. The subscript expressions must be monotonic with respect to all access dimensions for the descriptor to be considered monotonic. The monotonicity flag can have three possible values: *proven monotonic*, *proven non-monotonic*, and *unknown monotonic*. A descriptor whose monotonicity is *unknown monotonic* may cause a run-time monotonicity test to be carried out before optimizations based on it could be used.

5.8.2 The Execution Predicate

The ARD uses an execution predicate to represent the condition under which the access indicated by the LMAD occurs. This condition is extracted from the program text, as will be described in Section 6.1.8.2. It takes the form of a logical expression. If no execution predicate is attached to a descriptor, that is the same as if a `.TRUE.` expression were attached. It means that the memory access pattern described always happens in the given context.

5.8.3 The Correctness Predicate

The ARD needs a correctness predicate when there is some question as to the validity of the descriptor. The predicate expression gives the condition under which the descriptor is valid.

There are several reasons for using such a predicate. One would be the following case, in which the compiler cannot decide whether or not there is an overlap in the subscripting offset sequence.

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

It can be asserted that the descriptor

$$\mathcal{A}^{1,M}_{N-1,M(N-1)}+M+1$$

has no overlap, provided $M > N-1$. The expression $M > N-1$ could be used as the correctness predicate for the descriptor.

5.8.4 The Classification Predicate

As will be discussed in Section 6.1.7, the Memory Classification process sometimes must apply a predicate to an LMAD. When that happens, the predicate gets attached as the **classification predicate**. This expression can be used to disprove dependences and can be tested at run-time in some cases.

5.8.5 Original Reference Sites

The set of pointers to the original reference sites in the program, which produce the accesses described by the LMAD, are kept in the LMAD. These can be useful for the code generation pass of the compiler. For instance, if it is decided to privatize the access region of a descriptor, then the code generator could visit the original reference sites and generate code there to refer to the private version of the variable.

5.8.6 The Size of a Data Element

An ARD describes the access pattern in terms of which "data elements" are referred to, starting at a certain offset from an origin. These data elements must be of a constant size, given in this field of the descriptor. When operations are done between LMADs of two different sizes, the LMAD with the larger size must be converted to represent accesses to a data element with the smaller size.

5.8.7 Reduction/Induction Information

This data is used to assist the recognition of reductions and inductions, which will be described in Section 6.2.4. When the initial reduction pattern is found, the **reduction operator** field is

assigned with the operator which was found (addition, subtraction, or multiplication). When the operator is assigned, it is said that the reduction or induction has passed the level 1 check. The **possible reduction flag** or the **possible induction flag** can be set when the reduction or induction has passed the level 2 check.

5.9 ARD Notation

The format for writing an ARD uses the LMAD format, with extra notation attached. The three predicates: execution, correctness, and classification will be represented, plus the no-overlap characteristic and whether the descriptor represents a read-only, write-first, or read/write region in addition to the normal LMAD form. The form is as follows:

$$\begin{array}{l}[\{\text{predicate}\}_{exec}] \\ [\{\text{predicate}\}_{corr}] \\ [\{\text{predicate}\}_{class}]\end{array} \quad {}^{[r,w \text{ or } rw]}_{[\ \|\ ]}\mathcal{A}^{\delta_1,\cdots,\delta_d}_{\sigma_1,\cdots,\sigma_d}+\tau$$

All parts of the notation outside the realm of the LMAD are optionally written. The left-most parts are the three predicates, written in brackets ({}). The overlap flag position is at the lower left of the variable name. If two vertical bars (‖) are written there, it indicates that there is no internal overlap for the descriptor. At the upper left of the variable name, if an "r" is written there, it indicates that the descriptor is read-only. A "w" written there means write-first, and "rw" means read/write.

6 INTERPROCEDURAL PARALLELIZATION

An accurate memory access representation such as the ARD is of little use to compilers without a method to use it for optimizing programs. This chapter will describe a parallelization method whose accuracy depends on the accuracy of the intersection operation between memory access descriptors. The use of ARDs makes possible a very accurate intersection algorithm, so the parallelization is correspondingly accurate.

The approach is to summarize the memory activity of an arbitrary program section in such a way as to enable dependence analysis using only the summaries. If the summaries are translatable across subroutine boundaries, interprocedural parallelization is possible. Again, ARDs are crucial to this important capability, this time because they are easy to translate across procedure boundaries.

This chapter will describe a new method, called **Memory Classification Analysis** (MCA), for summarizing the memory activity in a section of code by classifying every memory location as to the type and order of accesses made to it by the code. First, the general method will be described, which can detect dependence between any two sections of code, based on the summaries. Then, a simplification of the general method will be shown to work for loop parallelization. Next, techniques to eliminate some of the dependences found will be discussed. Finally, a framework for using MCA to do interprocedural analysis will be described.

6.1 Memory Classification Analysis

The traditional notion of data dependence is based on classifying a memory location according to the type and order of accesses to it. This section extends this classification idea for the purpose of describing data dependence between arbitrary code sections, including loop iterations.

6.1.1 Traditional Data Dependence

Traditionally, the four types of data dependence are described by classifying two accesses to a single memory location, as in Table 6.1.

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

Table 6.1: Traditional data dependence definition.

Input dependences can safely be 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 be removed by transforming the original code through techniques such as induction variable analysis and reduction analysis.

6.1.2 Dependence Granularity

Data dependence is usually described as being between individual memory reference points in a program, but if the desire is to determine dependence between two larger sections of code, each of which contains several accesses to the memory location, the notion of dependence can be coarsened. Dependence between two reference sites within one of the code sections can be ignored, since a single processor will be executing it, automatically enforcing whatever dependence exists within the code section.

Definition 16 *A section of code within which the compiler can ignore dependences (since it will always be executed by a single processor) 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.

6.1.3 Dependence Summarization for Dependence Grain Parallelization

For medium- and coarse-grain parallelization, there can be many accesses to a single memory location in each grain. Instead of keeping track of the dependences between all possible pairs of references which have a reference site in each dependence grain, it is desired to represent them all with a single *representative dependence*.

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

Table 6.2: One possible representative dependence definition.

Definition 17 *A* **representative dependence** *is a single dependence between two dependence grains which represents all the dependences between the two grains due to individual reference sites, for a single memory location.*

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 dependences which have reference sites in each dependence grain, neither should a representative dependence exist. When one or more dependences exist between reference sites in the two grains, we must simply define ways to group them together such that an appropriate transformation may be used to deal with them as a group. For instance, if some combinations of dependences could all be removed by privatization, we can make a single memory-related representative dependence to represent them all.

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 to classify them as to dependence, according to Table 6.2.

When an input dependence exists between dependence grains, it can be ignored. When an anti dependence exists between dependence 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. Anti- and output dependences cause the grains to be serialized unless the first access in the later grain is a write. If it is, the dependence can be ignored as long as we privatize the variable involved. So, some anti and output dependences can be removed by privatization, while others cannot.

When a flow, or true dependence exists between grains, in general the grains must be serialized. However, the compiler can still parallelize in some cases. Certain patterns of memory accesses which result in flow dependences can be removed by a parallelizing compiler as long as

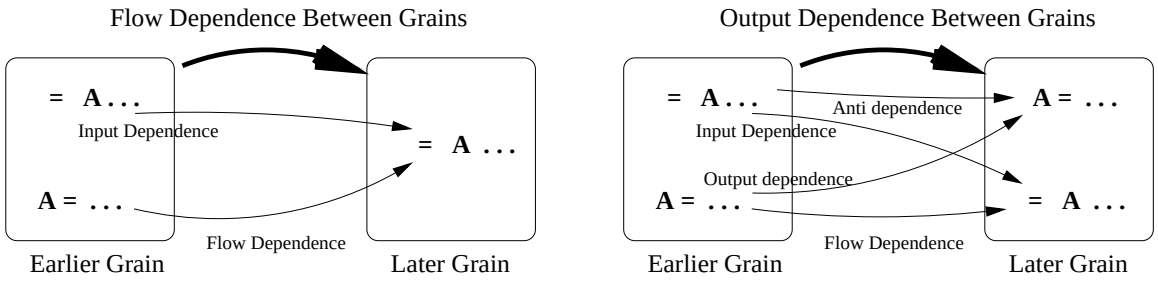

Figure 6.1: Dependence between grains depends on whether the two grains are read-only. The situation on the right shows a case where `A` can be privatized in the later grain, eliminating the output dependence.

it transforms the code in certain ways. Reductions and inductions, described in Chapter 2 and described in terms of the proposed compiler framework, in Sections 6.2.4 and 6.2.5, fall into this category.

Figure 6.1 illustrates dependence summarization.

6.1.4 Loop-based Dependence Analysis

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

There are no longer four cases, just two. The iteration is either read-only or it is not. However, 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 and one where a read is the first access. This gives three overall classes.

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 classified as **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

Access Type	Read-Only	Read-first/Write	Write-First
Dependence Type:	Input	Flow	Memory-related

Table 6.3: Loop-based representative dependence table.

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. This naturally leads to the loop-based dependence definition of Table 6.3, which contains fewer classes than the more general one of Section 6.1.3.

The name "Read-first/Write" will be simplified to Read/Write. Notice that Read/Write is the only case which prevents parallelization, and even in that case, reduction and induction patterns may still allow parallelization.

6.1.5 A More Effective Classification for General Dependence

The simplicity of the loop-based method described in the last section points to a classification scheme which is more effective than the one presented in Section 6.1.3 for representative dependences for the general dependence grain problem. Since the classification in Section 6.1.3 has the problem that only some memory-related dependences are removable by privatization (those which are written first in the later grain), it makes sense to use that fact in the classification.

If we use the classifications Read-Only, Write-First, and Read/Write, 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 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 Figure 6.2.

6.1.6 Establishing an Order Among Accesses

The crucial information needed for this kind of dependence analysis is the type of access done first to a memory location, and whether a write comes later. This information can be gathered by establishing an ordering of the accesses within the program. If a program contained only straight-line code without variables, establishing an ordering between accesses would be trivial. One could simply sweep through the program in "execution-order", keeping track of when the

	Read-Only later	Write-First later	Read/Write later
Read-Only earlier	Input	Memory-related	Flow
Write-First earlier	Flow	Memory-related	Flow
Read/Write earlier	Flow	Memory-related	Flow

Figure 6.2: A more effective way to classify dependence between two arbitrary dependence grains, using the classifications Read-Only, Write-First and Read/Write.

accesses happen. But branching statements and unknown variables make it more difficult to show that one particular access happens before another.

Take the following loop, for example:

```
for I=1 to N do {
    if (P) {
       A(I) = ...
    }
    if (Q) {
       ... = A(I) + ...
    }
}
```

Does the write happen before the read? It does if both `P` and `Q` are true. But if `Q` is true and `P` is false, 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**.

6.1.7 Memory Classification for Arrays

The techniques described above could be made to work for scalar variables by simply keeping track of the accesses to each location separately. However, the use of arrays in programs makes the problem more difficult. The use of symbolic expressions to index the arrays sometimes makes it impossible to know which array elements are being referred to, and a single array reference may refer to different elements at different times.

In order to make the correspondence between a memory location and a classification, in the presence of arrays, the focus of the analysis must simply be shifted. Instead of marking a class on a memory location, a memory location can be added to a class. So, instead of using a data structure for each memory location, a data structure may be used for each class. A symbolic description of the sequence of memory locations which are accessed in a particular way can be added to the appropriate class. Fortunately, the ARD provides an excellent mechanism for representing a sequence of memory locations.

So, one *summary set* is needed for each classification. According to Table 6.3, three summary sets are needed: Read-only (RO), Read/Write (RW), and Write-first (WF). In each summary set, a symbolic representation of the memory locations which are accessed in that way, derived from the program text, is kept.

A symbolic representation of the read and write accesses in a given statement can usually be easily determined. For instance, in the following Fortran assignment statement `S0`, for a single iteration of the surrounding loop, the variable `I` and the locations `B(I)` and `C(I)` are read-only, and the location `A(I+2)` is write-first.

```
                do I = 1, N
S0:                A(I+2) = B(I) * C(I)
                end do
```

The array references could be *expanded* by the loop index `I` to determine a symbolic representation of the locations being accessed in each way during the execution of the loop. For instance, during the execution of all iterations of the loop, the locations `B[1:N]` and `C[1:N]` are accessed in a read-only way, while the locations `A[3:N+2]` are accessed in a write-first way.

Expansion by the loop index would be sufficient for classifying memory accesses if there were no chance of cross-iteration dependences between reference sites. But since in most languages there is such a chance, memory classification within a loop must include a way to discover the

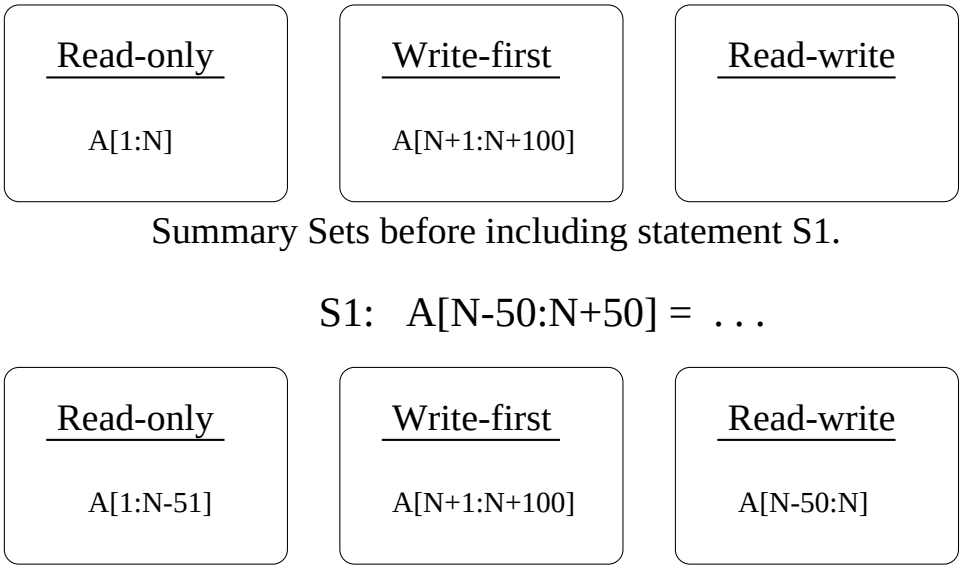

Figure 6.3: Distributing an access to the summary sets.

memory locations accessed in common by separate reference sites. This is done by intersecting the sets of locations accessed by the separate sites.

A symbolic intersection operation may be performed precisely between such sets in many cases, since quite often arrays are indexed with related variables.

For example, consider the situation shown in Figure 6.3. Let us assume that the summary sets have already recorded a Read-Only region of `A[1:N]` and a Write-First region of `A[N+1:N+100]`. The access to be classified is the write region `A[N-50:N+50]`, which will be called the "new write region". The region in common between the Read-Only set and the new write region must be placed in the Read/Write set, since those locations have a read followed by a write. The new write region must be reduced by the section which has been newly-classified as Read/Write, then what remains of the new write region must be intersected with the Write-First summary set. The region `A[N+1:N+50]` falls totally within the existing Write-First region, so no change is necessary for that set.

In general, the intersection algorithm must be capable of removing the intersecting region from the summary sets and the access to be classified, separating out the intersected region for moving it to a third summary set. An example of this is shown in Figure 6.4.

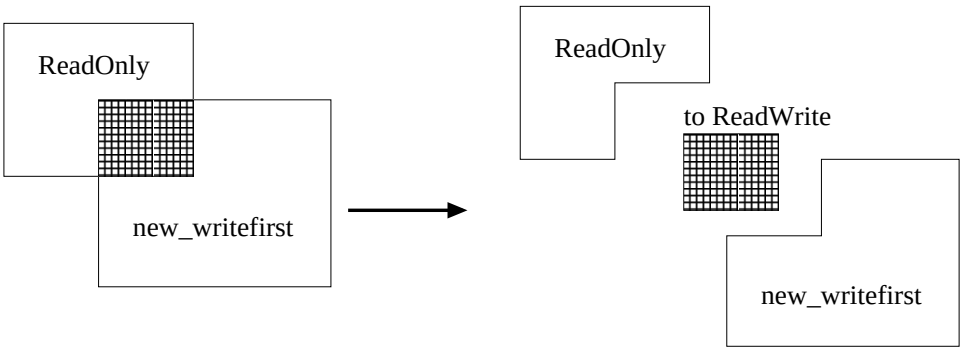

Figure 6.4: The general form of summary set intersection.

$$
\begin{aligned}
\mathrm{RO}_{new} &= \mathrm{RO}_{new} - \mathrm{RO} \\
\mathrm{RO}_{new} &= \mathrm{RO}_{new} - \mathrm{WF} \\
\mathrm{RO}_{new} &= \mathrm{RO}_{new} - \mathrm{RW} \\
\mathrm{RO} &= \mathrm{RO} \cup \mathrm{RO}_{new}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{WF}_{new} &= \mathrm{WF}_{new} - \mathrm{WF} \\
\mathrm{WF}_{new} &= \mathrm{WF}_{new} - \mathrm{RW} \\
\mathrm{RW} &= \mathrm{RW} \cup (\mathrm{WF}_{new} \cap \mathrm{RO}) \\
\mathrm{RO} &= \mathrm{RO} - (\mathrm{WF}_{new} \cap \mathrm{RO}) \\
\mathrm{WF}_{new} &= \mathrm{WF}_{new} - (\mathrm{WF}_{new} \cap \mathrm{RO}) \\
\mathrm{WF} &= \mathrm{WF} \cup \mathrm{WF}_{new}
\end{aligned}
\qquad
\begin{aligned}
\mathrm{RW}_{new} &= \mathrm{RW}_{new} - \mathrm{WF} \\
\mathrm{RW}_{new} &= \mathrm{RW}_{new} - \mathrm{RW} \\
\mathrm{RW} &= \mathrm{RW} \cup (\mathrm{RW}_{new} \cap \mathrm{RO}) \\
\mathrm{RO} &= \mathrm{RO} - (\mathrm{RW}_{new} \cap \mathrm{RO}) \\
\mathrm{RW}_{new} &= \mathrm{RW}_{new} - (\mathrm{RW}_{new} \cap \mathrm{RO}) \\
\mathrm{RW} &= \mathrm{RW} \cup \mathrm{RW}_{new}
\end{aligned}
$$

Figure 6.5: Classification of new summary sets RO_{new}, WF_{new}, and RW_{new} into the existing summary sets RO, WF, and RW.

In general, a read access to be classified must be reduced by its intersection with RO, WF, and RW, then the remaining locations added to RO. A write access to be classified must be reduced by its intersection with WF and RW, but the region which intersects with RO is removed from both RO and the write access, and moved to RW. The locations remaining in the write access must be added to WF. A read/write access to be classified is handled exactly like a write region to be classified, except that whatever remains in the read/write access after intersecting with RO, WF, and RW is moved to RW.

