An Implementation of Interprocedural Bounded Regular Section Analysis

Paul Havlak
Ken Kennedy

CRPC-TR-90063-S
March 22, 1991

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

This research has been supported by the National Science Foundation under grant CCR 88-09615, by the IBM Corporation, and by Intel Scientific Computers

1

An Implementation of Interprocedural Bounded Regular Section Analysis

Paul Havlak Ken Kennedy
Rice University
Department of Computer Science
Houston, TX 77251-1892

March 22, 1991

Abstract

Optimizing compilers should produce efficient code even in the presence of high-level language constructs. However, current programming support systems are significantly lacking in their ability to analyze procedure calls. This deficiency complicates parallel programming, because loops with calls can be a significant source of parallelism. We describe an implementation of *regular section analysis*, which summarizes interprocedural side effects on subarrays in a form useful to dependence analysis while avoiding the complexity of prior solutions. The paper gives the results of experiments on the LINPACK library and a small set of scientific codes.

Keywords: array sections, data dependence, interprocedural analysis, parallelization, range analysis

1 Introduction

A major goal of compiler optimization research is to generate code that is efficient enough to encourage the use of high-level language constructs. In other words, good programming practice

*This research has been supported by the National Science Foundation under grant CCR 88-09615, by the IBM Corporation, and by Intel Scientific Computers

should be rewarded with fast execution time.

The use of subprograms is a prime example of good programming practice that requires compiler support for efficiency. Unfortunately, calls to subprograms inhibit optimization in most programming support systems, especially those designed to support parallel programming in Fortran. In the absence of better information, compilers must assume that any two calls can read and write the same memory locations, making parallel execution nondeterministic. This limitation particularly discourages calls in loops, where most compilers look for parallelism.

Traditional interprocedural analysis can help in only a few cases. Consider the following loop:

```
      DO 100 I = 1, N
         CALL SOURCE(A,I,M)
         B(I) = A(INDEX(I),I)
100   CONTINUE
```

If `SOURCE` only modifies locations in the `I`th column of `A`, then parallel execution of the loop is deterministic. Classical interprocedural analysis only discovers which variables are used and which are defined as side effects of procedure calls. We must determine the *subarrays* that are accessed in order to safely exploit the parallelism.

In an earlier paper, Callahan and Kennedy proposed a method called *regular section analysis* for tracking interprocedural side-effects. Regular sections describe side effects to common substructures of arrays such as elements, rows, columns and diagonals [1, 2]. This paper describes an implementation of regular section analysis in the Rice Parallel Fortran Converter (PFC) [3], an automatic parallelization system that also computes dependences for the ParaScope programming environment[4]. The overriding concern in the implementation is that it be efficient enough to be incorporated in a practical compilation system.

Algorithm 1 summarizes the steps of the analysis, whiich is integrated with the three-phase interprocedural analysis and optimization structure of PFC [5, 6]. Regular section analysis added less than 8000 lines to PFC, a roughly 150,000-line PL/I program which runs under IBM VM/CMS.

The remainder of the paper is organized as follows. Section 2 compares various methods for representing side effects to arrays. Section 3 gives additional detail on the exact variety of bounded regular sections implemented. Sections 4 and 5 describe the construction of local sections and their propagation, respectively. Section 6 examines the performance of regular section analysis on two benchmarks: the LINPACK library of linear algebra subroutines and the Rice Compiler Evaluation

```
Local_Analysis:
    for each procedure
        for each array (formal parameter, global, or static)
            save section describing shape
            for each reference
                build ranges for subscripts
                merge resulting section with summary MOD or USE section
            save summary sections
        for each call site
            for each array actual parameter
                save section describing passed location(s)
            for each scalar (actual parameter or global)
                save range for passed value

Interprocedural_Propagation:
    solve other interprocedural problems
        call graph construction
        classical MOD and USE summary
        constant propagation
    mark section subscripts and scalars invalidated by modifications as ⊥
    iterating over the call sites
        translate summary sections into call context
        merge translated sections into caller's summary

Dependence_Analysis:
    for each procedure
        for each call site
            for each summary section
                simulate a DO loop running through the elements of the section
    test for dependences (Banerjee's, GCD)
```

ALGORITHM 1: Overview of Regular Section Analysis

Program Suite (RiCEPS), a set of complete application codes from a variety of scientific disciplines. Sections 7 and 8 suggest areas for future research and give our conclusions.

2 Interprocedural Array Side Effects

A simple way to make dependence testing more precise around a call site is to perform inline expansion, replacing the called procedure with its body [7]. This precisely represents the effects of the procedure as a sequence of ordinary statements, which are readily understood by existing dependence analyzers. However, even if the whole program becomes no larger, the loop nest which contained the call may grow dramatically, causing a time and space explosion due to the non-

linearity of array dependence analysis [8].

To gain some of the benefits of inline expansion without its drawbacks, we must find another representation for the effects of the called procedure. For dependence analysis, we are interested in the memory locations modified or used by a procedure. Given a call to procedure `p` at statement S_1 and an array global variable or parameter A, we wish to compute:

- the set $M^A_{S_1}$ of locations in A that may be modified via `p` called at S_1 and
- the set $U^A_{S_1}$ of locations in A that may be used via `p` called at S_1.

