Automatically Parallelizing the Main Loops in TFFT2 Via Access Region Analysis

Jay Hoeflinger

August 27, 1997

1 Introduction

This report will analyze one of the two main branches of the call tree of the SPEC program **TFFT2**. The other main branch is very similar, and the same type of analysis applies to it. First, we describe the *access region* representation used to describe very precisely the accesses to arrays in the program. Next, we show a strategy for using access regions for parallelizing programs, which involves

1. the precise interprocedural summarization of the array accesses made in the program,
2. the use of these summaries to privatize arrays and parallelize loops.

Finally, we describe the structure of the TFFT2 program and describe in detail how we can use our representation and parallelization strategy for parallelizing it.

The access region notation represents the array elements which are accessed due to the nested loops surrounding the access site. Operations on these access summaries will be used to combine them, to simplify their form, and to determine whether sets of accesses overlap.

2 Problems with triplet notation

Quite often within compilers, the set of array elements accessed by a section of code has been represented in *triplet notation*. Triplet notation has the form:

```
lower : upper : stride
```

which represents a sequence of integers, starting at *lower* and proceeding to *upper* in steps of *stride*. When triplet notation is used to represent accesses to an array, the accesses are typically expressed by using triplets in array subscript positions, for example:

```
A( 1:100:2 , 10:20:1, 1:3:1 )
```

The development of the access region notation was motivated by our observation that triplet notation is insufficient to accurately describe the values for array subscripts in some programs (such as **TFFT2**), where subscripts like

```
X(1+K+I*2**(L-1))
```

are found (in which `I`, `K`, and `L` are indices of the surrounding loops).

3 Access region notation

A slight generalization of triplet notation, which we call the access region notation, *is* able to represent such an access, however.

To summarize an array reference in a nested loop, we must first identify the set of *basic induction variables* [1], one per loop, in the loop nest. We will refer to such a basic induction variable for a loop as the *loop index*.

In the access region notation, we summarize the effect of each loop index on an array subscript expression by the *stride* and *span* due to the index, defined as follows. The difference between the values of the subscript expression at successive values of a loop index is the *stride* due to that index. The difference between the value of the subscript expression evaluated at the last value of the index and the value of the subscript expression evaluated at the first value of the index we call the *span* due to that index. The stride and span together form a *dimension* of the access, due to a single index. Note that the access dimensionality of an array may be different from the declared dimensionality of the array.

The set of elements referenced by a subscript expression within nested loops can be completely characterized by expressing the effect of all loop indices. The place where the access starts is called the "starting point", which is expressed as an element offset from the first element of the array (offset zero refers to the first element of the array).

So, we can describe the set of array elements being accessed through an array reference inside a nested loop with an *access descriptor*, expressed in the following form for the array `A`:

$$\mathcal{A}^{stride_1, stride_2, \cdots}_{span_1, span_2, \cdots} + start$$

where $stride_i$ and $span_i$ form the i-th access dimension, due to loop index i.

In addition, we will characterize a region as being either "write-first", "read-only" or "read/write" and either "overlap" or "no-overlap". We will refer to these as the *access characteristics* of the region.

Write-first refers to the situation where, for every element in the access region, a write occurs as the first access. Reads and other writes may follow or not, in any order, it doesn't matter. *Read-only* is a label for a region in which the only accesses are reads. *Read/write* is used for any other situation.

A *no-overlap* region, with respect to a given dimension, refers to the situation where different values of the loop index cause non-overlapping sets of locations to be accessed. All other regions are called *overlap* regions.

We will *annotate* the access region descriptor on the left upper part of the variable name with r to mean read-only, or w to mean write-first. No letter in that position refers to read/write. On the left lower part of the variable name, we will use "||" to refer to no-overlap. Absence of the "||" implies that one or more locations is repeated within the set of locations accessed.

