

# Validity of Interprocedural Data Remapping

Michał Cierniak  
Wei Li

Technical Report 642  
Department of Computer Science  
University of Rochester  
Rochester, NY 14627  
{cierniak,wei}@cs.rochester.edu

November 1996

## Abstract

Programming languages like Fortran or C define exactly the layout of array elements in memory. Programmers often use that definition to access the same memory via variables declared as arrays of different types. This is done, for instance, to access a slice of larger array or to access an data as a linear array, so that subscript computation is simpler giving performance improvements on some architectures and allowing the use of single procedure to perform some operation (e.g., copying) on variables of different types. For many real programs this practice makes changing the layout of an array impossible without violating the semantics of the program since the same memory block may be accessed via a variable of a different type — such an access will now receive wrong array elements.

On the other hand, changing array layout is often necessary to obtain good parallel performance or even to improve sequential performance by providing better cache locality. The techniques that achieve that range from manually inserted array distribution directives to automatic compiler transformations.

Our paper demonstrates that the problem of changing array layouts in the presence of multiple variables of different types accessing the same memory can be solved with our algorithms for 1) detecting *overlapping arrays*, 2) using procedure cloning to reduce overlapping, 3) array type coercion, and 4) code structure recovery. We describe the algorithms used in our compiler and present experimental results showing speedups which are not possible with other techniques.

# 1 Introduction

In this paper we describe how to decide whether changing the mapping of an array affects other data in a computer program. Our discussion will refer to Fortran programs, but similar techniques can be applied to other programming languages. It may be desirable to change a layout of an array for one of the following reasons:

- Data remapping may be used to improve cache re-use for accesses to that array.
- Data distribution may be specified to improve performance of a program parallelized using HPF directives.

We will use the term “array layout” to refer to both array mappings and distributions.

The impact of array remapping and distribution may be large. Choosing the right mapping on a shared-memory machine may improve performance of both sequential and parallel programs [5, 1, 9, 11]. The importance of the right array layout is even greater for NUMA (non-uniform memory access) parallel machines. Modern NUMA machines like Origin2000 provide mechanisms for specifying array layout [14]. In many cases data locality may be improved without changing the default array layout by *loop transformations* [13, 12, 15, 4]. However in some cases loop transformations cannot be applied due to data dependencies or synchronization in explicitly parallel program.

To change an array layout without changing the meaning of the program we must know if two arrays may access the same memory areas. The natural name for this property would be “aliasing.” In our paper we use the term “overlapping.” We decided against using the word “aliasing,” since it has an accepted meaning in programming language research which is not as wide as our meaning of overlapping. We say that two arrays overlap even if those two arrays cannot exist at the same time. Aliasing on the other hand describes a property at a given point of a program execution.

There exist techniques which let the compiler automatically choose<sup>1</sup> an array mapping [5, 1, 9, 11] or distribution [8, 10]. These techniques lose much of their effectiveness in the presence of aliasing or more generally in the presence of overlapping. Given an expression which references an array element, FORTRAN 77 calculates the address of a memory location which contains that element using fixed, language specified rules (column-major layout). When the compiler changes the layout of an array, some action must be performed to make references to the same memory area via a different variable consistent with the new mapping. This is necessary to ensure that the transformed program is semantically equivalent to original program. There are two basic approaches to this problem:

- Perform dynamic remapping (redistribution) and ensure that during the time that the layout of an array is different from the standard FORTRAN 77 layout, the array is not accessed via some other name. To ensure correctness, we still have to perform alias analysis. That alias analysis is much simpler than the complete overlap analysis tackled in our paper. This technique has a drawback of extra overhead caused by the dynamic remapping.

---

<sup>1</sup>The same problem exists of course if the mapping or distribution is specified explicitly by the programmer. Therefore our techniques may be applied with equal effectiveness as an aid for programmers who are willing to explicitly handle array layout.

- Changes are performed statically, at compile-time. This approach is preferred whenever possible since no extra overhead is incurred. The drawback of this technique is that for some programs it is not possible to disambiguate array overlapping.

Hybrid solutions are of course possible. We present an algorithm for performing static changes to array layouts by interprocedural analysis of overlapping arrays. Our algorithm generates an *Array Overlap Graph*. In that graph each vertex corresponds to one array name. There is an edge between two vertices if and only if the two arrays overlap.