We need comparable sets for simple statements as well. We can then test for dependence by intersecting sets. For example, there exists a true dependence from a statement S_1 to a following statement S_2, based on an array A, only if

$$M^A_{S_1} \cap U^A_{S_2} \neq \emptyset.$$

Several representations have been proposed for representing interprocedural array access sets. The contrived example in Figure 1 shows the different patterns that they can represent precisely. Evaluating these methods involves examining the complexity and precision of:

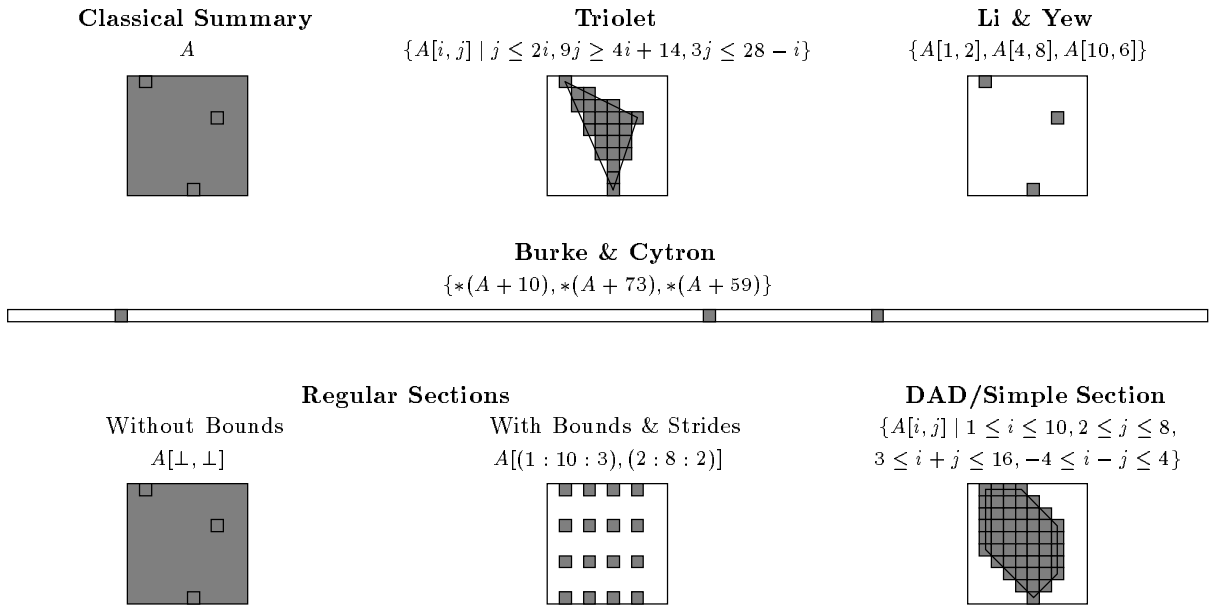

FIGURE 1: Summarizing the References $A[1,2]$, $A[4,8]$, and $A[10,6]$

- representing the sets $M^A_{S_1}$ and $U^A_{S_2}$,
- merging descriptors to summarize multiple accesses (we call this the *meet* operation, because most descriptors may be viewed as forming a lattice),
- testing two descriptors for *intersection* (dependence testing), and
- translating descriptors at call sites (especially when there are array reshapes).

Handling of recursion turns out not to be an issue. Iterative techniques can guarantee convergence to a fixed point solution using Cousot's technique of *widening operators* [9, 10]. Li and Yew proposed a preparatory analysis of recursive programs that guarantees termination in three iterations [11, 12]. Either of these methods may be adapted for regular sections.

2.1 True Summaries

True summary methods use descriptors whose size is largely independent of the number of references being summarized. This may make the descriptors and their operations more complicated, but limits the expense of translating descriptors during interprocedural propagation and intersecting them during dependence analysis.

Classical Methods The classical methods of interprocedural summary dataflow analysis compute MOD and USE sets indicating which parameters and global variables may be modified or used in the procedure [13, 14, 15]. Such summary information costs only two bits per variable. Meet and intersection may be implemented using single-bit or bit-vector logical operations. Also, there exist algorithms that compute complete solutions, in which the number of meets is linear in the number of procedures and call sites in the program, even when recursion is permitted [16].

Unfortunately, our experiences with PFC and PTOOL indicate that this summary information is too coarse for dependence testing and the effective detection of parallelism [1]. The problem is that the only access sets representable in this method are "the whole array" and "none of the array" (see Figure 1). Such coarse information limits the detection of data decomposition, an important source of parallelism, in which different iterations of a loop work on distinct subsections of a given array.

Triolet Regions Triolet, Irigoin and Feautrier proposed to calculate linear inequalities bounding the set of array locations affected by a procedure call [17, 18]. This representation and its intersection operation are precise for convex regions. Other patterns, such as array accesses with non-unit stride and non-convex results of meet operations, are given convex approximations.