For the purposes of this paper, we will refer to the ith stride as δ_i, the ith span as σ_i, and the starting point as τ, so the general form of a d-dimensional access region for array `A` is:

$$\begin{smallmatrix}[r,w]\\[\ \|\ ]\end{smallmatrix}\mathcal{A}^{\delta_1, \delta_2, \cdots \delta_d}_{\sigma_1, \sigma_2, \cdots \sigma_d} + \tau.$$

We will *normalize* a given access descriptor such that δ_i and σ_i are positive quantities. When the stride is negative, the starting point will be adjusted to make this possible. For instance, an access with a stride of -1 of 100 elements starting at element offset 99 (99, 98, 97, $\cdots$, 0) can be equivalently represented as a stride 1 access of 100 elements starting at element offset 0 (0, 1, 2, $\cdots$, 99). The order is not important, only which elements are accessed is important. So,

$$\mathcal{A}^{-1}_{-99}+99 \quad \text{will be normalized as} \quad \mathcal{A}^{1}_{99}+0.$$

As an example of the use of an access descriptor to represent the accesses in a nested loop, consider the access summary of the array A in the following loop nest:

```
REAL A(0:100)

DO I=1,N,2
   DO J=1,6
      A(3*I+J) = . . .
   END DO
END DO
```

The stride due to the I loop is 6, while the stride due to the J loop is 1. In addition, the starting point for the access structure is the element of A addressed when I and J have their first values (both being 1 in this case), and the offset is therefore 4. So, the access region representation of the accesses to A is:

$$\begin{matrix} w \\ \| \end{matrix} \mathcal{A}^{1,6}_{5,6(N-1)}+4.$$

4 Operations with access regions

4.1 Constructing the original access descriptor

The original construction of an access descriptor from an array reference in a program is done by constructing the access dimensions and a starting point expression. Usually, in the context of a single statement, a single array reference accesses a single location, in which the stride and span may be represented by (0,0). In other cases, the array reference might be in a CALL statement or a vector assignment statement, in which cases the appropriate dimensions for the access must be used.

The starting point expression is constructed by applying the array storage ordering function defined for the language being used (*column-major* for Fortran and *row-major* for C) along with the declared dimensions of the array, to the subscripting expression, giving an expression for the offset of the access from the first element of the array. It is important to remember to subtract the lower bound of each declared dimension, so that the result represents an offset from the beginning of the array. For example, given the following Fortran code fragment:

```
REAL D(128,128)

... D(I,2*J) ...
```

the access descriptor formed would be

$$\mathcal{D}^{0}_{0} + (I-1) + 2 * 128 * (J-1).$$

When the access descriptor is constructed for an array reference which is a parameter in a CALL statement, the union of all access descriptors for the formal parameter in that position are copied to the call site, the formal parameter names are translated to the corresponding actual parameter names, and the new starting point expression (as described above) is added to the starting point expression of each formal parameter descriptor.

4.1.1 Constructing the descriptor annotations

The annotations for the descriptor are constructed in a straightforward manner. If the reference is to a single array element, it obviously doesn't overlap and so is marked with the no-overlap characteristic. If the single-element reference is a read, the descriptor is marked read-only and if it is a write, it is marked write-first.

For the parameters to a `CALL` statement, all descriptor annotations are simply copied from the original descriptors.

The sections which follow, describing the operations aggregation, summarization, and coalescing, each include descriptions of how to calculate the access characteristics.

4.2 Aggregation

Another operation which is useful for access descriptors is *aggregation*. Aggregation is an operation which may be performed on two access descriptors which are *conjunctive*.

A formal definition of **conjunctive regions** is as follows:

Definition 1 [Conjunctive Regions] *Consider two d-dimensional access descriptors* $\mathbf{A} = \mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma_1,\sigma_2,\cdots,\sigma_d} + \tau$ *and* $\mathbf{A}' = \mathcal{A}^{\delta'_1,\delta'_2,\cdots,\delta'_d}_{\sigma'_1,\sigma'_2,\cdots,\sigma'_d} + \tau'$ *such that* $\tau \geq \tau'$. *If the dimensions of the two descriptors may be paired such that all d strides match and at least* $d-1$ *spans match, then we may select a candidate dimension from each descriptor for the aggregation. If only* $d-1$ *dimensions match, then the candidate dimensions are those for which the span did not match. If all d dimensions match, then any pair of dimensions with the same stride, one from each descriptor, may be selected as candidate dimensions. Let us refer to the candidate dimensions selected, as the stride/span pairs* $(\delta_{i_p}, \sigma_{i_p})$ *from descriptor* **A** *and* $(\delta'_{i_q}, \sigma'_{i_q})$ *from descriptor* **A'**. *If the following conditions hold*

1. δ_{i_p} *divides* $\tau - \tau'$, *and*
2. $\tau - \tau' \leq \sigma'_{i_q} + \delta'_{i_q}$,

then the descriptors are **conjunctive**.