Some vertices may be also marked as *taboo*. Arrays which are taboo cannot be safely remapped. The meaning of the graph is that the layout of an array affects all arrays which belong to the same connected component. Note that this is a conservative approach. In the following sections we give examples in which two arrays which belong to the same connected component could be remapped independently. We also present techniques which break those “false” paths between vertices of an Array Overlap Graph.

We observe that it is desirable to make the connected components as small as possible. Their size can be minimized by accurate analysis and by selective procedure cloning. Our algorithm also coerces the types of all arrays to the same type. We choose an array in a component which has most structure (the largest number of dimensions) and coerce all other arrays to that type. The reason for this is that data remapping and distribution optimizations work best when given most flexibility.

Rather than starting by presenting a complete algorithm, we will first show a simple case and then gradually refine the algorithm to handle more realistic cases. Section 2 shows how to construct an Array Overlap Graph in a simple way. Section 3 describes the use of selective procedure cloning to improve the accuracy of the graph. To use an Array Overlap Graph for data layout changes, it is desirable to coerce all overlapping arrays (i.e., all arrays in a connected component) to the type with “most structure.” We present an algorithm for type coercion in Section 4. Quality of the optimized code can be improved by *code recovery*. This technique is described in Section 5. We choose a set of benchmarks whose locality is not optimal and which cannot be optimized by loop transformations due to data dependencies and which cannot be optimized by data remapping without our techniques. We have implemented techniques in our compiler based on the Polaris compiler infrastructure [3]. Experimental results which show improvements in speed for those benchmarks are presented in Section 6. For the lack of space in this extended abstract, we do not include complete, formal algorithms for our analysis and transformations.

## 2 Array Overlap Graph

The main idea of array overlap analysis is similar to existing techniques for detecting aliases. For straightforward array overlap analysis we could simply use one of the alias analysis algorithms if we extended it so that arrays in different scopes are being considered. Alias analysis of FORTRAN 77 programs is simpler than general alias analysis due to the absence of pointers.

In our model we assume that array overlap can be introduced in the following ways:

- Aliasing between a formal parameter and an actual parameter.
- Aliasing between global arrays via the use of common blocks.

- Aliasing introduced by the EQUIVALENCE statement of FORTRAN 77.

We represent overlapping arrays with an *array overlap graph*. Each vertex of the graph corresponds to an array (a local variable of a procedure, a formal parameters, or a global variable). An edge is inserted when one of the three situation described above is detected during program analysis.

A connected component in our graph represents a conservative approximation of all possible names for arrays which are allocated in the same region of memory and share information. Remapping of an array will *possibly* affect other arrays in the same connected component. The uncertainty implied by the word “possibly” is caused by the fact that our graph is a conservative, safe approximation of the “overlaps” relation. Each edge corresponds to a real overlapping, but the transitivity does not necessarily imply real overlapping. Let us call the algorithm described in this section the *Simple Algorithm*. Our experiments have shown that the accuracy of the Simple Algorithm is insufficient to perform effective layout changes of arrays in real programs. There are two main reasons for the inaccuracies introduced in the graph by transitivity:

- Different arrays may be passed as parameters to a procedure and will end up in the same connected component via the corresponding formal parameter.
- Arrays may have different sizes. For instance two slices of a bigger array may directly not overlap, but since each of them overlaps with that bigger array, our algorithm assumes that all arrays in the connected component overlap.

The former problem is much more common in real programs, so we describe it more detail and show our solution in Section 3. The latter problem can be illustrated by the example shown in Figure 1. Arrays A and B occupy non-overlapping areas in memory. The simple algorithm presented

```

PROGRAM P
REAL WORK(200)
CALL Q(WORK(1), 10, 10)
CALL R(WORK(101), 100)
STOP
END

SUBROUTINE Q(A, N, M)
REAL A(N,M)
...
END

SUBROUTINE R(B, K)
REAL B(K)
...
END

```

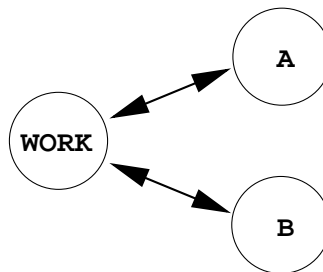


Figure 2: Array Overlap Graph

Figure 1: Example which shows that the analysis may be inaccurate.

in this section will build a graph with two edges: (A, WORK) and (B, WORK) as shown in Figure 2.