Operations on these regions are expensive; the meet operation requires finding the convex hull of the combined set of inequalities and intersection uses a potentially exponential linear inequality solver [19]. A succession of meet operations can also produce complicated regions with potentially as many inequalities as the number of primitive accesses merged together. Translation at calls sites is precise only when the formal parameter array in the called procedure maps to a (sub)array of the same shape in the caller. Otherwise, the whole actual parameter array is assumed accessed by the call. The region method ranks high in precision, but is too expensive because of its complex representation.

2.2 Reference Lists

Some proposed methods do not summarize, but represent each reference separately. Descriptors are then lists of references, the meet operation is list concatenation (possibly with a check for duplicates), and translation and intersection are just the repeated application of the corresponding operations on simple references. However, this has two significant disadvantages:

- translation of a descriptor requires time proportional to the number of references, and
- intersection of descriptors requires time quadratic in the number of references.

Reference list methods are simple and precise, but are asymptotically as expensive as in-line expansion.

Linearization Burke and Cytron proposed representing each multidimensional array reference by linearizing its subscript expressions to a one-dimensional address expression. Their method also retains bounds information for loop induction variables occurring in the expressions [20]. They describe two ways of implementing the meet operation. One involves merely keeping a list of the individual address expressions. The other constructs a composite expression that can be polynomial in the loop induction variables. The disadvantages of the first method are described above. The

second method appears complicated and has yet to be rigorously described. Linearization in its pure form is ill-suited to summarization, but might be a useful extension to a true summary technique because of its ability to handle arbitrary reshapes.

Atom Images Li and Yew extended Parafrase to compute sets of *atom images* describing the side effects of procedures [21, 11]. Like the original version of regular sections described in Callahan's thesis [2], these record subscript expressions that are linear in loop induction variables along with bounds on the induction variables. Any reference with linear subscript expressions in a triangular iteration space can be precisely represented, and they keep a separate atom image for each reference.

The expense of translating and intersecting lists of atom images is too high a price to pay for their precision. Converting atom images to a summary method would produce something similar to the regular sections described below.

2.3 Summary Sections

The precise methods described above are expensive because they allow arbitrarily large representations of a procedure's access sets. The extra information may not be useful in practice; simple array access patterns are probably more common than others. To avoid expensive intersection and translation operations, descriptor size should be independent of the number of references summarized. Operations on descriptors should be linear or, at worst, quadratic in the rank of the array. Researchers at Rice have defined several variants of *regular sections* to represent common access patterns while satisfying these constraints [2, 1, 22, 23].

Original Regular Sections Callahan's thesis proposed two regular section frameworks. The first, resembling Li and Yew's atom images, he dismissed due to the difficulty of devising efficient standardization and meet operations [2].

Restricted Regular Sections. The second framework, *restricted* regular sections [2, 1], is limited to access patterns in which each subscript is

- a procedure-invariant expression (with constants and procedure inputs),
- unknown (and assumed to vary over the entire range of the dimension), or

- unknown but diagonal with one or more other subscripts.

The restricted sections have efficient descriptors: their size is linear in the number of subscripts, their meet operation quadratic (because of the diagonals), and their intersection operation linear. However, they lose too much precision by omitting bounds information. While we originally thought that these limitations were necessary for efficient handling of recursive programs, Li and Yew have adapted iterative techniques to work with more general descriptors [12].

Bounded Regular Sections Anticipating that restricted regular sections would not be precise enough for effective parallelization, Callahan and Kennedy proposed an implementation of regular sections with bounds. That project is the subject of this paper. The regular sections implemented include bounds and stride information, but omit diagonal constraints. The resulting analysis is therefore less precise in the representation of convex regions than Triolet regions or the Data Access Descriptors described below. However, this is the first interprocedural summary implementation with stride information, which provides increased precision for non-convex regions.

The size of bounded regular section descriptors and the time required for the meet operation are both linear in the number of subscripts. Intersection is implemented using standard dependence tests, which also take time proportional to the number of subscripts.[1]

Data Access Descriptors Concurrently with our implementation, Balasundaram and Kennedy developed Data Access Descriptors (DADs) as a general technique for describing data access [22, 23, 24]. DADs represent information about both the shapes of array accesses and their traversal order; for our comparison we are interested only in the shapes. The *simple section* part of a DAD represents a convex region similar to those of Triolet *et al.*, except that boundaries are constrained to be parallel to one coordinate axis or at a 45° angle to two axes. Stride information is represented in another part of the DAD.

Data Access Descriptors are probably the most precise summary method that can be implemented with reasonable efficiency. They can represent the most likely rectangular, diagonal, triangular, and trapezoidal accesses. In size and in time required for meet and intersection they have

[1]This analysis ignores the *greatest common divisor* computation used in merging and intersecting sections with strides; this can take time proportional to the values of the strides.

complexity quadratic in the number of subscripts (which is reasonable given that most arrays have few subscripts).

The bounded sections implemented here are both less expensive and less precise than DADs. Our implementation can be extended to compute DADs if the additional precision proves useful.

3 Bounded Sections and Ranges

Bounded regular sections comprise the same set of rectangular subarrays that can be written using triplet notation in the proposed Fortran 90 standard [25]. They can represent sparse regions such as stripes and grids and dense regions such as columns, rows, and blocks.