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

Definition 2 [Aggregation Operation] *Let* **A** *and* **A'** *be the access descriptors which are conjunctive in Definition 1 with the two stride/span pairs* $(\sigma_{i_p}, \delta_{i_p})$ *and* $(\sigma'_{i_q}, \delta'_{i_q})$. *Then,* $\mathbf{A} \cup \mathbf{A}' = \mathcal{A}^{\delta_1,\cdots,\delta_{i_p},\cdots,\delta_d}_{\sigma_1,\cdots,\sigma_{i_p}+t,\cdots,\sigma_d} + \tau'$ *where* $t = \tau - \tau'$.

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

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

These access descriptors can be shown to be conjunctive, according to Definition 1 (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:

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

4.2.1 Propagating access characteristics within aggregation

The order of read and write accesses must be determined to evaluate the read/write access characteristic. If the accesses are only reads or only writes, then it is easy to mark the proper characteristic. However, when both reads and writes occur, we must determine which comes first and the extent of each. If it can be determined that even a single location is read before being written, the descriptor must be marked read/write. But if it can be determined that the write accesses happen before the reads and that the write access region *covers* the read access region, then we can mark the descriptor write-first. This subject has been extensively discussed in the array privatization literature, for instance in [2].

When two regions are determined to be **conjunctive**, as described in Definition 1, then the resulting descriptor can be marked with the no-overlap characteristic whenever

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

4.3 Summarization

The process of taking an access region defined by an access descriptor and forming a new access region which represents the region accessed during the execution of some outer loop is referred to as *summarizing* the access *to* the loop.

First, the stride of the access due to the particular loop index is determined. To do that, the incremented form of the descriptor's starting point expression is formed, and the original starting point expression is subtracted. Then, the difference is multiplied by the stride of the index. For instance, for the Fortran loop

$$ \texttt{DO I =}\ \mathit{init}, \mathit{high}, \mathit{loopstride} $$

if the starting point expression is a function of `I` (f(I)), then an incremented starting point expression is formed by replacing `I` by `I+1` and the original expression is subtracted from it, then multiplied by *loopstride*:

$$ \textbf{Stride}\ \text{due to}\ \texttt{I}: \qquad (f(I+1) - f(I)) \cdot \mathit{loopstride} $$

Next, the span of the access is calculated by multiplying the stride by the number of steps taken within the loop. the span would be

$$ \textbf{Span}: \qquad \textbf{Stride} \cdot \textbf{Steps} $$

The number of steps is calculated by forming an expression one less than the number of iterations. For the above `DO`-loop, the expression would be

$$ \textbf{Steps}: \qquad \texttt{MAX}((\mathit{high} - \mathit{init})/\mathit{loopstride}, 0). $$

Within loops where it can be proven that there is at least one iteration, the expression can be simplified to

$$ \textbf{Steps}: \qquad (\mathit{high} - \mathit{init})/\mathit{loopstride}. $$

Then, the new access descriptor is made by copying the dimensions of the original descriptor and adding a new dimension consisting of the stride and span just calculated. If the stride can be determined to be negative, the starting point expression of the new descriptor is formed by substituting the last value of the loop index, for the loop index variable, into the original starting point expression. Otherwise, the first value of the loop index is substituted for the loop index variable in the original starting point expression.

4.3.1 Propagating access characteristics within summarization

If the dimensions are sorted by strides, δ_1 being the smallest stride and δ_d being the largest stride, then if

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

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.

The read/write characteristics are not changed by summarization.

4.4 Coalescing

The next operation for the access region notation is one which determines when it is possible to reduce the number of dimensions in an access region descriptor. If it can be shown that two dimensions of an access descriptor fit together seamlessly (with no gaps), then the number of dimensions in the descriptor can be reduced by one.

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

Definition 3 [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 by eliminating the k-th dimension, we can form the access descriptor:*

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

This condition holds in the above example, where

$$\overset{w}{\parallel}\mathcal{A}^{1,6}_{5,6(N-1)}+4 \qquad \text{can be written as:} \qquad \overset{w}{\parallel}\mathcal{A}^{1}_{6(N-1)+5}+4$$

The coalescing operation serves to simplify the descriptor of the access pattern, making it easier to do operations with such descriptors.

4.4.1 Propagating access characteristics within coalescing

When a region can be coalesced, according to Definition 3 above, then it may be further determined that it has the no-overlap characteristic when, for the two dimensions j and k being coalesced,

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

Coalescing does not affect the read/write characteristics of the descriptor.