By transitivity, we will conclude that A and B are overlapping. While for many programs, we have to consider A and B as “overlapping”<sup>2</sup> even though they do not share memory, in some cases it is possible to use interprocedural array section data-flow analysis to prove that A and B do not share any information. For the lack of space, we do not give further details in this extended abstract.

### 3 Improving Graph Accuracy by Procedure Cloning

To make our Array Overlap Graph useful, we have to deal with a very common inaccuracy present in the graphs created by the Simple Algorithm from Section 2. In virtually all applications of significant size, it is easy to find a pattern like the one shown in Figure 3 (this code fragment, extracted from the NASA7 benchmark, is part of SPEC CFP92 [7] benchmark suite). As can be

```

SUBROUTINE CHOTST (ER, FP, TM)
  PARAMETER ( IDA=250, NMAT=250, M=4, N=40, NRHS=3 )
  COMMON /ARRAYS/ A(0:IDA, -M:0, 0:N),
$   B(0:NRHS, 0:NMAT, 0:N),
$   AX(0:IDA, -M:0, 0:N),
$   BX(0:NRHS, 0:NMAT, 0:N)
  ...
  LA = (IDA+1) * (M+1) * (N+1)
  LB = (NRHS+1) * (NMAT+1) * (N+1)
  ...
  CALL COPY (LA, AX, A)
  CALL COPY (LB, BX, B)
  ...
END

SUBROUTINE COPY (N, A, B)
  REAL*8 A(N), B(N)
  DO 100 I = 1, N
    B(I) = A(I)
100 CONTINUE
  RETURN
END

```

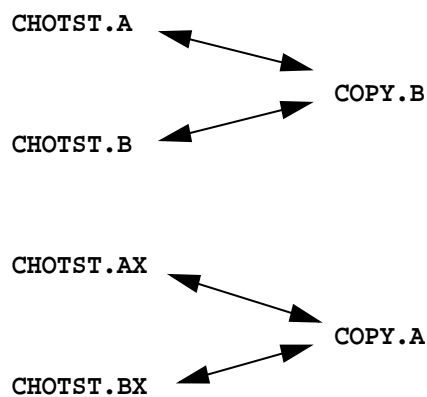


Figure 4: Array Overlap Graph

Figure 3: Code fragment from CHOLSKY.

seen in Figure 4, the Array Overlap Graph created by the Simple Algorithm does not allow us to remap the array CHOTST . B<sup>3</sup> without remapping CHOTST . A at the same time.

Given this code, it is indeed illegal to change the mapping of CHOTST . B without other changes to the program. The analysis of the program shows that the best locality of memory accesses could be achieved by changing the layout of CHOTST . B and leaving CHOTST . A in the column-major

<sup>2</sup>They are overlapping in the sense that if we change the layout of A then we have to change the layout of WORK and consequently we must change the layout of B.

<sup>3</sup>Our notation prefixes an array name with the procedure name to make clear which array we are referring to. Thus CHOTST . B means array B declared in the subroutine CHOTST.

mapping (this is due to accesses which are not shown in the included code fragment). We solve this problem by *procedure cloning*. For our working example from Figure 3, we create an exact clone of the procedure `COPY` and change one of the calls to use this new version of `COPY`. The transformed code is shown in Figure 5. We can see in the corresponding Array Overlap Graph

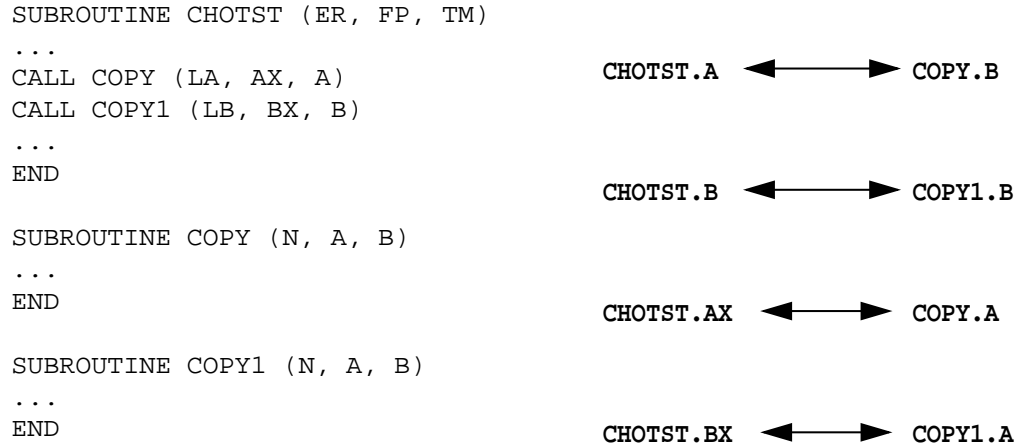


Figure 5: Code fragment from `CHOLSKY` after cloning.