This process of distributing new locations into the summary sets is called **classification**. Classification takes as input the three existing summary sets: RO, WF, and RW, the three summary sets to be distributed: RO_{new}, WF_{new}, and RW_{new} and produces new versions of the three summary sets RO, WF, and RW. The steps of classification, as described above, can be expressed in set notation as shown in Figure 6.5.

6.1.8 Classification Operations for the Elementary Contexts

To make whole-program classification possible in a systematic way, a program is assumed to be a series of nested contexts, as shown in Figure 6.6. If the language does not force this through its structure, then a program will have to be transformed into that form through a *regularization* process before classification takes place.

The elementary contexts, whose nesting will make up a whole program, are *simple statements*, `if` *statements*, *loops*, `call` *statements*, and *procedures*. The memory accesses for each of these contexts can be classified into the three summary sets. This section will describe the classification process in each of the elementary contexts.

6.1.8.1 Classification for Simple Statements

Simple statements are those whose read and write accesses can be determined by examining the text of a single statement. An assignment statement without function calls is an example of this type of statement. The read accesses form the RO_{new} set. The write accesses form the WF_{new} set. If any accesses are read first then written, they form the RW_{new} set.

6.1.8.2 Classification for `if` Statements

An `if` statement is a compound statement containing a conditional expression and two alternative blocks of statements. The `if` chooses one of the two successor blocks of statements to execute next, based on the evaluation of the conditional expression. If the conditional expression evaluates to `true`, the `then` block executes, and if it evaluates to `false`, the `else` block executes. After execution of either block, execution continues at the statement following the compound statement. The point at which execution proceeds after the `if` statement is a *join point* in the control flow graph.

To summarize an `if` statement, an execution predicate is applied to the access descriptors produced by both blocks, and the union of the descriptors for both successor blocks is computed. The `if` condition is applied to the representation of the accesses for the `then` block and the negation of the `if` condition is applied to the representation of the accesses for the `else` block.

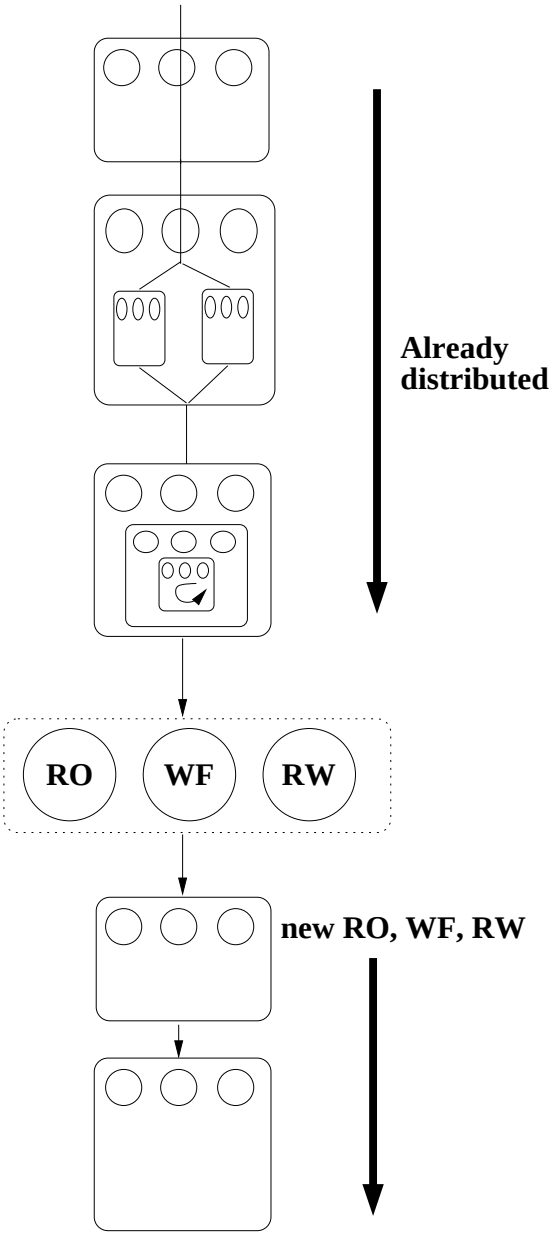

Figure 6.6: Memory Classification in a series of nested contexts.

It is possible to simplify the access representations after they are computed for an `if` statement by intersecting the descriptors and applying the **or** of the execution predicates to whatever is in the intersection. An example of this simplification is the following:

```
if (P) {
   A[1:N] =  . . .
   B[1:N] =  . . .
} else {
   A[1:N] =  . . .
}
```

The summary for this statement would include a WF set containing {P}A[1:N] and {P}B[1:N], (where P is the execution predicate), and {.not.P}A[1:N]. Intersecting these descriptors (and **or**ing the predicates) produces the {P or .not.P}A[1:N] and {P}B[1:N] in WF, which becomes simply A[1:N] and {P}B[1:N]. The removal of the execution predicate for A[1:N] reflects the fact that regardless of the value of the conditional expression, A[1:N] is written.

6.1.8.3 Classification for Loops

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, producing the three summary sets RO, WF, and RW. These summary sets represent the memory accesses for a single loop iteration, and are theoretically disjoint. They contain the symbolic form of the accesses, possibly parameterized by the index of the loop. Next, the summary sets must be *expanded* according to the loop index (as described in Section 5.2.1.2) so that the sets represent the locations accessed during the entire execution of the loop.

6.1.8.4 Classification for Procedures

Classification for a whole procedure is simply a classification sweep of the statements in the procedure in execution order. The resulting summary sets (RO, WF, and RW) are stored as a representation of the memory activity of the whole procedure.

6.1.8.5 Classification for `call` Statements

Classification for a `call` statement involves first the calculation of the access representation for the text of the `call` statement itself, then the retrieval of the summary sets for the procedure being called, matching formal parameters with actual parameters, and translating the summary sets involved from the called context to the calling context. If the symbols have further information which was derived during analysis of the subroutine (such as value range information), it must also be translated to the calling context. Section 5.6 describes how to translate ARDs across subroutine boundaries.

6.1.9 General Dependence Testing with Summary Sets

Given the symbolic summary sets RO_1, WF_1, and RW_1, representing the memory accesses for an earlier 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 6.2.

The intersections which must be done for each variable are:

$$\mathrm{RO}_1 \cap \mathrm{WF}_2$$
$$\mathrm{WF}_1 \cap \mathrm{WF}_2$$
$$\mathrm{RW}_1 \cap \mathrm{WF}_2$$
$$\mathrm{RO}_1 \cap \mathrm{RW}_2$$
$$\mathrm{WF}_1 \cap \mathrm{RO}_2$$
$$\mathrm{WF}_1 \cap \mathrm{RW}_2$$
$$\mathrm{RW}_1 \cap \mathrm{RO}_2$$
$$\mathrm{RW}_1 \cap \mathrm{RW}_2$$

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 first three intersections are non-empty, then the locations involved could be privatized to eliminate the memory-related dependence. If any of the last five intersections are non-empty, then a flow dependence exists between the grains.

For instance, a location written at least once in the earlier grain, and only read in the later grain would end up in one of the intersections $\mathrm{RW}_{earlier} \cap \mathrm{RO}_{later}$ or the intersection $\mathrm{WF}_{earlier} \cap \mathrm{RO}_{later}$, indicating a flow dependence between the grains.

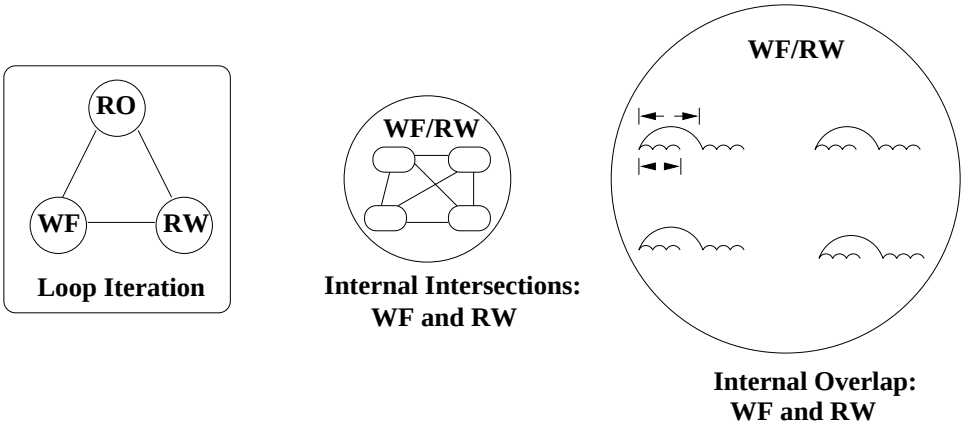

Figure 6.7: The Access Region Test.

6.1.10 Loop Dependence Testing with Summary Sets: The Access Region Test

As stated before, dependence testing between loop iterations is a special case of general dependence testing, described in the last section. Traditional loop parallelization considers each iteration to be one dependence grain, meaning all dependence grains have the same summary sets.

In this case, dependence between the grains (cross-iteration dependence) can occur in three ways: within one of the expanded descriptors, between two of the summary sets, or between two descriptors within one summary set. These three dependence checks make up the **Access Region Test** (ART), shown in Figure 6.7.

6.1.10.1 Overlap Within One Access Descriptor

The process of expanding a memory access descriptor by a loop index can cause an overlap within the descriptor to occur, as described in Section 5.3.2. This is a cross-iteration dependence because it appears as a result of the expansion operation, which models the access pattern caused by running all iterations of the loop.

6.1.10.2 Intersection of Two Summary Sets

There are only three summary sets to consider in loop dependence testing, instead of six, so there are only three intersections to try, instead of the eight required in general dependence testing. Since the disjoint summary sets before expansion represent the memory activity of one iteration, a non-empty intersection between two of the summary sets would indicate a cross-iteration dependence. Such a dependence involves at least one write operation, so none is an input dependence. The following intersections must be done:

$$\mathrm{RO} \cap \mathrm{WF}$$
$$\mathrm{RO} \cap \mathrm{RW}$$
$$\mathrm{WF} \cap \mathrm{RW}$$

6.1.10.3 Intersection of Descriptors Within One Summary Set

If two descriptors being put in the same summary set do not intersect initially, that doesn't mean that they will not intersect after they are expanded according to the loop index. Such an intersection would represent a cross-iteration dependence.

Intersection within the RO set would be a cross-iteration input dependence, which can be ignored, so there is no need to do the internal intersections within RO. Internal intersections for both WF and RW must be done, however.

An example is the following code:

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

When the two writes to array `A` are first assigned to a summary set, they do not overlap. The two write-first descriptors are $\mathcal{A}_0^0 + I - 1$ 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 both be assigned to the WF set. After expansion for I, (and creation of the dimension index I'), the normalized descriptors both become $\mathcal{A}_3^1 + 0$, which do intersect. This intersection would be found by attempting to intersect the descriptors within WF.

6.1.11 Loop-carried Dependence Handled by the Access Region Test

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 LMADs are represented as a single LMAD and moved to the ReadWrite 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 LMAD, no longer will be found to intersect. Intersections due to outer loops will be solely due to expansions for outer loop indices. This process is illustrated in Figure 6.8.

6.1.12 Uncertainty in the Classification Process

Sometimes it is not possible to unconditionally classify memory accesses. This happens when variables are used about which we have too little information, making it impossible to know how to classify a given access. An example is as follows:

```
do I = 1, N
   A(Z,I) = A(2,I) . . .
   A(1,I) = A(Q,I) . . .
end do
```

In this case, the `A(2,I)` reference gets classified as RO. Then the `A(Z,I)` reference must be classified. If nothing is known about the value of `Z`, it is likewise unknown whether it is equal to 2. If it were equal to 2, then the `A(2,I)` reference would be moved to the RW set and the `A(Z,I)` reference could be eliminated since it is represented by `A(2,I)`. However, if `Z` is not 2, then `A(2,I)` remains in RO and `A(Z,I)` would be classified WF.

This shows the need to do conditional classification. This involves determining all the possible classifications which could result from the attempt to classify a given reference, and attaching the condition under which each was valid. This condition is called the *classification condition*.

For instance, in the example above, the classification of `A(Z,I)` would result in `A(2,I)` being placed in the RW set with a classification condition of $Z \equiv 2$, and the assignment of the classification condition of $Z \neq 2$ to the reference `A(2,I)` in the RO set and to the `A(Z,I)`

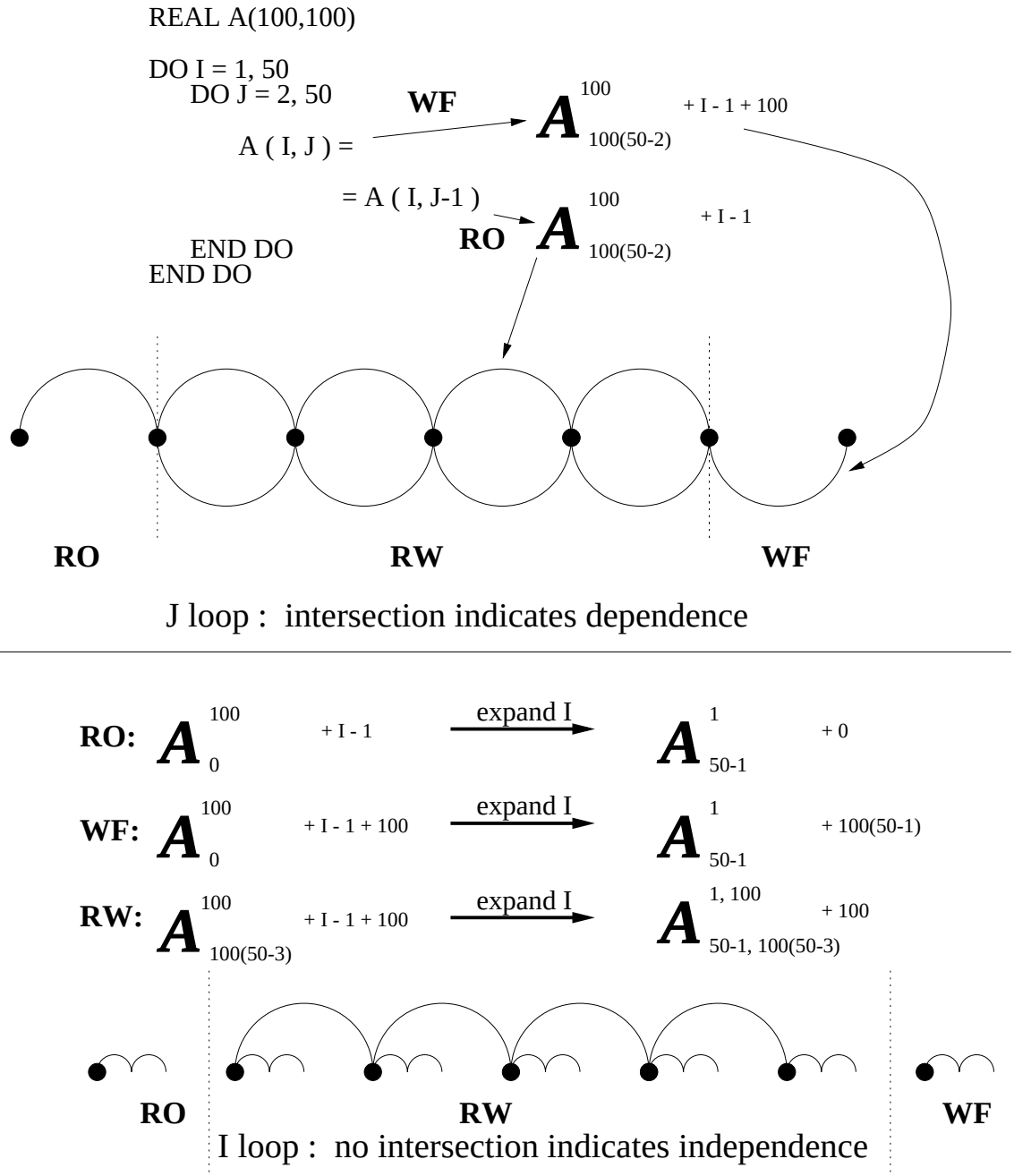

Figure 6.8: How the Access Region Test handles loop-carried dependence.

ReadOnly:	$\{Z \neq 2\}$`A(2,I)`	$\{Q \neq Z \wedge Q \neq 2 \wedge Q \neq 1\}$`A(Q,I)`
WriteFirst:	$\{Z \neq 2\}$`A(Z,I)`	$\{Z \neq 1 \wedge Q \neq 1\}$`A(1,I)`
ReadWrite:	$\{Z \equiv 2\}$`A(2,I)`	$\{Q \neq Z \wedge Q \neq 2 \wedge Q \equiv 1\}$`A(Q,I)`

Table 6.4: Summary sets illustrating conditional classification.

reference in the WF set. The summary sets which result from the classification of the above code can be seen in Table 6.4.

The intersection operation must take the classification conditions into consideration. One way is to form the **and** of the classification conditions of two ARDs when doing intersections. For instance, if the ReadOnly and the ReadWrite sets from Table 6.4 were intersected, even though the memory access part of $\{Z \neq 2\}$`A(2,I)` and $\{Z \equiv 2\}$`A(2,I)` intersects, the intersection should be reported as $\emptyset$ since $\{Z \neq 2\} \wedge \{Z \equiv 2\}$ is **false**. This corresponds to the fact that the two memory accesses can never be classified in those two sets at the same time.

Another way of taking the classification conditions into consideration is to use them to disprove a possible intersection. An example of this is the intersection of $\{Z \neq 2\}$`A(2,I)` with $\{Z \neq 2\}$`A(Z,I)`. In this case the intersection algorithm would be able to detect that an intersection would be non-empty only if $Z \equiv 2$, but since the **and** of the conditions results in $Z \neq 2$, that possibility is disproved, so the intersection should be returned as the empty set.

Using these classification conditions, it can be proved that no intersection occurs between the references in the above code. This is a case where a more traditional dependence test could easily determine a lack of dependence, by just noticing that the loop index I occurs by itself in one dimension of the array in every case, while MCA requires more extensive symbolic analysis.

When it is not possible to determine how to classify a given descriptor, and also not possible to determine a condition under which various possible classifications would be done, then the conservative action is to put all descriptors involved into the ReadWrite summary set. This will almost guarantee that the intersections within ReadWrite will be found non-empty, indicating a dependence.

6.1.13 Using the Classification Condition for Conditional Prefetching

One way to overlap computation with communication in a multiprocessor is to use an asynchronous prefetch command. This is typically used prior to a parallel loop, to fetch remote data which will be needed by a processor for executing its iterations of the loop. The algorithms that determine the data to prefetch suffer from the same imprecision as does the Classify algorithm. As discussed in Section 6.1.12, when unknown variables are used to index arrays, it sometimes cannot be determined whether or not the array reference accesses data previously accessed. If unnecessary prefetches are generated, machine time and resources are wasted.