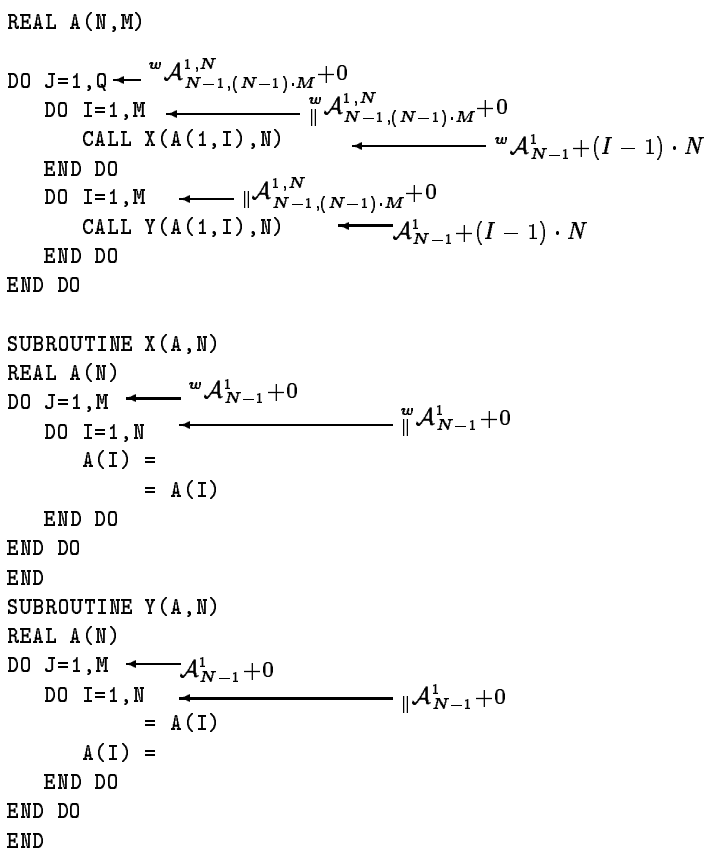

Figure 1: Code example for propagation of access characteristics

4.5 Parallelization using access characteristics

Once the access regions are computed for all loops and all routines are marked with the access characteristics, the loops may be parallelized based on the access characteristics alone. If an array is accessed in a loop and its descriptor is marked with either the read-only, write-first, or no-overlap characteristics, all potential dependences due to that array may be eliminated. This is obvious in the case of read-only. The no-overlap characteristic means that different values of the current loop index never cause a single array element to be accessed more than once, eliminating the chance for any dependence. The write-first characteristic means that the portion of the array represented by the access region may be privatized, eliminating all dependences associated with it.

As an example of propagating these characteristics interprocedurally, consider the code in Figure 1. The access descriptor for routine **X** is ${}^{w}\mathcal{A}^{1}_{N-1}+0$, and for routine **Y** is $\mathcal{A}^{1}_{N-1}+0$. When these are translated through their call sites, the outer I loop around each site causes the access to be no-overlap (${}^{w}_{\parallel}\mathcal{A}^{1,N}_{N-1,(N-1)\cdot M}+0$ and ${}_{\parallel}\mathcal{A}^{1,N}_{N-1,(N-1)\cdot M}+0$, for **X** and **Y** respectively). Then, when they are aggregated within the outer J loop in the main program, we find that the write-first access preceeds the read-write access, with the same extent, so the result is a write-first access: ${}^{w}\mathcal{A}^{1,N}_{N-1,(N-1)\cdot M}+0$ for the J loop. This means that the outer loop may be parallelized since the whole array A may be privatized to the outer loop, eliminating all apparent dependences for it in the call tree structure.

5 General parallelization strategy

The general strategy for parallelizing a program using access regions is to construct the access descriptors for the program by traversing the loop structure and the call-tree bottom-up, then to transform the program into parallel form by privatizing arrays and parallelizing loops based on the access descriptors from the top-down.

5.1 Bottom-up access descriptor construction

To begin the bottom-up construction pass, the call tree is topologically sorted, then the sorted list of routines is traversed from the leaf routines to the main routine. During the traversal, within a given routine, we scan the statement list and for each array reference found, we first make an access region descriptor representing the elements of that array referenced by the statement alone (ignoring any surrounding loops). Then, we traverse the loop structure from the point of reference to the procedure level, applying *region abstraction* to successively outer loops and finally to the whole routine.