Figure 6: Array Overlap Graph

(shown in Figure 6) that the dependency between `CHOTST.B` and `CHOTST.A` is gone.

A hasty conclusion from this simple example may be that we should clone all procedures at all call sites. Even a shallow examination of this idea shows that not only is complete cloning not possible for languages which allow recursive procedure calls, but it is not desirable to unnecessarily increase the code size of an application even if recursion is not a problem. Larger code size means increased storage requirements and increased load time, but most importantly, it means a larger memory footprint of the application. This of course will likely degrade the performance at all levels of memory hierarchies due to increased cache miss ratio and increased virtual memory paging. A small increase in code size is justified if the benefit of changing array layouts outweighs the associated cost.

To alleviate this problem we have developed an algorithm for selective cloning which only clones procedures when doing so will improve the accuracy of the Array Overlap Graph. To further reduce the code size, after the data remapping (or distribution) optimizations have been performed we examine, for every procedure, all clones and merge those which resulted in the same mapping for all its formal parameters and local variables, and which call equivalent clones at every corresponding call site.

The algorithm uses the concept of an *overlap graph signature* (OGS) which is assigned to every procedure. The basic idea is that the OGS describes the overlapping relation between formal parameters to the procedure. At every call site we find the overlapping relation between actual parameters and search the list of clones to the callee which have the same relation between its formal parameters. If we find one, we use the clone, otherwise we create a new clone and set its OGS to what we just computed for the actual parameters.

## 4 Coercing the Types of Overlapping Arrays

So far we have shown how to find a good approximation for the overlapping relationship. We have to do more to change the layout of arrays effectively. This section shows our approach to coercing all arrays in a connected component of the Array Overlap Graph to the same type. Since all overlapping arrays are coerced to the same type, we can guarantee correctness of data transformations by applying the same transformation to all arrays in a connected component of the Array Overlap Graph.

Consider again the code in Figure 5. The graph in Figure 6 shows that we can safely change the layout of `CHOTST.B` as long as we change `COPY1.B` accordingly. But those arrays have different types: `CHOTST.B` has three dimensions and `COPY1.B` — only one.

For a given connected component of the Array Overlap Graph we find an array with “most structure.” In our approach by “most structure” we mean “the largest number of dimensions.” Note that it is possible that overlapping arrays have *incompatible* types. Consider as an example  $P(3, 7)$  and  $Q(5, 4)$  — the only common type we could coerce those references to would be a linear (one-dimensional) array.<sup>4</sup> This however would make the array not suitable for layout changes. In that case our analysis gives up and marks all arrays in a connected component with incompatible types as *taboo*. Taboo arrays cannot be subject to any layout changes. This situation is of course possible, but — not surprisingly — very uncommon in real programs.

Coercing array types is performed by propagating types along edges until all arrays have the same type. Note that for aliasing caused by procedure parameter aliasing, the type may be propagated either from the caller to the callee or from the callee to the caller. For other types of aliasing (EQUIVALENCE statement and common blocks) the direction of type coercion is irrelevant. The rest of this section shows how the type coercion is performed. We use examples throughout the section to better convey the essence of the algorithm. All examples are taken from standard benchmarks to emphasize the relevance of the discussion to real programs.

We first consider the simpler case in which both arrays start at the same memory address. This case is described in Section 4.1. In Section 4.2 we extend the algorithm to handle partial array overlapping. Partial overlapping is common due to the practice of passing array slices to procedures which only need a portion of an array.

### 4.1 Arrays with the Same Base Address

We continue with our example from Figure 5. First we have to modify the call to `COPY1` to include all values needed to express the type of `CHOTST.B`.