The classification condition can be employed, in this situation, to generate conditional prefetches. For example, referring to the example in Section 6.1.12, it might be important to prefetch the array reference `A(Q,I)`. Through the classification conditions shown in Table 6.4, it can be seen that only under the condition $\{Q \neq Z \wedge Q \neq 2 \wedge Q \neq 1\}$ is `A(Q,I)` classified ReadOnly, and only under the condition $\{Q \neq Z \wedge Q \neq 2 \wedge Q \equiv 1\}$ is it classified ReadWrite. Together, these lead to the prefetch condition:

$$\{Q \neq Z \wedge Q \neq 2\}.$$

This is the condition under which it should be prefetched. In the other cases, it has either already been read (when $Q \equiv 2$), or already been written ($Q \equiv Z$).

6.1.14 Difficulty with a "Simple" Situation

Despite its success in "difficult" and complex situations, the Access Region Test can have trouble with "simple" situations without a little help. Take the following code for example:

```
real A(100,100)
do I = 1, Q
   do J = 2, 50
      A( I,J ) =
          = A( I,J-1 )
   end do
end do
```

There is obviously no dependence carried by the `do I` loop, since the loop index `I` appears by itself and in the same expression in both references to array `A` in the loop. Since the construction of the LMAD does not preserve the original form of the subscript expressions within the array references, this information can be lost. The difficulty arises when nothing is known about the values which the variable `Q` can take on. The Access Region Test would try to determine whether the variable `Q` is less than or equal to `100`. The Inbounds Assumption would guarantee that it is, but by losing the subscripting information, we may not be able to determine it. This would prevent parallelization of the loop.

This problem can be solved by saving the information that the loop index `I` is a **singleton index** at the time of LMAD creation. A singleton index is an index which never appears with other loop indices whenever it appears in a subscripting expression. This fact would guarantee that there can never be an overlap with respect to that index, allowing the intersection algorithm to prove there is no intersection due to the index.

6.2 Transformations and Analysis for Loop Parallelization

The classification operation has been designed to be closed in the sense that the outputs are of the same form as the inputs. This, combined with the defined dependence analysis procedure, makes it possible to produce a valid summary of the activity in an arbitrary-sized section of code which retains exactly the information needed for doing parallelization.

The basic idea is that a compiler would use a classification routine to classify the memory accesses in the main program. Whenever the classification routine encounters one of the contexts which are of interest, it recursively calls itself to classify that new context. The loop, the `if` statement, and the `call` statement are the contexts of interest, since MCA will be used to parallelize loops, do flow-sensitive analysis, and interprocedural analysis.

As part of the classification process, parallelization can be done. A classification context must correspond to the dependence grain which is to be parallelized, simply because the summary sets for those dependence grains are intersected in the parallelization method described in this thesis. The Classify algorithm, which is used as the basis for the interprocedural loop parallelization framework, is shown in Figure 6.9.

Algorithm Classify

```
Input:  A regularized program
Output: A loop-parallelized program
  Function classify( first_stmt, last_stmt, global_predicate, local_predicate):
    for ( stmt=first_stmt.next() to last_stmt) do
      switch(stmt.type() )
        case: ELSEIF:
          local_neg ← not local_predicate and local_neg;
          local_predicate ← elseif condition;
          continue;
        case: ELSE:
          local_neg ← not local_predicate and local_neg;
          local_predicate ← NULL;
          continue;
      endswitch;
      compute_access_rep ( stmt ); // Compute memory accesses for text of stmt

      if (stmt.type() == IF) {
        classify( stmt, endif-stmt,
          global_predicate and local_predicate and local_neg,
          if condition );
      }
      update_memory_sets(new_readonly, new_writefirst, new_readwrite,
        ReadOnly, ReadWrite, WriteFirst);
    endfor

    simplify_memsets( ); // employ ARD simplification ops

    if (stmt.type() == LOOP) {
      expand_by_loop ( ); // expand by loop index
      check_parallelization_loop ( );
    elseif (stmt.type() == IF) {
      summarize_IF_stmt ( );
    else
      summarize_to_procedure ( );
    }
  end Function classify
```

Figure 6.9: The classification algorithm of the unified parallelization framework.

Algorithm Classify support routines

```
update_memory_sets (new_readonly, new_writefirst, new_readwrite,
   ReadOnly, WriteFirst, ReadWrite)
   distribute_read( new_readonly, ReadOnly, ReadWrite, WriteFirst )
   distribute_write( new_writefirst, ReadOnly, ReadWrite, WriteFirst )
   distribute_readwrite( new_readwrite, ReadOnly, ReadWrite, WriteFirst )
end update_memory_sets

distribute_read (new_readonly, ReadOnly, WriteFirst, ReadWrite)
   forall  ARD pairs (new_readonly, WriteFirst) do
      if ( new_readonly.exec_pred() implies  WriteFirst.exec_pred() ) then
         discard  intersect(new_readonly,WriteFirst) from new_readonly
      else
         move  intersecting parts of new_readonly and WriteFirst to ReadWrite
      endif
   endforall
   forall  ARD pairs (new_readonly, ReadWrite) do
      if ( new_readonly ⊂ ReadWrite ) then
         discard  new_readonly
         if ( ReadWrite.possible_reduction() ) then
            ReadWrite.possible_reduction ← false
         endif
      endif
      if (  intersect(new_readonly, ReadWrite) ) then
         discard  intersect(new_readonly, ReadWrite) from new_readonly
         if ( either LMAD is reduction ) then
            make sure both are, otherwise cancel reductions
         endif
      endif
   endforall
   ReadOnly ← ReadOnly ∪ new_readonly
end distribute_read
```

Figure 6.10: Classify support routines.

Algorithm Classify support routines 2

```
distribute_write (new_writefirst, ReadOnly, WriteFirst, ReadWrite)
   forall  ARD pairs (new_writefirst, WriteFirst) do
         discard  intersect(new_writefirst,WriteFirst) from new_writefirst
   endforall
   forall  ARD pairs (new_writefirst, ReadWrite) do
      if ( new_writefirst.exec_pred() implies  ReadWrite.exec_pred() ) then
         if ( new_writefirst ⊂ ReadWrite ) then
            discard  new_writefirst
            if ( ReadWrite.possible_reduction() ) then
               ReadWrite.possible_reduction ← false
            endif
         endif
         if (  intersect(new_writefirst, ReadWrite) ) then
            discard  intersect(new_writefirst, ReadWrite) from new_writefirst
         endif
      endif
   endforall
   forall  ARD pairs (new_writefirst, ReadOnly) do
      if ( new_writefirst.exec_pred() implies  ReadOnly.exec_pred() ) then
         discard  intersect(new_writefirst,ReadOnly) from ReadOnly
         move  intersect(new_writefirst,ReadOnly) to ReadWrite
         discard  intersect(new_writefirst,ReadOnly) from new_writefirst
   endforall
   WriteFirst ← WriteFirst ∪ new_writefirst
end distribute_write
```

Figure 6.11: Classify support routines 2.

Algorithm Classify support routines 3

```
distribute_readwrite (new_readwrite, ReadOnly, WriteFirst, ReadWrite)
   forall  ARD pairs (new_readwrite, WriteFirst) do
      if ( new_readwrite.exec_pred() implies  WriteFirst.exec_pred() ) then
         if ( new_readwrite.possible_reduction() ) then
            new_readwrite.possible_reduction ← false
         endif
         discard  intersect(new_readwrite,WriteFirst) from new_readwrite
      endif
   endforall
   forall  ARD pairs (new_readwrite, ReadWrite) do
      if ( new_readwrite.exec_pred() implies  ReadWrite.exec_pred() ) then
         if ( new_readwrite ⊂ ReadWrite ) then
            discard  new_readwrite
            if ( not ReadWrite.possible_reduction() or
               not new_readwrite.possible_reduction() ) then
               ReadWrite.possible_reduction ← false
               new_readwrite.possible_reduction ← false
            endif
         endif
         if (  intersect(new_readwrite, ReadWrite) ) then
            discard  intersect(new_readwrite, ReadWrite) from new_readwrite
         endif
      endif
   endforall
   forall  ARD pairs (new_readwrite, ReadOnly) do
      if ( new_readwrite.exec_pred() implies  ReadOnly.exec_pred() ) then
         if ( new_readwrite.possible_reduction() ) then
            new_readwrite.possible_reduction ← false
         endif
         discard  intersect(new_readwrite,ReadOnly) from ReadOnly
         move  intersect(new_readwrite,ReadOnly) to ReadWrite
         discard  intersect(new_readwrite,ReadOnly) from new_readwrite
   endforall
   ReadWrite ← ReadWrite ∪ new_readwrite
end distribute_write
```

Figure 6.12: Classify support routines 3.

6.2.1 Regularizing the Program

The program must be preconditioned-conditioned or *regularized* by identifying loops and making all loops and procedures single-entry/single-exit, prior to using the Classify algorithm.

Identifying loops in the program may be done with *interval analysis* [1]. Many languages have explicit looping constructs, such as the `do` loop in Fortran, which makes identifying those loops easy. However, the existence of an `if` statement and a `goto` statement in a language makes it possible to program arbitrarily complex looping patterns, so these loops must be identified by interval analysis.

Interval analysis identifies the set of nested loops which represent the program structure. Interval analysis also identifies program patterns which cannot be represented as a nested structure – these programs have what is known as an *irreducible flow graph*. When this is the case, it is always possible to transform such a program into a program with a nested structure by a technique known as *node-splitting* [1]. Programs with an irreducible flow graph are believed to be very rare [1], so this is not expected to be a problem.

After interval analysis, the program is a collection of nested loops and `if` statements. All such loops and procedures must be made single-entry/single-exit, so that both forward and backward analysis will find a single point at which to look for summary sets representing accesses within those contexts.

6.2.2 Other Key Transformations

In addition to the regularization transformation, it can be useful to perform some other analysis/transformation passes prior to the classification process. The elimination of dead code is always useful in that it cuts down the amount of processing which must be done. Interprocedural value propagation [8] propagates expressions from definition points to use points interprocedurally. A conversion to Static Single Assignment (SSA) form [15] embeds the def-use chains for scalar variables in the names of the variables, which supports the symbolic analysis process by making clear whether two uses of the same variable hold the same value or may not. Range propagation [8] stores value ranges for scalar variables, to aid in symbolic analysis.

6.2.3 Privatization

As described in Section 2.2.1, privatization is the transformation by which dependences involving certain variables are removed, by making one copy of the variable per processor.

The conditions for recognizing privatizable memory locations are that the locations are written before being read in a single iteration of the loop, and that the location be accessed in more than one iteration of the loop (leading to an apparent cross-iteration dependence). These two conditions are discovered by MCA through the WriteFirst classification and the detection of a dependence through the Access Region Test.

The program situations requiring proof of logical implication, which Tu [38] proposed specifically for array privatization, are now handled as a natural part of MCA, to the benefit of all analysis.

The analysis necessary to determine whether last value assignment needs to be done is essentially **liveness analysis** [1]. It must be determined whether a privatized variable is *live* immediately after the loop.

6.2.4 Reduction Analysis

The general idea of reduction recognition is discussed in Section 2.2.2. Reduction recognition is easily incorporated into a structure involving MCA and the ART because each reference to a memory location involved in a reduction operation is classified ReadWrite, only appears in a loop in statements which have the "reduction pattern" (all must use the same reduction operator), and a dependence is detected for that memory location. The MCA handles the classification, and the ART handles the dependence detection. The only additional processing necessary consists of two additional checks:

Level 1 The reduction pattern must be recognized when the original LMAD is constructed from an assignment statement, and the reduction operator is stored.

Level 2 During the distribution of LMADs to the proper memory access classes (in MCA), each LMAD for a given symbol is intersected with each other for that symbol. At each intersection operation, a check can be made to ensure that no other reference to that symbol within the loop is of a non-reduction type, and all use the same reduction operator.

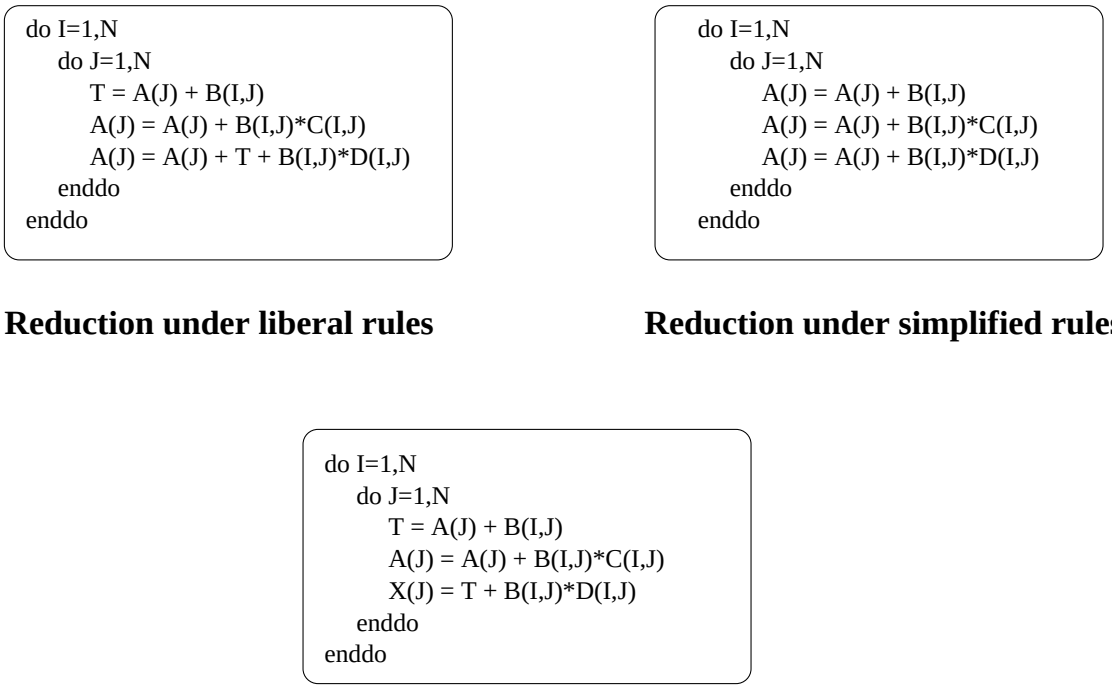

Figure 6.13: Comparing liberal and conservative rules for reduction recognition.

These conditions for recognizing a reduction are slightly more conservative than necessary, although many compilers use this approach. See Figure 6.13 for a comparison of these reduction situations.

There is no reason why the more liberal rules could not be used within the ART, at the expense of a more complex algorithm and data structure.

6.2.5 Induction Analysis

Induction analysis was described in Section 2.2.3. The induction form is very similar to the reduction form, but with three important differences:

1. The induction variable must be of type integer.
2. The expression being added to the induction variable must be an integer constant.
3. The induction variable has no restrictions about being used elsewhere in the loop in a non-induction way.

Like the reduction location, the induction location would be classified by MCA in the ReadWrite summary set, and have a dependence found by the ART.

One problem faced by people who implement a separate induction recognition pass is not knowing which loop will eventually be made parallel. This is a problem because the closed form of an induction variable is more computationally expensive to evaluate than is the original form. Take, for example, the following loop:

```
 do I=1,N
    do K=I,100
       do J=1,K
          L = L  + 1
S1:       B(L) = . . .
S2:       . . .
       end do
    end do
    L = L + 1
S3: B(L) = . . .
S4: . . .
 end do
```

The Polaris induction pass transforms statements S1 and S3 into the following:

```
S1:  B((-5051)+j+l0+(i+((-2)*i+(-1)*i**3)/3+k**2+(-1)*k)/2+5051*i) = . . .
S3:  B(l0+(i+((-2)*i+(-1)*i**3)/3)/2+5051*i) = . . .
```

This eliminates the dependence, but is very computationally intensive. The triangular inner loop nest causes most of the complexity. Assuming that only one loop in a nest can be parallelized, if it is decided to parallelize the `do I` loop (serializing the inner loops), the closed form could become much simpler. However, if the induction pass is run prior to dependence analysis, it cannot know which loop will be parallelized. Some compilers (including Polaris) solve this by generating the full closed form everywhere.

Compare the following two loops:

```
$parallel do (I)
  L0 = L
  do I=1,N
     do K=I,100
        do J=1,K
 S1:      B((-5051)+J+L0+(I+((-2)*I+(-1)*I**3)/3+K**2+(-1)*K)/2+5051*I) = . . .
 S2:      . . .
        end do
     end do
 S3: B(L0+(I+((-2)*I+(-1)*I**3)/3)/2+5051*I) = . . .
 S4: . . .
  end do

$parallel do (I)
  do I=1,N
     do K=I,100
        do J=1,K
           L = L + 1
 S1:       B(L) = . . .
 S2:       . . .
        end do
     end do
 S3: B(L0+(I+((-2)*I+(-1)*I**3)/3)/2+5051*I) = . . .
 S4: . . .
  end do
```

The second is obviously more efficient. This code is possible if the closed-form for the induction is generated only within the parallel loop.

The recognition of a possible induction can be incorporated into a structure involving MCA and the ART in much the same way as is reduction recognition. After an induction is recognized, if the loop is chosen for parallelization, the generation of the closed-form for the induction can be done.

6.2.6 Run-time Dependence Analysis

Dependence analyzers typically return three results:

1. a dependence exists
2. no dependence exists
3. unknown

If the result is "unknown", then sometimes a condition can be extracted which, when evaluated at run-time, can test the dependences which are in question, then the loop can be conditionally parallelized by using the condition in an `if` statement which chooses between a serial and a parallel version of the original loop.

The correctness predicate, the classification condition, the imprecise-sorting flag, and the monotonicity flag within the ARD, plus the conditions derived from proving that one condition implies another, can all trigger the generation of a run-time dependence check.

These conditions can be collected from a set of ARDs which remain on the loop after parallelization. The conditions can be combined, simplified, and included in an `if` statement which chooses between the parallel and serial forms of a loop. The imprecise-sorting flag could generate a run-time sort of the access dimensions and a run-time overlap check, probably in the form of a library call whose return value is tested in the `if` condition. Similarly, if the monotonicity flag is marked "unknown monotonic", then a monotonicity check could be generated at run-time.

An example of a loop which could use such a run-time check would be the following:

```
      do 960 I=1,NNN
         SX(NS(I))=AX(I)*RF
         SY(NS(I))=AY(I)*RF
         SZ(NS(I))=AZ(I)*RF
  960 continue
```

The three arrays `SX`, `SY`, and `SZ` are dependence-free if it can be proved that `NS` contains no two values which are the same. The ART presently generates a monotonicity test for `NS` as a condition for parallelization, which is correct, but in the future it may be possible to relax that test into one which simply checks that the arrays have no two values the same.