Region abstraction involves applying the following three operations (already described in Section 4), within a given loop or, in the last step, within the whole procedure:

1. [**Summarization**] Summarize the access regions remaining after aggregation by applying the loop bounds to the access descriptors (there is nothing for this operation to do at the procedure level).
2. [**Aggregation**] Attempt to combine separate access regions for a single array into fewer access regions by applying aggregation to conjunctive regions.
3. [**Coalescing**] Attempt to simplify the summarized access regions by applying coalescing where appropriate. If at least one access descriptor was coalesced in this step, when this step is finished, go back to step 2.

5.2 Top-down parallelization

Once the access regions are computed for the whole program, the parallelization process proceeds down the call tree from the main program to the leaf routines. The routines are visited via the topologically-sorted list used in the bottom-up phase, but in the reverse order.

Within each routine, the statement list is scanned and whenever a loop is encountered, the list of access descriptors attached to it is checked for access annotations. As stated in Section 4.5, dependences due to an array reference may be removed when any of the access characteristics "no-overlap", "write-first", or "read-only" are marked on the descriptor. Whenever the write-first annotation occurs, the array involved is privatized, and code to copy the last-value of the private array to the shared array is added. If any access region which is attached to the loop lacks an annotation, then the loop must be serialized, or checked further by a different dependence testing technique.

6 Parallelizing TFFT2

6.1 The TFFT2 code structure

The **TFFT2** code structure is straightforward, but several difficulties must be addressed by compilers which try to parallelize this code. Please refer to Figure 2 for the overall structure of this branch of TFFT2. Simplified versions of the six routines considered in this report are presented in Figures 3 through 5. The following difficulties for compilers are caused by the TFFT2 program structure:

1. There are five code levels due to the subroutine calls inside loops.
2. Array reshaping occurs at the call to `FFTZ2`, where parameter six changes from one to four dimensions, and the dimensions change with new values of the outer loop bounds. The first dimension of the callee's

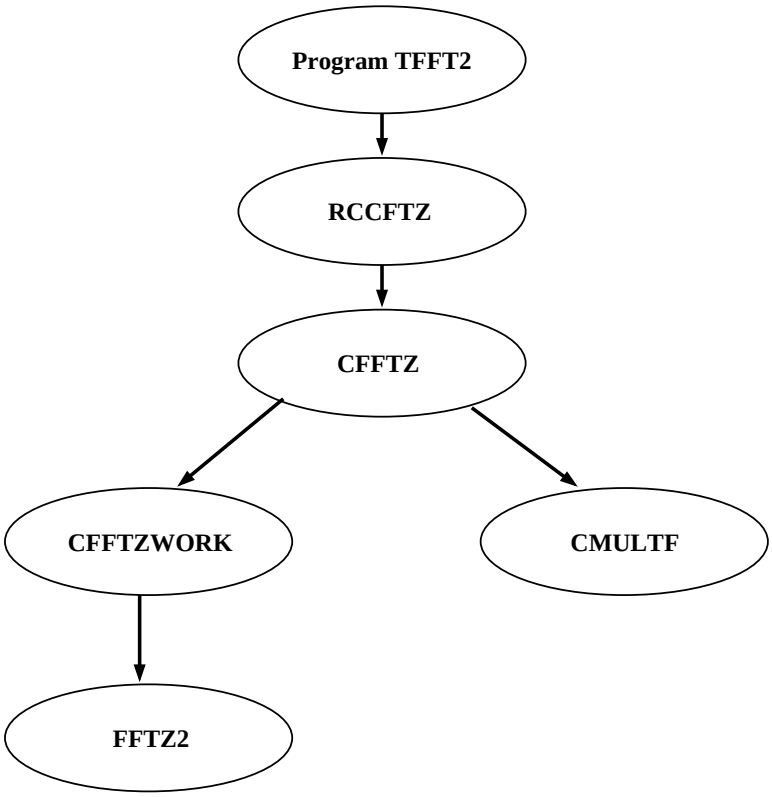

Figure 2: Call tree for the branch of TFFT2 described in this paper

```
PROGRAM TFFT2
COMMON S(2**20),U(2*2**20), X(2**20+2), Y(2**20+2)

DO II=1,IT <-- X(1,2**M-1+0),   Y(1,2**M-1+0)
   .
CALL RCFFTZ (1,M,U,X,Y)
   .
END DO
   .
END
SUBROUTINE RCFFTZ (IS, M, U, X, Y) <-- X(1,2**M-1+0),   Y(1,2**M-1+0)
DIMENSION U(1), X(1), Y(1)
  .
CALL CFFTZ (IS, M-1, U, Y, X)
  .
END
```