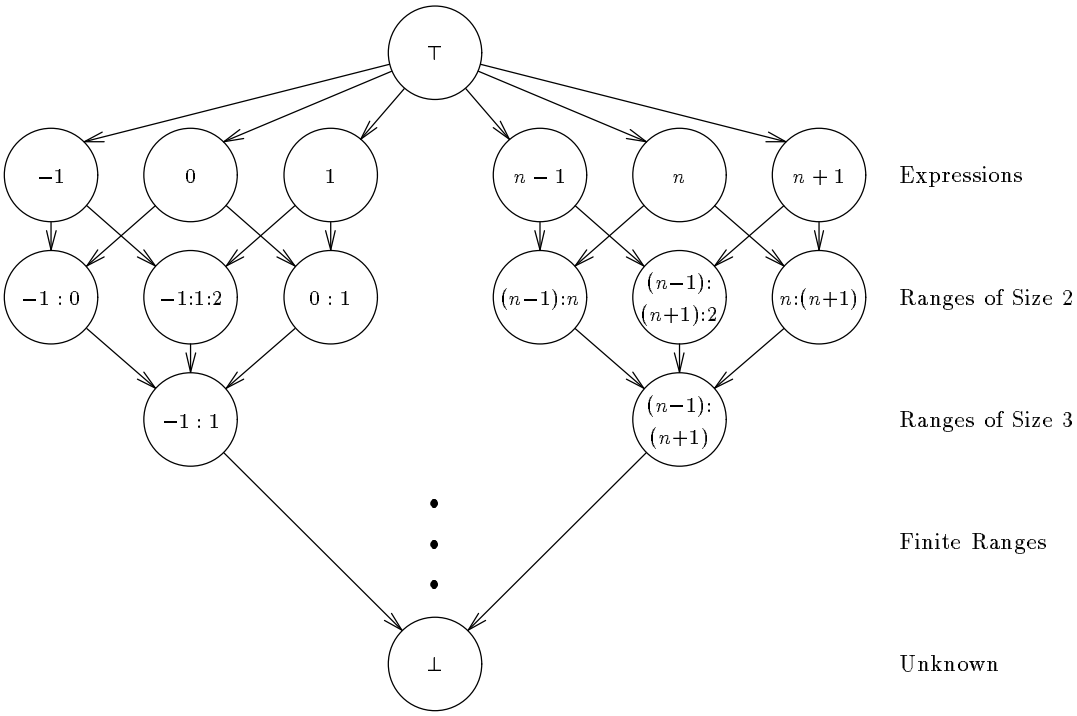

FIGURE 2: Lattice for Regular Section Subscripts

3.1 Representation

The descriptors for bounded regular sections are vectors of elements from the subscript lattice in Figure 2. Lattice elements include:

- invariant expressions, containing only constants and symbols representing the values of parameters and global variables on entry to the procedure;
- ranges, giving invariant expressions for the lower bound, upper bound, and stride of a variant subscript; and
- $\perp$, indicating no knowledge of the subscript value.

While ranges may be constructed through a sequence of meet operations, the more common case is that they are read directly from the bounds of a loop induction variable used in a subscript.

Since no constraints between subscripts are maintained, merging two regular sections for an array of rank d requires only d independent invocations of the subscript meet operation. We test for intersection of two sections with a single invocation of standard d-dimensional dependence tests. Translation of a formal parameter section to one for an actual parameter is also an $O(d)$ operation (where d is the larger of the two ranks).

3.2 Operations on Ranges

Ranges are typically built to represent the values of loop induction variables, such as I in the following loop.

```
DO I = l, u, s
    A(I) = B(2*I+1)
ENDDO
```

We represent the value of I as $[l : u : s]$. While l and u are often referred to as the lower and upper bound, respectively, their roles are reversed if s is negative. We can produce a standard lower-to-upper bound form if we know $l \leq u$ or $s \geq 1$; this operation is described in detail in Algorithm 2. Standardization may cause loss of information; therefore, we postpone standardization until it is required by some operation, such as merging two sections.

```
function standardize([l : u : s])
begin
    diff = u − l
    perfect = false
    if diff and s are both constant then
        if sign(diff) ≠ sign(s) then return(⊤) /* empty range */
        direction = sign(diff)
        u = u - direction * (abs(diff) mod abs(s))
        perfect = true

    else if diff is constant then direction = sign(diff)
    else if s is constant    then direction = sign(s)
    else return(⊥)

    select direction
        when = 0 return(l)
        when > 0 return([l : u : s])
        when < 0 if perfect then return([u : l : (−s)])
                            else  return([u : l : 1])
    end select
end.
```

ALGORITHM 2: Standardizing a Range to Lower-Bound-First Form

Expressions in ranges are converted to ranges; for example, `2*I+1` in the above loop is represented as $[(2 * l + 1) : (2 * l + 1) : (2 * s)]$. Only invariant expressions are accurately added to or multiplied with a range; Algorithm 3 constructs approximations for sums of ranges.