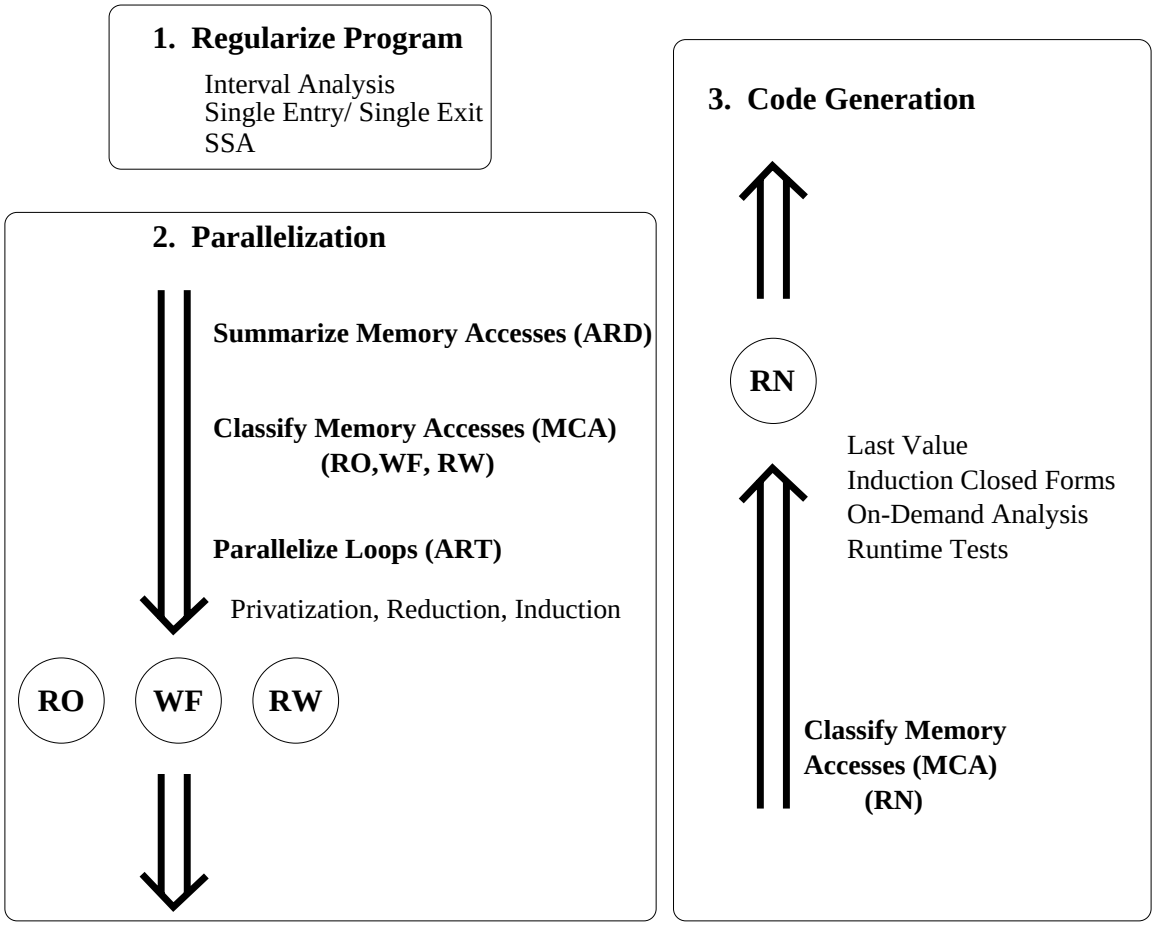

Figure 6.14: The Summary Set Framework structure for a compiler based on Memory Classification Analysis, using the summary sets ReadOnly (RO), WriteFirst (WF), ReadWrite (RW), and ReadNext (RN).

6.3 A Framework for Interprocedural Analysis

Now, the task is to define the framework which combines parallelization, privatization, reduction and induction recognition, and demand-driven analysis. This framework will be referred to as the **Summary Set Framework**. A pictorial representation of this compiler framework is shown in Figure 6.14. The Summary Set Framework algorithm is given in Figure 6.16.

The framework involves three passes. The first is a regularization pass, which prepares the program for parallelization, the second is a classification pass, which proceeds top-down and forward through the code, and the third is a top-down and backward code-generation pass.

Component	Necessary or Optional	Whole Prog or Single Prog Unit	Usefulness
Interval Analysis	Necessary	Single Prog Unit	Find loops-form nested structure
Control Flow Norm.	Necessary	Whole Prog	guarantees single-exit point
SSA form	Necessary	Single Prog Unit	Demand-driven deeper analysis - correct symbolic analysis - generate induction closed forms
Interprocedural Value Propagation	Optional	Whole Prog	Increased symbolic accuracy
Range Propagation	Optional	Single Prog Unit	Increased symbolic accuracy
Deadcoding	Optional	Single Prog Unit	Increased processing speed
Subroutine cloning	Optional	Whole Prog	Increased symbolic accuracy
Topological Sort	Optional	Single Prog Unit	Reduced traversal complexity

Table 6.5: Some potential components of the regularization pass in the proposed compiler framework.

6.3.1 Regularization Pass

The regularization pass prepares the source code for the parallelization and code generation passes. Interval analysis is required in this part, as is the transformation of all loops and procedures into single-entry/single-exit form. Conversion to SSA form is required for correct symbolic analysis, generating induction closed-forms and demand-driven deeper analysis. A number of other analyses and transformations are useful, but optional. Table 6.5 shows a set of components which would be useful to employ in the regularization pass.

6.3.2 Classification Pass

The classification pass employs the MCA algorithm, starting from the main entry point in the code and proceeding forward in execution order. If the code was topologically sorted during the regularization pass, then the classification algorithm may be run as a single sweep of the program, traversing the statements in lexical order. If the code was not topologically sorted, then in order to make sure the classification follows execution order, a traversal of the control flow graph must be done, via depth-first or breadth-first search.

When the header statement for a program context (`do` statement, `if` statement, `call` statement) is encountered, the classification algorithm calls itself recursively to produce the summary sets for that context.

As detailed in the classification algorithm of Figure 6.9, parallelization (in our case, loop parallelization in the form of the ART) takes place along with classification. Loops are marked either serial or parallel, and if they are parallel, variables are marked "private", and reductions and inductions are marked. Any conditions which need to be tested at run-time are also noted.

Because a `call` statement causes a recursive call to the classification routine, which either computes the summary sets of the subroutine or gathers those which were calculated by a previous call, this produces essentially a bottom-up calculation of summary sets on the program call-tree. Every loop in which no dependence can be found is marked "parallel".

6.3.3 Code-generation Pass

The final pass starts at the common exit point of the main program and proceeds backwards, following the reverse of the execution path of the program. If the program was topologically sorted, this pass can proceed along the reverse lexical order of the code. If it was not topologically sorted, then a reverse depth-first or breadth-first traversal will be necessary.

This pass has three purposes:

1. to calculate the ReadNext summary set (for generating last-value assignments),
2. to do on-demand deeper analysis of the program, and
3. to decide which loops will be made parallel and generate the appropriate code.

6.3.3.1 The ReadNext calculation

The ReadNext summary set is the set, calculated at every point in the program, of all the memory locations which are read next along some execution path following that point. This is essentially the same as the traditional *liveness* [1] calculation. The purpose of the ReadNext summary set is to determine which private locations need to have a last value copy inserted for them.

The calculation of ReadNext is a simple one:

$$\text{ReadNext} \leftarrow (\text{ ReadNext } - \text{ WriteFirst }) \cup (\text{ ReadOnly } \cup \text{ ReadWrite })$$

The conservative thing to do in this calculation is to subtract less write locations and add more read locations. In other words, if the subtraction or union cannot be done precisely, the calculation must err on the side of removing too few locations during the subtraction of WriteFirst, and err on the side of adding too many locations during the union with ReadOnly and ReadWrite.

The form above is conservatively correct since in the ReadOnly and WriteFirst summary sets, the order of reading and writing is known, and with the ReadWrite set, which is used to hold both legitimate read-first-then-write locations and also locations for which the order is unknown, it is properly assumed that a read happens first, so that the error is on the side of enlarging the ReadNext set.

The ReadNext calculation can be done using summary sets for either individual statements or larger code sections. This allows the calculation to, for instance, use the summary sets for a whole subroutine at a `call` statement.

6.3.3.2 On-Demand Deeper Analysis

Conditions which are gathered during the normal operation of the classification pass can be either tested during the code generation pass by sophisticated analysis algorithms, or generated as run-time dependence tests. The code generation pass can decide which approach is better by taking into account a number of factors, such as the perceived "importance" of the loop, the effort which the user wants the compiler to use, etc. If the avoidance of a run-time test seems important, the compiler can launch more intensive "deeper analysis" in an attempt to prove the condition at compile-time.

An example of a situation where on-demand analysis would be useful can be found in the Perfect Benchmarks code MDG, loop INTERF_do1000, a simplified version of which is shown in Figure 6.15.

The only difficulty which the ART has in parallelizing this code is for the variable `RL`. There is a write-first region $\{\texttt{RS}[6:9] \leq \texttt{CUT2}\}\texttt{RL}[6:9]$ for statement `S1`, and a read-only region $\{\texttt{KC} \equiv 0\}\texttt{RL}[6:9]$ for statement `S2`. Since it is advantageous for us to classify something WriteFirst and the ReadOnly region for `S2` is a subregion of the WriteFirst region from `S1`, the classification pass would register the need to prove that $\{\texttt{KC} \equiv 0\}$ implies $\{\texttt{RS}[6:9] \leq \texttt{CUT2}\}$.

```
S4:           KC=0
              do 1110 K=1,9
              RS(K)=XL(K)*XL(K)+YL(K)*YL(K)+ZL(K)*ZL(K)
S5:      1110 if(RS(K).gt.CUT2) KC=KC+1
S3:              . . .
              do       1130       K=2,5
              if(RS(K).ge.CUT2) go to 11
              FF(K)=-QQ2/(RS(K)*SQRT(RS(K)))-REF2
              VIR=VIR+FF(K)*RS(K)
           11 if(RS(K+4).gt.CUT2) GO TO 1130
S1:           RL(K+4)=SQRT(RS(K+4))
              FF(K+4)=QQ/(RS(K+4)*RL(K+4))+REF1
              VIR=VIR+FF(K+4)*RS(K+4)
         1130 continue
              if(KC.NE.0) go to 1140
                 . . .
              do 1140 K=11,14
S2:           FTEMP=AB2*EXP(-B2*RL(K-5))/RL(K-5)
                 . . .
              RL(K)=SQRT(RS(K))
              FF(K)=(AB3*EXP(-B3*RL(K))-AB4*EXP(-B4*RL(K)))/RL(K)
              VIR=VIR+FF(K)*RS(K)
         1140 continue
```

Figure 6.15: Simplified code from MDG, illustrating the possibility of using on-demand deeper analysis to prove $\{\texttt{KC} \equiv 0\} \Rightarrow \{\texttt{RS}[6:9] \leq \texttt{CUT2}\}$, thus allowing privatization of RL and parallelization of the loop.

By using the SSA def-use links, the compiler can find the join point in the control flow graph at statement S3, where the version of KC defined in statement S4 joins with the version of KC defined in statement S5. The S5 version of KC would have been marked as a possible induction variable, with increment 1 and execution predicate {RS[1 : 9] > CUT2}. It is conceivable that a general implication prover could use this information to infer that KC equals 0 only if RS[1 : 9] ≤ CUT2, which includes the condition it is trying to prove: RS[6 : 9] ≤ CUT2, thereby allowing the classification of RL[6:9] to be unconditionally WriteFirst, and therefore allowing the privatization of that section, removing the dependence on that section, and allowing the parallelization of INTERF_do1000.

This kind of primitive theorem-proving is potentially intensive analysis, and therefore should only be performed on-demand, when it appears to be beneficial, not as part of the classification algorithm itself. In this case it would be beneficial, because INTERF_do1000 is a very large loop (in terms of program text), and from running the program it may be seen that it is the most time-consuming loop in the program.

6.3.3.3 Deciding Which Loops Should Be Parallelized

Since the code-generation pass takes place top-down, it sees the outer-most loops of a given loop nest first. This gives it the chance to parallelize outer-most loops first, and if a loop is not parallelized, to avoid generating any parallelization code, such as the closed form of an induction variable.

When the code-generation pass decides that a loop should be parallelized, it calculates the last-value assignments needed by intersecting the set of memory locations being privatized with the ReadNext set immediately after the loop:

$$\text{LastValue} \leftarrow \text{Private} \cap \text{ReadNext}_{after}$$

In addition, any locations which were marked for reduction get the proper code generated for them. Any memory locations which were marked for induction must get the proper closed form generated for them.

Algorithm Summary Set Framework

Input: A program
Output: A parallelized program
Algorithm:

```
SSF( )
    regularization(main_prog )
    classification(main_prog )
    generate_code(main_prog )
end SSF

regularization( )
    interval_analysis( )
    control_flow_normalization( )
    convert_to_SSA_form( )
    others( )
end regularization

others( )
    choose from:
    interprocedural_value_propagation( )
    intraprocedural_range_propagation( )
    subroutine_cloning( )
    dead_coding( )
end others

control_flow_normalization( )
    topologically_sort_basic_blocks( )
    produce_single_exit_routines( )
end control_flow_normalization
generate_code( )
    compute_last_value( )
    on_demand_analysis( )
    run_time_code( )
    induction_code( )
    reduction_code( )
end generate_code( )
```

Figure 6.16: The Summary Set Framework algorithm.

Algorithm Compute Last Value

```
Input:  A program with ARDs representing private memory locations
Output: A program with last-value copies
Algorithm:
  compute_last_value( last_stmt, first_stmt, global_predicate, local_predicate)
    for ( stmt=last_stmt.next() to first_stmt do
      switch(stmt.type() )
        case: ELSEIF:
          determine_predicate( stmt, global_predicate, local_predicate, local_neg )
          continue
        case: ELSE:
          determine_predicate( stmt, global_predicate, local_predicate, local_neg )
          continue
      endswitch
      compute_access_rep ( stmt ) // Compute memory accesses for text of stmt

      if stmt.type() == ENDIF then
        determine_predicate( stmt, global_predicate, local_predicate, local_neg )
        reverse_classify( stmt, endif-stmt,
          global_predicate and local_predicate and local_neg,
          if condition )
      endif
      compute_readnext(new_readonly, new_writefirst, new_readwrite,
        ReadOnly, ReadWrite, WriteFirst, ReadNext)
    endfor

    simplify_memsets( ) // employ ARD simplification ops

    if stmt.type() == ENDLOOP then
      expand_by_loop ( ) // expand by loop index
      if ( decide_parallel ( ) ) then // demand-driven propagation, generate runtime tests
        determine_last_value ( ) // Private ∩ ReadNext
      endif
    elseif stmt.type() == ENDIF then
      summarize_IF_stmt ( )
    else
      summarize_to_procedure ( )
    endif
  end compute_last_value
```

Figure 6.17: The algorithm to compute last value copies for private variables.

6.3.4 Precision of the Analysis

Notice that all classification and dependence analysis in the Summary Set Framework is based on the summary set intersection operation. Therefore, the accuracy of the intersection operation directly determines the accuracy of both classification and dependence analysis in this framework. This is a good thing in that it offers a single point where concentration of effort can improve the accuracy of the overall analysis.

6.3.5 Conservative Operations

Inevitably, some of the symbolic set operations will be imprecise. This can be due to a shortcoming of the symbolic manipulation routines within the compiler or simply due to unknown or unrelated variables being used to represent the accesses of the program. For instance, how do you intersect `A[1:N]` with `A[R:S]`? All operations within the compiler must accept a *conservative direction* parameter, which indicate what to do when the answer is just *"I don't know!"*.

For intersection, sometimes the conservative thing to do is report that the accesses do not overlap at all. This is the correct choice when the intersection is being done to simplify the representation (for example, at a join point in the program flow graph, as was discussed in Section 6.1.8.2. At other times the conservative thing to do is to report that the accesses intersect. This is the correct choice when an empty intersection allows us to eliminate a possible dependence (for example, during dependence analysis).

It is crucial to this process that all operations be aware of the proper conservative direction for the result.

6.4 A General Multi-Dimensional Intersection Algorithm

Intersecting two arbitrary LMADs is quite difficult. But if two LMADs are stride-equivalent or semi-stride-equivalent, then they are similar enough to make the intersection algorithm tractable.

The algorithm accepts two stride-equivalent LMADs. If the two LMADs are semi-stride-equivalent, then 0-span dimension(s) can be safely inserted into the LMAD with fewer dimensions (according to Theorem 3) to make them stride-equivalent.

Let us assume two input LMADs which have all dimensions precisely sorted, $\text{LMAD}_{\text{left}} = \mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma_1,\sigma_2,\cdots,\sigma_d} + \tau$ and $\text{LMAD}_{\text{right}} = \mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma'_1,\sigma'_2,\cdots,\sigma'_d} + \tau'$. The intersection routine is called with the two LMADs as parameters, such that $\text{LMAD}_{\text{left}}$ has a base offset which is less than that of $\text{LMAD}_{\text{right}}$, and the number of the dimension, d, to work on (initially the dimension d is the dimension with the largest stride):

$$\textbf{Intersect}(\text{LMAD}_1, \text{LMAD}_2, d)$$

The routine treats the LMADs as if they were stride-1 over their whole extent through the given dimension. By comparing the two extents, it can be determined whether they overlap in any way, as shown in Figure 6.18, part A. If not, it can safely report that the intersection is empty. If they do overlap, then the routine eliminates the given dimension from each, possibly adjusts the offset of the left-most LMAD, and calls itself recursively to work on the next-inner dimension, as shown in Figure 6.18, part B. Finally, when the dimension to work on is the inner-most, as in Figure 6.19, it can make the final determination as to whether there is an intersection between the two. If there is, it can determine what the intersection is, since this is a simple one-dimensional case (such as can be easily implemented for traditional triplet notation). The result LMAD for the intersection is returned. As each recursion returns, a dimension is added to the result descriptor.

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

Refer to Figure 6.20 for the full algorithm.

6.5 Comparing the Access Region Test with Other Dependence Tests

This section will give an informal argument that the Access Region Test is more powerful than the combination of the Range Test and the GCD test.

The Range Test compares two subscript expressions (call them f and g) to determine whether they could take on the same values at any point within the range of any of the surrounding loops. The Range Test uses two tests to determine whether f and g overlap, as discussed in Section 2.3.4.