Figure 3: 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 X(1)_{2^{1+M-M/2}-1}+0,     ‖ 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 ‖ X(1)_{2^{M+1}-1}+0,     w ‖ 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 X(1)_{2^{1+M/2}-1}+0,     ‖ 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)    <---  X(1)_{2^{M+1}-1}+0,     w 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 4: 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/2))
         = X(1+K+I*2**(L-1)+2**(M/2)+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 5: Lowest level routines FFTZ2 and CMULTF (simplified)

array grows exponentially with the outer loop bound and the third dimension shrinks exponentially with the outer loop bound.

3. 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).

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

6.2 Access region summaries of the code (Bottom up pass)

We will start at the bottom of the call tree and proceed up, summarizing the accesses as we go. Array U is only read in this part of the program, so we won't consider it in the rest of this summary.

6.2.1 Leaf routines FFTZ2 and CMULTF

For the routine FFTZ2, refer to Figure 6 for a diagram of the operations on the access regions for variable X, and to Figure 7 for a diagram of the operations on the access regions for variable Y.

Starting in routine FFTZ2, we find 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 6, notice that each of these regions can be coalesced, then aggregation can be applied in two stages, to produce the final combined result $\overset{r}{\|}\mathcal{X}^{1}_{2^{m+1}-1}+0$.

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**, we get the following access descriptors: $\overset{w}{\|}\mathcal{Y}^{1,2^l}_{2^{l-1}-1,2^m-2^l}+0$, $\overset{w}{\|}\mathcal{Y}^{1,2^l}_{2^{l-1}-1,2^m-2^l}+2^m$, $\overset{w}{\|}\mathcal{Y}^{1,2^l}_{2^{l-1}-1,2^m-2^l}+2^{l-1}$, and $\overset{w}{\|}\mathcal{Y}^{1,2^l}_{2^{l-1}-1,2^m-2^l}+2^m+2^{l-1}$. As shown in Figure 7, first we can do aggregation between pairs of access descriptors, then coalescing simplifies the two results, and finally aggregation can be applied again, to get the final result $\overset{w}{\|}\mathcal{Y}^{1}_{2^{m+1}-1}+0$.

The net result is that for the routine **FFTZ2**, argument X is read with region $\overset{r}{\|}\mathcal{X}^{1}_{2^{m+1}-1}+0$ and argument Y is written with region $\overset{w}{\|}\mathcal{Y}^{1}_{2^{m+1}-1}+0$. This is a surprising result since we see that the argument L does not affect either region, even though at first glance the code within **FFTZ2** in Figure 5 would make it seem otherwise. The powerful simplification afforded by the aggregation and coalescing operations allows us to see the true nature of the accesses within **FFTZ2**.

Next, we consider the routine **CMULTF** (refer again to Figure 5), 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 $\overset{w}{\|}\mathcal{Y}^{1}_{N-1}+N$. We can use aggregation to fuse these together as $\overset{w}{\|}\mathcal{Y}^{1}_{2N-1}+0$. The same operations produce a READ access descriptor for X of $\overset{r}{\|}\mathcal{X}^{1}_{2N-1}+0$.