We choose all subscripts of the recovered type to start from the default value for a given language<sup>5</sup>

---

<sup>4</sup>We can of course convert any array to any type by first linearizing it and then recovering the necessary structure. For our example, assume that we want to coerce  $Q$  to have the same type as  $P$ , so that the new type is now  $Q'(3, 7)$ . A reference  $Q(i, j)$  would be first linearized to  $Q'(i+5*(j-1))$  and then the reference using the new type would be  $Q'(\text{MOD}(i+5*(j-1)-1, 3)+1, (i+5*(j-1)-1)/3+1)$ . We expect that the index computation would become too expensive. The discussion is not very interesting since we have not yet seen an application which would require that type of coercion.

<sup>5</sup>This is an arbitrary decision with a goal of making the transformed source code more readable for programmers used to the particular language. Any lower bound could be used. All calculations must be made for lower bounds equal to

(for Fortran this value is 1). Using a constant lower bound saves us passing the parameters for lower bounds. In this example, the recovered type will be

```
REAL*8 B(NRHS+1, NMAT+1, N+1)
```

In general we would have to add `CHOTST.NRHS`, `CHOTST.NMAT` and `CHOTST.N` to the parameter list, but as an optimization, if array bounds are compile-time constants, we use them directly rather than passing them as parameters. In our example all those values are constant, so the new type of `COPY1.B` is

```
REAL*8 B(4, 251, 41)
```

As another optimization (which is not applicable in this example), we always check if a value is already present in the parameter list before we add a new argument.

We modify all array references by first linearizing the reference and then generating the proper subscripts by using a combination of integer division and modulo operations. In the computation, we use two concepts defined in [6]: a *mapping vector* and a *subscript range vector*. Every recovered subscript expression is obtained by first dividing the linearized subscript expression by the mapping vector entry for this dimension and then computing the remainder of the division by the subscript range vector entry for the same dimension (all those operations must be performed on zero-based subscripts). This can in general lead to subscripts which are expensive to compute, but we have developed optimizations which completely eliminate all division and modulo operations for the cases which occur in practice (see Section 6 for a partial list of programs used to evaluate our technique).

Two optimizations are used by us to simplify subscript computation:

- We eliminate the need to linearize multidimensional arrays before coercing them to the new type. This technique is described in Section 4.2.
- We recover the code structure to match the new array type. Section 5 demonstrates this technique.

## 4.2 Arrays with Different Base Addresses

Consider the example in Figure 7. This code fragment is extracted from one of the Perfect benchmarks — ADM (the same program is now part of SPEC95 as 141.apsi). We want to propagate the type of `DKZMH.DKZM` to `SMOOTH.F` and then to `HORSMT.F`. The first part follows the approach described in Section 4.1. We do not have to add new parameters to the call to `SMOOTH` since they are all already present. The reference `F(MLAG)` is converted by Polaris into `F(1+II*NX*NY)` as part of the induction variable elimination. Code structure recovery (see Section 5) fails for this loop, so we convert this reference to the new type by using division and modulo. The recovered reference (before simplification) is

```
F(MOD(1+II*NX*NY-1, NX)+1,
  MOD((1+II*NX*NY-1)/NX, NY)+1,
  (1+II*NX*NY-1)/(NX*NY)+1)
```

---

zero, so often we have to convert the subscript to a zero-based one and then after the necessary calculations, we convert it back to a one-based subscript.

```

SUBROUTINE DKZMH(DKZM,NX,NY,NZ)
REAL DKZM(NX,NY,NZ)
CALL SMOOTH(DKZM,NX,NY,NZ)
END

```

```

SUBROUTINE SMOOTH(F,NX,NY,NZ)
REAL F(1)
MLAG=1
DO II=1,NZ-2
  MLAG=MLAG+NX*NY
  CALL HORSMT(NX,NY,F(MLAG))
END DO
END

```

```

SUBROUTINE HORSMT(NX,NY,F)
REAL F(NX,NY)
DO J=1,NY
  DO I=2,NX-1
    F1=F(I,J)
  END DO
END DO
END

```

Figure 7: Code fragment from ADM before type coercion.

```

SUBROUTINE DKZMH(DKZM,NX,NY,NZ)
REAL DKZM(NX,NY,NZ)
CALL SMOOTH(DKZM,NX,NY,NZ)
END

```

```

SUBROUTINE SMOOTH(F,NX,NY,NZ)
REAL F(NX,NY,NZ)
MLAG=1
DO II=1,NZ-2
  MLAG=MLAG+NX*NY
  CALL HORSMT(NX,NY,F,1+II,NZ)
END DO
END

```

```

SUBROUTINE HORSMT(NX,NY,F,SUB,NZ)
REAL F(NX,NY,NZ)
DO J=1,NY
  DO I=2,NX-1
    F1=F(I,J,SUB)
  END DO
END DO
END

```

Figure 8: Code fragment from ADM after type coercion.