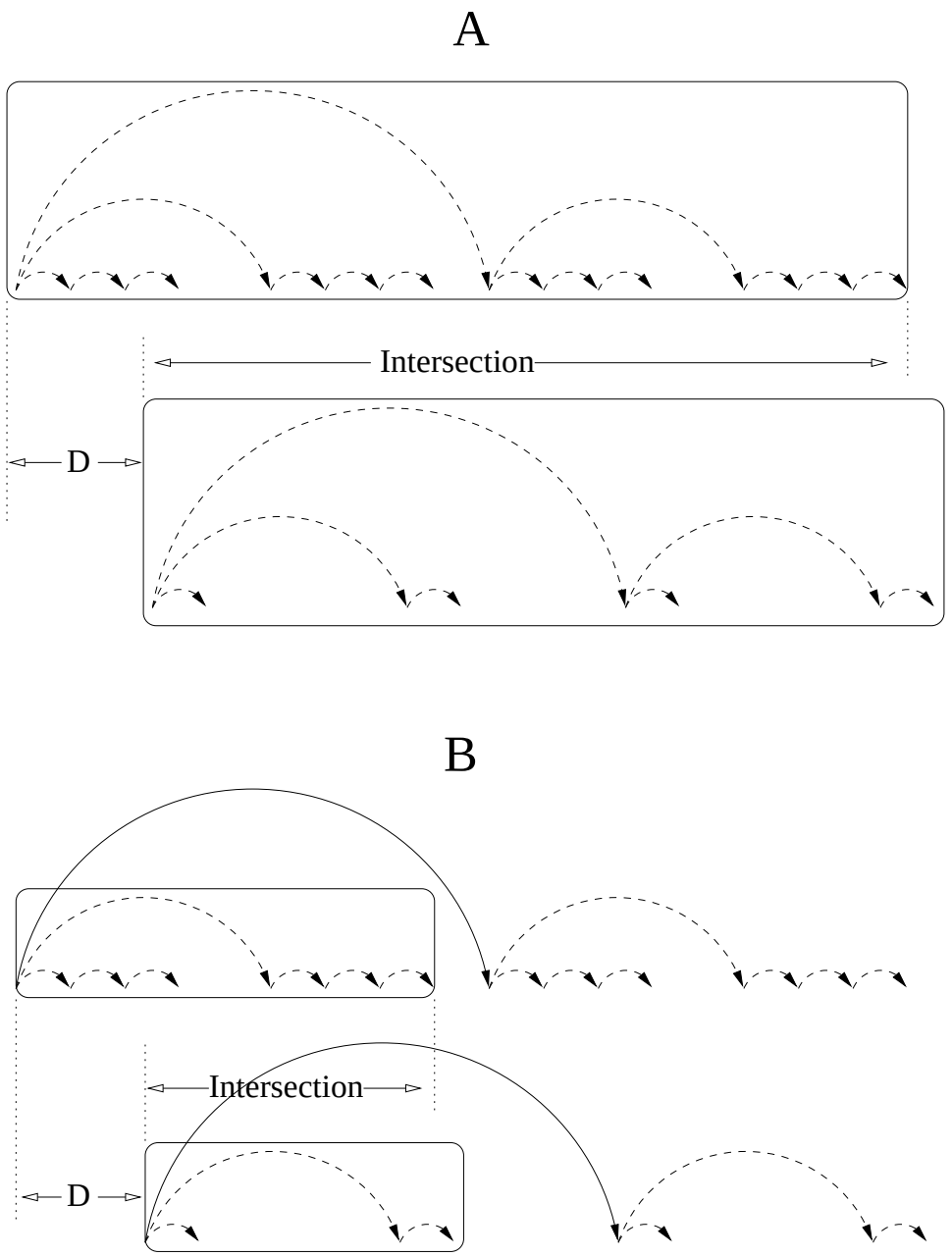

Figure 6.18: The multi-dimensional recursive intersection algorithm, considering the whole extent of the two access patterns (Part A), then recursing inside to consider the next inner dimension (Part B).

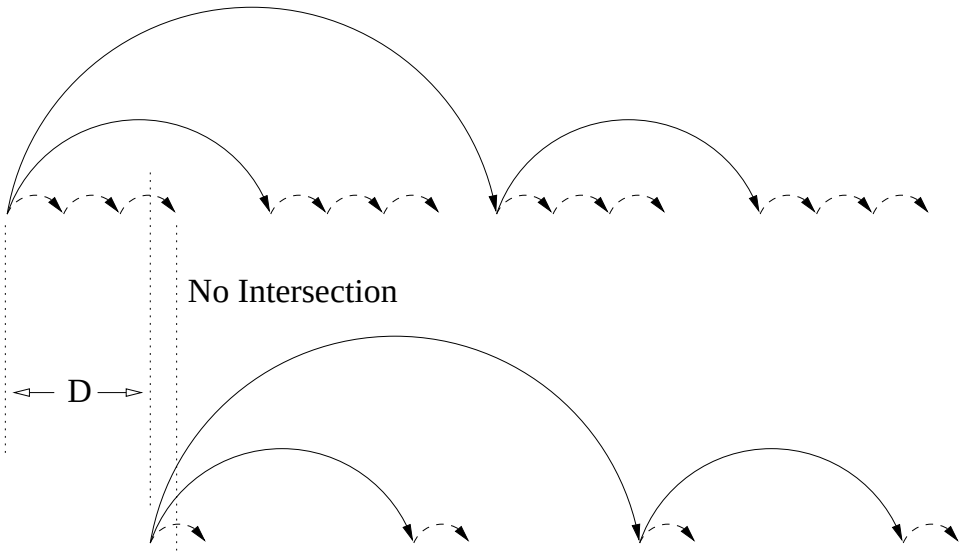

Figure 6.19: The multi-dimensional recursive intersection algorithm, considering the inner-most dimension, finding no intersection.

If the minimum of one of the functions is greater than the maximum of the other, then the Range Test reports that there is no overlap. This case is illustrated in Figure 6.21, part A. If any interleaving of the values of the functions happens, as in Figure 6.21, part B, then it uses the other test to check for overlap. The other test expresses the minimum and maximum of f and g in terms of the loop index of the loop being tested for independence. It checks the maximum of one function against the minimum expression of the other function with the loop index in question incremented by its stride value. On the figure, it checks that point 9 is greater than point 4 and that point 10 is greater than point 2. In the case of part B of Figure 6.21, these conditions are both true.

However, this is not the case in part A or part B of Figure 6.22. These are cases which the Range Test cannot prove independent, although it is obvious from the figure that they are. Part B of Figure 6.22 might be provably independent by the GCD Test as long as the GCD of the two strides is greater than 1. If it is, then it can be proven that the GCD of the strides does not divide the difference between the starting points, since the difference is less than the smallest stride. But, if the GCD of the strides is 1, as is often the case in programs, the GCD could not determine the independence of Part B either.

All four of these cases can be determined to be independent by the Access Region Test. The multi-dimensional intersection algorithm would find the intersection empty in each case, since at the inner-most access dimension they can be found not to overlap.

This shows that the Range Test's method of logical loop interchanging is not as flexible as the Access Region Test. Loop interchanging cannot help in part A of Figure 6.22, since the two loops produce different strides in the two expressions.

It is interesting to note that the MAX and MIN functions of the Range Test need to know the monotonicity of the subscript function with respect to a given loop index to determine whether to substitute the upper or lower loop bound for the loop index in the subscript expression. They must also find the stride due to a loop index. Blume states in his thesis that a naive implementation of this test can be inefficient, because it would have to continually compute these values. Instead, he computes the values only once and saves them for subsequent uses. The use of the LMAD serves this function of computing monotonicity, stride, and span only once (when the LMAD is constructed) and then has them always available for doing the testing.

The loop permutation done by the Range Test is avoided by sorting the access dimensions according to the stride. This causes the overlap test to check the dimensions in the correct order without needing the permutations.

6.5.1 The ART as an Alternative to Equation-Solving Tests

Notice that the Access Region Test does no equation-solving to determine dependence. This is in stark contrast to most other dependence tests. Taking the place of equation-solving is the multi-dimensional intersection algorithm, which attempts to solve the problem directly, instead of through an equation-solving regimen. This directness allows the ART to work without imposing the constraints which equation-solving requires, such as using only affine expressions.

Algorithm Intersect

Input: Two LMADs, with properly nested, sorted dimensions :
$\text{LMAD}_{\text{left}} = \mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma_1,\sigma_2,\cdots,\sigma_d} + \tau$, $\text{LMAD}_{\text{right}} = \mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma'_1,\sigma'_2,\cdots,\sigma'_d} + \tau'$ ($\tau \leq \tau'$ and $\delta_i \leq \delta_{i+1}$),
the number of the dimension to work on: $k (0 \leq k \leq d)$
Output: Intersection LMAD list
Algorithm:

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

Figure 6.20: The recursive, multi-dimensional intersection algorithm for LMADs. See Section 5.3.1 for a definition of the function *width*.

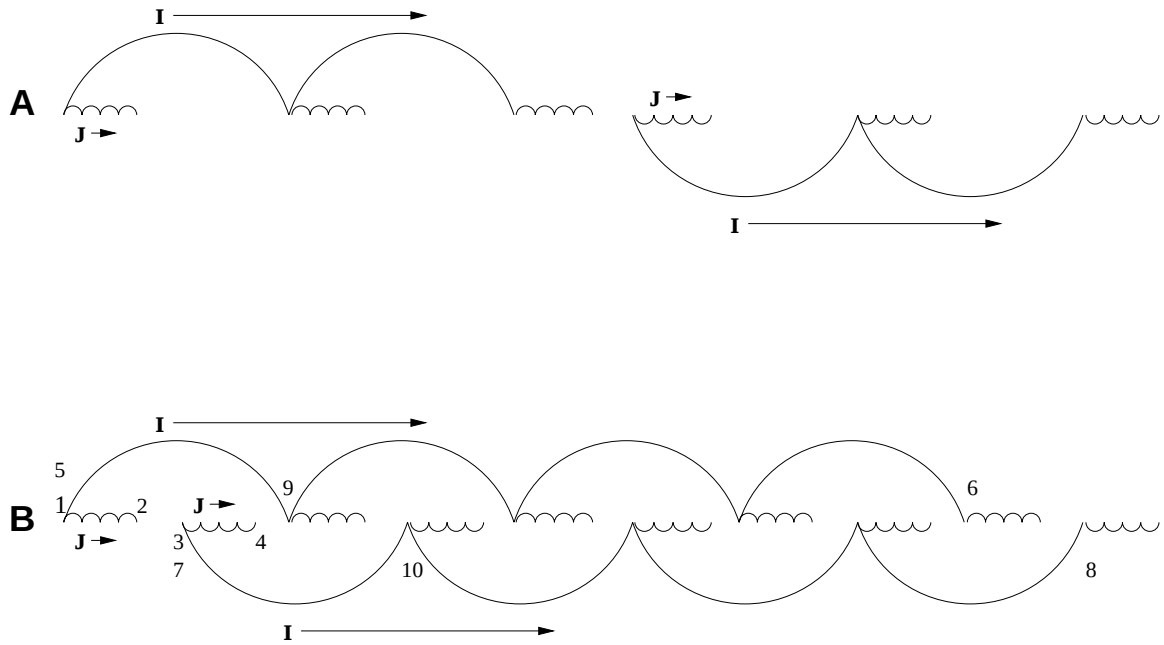

Figure 6.21: Memory access diagrams for access patterns which the both the Range Test and the Access Region Test can prove independent.

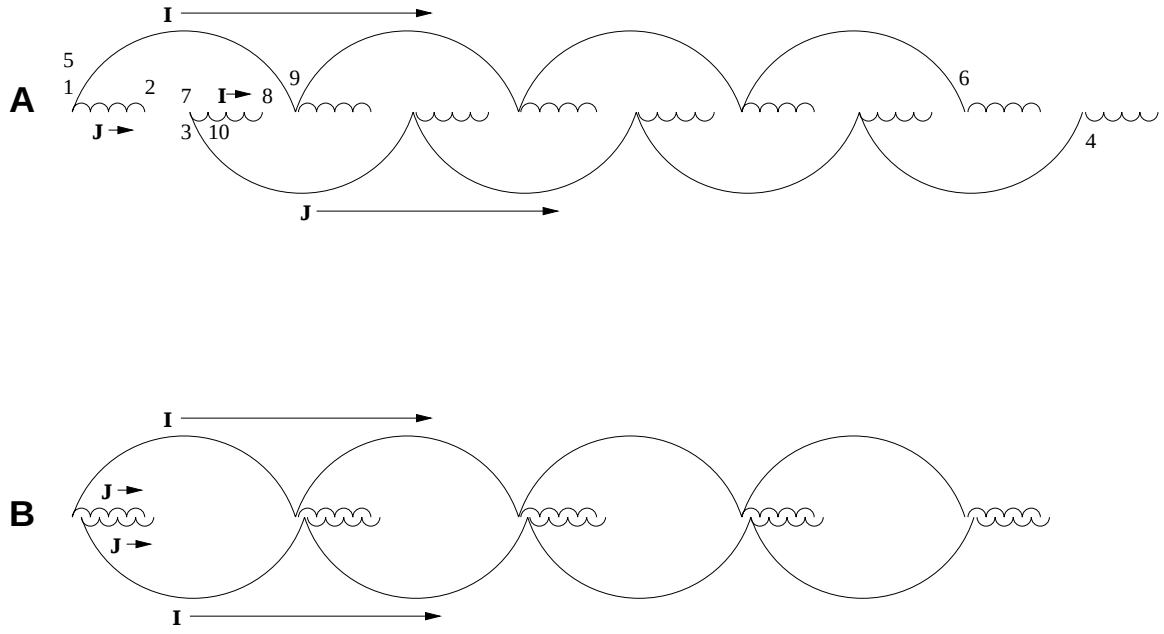

Figure 6.22: Memory access diagrams for access patterns which the Range Test cannot prove are independent, while the Access Region Test can.

7 IMPLEMENTATION ISSUES

7.1 Representation of Summary Sets

The summary sets are simply lists of ARDs. When a new descriptor is added to the set, it should be compared against each existing descriptor in the list with the subregion operation, to determine whether the memory locations are already represented in the set. If so, the new descriptor can be simply discarded.

The read-only, write-first, and read-write list for a given program variable are stored together in a `SymbolAccess` object. Each statement in the program has a keyed structure called a SymbolAccessMap which stores a SymbolAccess object for every symbol referenced in the statement.

7.2 Optimizations

Optimizing the operations on the summary sets is crucial to the success of the Summary Set Framework. If the summary sets grow large, doing any operation which requires $\mathcal{O}(n^2)$ comparisons between ARDs will dominate the compile time of the program. For this reason, the simplification operations must be used aggressively to reduce the number of dimensions per descriptor and the number of descriptors. Any linear time effort which reduces the number of descriptors or dimensions is well worth the time spent, since it reduces n in the $\mathcal{O}(n^2)$ processing step.

Therefore, the emphasis must be placed on approximate, linear algorithms for doing ARD simplification. The insertion-sorting of dimensions during construction of ARDs enables such algorithms. One is a linear algorithm for discovering whether one LMAD is a subregion of another. If we assume that an ARD which is equivalent or semi-equivalent to another has the same sorting order for its dimensions, then a linear comparison of dimensions between the two is all that is necessary to check for the subregion relationship. When an ARD is being inserted into a list of ARDs, we can make a quick linear scan of the existing ARDs to determine whether the new one is a subregion of an existing one. If so, it need not be inserted.

Another way to reduce processing time is to use flags within the ARD to record information which could otherwise only be discovered through lengthy processing. An example is to mark a descriptor when it is constructed as containing expressions involving subscripted-subscripts. Since very little can be done at present with dimensions involving subscripted-subscripts, checking for this flag before attempting to simplify or intersect such a descriptor can allow the algorithm to give up. The processing of subscripted-subscript references could still be done in the code-generation pass, when we might decide to attempt to generate a run-time test, or do deeper analysis for the values in the subscripting array.

7.2.1 Similarity Graphs

A Similarity Graph is a mechanism for keeping track of the relationships between ARDs. Sometimes an algorithm calls for the computation of relationships among all possible pairs of ARDs in a single list, and other times for the relationships between pairs of ARDs from two lists. A `SimGraph` object is used for the former type, while a `SimBiGraph` object is used for the latter type.

The mechanism for constructing these graphs does a full dimension matching between each pair of ARDs and stores the Similarity Type for each pair, as well as the dimension correspondence, from the point of view of each ARD. As the Similarity Graph is being constructed, it uses a transitive closure mechanism to reduce the amount of redundant work it must do. The observation which enables this is the fact that if we already know A is equivalent to B, and B is equivalent to C, then we need no comparison between A and C to know that A is equivalent to C, as shown in Figure 7.1. We can even construct the correspondence between the dimensions of A and C by using the other correspondences between the dimensions, as shown in Figure 7.2.

Both the `SimGraph` and `SimBiGraph` objects are iterated through by using an iterator object as a convenient way to iterate through all the possible pairs of ARDs. The iterator keeps track of which pairs have already been visited, and can react to changes in the Similarity Graph caused by changes which happen to the ARDs (by skipping ARD pairs in which one or both ARDs have been eliminated – by simplification operations, for instance). The Similarity Graph can even trigger the full recalculation of a relationship between two ARDs if they are visited and have changed enough to invalidate the stored relationship.

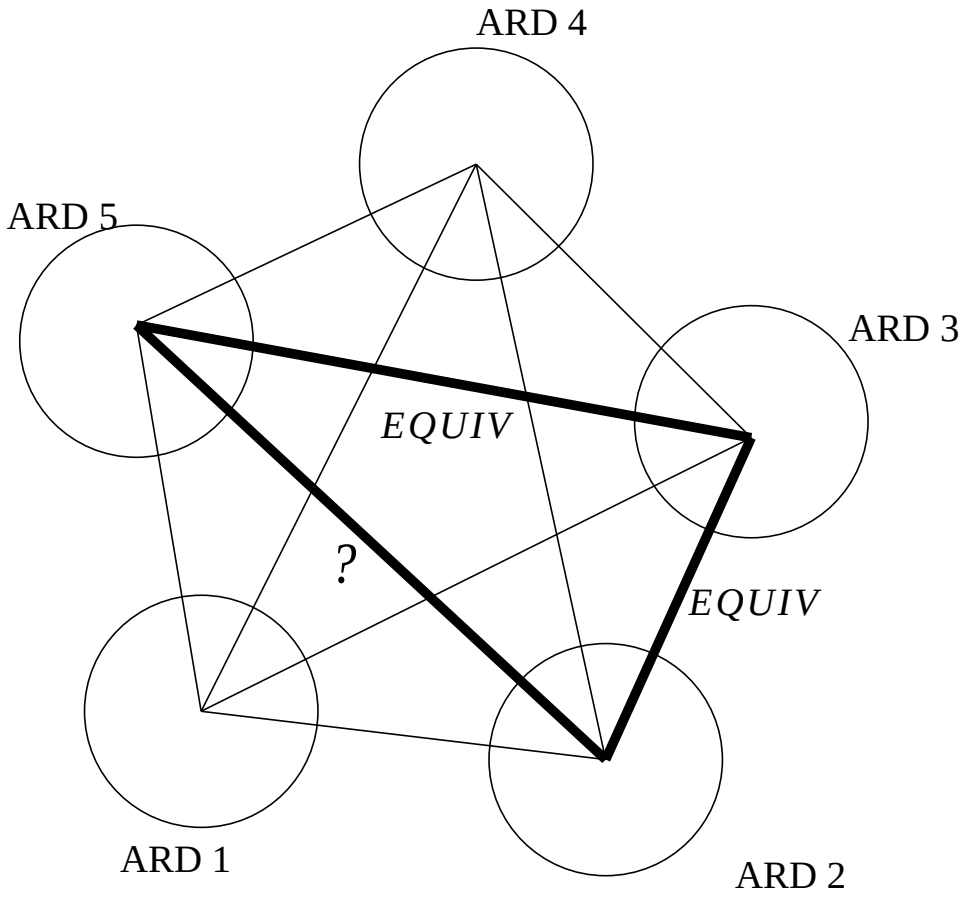

Figure 7.1: The Similarity Graph stores relationships between each pair of ARDs. In this situation, we can use transitive closure to determine that ARD5 is equivalent to ARD2 without needing to do the matching.