Ranges are merged by finding the lowest lower bound and the highest upper bound, then correcting the stride. An expression is merged with a range or another expression by treating it as a range with a lower bound equal to its upper bound. Algorithm 4 thus computes the same result for $1 \wedge 3 \wedge 5$ as for $[1 : 5 : 4] \wedge 3$; namely, $[1 : 5 : 2]$.

The most interesting subscript expressions are those containing references to scalar parameters and global variables. We represent such symbolic expressions as global value numbers so that they may be tested for equality by the standardization and merge operations.

4 Local Analysis

For each procedure, we construct symbolic subscript expressions and accumulate initial regular sections with no knowledge of interprocedural effects. The precision of our analysis depends on

```
function build_range(e)
begin
    if e is a leaf expression (constant, formal, or global value; or ⊥) then
        return(e)
    for each subexpression s of e
        replace s with build_range(s)
    select form of e
        when [l1 : u1 : s1] + [l2 : u2 : s2]      [l1' : u1' : s1'] = standardize([l1 : u1 : s1])
                                                  [l2' : u2' : s2'] = standardize([l2 : u2 : s2])
                                                  return([(l1' + l2') : (u1' + u2') : gcd(s1', s2')])
        when a + [l : u : s] or [l : u : s] + a   return([(a + l) : (a + u) : s])
        when a * [l : u : s] or [l : u : s] * a   return([(a * l) : (a * u) : (a * s)])
        otherwise return(⊥)
    end select
end.
```

ALGORITHM 3: Moving Ranges to the Top Level of an Expression

recording questions about side effects, but not answering them until the results of other interprocedural analyses are available.

4.1 Symbolic Analysis

Constructing regular sections requires the calculation of symbolic expressions for variables used in subscripts. While there are many published algorithms for performing symbolic analysis and global value numbering [26, 27, 28], their preliminary transformations and complexity make them difficult to integrate into PFC. Our implementation builds global value numbers with the help of PFC's existing dataflow analysis machinery.

Leaf value numbers are constants and the global and parameter values available on procedure entry. We build value numbers for expressions by recursively obtaining the value numbers for subexpressions and reaching definitions. Value numbers reaching the same reference along different def-use edges are merged. If either the merging or the occurrence of an unknown operator creates a unknown ($\perp$) value, the whole expression is lowered to $\perp$.

Induction variables are recognized by their defining loop headers and replaced with the inductive range. (Auxiliary induction variables are currently not identified.) For example, consider the following code fragment.

```
      SUBROUTINE S1(A,N,M)
```

```
function merge(a, b)
begin
    if a = ⊥ or b = ⊥ then return(⊥)
    if a = ⊤ or a = b  then return(b)
    if b = ⊤           then return(a)

    if a is a range then let [l_a, u_a, s_a] = standardize(a)
                    else  let [l_a, u_a, s_a] = [a, a, ⊤]
    if b is a range then let [l_b, u_b, s_b] = standardize(b)
                    else  let [l_b, u_b, s_b] = [b, b, ⊤]

    l' = min(l_a, l_b)                    /* min, max, gcd, and abs can return ⊥ */
    u' = max(u_a, u_b)
    s' = gcd(s_a, s_b, abs(l_a − l_b))    /* gcd(⊤, a) returns a */

    if l' = ⊥       then return(⊥)
    else if s' = ⊥ then return([l' : u' : 1])
    else                return([l' : u' : s'])
end.
```

ALGORITHM 4: Merging Expressions and Ranges

```
        DIMENSION A(N)
        DO I = 1, N
            A(M*I) = 0.0
        ENDDO
        RETURN
    END
```

Dataflow analysis constructs def-use edges from the subroutine entry to the uses of `N` and `M`, and from the `DO` loop to the use of `I`. It is therefore simple to compute the subscript in `A`'s regular section: $\mathtt{M} * [1 : \mathtt{N} : 1]$, which is converted to the range $[\mathtt{M} : \mathtt{M} * \mathtt{N} : \mathtt{M}]$ (the names `M` and `N` are actually replaced by their formal parameter indices). Note that expressions that are nonlinear during local analysis may become linear in later phases, especially after constant propagation.

4.2 Avoiding Compilation Dependences

To construct accurate value numbers, we require knowledge about the effects of call sites on scalar variables. However, using interprocedural analysis to determine these effects can be costly.

A programming support system using interprocedural analysis must examine each procedure

at least twice:[2] once when gathering information to be propagated between procedures, and again when using the results of this propagation in dependence analysis and/or transformations. By precomputing the local information, we can construct an interprocedural propagation phase which iterates over the call graph without additional direct examination of any procedure.

To achieve this minimal number of passes, all interprocedural analyses must gather local information in one pass, without the benefit of each others' interprocedural solutions. However, to build precise local regular sections, we need information about the side effects of calls on scalars used in subscripts. In the following code fragment, we must assume that `M` is modified to an unknown value unless proven otherwise:

```
SUBROUTINE S1(A,N,M)
    DIMENSION A(N)
    CALL CLOBBER(M)
    A(M) = 0.0
    RETURN
END
```

To achieve precision without adding a separate local analysis phase for regular sections, we build regular section subscripts as if side effects did not occur, while annotating each subscript expression with its hazards, or side effects that would invalidate it. We thus record that `A(M)` is modified, with the sole parameter of `CLOBBER` as a hazard on `M`. During the interprocedural phase, after producing the classical scalar side effect solution, but before propagating regular sections, we check to see if `CLOBBER` may change `M`. If so, we change `S1`'s array side effect to `A(`$\perp$`)`. A similar technique has proven successful for interprocedural constant propagation in PFC [31, 6].