Symbolic simplification (which is also part of the Polaris compiler) yields  $F(1, 1, 1+II)$ . Therefore the new modified call statement is:

```
CALL HORSMT(NX,NY,F(1, 1, 1+II))
```

In this example the base addresses of `SMOOTH.F` and `HORSMT.F` are different. We coerce their types by modifying the call to pass the address to the start of `SMOOTH.F` rather than to a slice of it. But now we have to add new parameters to the call to pass the offset in `SMOOTH.F` which is implied by  $F(1, 1, 1+II)$ . We should pass all subscripts as additional parameters to the call. As an optimization we eliminate compile-time constants and subscripts which are equal to the lower bound in the particular dimension (recall that we always recover types so that they start from 1). In our case the first two subscripts of  $F(1, 1, 1+II)$  can be eliminated by any of the above optimizations. We only have to add  $(1+II)$  to the parameter list (this is in addition to passing `NZ` which is needed to declare the array). The new lists of actual and formal parameters are:

```

CALL HORSMT(NX,NY,F,1+II,NZ)
...
SUBROUTINE HORSMT(NX,NY,F,SUB,NZ)

```

In general, to coerce original, two-dimensional array references to the new type *and* account for the lost offset between the base addresses of the two arrays, we have to:

1. Linearize the old reference:  $F(1+(I-1)+(J-1)*NX)$ .

2. Add the offset:  $F(1 + (I-1) + (J-1) * NX + SUB * NX * NY)$ .
3. Apply the right combination of modulo and division operations and perform symbolic simplification to obtain:  $F(I, J, SUB)$ . Note that to perform this simplification we have to use the fact that the range of  $I$  is  $(2 \dots NX - 1)$  and that the range of  $J$  is  $(1 \dots NY)$ .

The recovered code is shown in Figure 8. For this example, it is possible to perform the coercion in a faster way and our implementation uses this as an optimization. Our compiler checks that the dimensions of the old type of `HORSMT.F` are identical to the two leftmost dimensions of its new type. Therefore we can leave the left two subscripts of every reference to `HORSMT.F` unchanged and simply add the missing subscript for the third dimension.

## 5 Code Structure Recovery

This section describes an optimization for a special case which is common enough to deserve a custom treatment.

Consider again the example from Figures 3 and 5. We have shown in Section 4.1 how to coerce the array `COPY1.B` to the type of `CHOTST.B`. If we then use the standard way of transforming every reference to `COPY1.B` by using division and modulo operations, we will obtain the new version of `COPY1` shown in Figure 9 (here we assumed that `COPY1.A` was coerced to the type of `CHOTST.BX`). The subscripts to both `COPY1.B` and `COPY1.A` are being calculated in an expensive

```

SUBROUTINE COPY1(N, A, B)
  INTEGER*4 I, I0, I1, I2, N
  REAL*8 A(4,251,41), B(4,251,41)
  DO I = 1, N
    B(1+MOD(I-1,4),
>    1+MOD((I-1)/4,251),
>    1+(I-1)/1004) =
>    A(1+MOD(I-1,4),
>    1+MOD((I-1)/4,251),
>    1+(I-1)/1004)
  END DO
  RETURN
END

```

Figure 9: Procedure `COPY1` without code structure recovery.

```

SUBROUTINE COPY1(N, A, B)
  INTEGER*4 I, I0, I1, I2, N
  REAL*8 A(4,251,41), B(4,251,41)
  DO I0 = 1, 41
    DO I1 = 1, 251
      DO I2 = 1, 4
        B(I2,I1,I0)=A(I2,I1,I0)
      END DO
    END DO
  END DO
  RETURN
END

```

Figure 10: Procedure `COPY1` with code structure recovery.

way. We note that trip count of the loop `I` is equal to the size of the arrays. In this special we can replace the loop nest of the depth corresponding to the number of dimensions of the original type (in our example — one) with a loop nest which matches the new type (of depth three in our example). We call this *code structure recovery*, since it is complementary to the *data structure recovery* [6]. Procedure `COPY1` after applying code structure recovery is shown in Figure 10. This special case is common enough that our compiler recognizes it and performs the necessary code transformation. Currently our implementation attempts it only if the original loop nest is of depth one, the array

type before coercion has only one dimension and at least one of the references to that array has a subscript which is equal to the loop index variable.