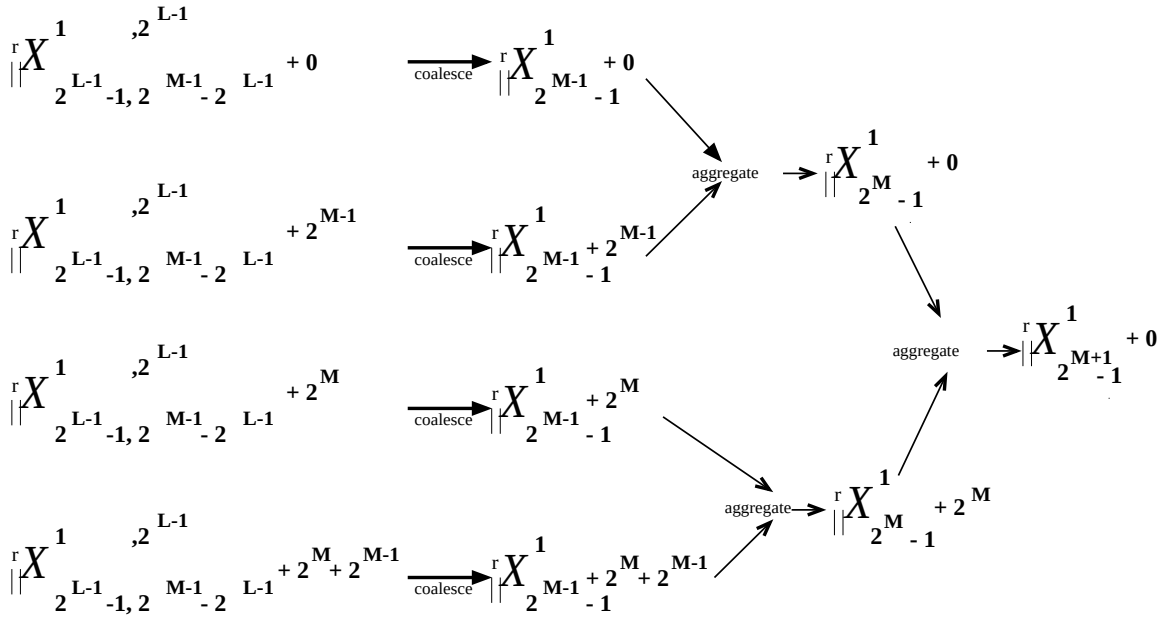

Figure 6: Combination of access descriptors for variable X in routine FFTZ2

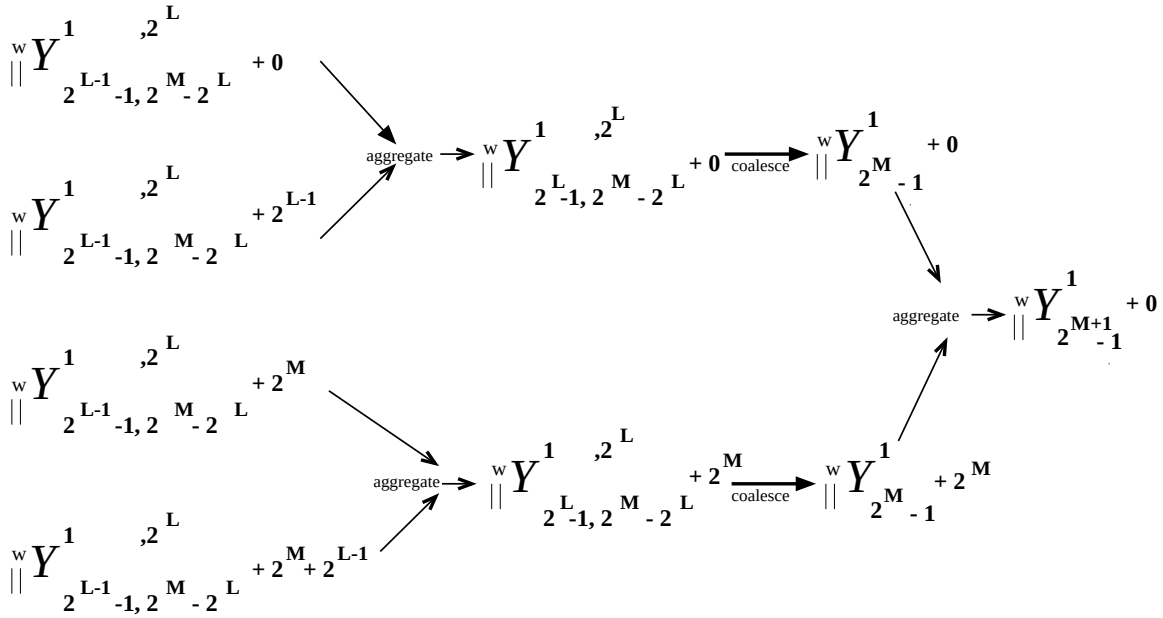

Figure 7: Combination of access descriptors for variable Y in routine FFTZ2

6.2.2 Middle-level routines CFFTZWORK and CFFTZ

Working our way up to the top of the call tree, the next stop is subroutine **CFFTZWORK**, where **FFTZ2** is called (refer to Figure 4). Since the access regions for the fifth and sixth parameters to **FFTZ2** depend only on the value of the third parameter and that value (M) does not vary within the L0 loop, and the starting point 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.