Hazards must be recorded with each scalar expression saved for use in regular section analysis: scalar actual parameters and globals at call sites as well as array subscripts. When we merge two expressions or ranges, we take the union of their hazard sets.

4.3 Building Summary Regular Sections

With the above machinery in place, the USE and MOD regular sections for the local effects of a procedure are constructed easily. In one pass through the procedure, we examine each reference to

[2]This is not strictly true; a system computing only summary information (USE, MOD) or context information (ALIAS) can make do with one pass. Both the PFC and $\mathbb{R}^n$/ParaScope systems perform summary and context analysis, as well as constant propagation, and therefore require at least two passes [30, 5, 6].

a formal parameter, global, or static array. The symbolic analyzer provides value numbers for the subscripts on demand; the resulting vector is a regular section. After the section for an individual reference is constructed, it is immediately merged with the appropriate cumulative section(s), then discarded.

5 Interprocedural Propagation

Regular sections for formal parameters are translated into sections for actual parameters as we traverse edges in the call graph. The translated sections are merged with the summary regular sections of the caller, requiring another translation and propagation step if this changes the summary. To extend our implementation to recursive programs and have it terminate, we must bound the number of times a change occurs.

5.1 Translation into a Call Context

If we were analyzing Pascal arrays, mapping the referenced section of a formal parameter array to one for the corresponding actual parameter would be simple. We would only need to replace formal parameters in subscript values of the formal section with their corresponding actual parameter values, then copy the resulting subscript values into the new section. However, Fortran provides no guarantee that formal parameter arrays will have the same shape as their actual parameters, nor even that arrays in common blocks will be declared to have the same shape in every procedure. Therefore, to describe the effects of a called procedure for the caller, we must translate the referenced sections according to the way the arrays are reshaped.

The easiest translation method would be to linearize the subscripts for the referenced section of a formal parameter, adding the offset of the passed location of the actual parameter [20]. The resulting section would give referenced locations of the actual as if it were a one-dimensional array. However, if some subscripts of the original section are ranges or non-linear expressions, linearization contaminates the other subscripts, greatly reducing the precision of dependence analysis. For this reason, we forego linearization and translate significantly reshaped dimensions as $\perp$.

Algorithm 5 shows one method for translating a summary section for a formal parameter `F` into a

```
function translate(bounds_F, ref_F, bounds_A, pass_A)
begin
    if ref_F = ⊤ then return(⊤)
    consistent = true
    for i = 1 to rank(A)
        if not consistent then ref_A[i] = ⊥
        else if i > rank(F) then if consistent then ref_A[i] = pass_A[i]
                                                else  ref_A[i] = ⊥
        else
            replace scalar formal parameters in bounds_F and ref_F
                    with their corresponding actual parameters
            bounds_i= bounds_F[i] - lo(bounds_F[i]) + pass_A[i]
            ref_i= ref_F[i] - lo(bounds_F[i]) + pass_A[i]
            consistent = (bounds_i = bounds_A[i])
            if consistent then ref_A[i] = ref_i
                        else  if stride(ref_i) = hi(bounds_A[i + 1]) - lo(bounds_A[i + 1]) then
                                    /* delinearization is possible */
                              if i = rank(F) and ((ref_i fits in bounds_A[i]) or assume_fit) then
                                    ref_A[i] = ref_i
                              else ref_A[i] = ⊥ /* safe */
    end for
    return(ref_A)
end.
```

ALGORITHM 5: Translating a Summary Section

section for its corresponding actual parameter A. Translation proceeds from left to right through the dimensions, and is precise until a dimension is encountered where the formal and actual parameter are *inconsistent* (having different sizes or non-zero offset). The first inconsistent dimension is also translated precisely if it is the last dimension of F and the referenced section subscript value(s) fit in the bounds for A. Delinearization, which is not implemented, may be used to recognize that a reference to F with a column stride the same as the column size of A corresponds to a row reference in A.

5.2 Treatment of Recursion

The current implementation handles only non-recursive Fortran. Therefore, it is sufficient to proceed in reverse invocation order on the call graph, translating sections up from leaf procedures to their callers. The final summary regular sections are built in order, so that incomplete regular sections need never be translated into a call site. However, the proposed Fortran 90 standard allows

recursion [25], and we plan an extension or re-implementation that will handle it. Unfortunately, a straightforward iterative approach to the propagation of regular sections will not terminate, since the lattice has unbounded depth.

Li and Yew [11] and Cooper and Kennedy [16] describe approaches for propagating subarrays that are efficient regardless of the depth of the lattice. However, it may be more convenient to implement a simple iterative technique while simulating a bounded-depth lattice. If we maintain a counter with the summary regular section for each array and procedure, then we can limit the number of times we allow the section to become larger (lower in the lattice) before going to $\perp$. The best way to do this is by keeping one small counter (e.g., two bits) per subscript. Variant subscripts will then go quickly to $\perp$, leaving precise subscripts unaffected. If we limit each subscript to being lowered in the subscript lattice k times, then an array of rank d will have an effective lattice depth of $kd + 1$.