In our example of `CHOTST.B` and `COPY1.B` in Figure 5 our analysis first computes the sizes of both arrays. To do this, we express the size of `COPY1.B` in the terms of variables in the caller (i.e., `CHOTST`). We match the formal parameter `N` with the actual parameter `LB`. Then we use data-flow analysis to substitute the expression which is used as the actual parameter until it is expressed exclusively in terms of compile-time constants and formal parameters of `CHOTST` (since only compile-time constants and formal parameters may be used in the declaration of array dimensions). If this part of the analysis fail, we are not able to recover the code structure. Once we have expressions for the sizes of `COPY1.B` before and after coercion, we symbolically compare the two expressions. If their values are equal, we replace the original loop nest with a new one which matches the new type of the array.

Then every array reference which has a subscript equal to the original loop index is replaced by a reference with subscripts which are the loop variables of the new loop nest. At the start of the loop body we also compute the value of the variable which used to be the index variable of the original loop as a function of the new loop variables. This value is used in all other occurrences of the original loop variable. This assignment is not present in Figure 5 because it was removed by the dead code elimination phase.

Accessing multidimensional arrays may have higher costs than accessing linear arrays. Therefore in our compiler we record which arrays have been remapped during the optimization phase and all procedures which do not contain references to any remapped array are being replaced again with the original procedure. For instance in the Cholesky decomposition kernel, a fragment of which we show in Figure 5, initially array `COPY.B` is being coerced to the type of `CHOTST.A` and `COPY.A` is being coerced by `CHOTST.AX`. However, since none of those arrays is being remapped, after the optimization phase, the original version of `COPY` is being used again in the call:

```
CALL COPY (LA, AX, A)
```

## 6 Experimental Results

Our paper does not describe how to use array remapping or distribution for locality optimizations or parallelization, nevertheless we have decided to include some preliminary results we got on small (hundreds lines of source code) benchmarks. To optimize larger programs, we have to refine our algorithms for unified data and control transformations [5]. That problem is however beyond the scope of this paper. To demonstrate the benefits of our techniques, we found a set of programs which cannot be optimized by existing locality optimizations. Loop transformations cannot be applied because of data dependencies and data transformations cannot be used without the techniques described here due to array overlapping. For our benchmarks we had to apply most of the techniques described in this paper. The resulting speedups were obtained by applying unified data and control transformations [5]. We ran the programs sequentially on a DEC AlphaStation 2100 using standard data sizes supplied with the benchmarks. The standard data sizes are small for cache sizes found in contemporary computers. For instance, the primary cache of the Alpha processor in our AlphaStation has the size of 16KB. The array sizes in our benchmarks are comparable with the cache size, therefore we expect even bigger performance gains for larger data sets — especially if they exceed L2 cache sizes.