We can easily see that since both calls use the same value of M, 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 **CFFTZWORK**.

The next stop up the call tree is in subroutine **CFFTZ** (again refering to Figure 4), 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 we found to have the "write-first" characteristic. Translating parameter values, we find that the access descriptor at the call site for X is $\overset{w}{\|}\mathcal{X}^1_{2^{m-m/2+1}-1}+0$. 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 $\|\mathcal{Y}^{1,2^{m-m/2+1}}_{2^{m-m/2+1}-1,2^{m+1}-2^{m-m/2+1}}+0$. Applying coalescing to that descriptor results in the simplified descriptor $\|\mathcal{Y}^1_{2^{m+1}-1}+0$. 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**, we can construct the access region for Y in the CALL statement, which is $\overset{w}{\|}\mathcal{Y}^1_{2\cdot n1-1}+i\cdot 2^{1+m/2}$. Summarizing this to the outer loop, it becomes $\overset{w}{\|}\mathcal{Y}^{1,2^{1+m/2}}_{2\cdot n1-1,2^{m+1}-2^{1+m/2}}+0$. 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 $\overset{w}{\|}\mathcal{Y}^1_{2^{m+1}-1}+0$. So, the access to X is of the same form: $\overset{r}{\|}\mathcal{X}^1_{2^{m+1}-1}+0$. 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 easily 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, we must combine 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 ${}^{w}\mathcal{X}^{1}_{2^{1+m/2}-1}+0$. They may be combined as $\mathcal{X}^{1}_{2^{m+1}-1}+0$.

For the array Y, the three summaries are ${}_{\|}\mathcal{Y}^{1}_{2^{m+1}-1}+0$, ${}^{w}_{\|}\mathcal{Y}^{1}_{2^{m+1}-1}+0$, ${}_{\|}\mathcal{Y}^{1}_{2^{m+1}-1}+0$. These regions may be combined as $\mathcal{Y}^{1}_{2^{m+1}-1}+0$.

6.2.3 Highest level routines

In the routine **RCFFTZ**, we translate the parameters in the call to **CFFTZ**, so the descriptors turn out to be $\mathcal{X}^{1}_{2^{m}-1}+0$ and $\mathcal{Y}^{1}_{2^{m}-1}+0$.

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 in 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 $\mathcal{Y}^{1}_{2^{m}-1}+0$.

6.3 Interprocedural parallelization (Top-down pass)

The interprocedural parallelization pass starts at the **TFFT2** main program. The summaries on the loop in the main program indicate overlapping read/write regions for both X and Y, so it is not possible to parallelize that loop.

Descending into the call tree, through **RCFFTZ** (which has no loops), to **CFFTZ**, in the first loop we find all parameters to the call to be read-only except Y and X. Y was summarized as "non-overlapping read/write", and X is summarized as "write-first", so, by privatizing X, the first loop is parallelizable. The second loop calls **CMULTF** with all parameters read-only except Y, which is summarized as "non-overlapping write-first", therefore it is parallelizable. The third loop is much the same as the first loop, with X "write-first" (and therefore privatizable) and Y "non-overlapping read/write", so it is also parallelizable.

We can continue descending the call tree doing the same kind of analysis if the target machine can use more than one level of parallelization.

7 Conclusion

We have presented a new way of representing array accesses within a program section, called access region notation. This representation accurately represents the access pattern driven by a set of nested loops surrounding an array reference, makes it easy to translate the access pattern through a subroutine call, and makes use of powerful operations to both combine and simplify the representations. This representation is more accurate than triplet notation, can easily summarize accesses across procedure boundaries, and can represent all the information needed to eliminate possible dependences within loops.

We have presented a two-pass method for interprocedurally parallelizing a program, based on the access region notation. The first interprocedural pass is a bottom-up pass on the call tree for constructing access descriptors, and marking them as to their characteristics, such as "non-overlapping write-first" or "read-only". The top-down interprocedural pass simply checks the descriptors for annotations, using them to parallelize at the outer-most level possible.

Finally, we show in detail how the the SPEC benchmark program TFFT2 can be parallelized using these techniques, despite the difficult problems which TFFT2 presents to a compiler.

References

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

[2] Peng Tu. Automatic Array Privatization and Demand-Driven Symbolic Analysis. PhD Thesis, University of Illinois at Urbana-Champaign, 1995.