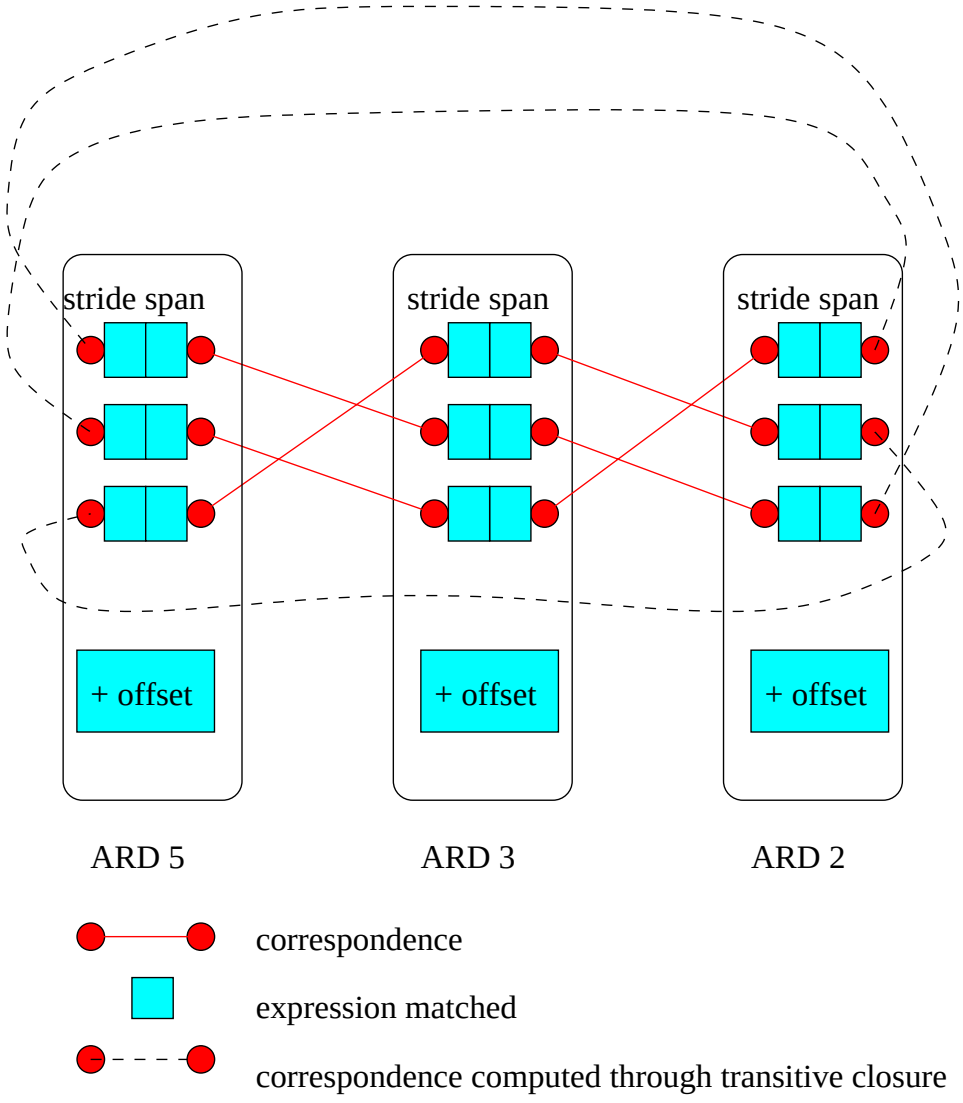

Figure 7.2: Calculating the dimension-matching between two ARDs through transitive closure.

7.2.2 Optimizations for Scalars

Scalars are treated similarly to arrays in the Summary Set Framework, in that they are represented by an LMAD. The base offset is zero and there are no access dimensions. This makes the framework more general in that it can handle aliasing between array elements and scalars, but it potentially adds a lot of extra descriptors to be processed. This could cause a lot of extra work for the framework. A check should always be made as to whether a scalar is being processed, since it needs much less complex processing. One example of this is the use of complicated mechanisms like the Similarity Graph (see Section 7.2.1) to compare two scalar ARDs. The expense of constructing a Similarity Graph is certainly not warranted to compare two zero-dimension, zero-offset descriptors. A simple check for the fact that a scalar is involved can avoid the expense.

7.2.3 Compiler Effort

It seems to be a good idea to place a limit on the size of summary sets for which simplification and intersection will be attempted. For example, if a summary set has grown to include a large number of ARDs, say 1000, nearly any $\mathcal{O}(n^2)$ process, whether it be simplification or intersection, will take an enormous amount of time. Chances are good that, since those 1000 have survived despite aggressive simplification, the ARDs will not be similar enough to get an accurate result (even if there are no dependences involved!). Therefore, the user should be given some kind of parameter which can be called a "compiler effort" value. In the current implementation of Polaris, this takes the form of a value between 1 and 10. An effort of 1 places the smallest limit on the size of an ARD list for which simplification and intersection will be attempted. The limit gets larger for each successive value until 10, which indicates *no limit*.

When the ARD list size exceeds the compiler effort limit, the operation to be done is simply abandoned, and the most conservative result is generated.

8 EXPERIMENTS

Many of the algorithms described in this thesis were implemented inside the existing Polaris compiler. Many codes have been compiled with the new compiler. The parallelization results have been mostly very good. On the other hand, the compilation time necessary for various codes has varied wildly. For some codes it has been very short, while for others it has been very long. For instance, compiling the SPEC cfp95 code TURB3D, required 20 hours for the "old" Polaris to compile (due to subroutine inlining, the source code grew from 2100 lines to 20,000 lines), but required only 5 minutes to compile using MCA with the ART, while compiling the Perfect Benchmark code ARC2D required 2 hours.

The ART version of Polaris was used to perform a series of experiments for four purposes:

1. to understand the extent of the representational accuracy problem for triplet notation,
2. to determine how much LMAD simplification was possible by using the coalescing and aggregation operations,
3. to see how the parallelization with the Access Region Test of a set of benchmark codes compares to that of the older version of Polaris (using the Range Test), and
4. to determine, in detail, how well the ART is able to parallelize a particular code, a two-dimensional FFT program called TFFT2, from the preliminary version of the SPEC cfp95 benchmarks.

8.1 Representational Accuracy Experiment

Fourteen Fortran77 programs were chosen in an attempt to study the magnitude of each of the sources of inaccuracy for triplet notation. These included codes from the SPEC cfp95 and Perfect benchmarks, and one from a set of production codes obtained from the National Center for Supercomputing Applications (NCSA) to study. After applying interprocedural value propagation, induction variable substitution, and forward substitution within these codes, the LMAD representation was used for memory accesses at each memory reference site. Each de-

scriptor was then expanded for each of its enclosing `do` loops. The number of expanded LMADs which would not be representable in triplet notation was counted. Finally, their percentage with respect to the total number of summaries was plotted in Figure 8.1.

For this analysis, array accesses that could not be represented by triplet notation were divided into the following five categories, as described in Chapter 4:

subscripted-subscripts – accesses due to references with subscripted-subscript expressions;

non-affine – accesses due to references with non-affine subscript expressions;

triangular affine – accesses due to references within a triangular loop;

coupled-subscripts – accesses due to references with coupled-subscript expressions;

multiple index affine – accesses due to references containing multiple indices in a subscript position.

In this classification scheme, each category excludes those above it. For instance, a reference with a subscripted-subscript inside a triangular loop would be counted as subscripted-subscript, and not triangular affine.

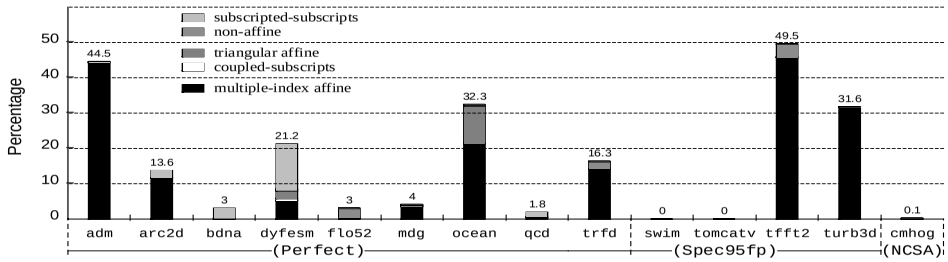

Figure 8.1: Percentage of non-triplet-representable access summaries versus total number of access summaries.

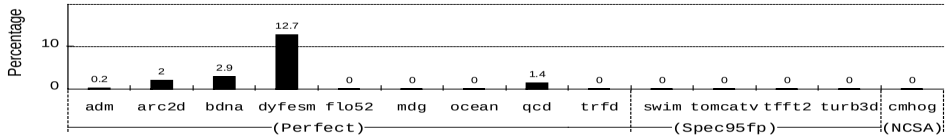

Figure 8.2: Percentage of access summaries which are not provably-monotonic versus total number of access summaries.

As discussed in Section 5.2.1.2, in order for the the span to be guaranteed to represent the true distance moved for a dimension, the subscript function must be monotonic for that index.

Thus, to see how often the LMADs have accurate spans, the percentage of array accesses that were provably monotonic at compile time was determined.

By their nature, of the five categories of references that are not accurately representable by triplet notation, all but subscripted-subscripts and non-affine are intrinsically monotonic. All the non-affine references in the set of test codes were checked and, unexpectedly, every one of those accesses was provably monotonic. Only the subscripted-subscripts were not provably monotonic at compile time. This data is presented in Figure 8.2.

This data shows that a large percentage of accesses are monotonic in each of our benchmark codes. This tends to support the argument that the LMAD is an appropriate representational form, at least for these benchmark codes.

8.2 The Effect of Simplification on LMADs

In an attempt to determine how much simplification can be achieved by the techniques of coalescing and contiguous aggregation, the simplification which was produced by coalescing and contiguous aggregation for our set of test codes was measured. All LMADs were computed interprocedurally without applying any simplification techniques. The LMADs produced at all loop headers and `call` statements were counted, and the number of dimensions used for all LMADs was totaled.

Then, all LMADs were computed interprocedurally, but both coalescing and contiguous aggregation were applied iteratively during the process until no more simplification was possible, and again the number of LMADs and the dimensions used in them were counted.

The reduction in the total number of dimensions was chosen as a measure to indicate the amount of simplification which had been performed, because it captures both the reduction in the number of LMADs (through aggregation) and the reduction in the number of dimensions (through coalescing).

The results, in Figure 8.3, show that a significant amount of simplification can be achieved in most cases.

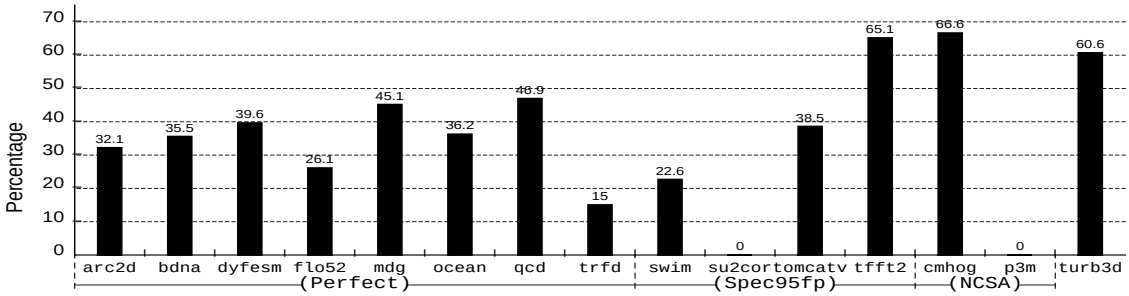

Figure 8.3: Percentage reduction in total number of LMAD dimensions by coalescing and contiguous aggregation.

8.3 Comparing ART Parallelization with Range Test Parallelization

The Access Region Test version of Polaris and the "old" version of Polaris were used to compile eleven codes. A comparison was made between the parallelization summaries in the two cases. When both versions serialized a given loop, or both versions parallelized it, it was left out of the tables in this section. The one exception to this rule is that when the old version of Polaris parallelized a loop without a run-time test and the ART version parallelized it with a run-time test, that was noted in the table. When one version parallelized a loop and the other version made it serial, the reason for the discrepancy was determined and an entry was made in one of the following tables for the loop.

8.3.1 An Explanation of the Notes in the Tables

There are three situations, described in the paragraphs that follow, where the Access Region Test fails to find parallelism, while the old Polaris does find it:

- SSA deeper analysis
- classification problem
- range info

There are four situations, described in the paragraphs that follow, where the ART finds parallelism, while the old Polaris could not:

- Range Test limitation
- privatization

PROGRAM-subroutine/loop	Range Test	ART	Notes
CLOUD3D -			
No parallelization differences			
TFFT2 -			
CFFTZ0_do#2	serial	parallel	Range Test limitation
RCFFTZ_do110	serial	parallel	Range Test limitation
CRFFTZ_do100	serial	parallel	Range Test limitation
CFFTZ_do#2	serial	parallel	Range Test limitation
FFTZ2_do100	serial	parallel	Range Test limitation
FFTZ2_do100/2	serial	parallel	Range Test limitation
CFFTZINIT_do#2	serial	parallel	Range Test limitation
TRFD -			
OLDA_do100	parallel	parallel	ART runtime test
OLDA_do300	parallel	parallel	ART runtime test

Table 8.1: A comparison of Range Test and ART parallelization for the programs CLOUD3D from NCSA, TFFT2 from a preliminary version of the SPEC FP95 benchmarks, and TRFD from the Perfect benchmarks.

- reduction
- run-time test.

SSA Deeper Analysis The "SSA deeper analysis" notation refers to the need to use the def-use chains of the SSA form for proving or disproving conditions using on-demand analysis in the code generation pass. This is discussed in detail in Section 6.3.3.2. This has not yet been implemented in Polaris.

Classification Problem This notation in the tables refers to situations in which it is not possible to classify a particular array reference due to unknown variables. This is discussed in detail in Section 6.1.12. This problem is solved by determining the conditions under which the reference should be classified in each possible category, then attaching a classification condition and inserting the ARD into the proper category.

These classification conditions can then be used to disprove dependence, as discussed in Section 6.1.12.

This has also not been implemented yet.

PROGRAM-subroutine/loop	Range Test	ART	Notes
MDG -			
INTERF0_do1000	parallel	serial	SSA deeper analysis
POTENG_do2000	parallel	serial	SSA deeper analysis
MDG_do210	parallel	parallel	ART runtime test
INTRAF0_do1000	serial	parallel	Range Test limitation
INTRAF0_do1300	serial	parallel	ART runtime test
CNSTNT_do1100	serial	parallel	Range Test limitation (triang)
INTRAF_do1000	serial	parallel	Range Test limitation
INTRAF_do1300	serial	parallel	ART runtime test
FLO52 -			
BCFAR_do20	parallel	serial	classification problem
BCFAR_do30	parallel	serial	classification problem
BCFAR_do40	parallel	serial	classification problem
BCWALL_do30	parallel	serial	classification problem
COLLC_do10_IV1	parallel	parallel	ART runtime test
COLLC_do30_IV1	parallel	serial	classification problem
COLLC_do40_IV1	parallel	serial	classification problem
DFLUX_do30	parallel	serial	classification problem
EULER_do50	parallel	serial	classification problem
EULER_do60	parallel	serial	classification problem
METRIC_do15	parallel	serial	classification problem
PSMOO_do40_IV1	parallel	serial	classification problem
PSMOO_do40_IV2	parallel	serial	classification problem
PSMOO_do80_IV1	parallel	serial	classification problem
PSMOO_do80_IV2	parallel	serial	classification problem
STEP_do40	parallel	serial	classification problem
HYDRO2D -			
No parallelization differences			

Table 8.2: A comparison of Range Test and ART parallelization for the programs MDG, FLO52 from the Perfect benchmarks, and HYDRO2D from the SPEC FP'95 benchmarks.

PROGRAM-subroutine/loop	Range Test	ART	Notes
OCEAN -			
FTRVMT7_do103	serial	parallel*	Range Test limitation
FTRVMT7_do#2	serial	parallel*	Range Test limitation
FTRVMT7_do106	serial	parallel*	Range Test limitation
FTRVMT7_do108	serial	parallel*	Range Test limitation
FTRVMT7_do#4	serial	parallel*	Range Test limitation
FTRVMT7_do1060	serial	parallel*	Range Test limitation
FTRVMT7_do#5	serial	parallel*	Range Test limitation
FTRVMT7_do1061	serial	parallel*	Range Test limitation
FTRVMT6_do103	serial	parallel*	Range Test limitation
FTRVMT6_do#2	serial	parallel*	Range Test limitation
FTRVMT6_do106	serial	parallel*	Range Test limitation
FTRVMT6_do108	serial	parallel*	Range Test limitation
FTRVMT6_do#4	serial	parallel*	Range Test limitation
FTRVMT6_do1060	serial	parallel*	Range Test limitation
FTRVMT6_do#5	serial	parallel*	Range Test limitation
FTRVMT6_do1061	serial	parallel*	Range Test limitation

Table 8.3: A comparison of Range Test and ART parallelization for the OCEAN program from the Perfect benchmarks, part 1. "parallel*" means that the loop would be parallelized by the ART with a modest improvement in symbolic expression simplification.

PROGRAM-subroutine/loop	Range Test	ART	Notes
OCEAN -			
FTRVMT4_do103	serial	parallel*	Range Test limitation
FTRVMT4_do#2	serial	parallel*	Range Test limitation
FTRVMT4_do106	serial	parallel*	Range Test limitation
FTRVMT4_do108	serial	parallel*	Range Test limitation
FTRVMT4_do#4	serial	parallel*	Range Test limitation
FTRVMT4_do1060	serial	parallel*	Range Test limitation
FTRVMT4_do#5	serial	parallel*	Range Test limitation
FTRVMT4_do1061	serial	parallel*	Range Test limitation

Table 8.4: A comparison of Range Test and ART parallelization for the OCEAN program from the Perfect benchmarks, part 2. "parallel*" means that the loop would be parallelized by the ART with a modest improvement in symbolic expression simplification.