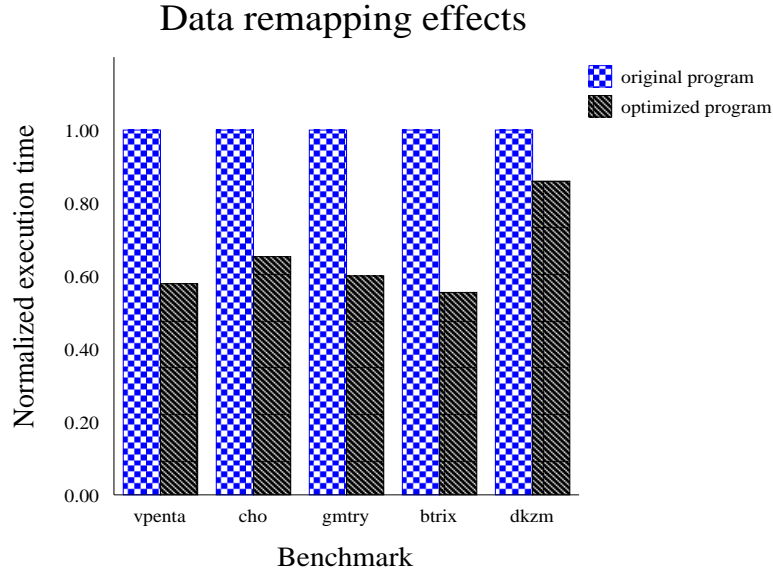


Figure 11: Speedups for selected benchmarks

Our benchmarks include four kernels from the SPEC CFP92 [7] benchmark suite: `vpenta`, `cho`, `gmtry` and `btrix`. The fifth benchmark — `dkzm` — is a code fragment extracted from one of the Perfect [2] benchmarks — ADM (the same program is now part of SPEC95 as `141.apsi`). Our benchmark includes all operations performed on the `DKZM` array. That array is accessed via different names — we had to include five procedures which perform work on `DKZM` and some other procedures which declare that array but never use it in a way other than passing it as a parameter to some other procedure.

Our optimizations cut the execution time by 14%–45%. Relatively small improvement for `dkzm` is an effect of the data size. Standard input data define the array `DKZM` to be of size  $64 \times 1 \times 16$  which corresponds to 8KB. The primary data cache in our processor is twice that size, therefore even an unoptimized version of the benchmark ends up having the array in the cache most of the time. As we noted above, we expect even better speedups for larger data sets.

## 7 Conclusions

Techniques developed by us allow safe changes in the layout of arrays in the presence of overlapping caused by parameter passing, common blocks and `EQUIVALENCE` statements. Overlapping implied in the original program may be reduced by procedure cloning while minimizing the number of copies of clones of every procedure. Coercing all overlapping arrays to the type with most structure allow the largest flexibility possible in data remapping and distribution. Since all overlapping arrays are coerced to the same type, we can guarantee correctness of data transformations by applying the same transformation to all arrays in a connected component of the Array Overlap Graph. Symbolic expression simplification and code structure recovery make subscript computation inexpensive.

The implementation of those techniques made possible data transformations for programs

which previously could not be optimized with data remapping or parallelized with the use of data distribution directives. Our optimizing compiler implements the techniques described in the paper. We observe good speedups for the chosen set of benchmarks transformed with our compiler.

## Acknowledgements

This work was supported in part by an NSF Research Initiation Award (CCR-9409120) and ARPA contract F19628-94-C-0057. We would like to thank the developers of Polaris [3] for providing an excellent tool for compiler optimizations.

## References

- [1] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and Computation Transformations for Multiprocessors. In *Proceedings of the Fifth ACM Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.
- [2] M. Berry and others. The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):9–40, Fall 1989.
- [3] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The Next Generation in Parallelizing Compilers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing: 7th International Workshop*, volume 892 of Lecture Notes in Computer Science. Springer-Verlag, Berlin/Heidelberg, 1995.
- [4] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler Optimizations for Improving Data Locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, CA, October 1994.
- [5] M. Cierniak and W. Li. Unifying Data and Control Transformations for Distributed Shared-Memory Machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.
- [6] M. Cierniak and W. Li. Recovering Logical Structures of Data. In C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing: 8th International Workshop*, volume 1033 of Lecture Notes in Computer Science. Springer-Verlag, Berlin/Heidelberg, 1996.
- [7] K. M. Dixit. The SPEC benchmarks. In *Parallel Computing*, pages 1195–1209, 1991.
- [8] M. Gupta and P. Banerjee. PARADIGM: A Compiler for Automatic Data Distribution on Multicomputers. In *1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

- [9] T. E. Jeremiassen and S. J. Eggers. Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations. In *Proceedings of the Fifth ACM Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.
- [10] K. Kennedy and U. Kremer. Automatic Data Layout for High Performance Fortran. In *Proceedings Supercomputing '95*, San Diego, CA, December 1995.
- [11] S. Leung and J. Zahorjan. Optimizing Data Locality by Array Restructuring. TR-95-09-01, Department of Computer Science and Engineering, University of Washington, September 1995.
- [12] W. Li and K. Pingali. A Singular Loop Transformation Framework Based on Non-Singular Matrices. *International Journal of Parallel Programming*, 22(2), April 1994.
- [13] W. Li. Compiler Cache Optimizations for Banded Matrix Problems. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.
- [14] Silicon Graphics, Inc. MIPSPro Fortran Programmer's Guide. 1996.
- [15] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, June 1991.