Since each summary regular section is lowered at most $O(kd)$ times, each associated call site is affected at most $O(kdv)$ times (each time involving an $O(d)$ merge), where v is the number of *referenced* global and parameter variables. In the worst case, we then require $O(kd^2ve)$ subscript merge and translation operations, where e is the number of edges in the call graph. This technique allows us to use a lattice with bounds information while keeping time complexity comparable to that obtained with the restricted regular section lattice.

6 Experimental Results

The precision, efficiency, and utility of regular section analysis must be demonstrated by experiments on real programs. Our current candidates for "real programs" are the LINPACK library of linear algebra subroutines [32], the Rice Compiler Evaluation Program Suite, and the Perfect Club benchmarks [33]. We ran the programs through regular section analysis and dependence analysis in PFC, then examined the resulting dependence graphs by hand and in the ParaScope editor, an interactive dependence browser and program transformer [4].

LINPACK Analysis of LINPACK provides a basis for comparison with other methods for analyzing interprocedural array side effects. Both Li and Yew [21] and Triolet [18] found several parallel calls in LINPACK using their implementations in the University of Illinois translator, Parafrase.

LINPACK proves that useful numerical codes can be written in the modular programming style for which parallel calls can be detected.

RiCEPS The Rice Compiler Evaluation Program Suite is a collection of 10 complete applications codes from a broad range of scientific disciplines. Our colleagues at Rice have already run several experiments on RiCEPS. Porterfield modeled cache performance using an adapted version of PFC [34]. Goff, Kennedy and Tseng studied the performance of dependence tests on RiCEPS and other benchmarks [35]. Some RiCEPS and RiCEPS candidate codes have also been examined in a study on the utility of inline expansion of procedure calls [8]. The six programs studied here are two RiCEPS codes `linpackd` and `track`) and four codes from the inlining study.

Perfect Club Benchmarks This suite was originally collected for benchmarking the performance of supercomputers on complete applications. While we hope to test the performance of our implementation on these programs, a delay in receiving them prevented us from obtaining more than very preliminary results for this paper.

6.1 Precision

The precision of regular sections, or their correspondence to the true access sets, is largely a function of the programming style being analyzed. LINPACK is written in a style which uses many calls to the BLAS (basic linear algebra subroutines), whose true access sets are precisely regular sections. We did not determine the true access sets for the subroutines in RiCEPS, but of the six programs analyzed, only `dogleg` and `linpackd`, which actually call LINPACK, exhibited the LINPACK coding style.

While there exist regular sections to precisely describe the effects of the BLAS, our local analysis was unable to construct them under complicated control flow. With changes to the BLAS to eliminate unrolled loops and the conditional computation of values used in subscript expressions, our implementation was able to build minimal regular sections that precisely represented the true access sets. The modified `DSCAL`, for example, looks as follows:

```
SUBROUTINE DSCAL(N, DA, DX, INCX)
    DOUBLE PRECISION DA, DX(*)
    IF (N .LE. 0) RETURN
```

```
      DO I = 1, N*INCX, INCX
          DX(I) = DA * DX(I)
      ENDDO
      RETURN
END
```

Obtaining precise symbolic information is a problem in all methods for describing array side effects. Triolet made similar changes to the BLAS; Li and Yew avoided them by first performing interprocedural constant propagation. The fundamental nature of this problem indicates the desirability of a clearer Fortran programming style or more sophisticated handling of control flow (such as that described in Section 7).

6.2 Efficiency

We measured the total time taken by PFC to analyze the six RiCEPS programs.[3] Parsing, local analysis, interprocedural propagation, and dependence analysis were all included in the execution times. Table 1 compares the analysis time required using classical interprocedural summary analysis alone ("IP only") with that using summary analysis and regular section analysis combined ("IP + RS").[4]

program name	Lines	Procs	IP only	IP +RS	% Change
`efie`	1254	18	209	232	+10
`euler`	1113	13	117	138	+15
`vortex`	540	19	65	87	+25
`track`	1711	34	191	225	+15
`dogleg`	4199	48	272	377	+28
`linpackd`	355	10	28	44	+36
total	9172	142	882	1103	+25

TABLE 1: Analysis times in seconds (PFC on an IBM 3081D)

[3]While we were able to run most of the Perfect programs through PFC, we have not yet obtained reliable timings on the recently-upgraded IBM system at Rice.

[4]We do not present times for the dependence analysis with no interprocedural information because it performs less analysis on loops with call sites. Discounting this advantage, the time taken for classical summary analysis seems to be less than 10 percent.

The most time time-consuming part of our added code is the local symbolic analysis for subscript values, which includes an invocation of dataflow analysis. More symbolic analysis would improve the practicality of the entire method. Overall, the additional analysis time is comparable to that required to analyze programs after heuristically-determined inline expansion in Cooper, Hall and Torczon's study [8].