PROGRAM-subroutine/loop	Range Test	ART	Notes
OCEAN -			
FTRVMT_do103	serial	parallel*	Range Test limitation
FTRVMT_do#2	serial	parallel*	Range Test limitation
FTRVMT_do106	serial	parallel*	Range Test limitation
FTRVMT_do108	serial	parallel*	Range Test limitation
FTRVMT_do#4	serial	parallel*	Range Test limitation
FTRVMT_do1060	serial	parallel*	Range Test limitation
FTRVMT_do#5	serial	parallel*	Range Test limitation
FTRVMT_do1061	serial	parallel*	Range Test limitation

Table 8.5: A comparison of Range Test and ART parallelization for the OCEAN program from the Perfect benchmarks, part 3. "parallel*" means that the loop would be parallelized by the ART with a modest improvement in symbolic expression simplification.

PROGRAM-subroutine/loop	Range Test	ART	Notes
BDNA -			
RESTAR0_do20	serial	parallel	privatization (code generation)
RESTAR0_do150	serial	parallel	reduction (runtime)
RESTAR0_do700	serial	parallel	reduction (runtime)
ACTFOR_do238	serial	parallel	ART runtime test
ACTFOR_do240	parallel	serial	SSA deeper analysis
ACTFOR_do110	serial	parallel	reduction
CNVERT_do40	serial	parallel	privatization (code generation)
CNVERT_do50	serial	parallel	privatization (code generation)
MPOLES_do100	serial	parallel	reduction (code generation)
SITES_do960	serial	parallel	privatization (runtime)
UPDATE_do100	serial	parallel	runtime
UPDATE_do100/2	serial	parallel	runtime

Table 8.6: A comparison of Range Test and ART parallelization for the BDNA program from the Perfect benchmarks.

PROGRAM-subroutine/loop	Range Test	ART	Notes
ARC2D -			
BC_do60	parallel	serial	range info
BC_do70	parallel	serial	range info
BICONG_do6	parallel	serial	range info
FILERY_do39	parallel	serial	range info
YPENT2_do12	parallel	serial	range info
YPENT2_do22	parallel	serial	range info
YPENT2_do13	parallel	serial	range info
TOMCATV -			
No parallelization differences			
SWIM -			
No parallelization differences			

Table 8.7: A comparison of Range Test and ART parallelization for the ARC2D program from the Perfect benchmarks and the programs TOMCATV and SWIM from the SPEC FP95 benchmarks.

Range Info This notation refers to the need for improved symbolic range information made available to the compiler. The Access Region Test can theoretically determine independence in each case, provided it has access to sufficient symbolic range information for the variables involved.

An example of such a situation is as follows:

```
      real F(JDIM,KDIM,2)
      K = KU-1
      do 12 J = JL,JU
        . . .
      F(J,K,1) = (F(J,K,1) - LD2*F(J,K-2,1) - LD1*F(J,K-1,1))*LDI
      F(J,K,2) = (F(J,K,2) - LD2*F(J,K-2,2) - LD1*F(J,K-1,2))*LDI
        . . .
12    continue
```

Here, it is easy to see that the references to `F` do not involve a dependence, but ART parallelization requires us to compare `JU-JL` against `JDIM`. From the original form it is easy to see that $\texttt{JU} - \texttt{JL} \leq \texttt{JDIM}$ due to the Inbounds Assumption. In the ARD form, however, the original

declared dimensions are lost, and the ART must rely on the range information to retain the proper relationships.

An alternative way to solve this problem is to mark `J` as a singleton index, as discussed in Section 6.1.14.

Range Test Limitation These cases were of two types:

1. non-affine subscript expressions
2. access patterns not analyzable by the Range Test

The non-affine subscript expressions were found in the TFFT2 program. These used expressions involving $2^{\mathtt{L}}$, where `L` is the index of an outer loop. This will be discussed in detail in Section 8.4. These patterns were parallelized by the ART primarily through the use of simplification operations on the LMADs involved.

The non-analyzable access patterns occurred mostly in the OCEAN program. In OCEAN, the loops which the Range Test could not parallelize were in the form of pattern `B` from Figure 6.22.

Privatization As mentioned above, these cases typically involved a complicated last-value assignment, such as would be needed in the following:

```
      do 20 N=1,NTYPES
      if(OTTRAN(N).ne.TTRAN(N)) ICHECK=.TRUE.
20    if (OTROT(N).ne.TROT(N))  ICHECK=.TRUE.
```

Here, `ICHECK` is privatized by the ART, depending on the code generator to generate the proper last-value assignment. This is just a parallel search, and since it only assigns a single value to the variable, the code generator could even just parallelize the loop *without* privatizing `ICHECK`, avoiding altogether the last-value problem. It could also cause the loop to be aborted by the first processor which sets `ICHECK` to `.TRUE.`.

Reduction As mentioned above, the situations marked as reductions in this category are not normally thought of as reductions, but actually refer to a way to parallelize a loop with subscripted-subscripts without knowing the contents of the subscripting array.

An example of the situation is:

```
      do 40 I=0,NOP-1
         T(IX(IW+0)+I)=T(IX(IW+0)+I)*SDT
         T(IX(IW+1)+I)=T(IX(IW+1)+I)*SDT
         T(IX(IW+2)+I)=T(IX(IW+2)+I)*SDT
   40 continue
```

The ART generates both a run-time monotonicity check for `IX` and a potential reduction for `T`. These accesses could be considered a reduction if `IX` contains the same value in more than one location, since the references are in the reduction form and there would be a dependence in the loop. If there is no overlap in the `IX` array, however, the loop can be run as a completely parallel loop. The code generator would have to recognize these situations and decide whether it was worth the expense to test `IX` at run-time for choosing between the reduction and the completely parallel loops.

Run-time This notation refers to cases where the "old" Polaris did not parallelize a loop, but the ART did so with a run-time test involved. The various things which could be tested at run-time are monotonicity, the overlap condition, the classification condition, the correctness condition, and a condition which could prove that one condition implies another.

8.3.2 Overall Results

The results show that the Access Region Test matches the results of the old Polaris, and in some cases, surpasses them. All the cases where the old Polaris can parallelize and the ART cannot are fixable by the addition of demand-driven analysis, the implementation of conditional classification, or an increased-precision range analysis.

Some of the cases where the Access Region Test can parallelize a loop are simply beyond the capabilities of the Range Test, the Omega Test and the compilers whose dependence analysis is based on Fourier-Motzkin linear constraint solvers. In addition, the run-time dependence condition extraction capabilities of the Access Region Test have been shown to enable parallelism where Polaris was unable to find it before.

8.4 Parallelizing TFFT2

The TFFT2 code, from a preliminary version of the SPEC cfp95 benchmarks, is a good showcase for the capabilities of the Access Region Test. The main loop nests consist of five nested loops in which the call tree is four subroutines deep. Refer to Figure 8.4 for the overall `call` structure of one of the main loop nests. Simplified versions of the six routines considered in this section are presented in Figures 8.5 through 8.7.

The following difficulties for compilers are caused by the TFFT2 program structure:

1. Array reshaping occurs at the call to `FFTZ2`, where parameter six changes from one to four dimensions, and the dimensions are dependent on an outer loop index. The first dimension of the callee's array grows exponentially with the outer loop index and the third dimension shrinks exponentially with the outer loop index.

2. The inner-most loops in this part of the call tree, those inside `FFTZ2`, have loop bounds involving 2 raised to the power of an outer loop index (the loop in `CFFTZWORK`).

3. The starting point of the reshaped four-dimensional array within `CFFTZWORK` depends on the loop index.

4. The loop bounds on the two inner-most loops (within `FFTZ2`) both are exponential expressions involving a formal parameter derived from the loop index of an outer loop in the calling routine. So, they are "triangular" loops in the sense that their upper bounds depend on an outer loop index, but the shape of the iteration space would not look triangular.

The Range Test cannot parallelize any loops in the main loop nests, not even the inner-most loops of the bottom-most subroutine in the call tree. The Access Region Test can parallelize all of the loops which are parallel in this nest, up to the fourth loop level. The fifth (outer-most) loop is not parallel.

8.4.1 Access Region Summaries of the Code

This analysis will start at the bottom of the call tree and proceed up, summarizing the accesses along the way. Array `U` is read-only in this part of the program, so it won't be considered.

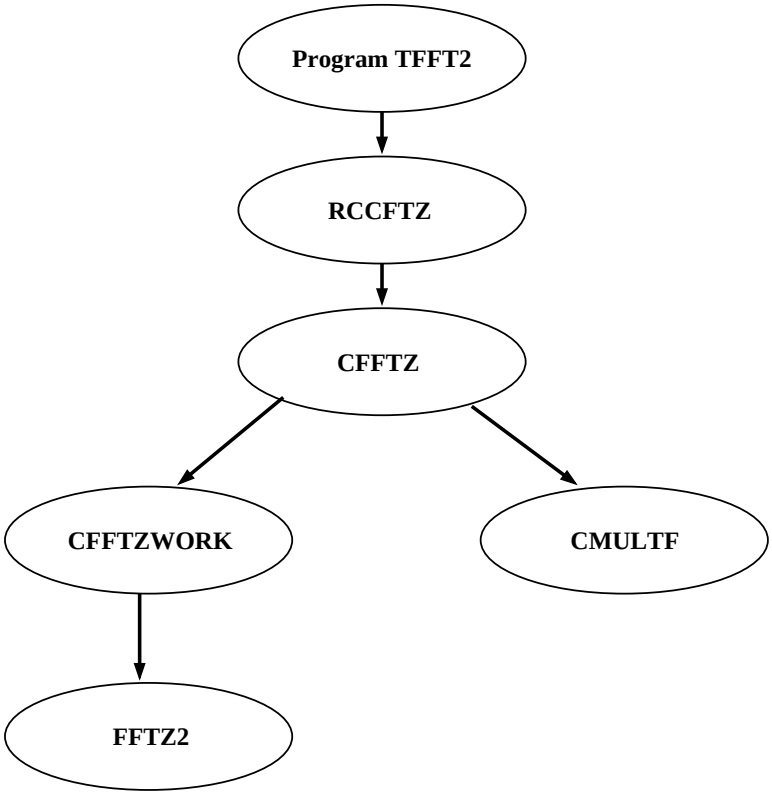

Figure 8.4: Call tree for the branch of TFFT2 described in this thesis.

```
program TFFT2
common S(2**20),U(2*2**20), X(2**20+2), Y(2**20+2)

do II=1,IT   <-- X(2^M-1 + 1) + 0,   Y(2^M-1 + 1) + 0
   .
call RCFFTZ (1,M,U,X,Y)
   .
end do
   .
end
subroutine RCFFTZ (IS, M, U, X, Y)  <-- X(2^M-1 + 1) + 0,   Y(2^M-1 + 1) + 0
dimension U(1), X(1), Y(1)
  .
call CFFTZ (IS, M-1, U, Y, X)
  .
end
```

Figure 8.5: Highest level routines - main program and RCFFTZ (simplified).

```
subroutine CFFTZ (IS, M, U, X, Y)
dimension U(1), X(1), Y(1)
  .
do I=0,2**(M/2)-1  <--  ${}^{w}\mathcal{X}^1_{2^{1+M-M/2}-1}+0, \quad {}_{\|}\mathcal{Y}^1_{2^{M+1}-1}+0$

  .
  call CFFTZWORK (IS, M-M/2, U(1+3*2**(1+M)/2), Y(1+I*2**(1+M-M/2)), X)
  .
end do
do I=0,2**(M-M/2)  <--  ${}^{r}_{\|}\mathcal{X}^1_{2^{M+1}-1}+0, \quad {}^{w}_{\|}\mathcal{Y}^1_{2^{M+1}-1}+0$
  .
  call CMULTF (IS,N1,U(1+2**(1+M)/2 + I*2**(1+M/2)),
               X(1+I*2**(1+M/2)), Y(1+I*2**(1+M/2)))
  .
end do
do I=0,2**(M-M/2)-1  <--  ${}^{w}\mathcal{X}^1_{2^{1+M/2}-1}+0, \quad {}_{\|}\mathcal{Y}^1_{2^{M+1}-1}+0$
  call CFFTZWORK (IS,M/2, U(1+7*2**(1+M)/4), Y(1+I*2**(1+M/2)), X)
end do
end
subroutine CFFTZWORK (IS, M, U, X, Y)  <--  $\mathcal{X}^1_{2^{M+1}-1}+0, \quad {}^{w}\mathcal{Y}^1_{2^{M+1}-1}+0$
dimension 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
end
```

Figure 8.6: Middle level routines CFFTZ and CFFTZWORK (simplified).

```
subroutine FFTZ2 (IS, L, M, U, X, Y)  ⟵ ∥^r X^1_{2^{M+1}-1}+0,   ∥^w Y^1_{2^{M+1}-1}+0
dimension U(*), X(*), 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
end
subroutine CMULTF (IS, N, U, X, Y)  ⟵ ∥^r X^1_{2·N1-1}+0,   ∥^w Y^1_{2·N1-1}+0
dimension U(*), X(*), Y(*)

do I=1,N
   Y(I) = U(2*I-1)*X(I) - U(2*I)*X(I+N)
   Y(I+N) = U(2*I-1)*X(I+N) + U(2*I)*X(I)
end do
end
```

Figure 8.7: Lowest level routines FFTZ2 and CMULTF (simplified).

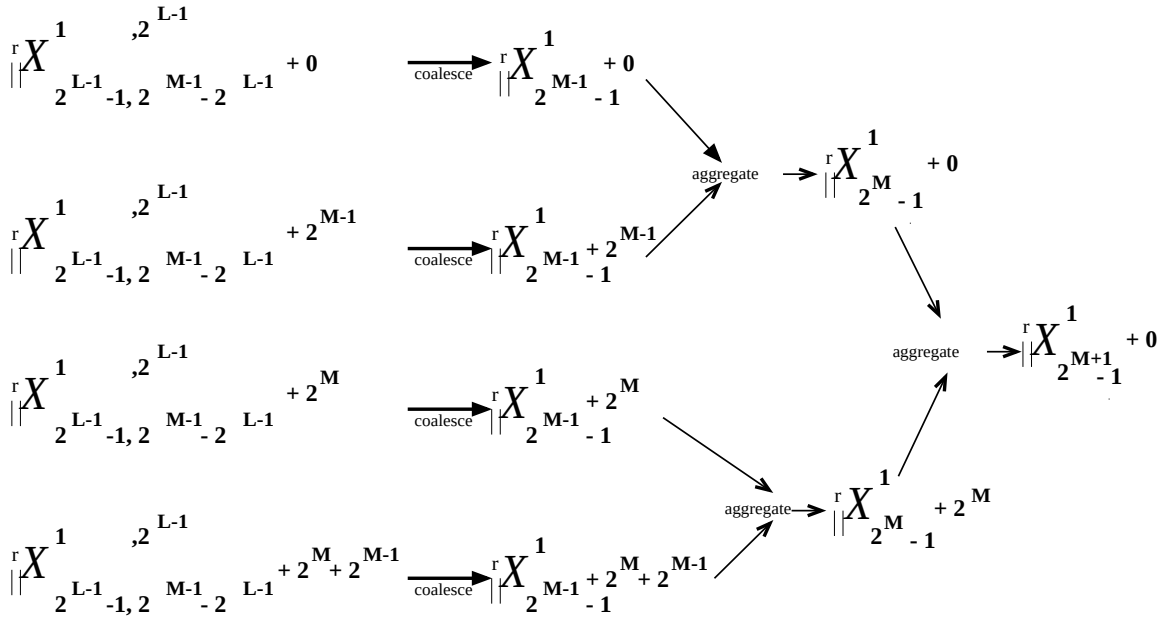

Figure 8.8: Combination of access descriptors for variable X in routine FFTZ2.

8.4.1.1 Leaf Routines FFTZ2 and CMULTF

Starting in routine `FFTZ2`, there are four accesses to both array `X` and array `Y`. Even though array `X` is declared one-dimensional and array `Y` is declared four-dimensional, the accesses to both are two-dimensional since there are two nested loops in this routine.

The references to `X` (all read accesses) in the inner-most loop are:

```
= 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)
```

The index `K` drives a stride-one access within `X`, while the index `I` has a coefficient of `2**(L-1)` in each case, so it drives an access dimension with a stride of `2**(L-1)`. The loop bounds determine the span of each dimension, so the resulting summaries of the four accesses within the outermost loop in routine **FFTZ2** become $\overset{r}{\|}\mathcal{X}^{1,2^{L-1}}_{2^{L-1}-1,2^{M-1}-2^{L-1}}+0$, $\overset{r}{\|}\mathcal{X}^{1,2^{L-1}}_{2^{L-1}-1,2^{M-1}-2^{L-1}}+2^{M}$, $\overset{r}{\|}\mathcal{X}^{1,2^{L-1}}_{2^{L-1}-1,2^{M-1}-2^{L-1}}+2^{M-1}$, and $\overset{r}{\|}\mathcal{X}^{1,2^{L-1}}_{2^{L-1}-1,2^{M-1}-2^{L-1}}+2^{M}+2^{M-1}$.

Referring to Figure 8.8, notice that each of these regions can be coalesced, then aggregation can be applied in two stages, to produce the final combined result $\overset{r}{\|}\mathcal{X}^{1}_{2^{m+1}-1}+0$.

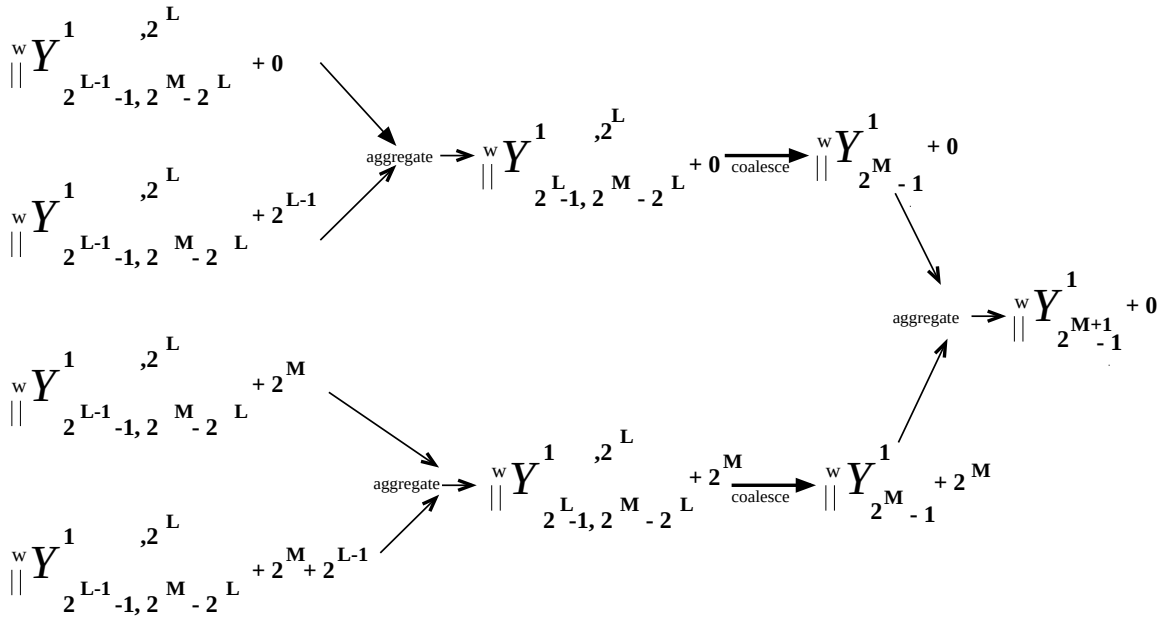

Figure 8.9: Combination of access descriptors for variable Y in routine FFTZ2.

The accesses to `Y` (which are all writes in **FFTZ2**) may be similarly summarized. The raw accesses are:

```
Y(k, 0, i, 0) =
Y(k, 0, i, 1) =
Y(k, 1, i, 0) =
Y(k, 1, i, 1) =
```

Summarizing these accesses to the outermost loop in **FFTZ2**, the following access descriptors result: ${}^{w}_{\|}\mathcal{Y}^{1,2^l}_{2^{l-1}-1,2^m-2^l}+0$, ${}^{w}_{\|}\mathcal{Y}^{1,2^l}_{2^{l-1}-1,2^m-2^l}+2^m$, ${}^{w}_{\|}\mathcal{Y}^{1,2^l}_{2^{l-1}-1,2^m-2^l}+2^{l-1}$, and ${}^{w}_{\|}\mathcal{Y}^{1,2^l}_{2^{l-1}-1,2^m-2^l}+2^m+2^{l-1}$. As shown in Figure 8.9, first aggregation can be done between pairs of access descriptors, then coalescing simplifies the two results, and finally aggregation can be applied again, to get the final result ${}^{w}_{\|}\mathcal{Y}^{1}_{2^{m+1}-1}+0$.

The net result is that for the routine **FFTZ2**, argument `X` is read with region ${}^{r}_{\|}\mathcal{X}^{1}_{2^{m+1}-1}+0$ and argument `Y` is written with region ${}^{w}_{\|}\mathcal{Y}^{1}_{2^{m+1}-1}+0$. This is a surprising result since the argument `L` does not affect either region, even though at first glance the code within **FFTZ2** in Figure 8.7 would make it seem otherwise. The powerful simplification afforded by the aggregation and coalescing operations exposes the true nature of the accesses within **FFTZ2**.

Next, consider the routine **CMULTF** (refer again to Figure 8.7), which is another leaf in the call tree. The access patterns are fairly simple, compared to those in **FFTZ2**. `Y` is written in two places, one with a descriptor of $\overset{w}{\|}\mathcal{Y}^1_{N-1}{+}0$ and the other with a descriptor of $\overset{w}{\|}\mathcal{Y}^1_{N-1}{+}N$. Aggregation can be used to fuse these together as $\overset{w}{\|}\mathcal{Y}^1_{2N-1}{+}0$. The same operations produce a read-only access descriptor for `X` of $\overset{r}{\|}\mathcal{X}^1_{2N-1}{+}0$.

8.4.1.2 Middle-level Routines CFFTZWORK and CFFTZ

Next, consider subroutine **CFFTZWORK**, where **FFTZ2** is called (refer to Figure 8.6). Since the access regions for the fifth and sixth parameters to **FFTZ2** depend only on the value of the third parameter (`M`) and that value does not vary within the `L0` loop, and the base offset aggregation and coalescing operations exposes the true nature of the accesses within **FFTZ2**.

Next, consider the routine **CMULTF** (refer again to Figure 8.7), which is another leaf in the call tree. The access patterns are fairly simple, compared to those in **FFTZ2**. `Y` is written in two places, one with a descriptor of $\overset{w}{\|}\mathcal{Y}^1_{N-1}{+}0$ and the other with a descriptor of $\overset{w}{\|}\mathcal{Y}^1_{N-1}{+}N$. Aggregation can be used to fuse these together as $\overset{w}{\|}\mathcal{Y}^1_{2N-1}{+}0$. The same operations produce a read-only access descriptor for `X` of $\overset{r}{\|}\mathcal{X}^1_{2N-1}{+}0$.

8.4.1.3 Middle-level Routines CFFTZWORK and CFFTZ

Next, consider subroutine **CFFTZWORK**, where **FFTZ2** is called (refer to Figure 8.6). Since the access regions for the fifth and sixth parameters to **FFTZ2** depend only on the value of the third parameter (`M`) and that value does not vary within the `L0` loop, and the base offset of each parameter does not vary, the same region is being accessed in each iteration for each call site. This causes the access regions from the subroutine to lose their "no-overlap" characteristic when they are summarized to the `L0` loop.

Both calls use the same value of `M`, so identical regions are accessed in both calls, the only difference being that the parameters `X` and `Y` are reversed between the two calls. The parameter `Y` is "write-first" in the first call, then the same region is "read-only" in the second call. This gives the summary for `Y` the "write-first" characteristic, since the whole region read within `Y` is written first. The parameter `X` is "read-only" during the first call, then "write-first" during the second call, making it "read-write" when summarized to the `L0` loop in **CFFTZWORK**.

Next, consider subroutine **CFFTZ** (again referring to Figure 8.6), where there are three loops to deal with. The first calls **CFFTZWORK**:

```
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+(-1)*(m/2))), x)
end do
```

The array `X` is passed to **CFFTZWORK** in the subscript position which was found to have the "write-first" characteristic. Translating parameter values, the access descriptor at the call site for `X` is ${}^{w}_{\parallel}\mathcal{X}^{1}_{2^{m-m/2+1}-1}+0$. Summarizing this to the outer loop causes it to lose "no-overlap", since the access descriptor does not involve `i` at all.

The array `Y` is passed in the position which is read first, then written. The starting address of the `Y` parameter which is sent to the subroutine depends on the value of the index (`i`) of the surrounding loop. The access descriptor of the `call` itself, pulled from the subroutine summary and translated is $\mathcal{Y}^{1}_{2^{m-m/2+1}-1}+i\cdot 2^{m-m/2+1}$. Summarizing that to the level of the outer loop gives the access descriptor ${}_{\parallel}\mathcal{Y}^{1,2^{m-m/2+1}}_{2^{m-m/2+1}-1,2^{m+1}-2^{m-m/2+1}}+0$. Applying coalescing to that descriptor results in the simplified descriptor ${}_{\parallel}\mathcal{Y}^{1}_{2^{m+1}-1}+0$. Note that there is no overlap between the intervals read and written in different iterations of the outer `i` loop. So, the access region for `Y` can be marked "no-overlap".

The second loop in **CFFTZ** makes a call to **CMULTF**:

```
do I = 0, 2**(M-M/2)-1
 call CMULTF(IS, N1, U(1+2**M+I*2**(1+M/2)), X(1+I*2**(1+M/2)), Y(1+I*2**(1+M/2)))
end do
```

In **CMULTF**, both `X` and `Y` are passed with the same starting point in the call statement and the same region is accessed for both within **CMULTF**, so the same analysis applies to both. From the summary of **CMULTF**, the access region for `Y` can be translated to the `CFFTZ` context due to the `call` statement, which is ${}^{w}_{\parallel}\mathcal{Y}^{1}_{2\cdot n1-1}+i\cdot 2^{1+m/2}$. Summarizing this to the outer loop, it becomes ${}^{w}_{\parallel}\mathcal{Y}^{1,2^{1+m/2}}_{2\cdot n1-1,2^{m+1}-2^{1+m/2}}+0$. This generates a no-overlap sequence as long as $2\cdot n1-1<2^{1+m/2}$ (the no-overlap condition for summarization). Since $n1$ is set to $2^{m/2}$ earlier in this routine, making $2\cdot n1-1$ equal $2^{1+m/2}-1$, that is true. The coalesced version of the access descriptor for `Y` is ${}^{w}_{\parallel}\mathcal{Y}^{1}_{2^{m+1}-1}+0$. So, the access to `X` is of the same form: ${}^{r}_{\parallel}\mathcal{X}^{1}_{2^{m+1}-1}+0$. The only difference is that `X` is read-only, while `Y` is write-only.

The third loop in **CFFTZ** contains another call to **CFFTZWORK**:

```
do i = 0, 2**(M-M/2)-1
 call CFFTZWORK(IS, M/2, U(1+(7*2**(1+M))/4), Y(1+I*2**(1+M/2)), X)
end do
```

The access descriptor for `Y` at the call site is $\mathcal{Y}^1_{2^{1+m/2}}+1+i\cdot 2^{1+m/2}$. Summarizing to the outer loop, it becomes ${}_{\|}\mathcal{Y}^{1,2^{1+m/2}}_{2^{1+m/2}-1,2^{m+1}-2^{1+m/2}}+0$. This can be coalesced to ${}_{\|}\mathcal{Y}^1_{2^{m+1}-1}+0$, since it satisfies the no-overlap condition. The access descriptor for `X` can be derived from the routine-level summary in **CFFTZWORK**, becoming ${}^{w}\mathcal{X}^1_{2^{1+m/2}-1}+0$.

The access regions must next be summarized at the routine level of **CFFTZ**. For the array `X`, the summaries of the three loops: ${}^{w}\mathcal{X}^1_{2^{m-m/2+1}-1}+0$, ${}^{r}\mathcal{X}^1_{2^{m+1}-1}+0$, and ${}^{w}\mathcal{X}^1_{2^{1+m/2}-1}+0$ must be combined. They may be combined as $\mathcal{X}^1_{2^{m+1}-1}+0$.

8.4.1.4 Highest Level Routines

In the routine **RCFFTZ**, the parameters are translated due to the call to **CFFTZ**, so the descriptors turn out to be $\mathcal{X}^1_{2^m-1}+0$ and $\mathcal{Y}^1_{2^m-1}+0$ (both carrying an overlap, indicating dependence).

Finally, in the highest level in the call tree, the main program for **TFFT2**, a loop surrounds the call site of **RCFFTZ**, but the loop index of that loop is not involved in the call, so each iteration accesses the same locations. The access patterns summarized to the outer loop would be identical: $\mathcal{X}^1_{2^m-1}+0$ and $\mathcal{Y}^1_{2^m-1}+0$ (again, both carry an overlap, indicating dependence).

9 THE CONTRIBUTIONS OF THIS THESIS AND FUTURE WORK

This thesis describes the Access Region Descriptor (ARD), which can represent memory accesses more precisely than traditional triplet notation. The ARD distills program code for array accesses into the information critical to several important compiler analysis techniques. This descriptor is not constrained by the declared dimensions of an array and therefore eliminates the array reshaping problem when translating array accesses from one program context to another. The ease with which accesses can be translated across procedure boundaries makes it an ideal vehicle for interprocedural analysis. The ARDs can precisely represent non-affine expressions, making possible parallelization in situations where no other known dependence test has been successful. The precision of the ARD and its simplification operations make it a useful vessel within algorithms for generating data movement messages.

This thesis also describes Memory Classification Analysis (MCA) which distills the important information about the relationships between memory accesses in a program. These relationships are embodied in the summary sets of ARDs produced by MCA. It is shown herein that this information is exactly what is needed for doing dependence analysis, and not coincidentally, also for privatization, reduction and induction analysis, three of the most important parallelism-enabling transformations used in parallelizers.

The Access Region Test (ART) is also described. This is the algorithm which uses the summary sets of ARDs produced by MCA to parallelize programs by combining privatization, reduction and induction analysis. It is shown, through argument and experiment, to be more powerful than a combination of the Range Test [8] and the GCD Test.

The various components of MCA are able to generate conditions when they cannot be certain of their analysis. These conditions can be combined within the code generation pass for doing deeper, demand-driven analysis, or for generating a runtime dependence test. As shown in Section 6.1.13, the classification condition can also be used to assist in generating conditional prefetch commands.

The combination of the ARD, the ART and MCA can precisely represent subscripted-subscript access within loops, and can sometimes parallelize loops containing such accesses.

Experiments are reported here which show that these techniques can parallelize more loops than can the Range Test. All loops without a dependence in the routine TFFT2 can be parallelized despite its heavy use of non-affine subscript expressions, nested loops containing subroutine calls, complicated array reshaping, and triangular loops. The difficult and important FTRVMT/`do_109` loop in the Perfect benchmark routine OCEAN can also be fully parallelized (representing about 40% of the serial execution time). The TURB3D code from the SPEC FP'95 benchmarks, which grew from 2100 lines to 20,000 lines with full subroutine inlining and required 20 hours to process by Polaris, can be parallelized by the algorithms described here in just 5 minutes.

9.1 Future Work

This work suggests a number of interesting avenues for future research.

9.1.1 Code Generation Pass

First, the code generation pass should be implemented. This will be an interesting component in and of itself. Designing an algorithm which can collect the run-time test conditions and decide what to do based on them would be an interesting task. Such an algorithm would have to balance the effort desired by the user with information it can gather about the "importance" of the loop in question and the conditions it needs to prove, to decide how to proceed.

The code generation pass will need to have several demand-driven deeper analysis algorithms at its disposal for proving conditions, generating the induction closed forms, and gathering definitions of subscripting arrays.

One goal would be to handle irregular problems by using the demand-driven deeper analysis to discover facts about subscripting arrays, and to generate runtime tests for whatever facts cannot be discovered.

Also, an interesting sub-task would be to explore the question of how to combine the various runtime test conditions into the proper runtime test, which is both minimal and appropriate.

9.1.2 Symbolic Infrastructure

Secondly, the symbolic infrastructure of Polaris must be improved. There are still bits of "obvious" information which Polaris does not infer from the structure of the program. Symbolic expression simplification can also be improved. The whole expression infrastructure could be optimized. An interesting question is "How should symbolic expression manipulation be structured to make it powerful enough to support MCA and the ART, yet fast enough to use in a commercial compiler?".

9.1.3 Communication Generation

Thirdly, issues in generating precise data movement should be studied. The precise memory access representation and defined simplification techniques of the ARD makes it a natural for use in generating data messages for moving data between processors. The conditions which are produced by the various components within MCA could be used to generate different messages under different conditions.

BIBLIOGRAPHY

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[2] S. Amarasinghe. *Parallelizing Compiler Techniques Based on Linear Inequalities*. PhD thesis, Stanford University, January 1997.

[3] J. Backus. The History of FORTRAN I, II, and III. *Annals of the History of Computing*, 1(1), July 1979.

[4] 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.

[5] Utpal Banerjee. *Loop Transformations for Restructuring Compilers*. Kluwer Academic Publishers, Norwell, Massachussats, 1993.

[6] Utpal Banerjee. *Dependence Analysis*. Kluwer Academic Publishers, Norwell, MA, 1997.

[7] 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.

[8] William Blume. *Symbolic Analysis Techniques for Effective Automatic Parallelization*. PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, June 1995.

[9] William Blume and Rudolf Eigenmann. Symbolic Range Propagation. *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

[10] William Blume and Rudolf Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. *Proceedings of Supercomputing '94*, pages 528–537, November 1994.

[11] M. Burke and R. Cytron. Interprocedural Dependence Analysis and Parallelization. *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 162–175, July 1986.

[12] D. Callahan and K. Kennedy. Analysis of Interprocedural Side Effects in a Parallel Programming Environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[13] B. Creusillet and F. Irigoin. Interprocedural Array Region Analyses. In *Lecture Notes in Computer Science*. Springer Verlag, New York, New York, August 1995.

[14] B. Creusillet and F. Irigoin. Exact vs. Approximate Array Region Analyses. In *Lecture Notes in Computer Science*. Springer Verlag, New York, New York, August 1996.

[15] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, pages 451–490, October 1991.

[16] G. Dantzig and B.Eaves. Fourier-Motzkin Elimination and its Dual. *Journal of Combinatorial Theory*, pages 288–297, 1973.

[17] 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.

[18] Dennis Gannon and Ko-Yang Wang. Using AI Techniques to Resturcture Programs for Different Parallel Architectures. In Kai Hwang; Douglas Degroot, editor, *AI and Supercomputing Systems*, 1987.

[19] J. Grout. Inline Expansion for the Polaris Research Compiler. Master's thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, May 1995.

[20] J. Gu, Z. Li, and G. Lee. Symbolic Array Dataflow Analysis for Array Privatization and Program Parallelization. *Proceedings of Supercomputing '95*, December 1995.

[21] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[22] M. Hall, B. Murphy, S. Amarasinghe, S. Liao, and M. Lam. Interprocedural Analysis for Parallelization. *Proceedings of 8th Workshop on Language and Compilers for Parallel Computing*, August 1995.

[23] M. Hall, B. Murphy, S. Amarasinghe, S. Liao, and M. Lam. Detecting Coarse-grain Parallelism Using An Interprocedural Parallelizing Compiler. *Proceedings of Supercomputing '95*, December 1995.

[24] P. Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Rice University, May 1994.

[25] Chris Huson. An Inline Subroutine Expander For Parafrase. Master's thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, December 1982.

[26] Z. Li and P. Yew. Efficient Interprocedural Analysis for Program Parallelization and Restructuring. *Proceedings of the SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, July 1988.

[27] 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.

[28] D. Maydan, S. Amarasinghe, and M. Lam. Array Data-Flow Analysis and its Use in Array Privatization. *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languges*, January 1993.

[29] 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.

[30] Y. Paek, J. Hoeflinger, and D. Padua. Simplification of Array Access Patterns for Compiler Optimizations. In *Proceedings of the ACM SIGPLAN 98 Conference on Programming Language Design and Implementation*, June 1998.

[31] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B Leung, and D. Schouten. The Structure of Parafrase-2: An Advanced Parallelizing Compiler for C and Fortran. In *Lecture Notes in Computer Science*. Springer Verlag, New York, New York, August 1990.

[32] 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.

[33] W. Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8), August 1992.

[34] Z. Shen, Z. Li, and P. Yew. An Empirical Study of Fortran Programs for Parallelizing Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):350–364, July 1990.

[35] R. Triolet, F. Irigoin, and P. Feautrier. Direct Parallelization of Call Statements. *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 176–185, 1986.

[36] P. Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis*. PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, May 1995.

[37] P. Tu and D. Padua. Automatic Array Privatization. August 1993.

[38] 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.

VITA

Jay Philip Hoeflinger was born on December 24, 1951 in Belleville, Illinois. He began his long association with the University of Illinois in 1970 as a college freshman. He received the degree of Bachelor of Science in Computer Science in 1974, then a Master of Science degree in Computer Science in 1977. He worked at the Illinois State Geological Survey as a scientific programmer from 1972 through 1980. He joined Colwell Systems in Champaign, Illinois in 1980 to work as a systems programmer, then as Manager of the Systems Programmers. In 1985, he left Colwell to return to the University of Illinois, specifically at the newly-formed Center for Supercomputing Research and Development. He worked there in Professor Padua's compiler group until the Center's demise, at which time the group became a part of the University's Coordinated Sciences Laboratory. In 1997, the group moved to the Department of Computer Science. In 1995, Jay decided to enroll in the Ph.D. program at the University of Illinois. In August, 1998, he began work as a Senior Research Scientist at the Center for Simulation of Advanced Rockets at the University of Illinois.