We have not seen published execution times for the array side effect analyses implemented in Parafrase by Triolet and by Li and Yew, except that Li and Yew state that their method runs 2.6 times faster than Triolet's [21]. Both experiments were run only on LINPACK; it would be particularly interesting to know how their methods would perform on complete applications.

6.3 Utility

We chose three measures of utility:

- reduced numbers of dependences and dependent references,
- increased numbers of calls in parallel loops, and
- reduced parallel execution time.

Reduced Dependence Table 2 compares the dependence graphs produced using classical interprocedural summary analysis alone ("IP") and summary analysis plus regular section analysis

	All Dependences			Array Dep. on Calls in Loops					
				loop carried			loop independent		
source	IP	RS	% ⇓	IP	RS	% ⇓	IP	RS	% ⇓
`efie`	12338	12338		177	177		81	81	
`euler`	1818	1818		70	70		30	30	
`vortex`	1966	1966		220	220		73	73	
`track`	4737	4725	0.25	68	67	1.5	27	26	3.7
`dogleg`	1858	1675	9.8	226	168	25.7	80	59	26.2
`linpackd`	496	399	19.6	191	116	39.3	67	45	32.8
RICEPS	23213	22921	1.25	952	818	14.1	358	314	12.3
LINPACK	12336	11035	10.5	3071	2064	32.8	1348	1054	21.8

TABLE 2: Effects of Regular Section Analysis on Dependences

("RS").[5]

LINPACK was analyzed without interprocedural constant propagation, since library routines may be called with varying array sizes. The first set of three columns gives the sizes of the dependence graphs produced by PFC, counting all true, anti and output dependence edges on scalar and array references in DO loops (including those references not in call sites). The other sets of columns count only those dependences incident on array references in call sites in loops, with separate counts for loop-carried and loop-independent dependences. Preliminary results for eight of the 13 Perfect benchmarks indicate a reduction of 0.6 percent in the total size of the dependence graphs.[6]

Parallelized Calls Table 3 examines the number of calls in LINPACK which were parallelized after summary interprocedural analysis alone ("IP"), after Li and Yew's analysis [21], and after regular section analysis ("RS"). (Triolet's results from Parafrase resembled Li and Yew's.) Most (17) of these call sites were parallelized in ParaScope, based on PFC's dependence graph, with no transformations being necessary. The eight parallel call sites detected with summary interprocedural analysis alone were apparent in ParaScope, but exploiting the parallelism requires a variant of statement splitting that is not yet supported. Starred entries ($\star$) indicate parallel calls which were precisely summarized by regular section analysis, but which were not detected as parallel due to a deficiency in PFC's symbolic dependence test for triangular loops. One call in `QRDC` was mistakenly parallelized by Parafrase [36].

These results indicate, at least for LINPACK, that there is no benefit to the generality of Triolet's and Li and Yew's methods. Regular section analysis obtains exactly the same precision, with a different number of loops parallelized only because of differences in dependence analysis and transformations.

Improved Execution Time Two calls in the RICEPS programs were parallelized: one in `dogleg` and one in `linpackd`. Both were the major computational loops (`linpackd`'s in `DGEFA`, `dogleg`'s in

[5]The dependence graphs resulting from no interprocedural analysis at all are not comparable, since no calls can be parallelized and their dependences are collapsed to conserve space.

[6]Sections are not yet propagated for arrays in common blocks. This deficiency probably resulted in more dependences for the larger programs.

routine name	calls in DO loops	Parallel Calls		
		IP	Li-Yew	RS
·GBCO	8	1	1	1
·GECO	8	1	1	1
·PBCO	8	1	1	1
·POCO	8	1	1	1
·PPCO	8	1	1	1
·SICO	1	1	1	1
·SPCO	1	1	1	1
·TRCO	4	1	1	1
·GBFA	3		1	1
·GEDI	4		1	1
·GEFA	3		1	1
·PODI	4		2	2
·QRDC	9		5	4
·SIDI	6		3	*3
·SIFA	3		3	*3
·SVDC	15		3	7
·TRDI	4		—	1
other	47			
total(36)	144	8	27	31

TABLE 3: Parallelization of LINPACK

DQRDC).[7] Running linpackd on 19 processors with the one call parallelized was enough to speed its execution by a factor of five over sequential execution on the Sequent Symmetry at Rice. Further experiments on improvements in parallel execution time await our acquisition of more Fortran codes written in an appropriate style.

7 Future Work

More experiments are required to fully evaluate the performance of regular section analysis on complete applications and find new areas for improvement. Based on the studies conducted so far, extensions to provide better handling of conditionals and flow-sensitive side effects seem promising.

7.1 Conditional Symbolic Analysis

Consider the following example, derived from the BLAS:

```
SUBROUTINE D(N, DA, DX, INCX)
    DOUBLE PRECISION DA, DX(*)
    IF (INCX .LT. 0) THEN
        IX = (-N+1)*INCX + 1
    ELSE
        IX = 1
    ENDIF
    DO I = 1, N
       DX(IX) = DA * DX(IX)
       IX = IX + INCX
    ENDDO
    RETURN
END
```

The two computations of the initial value for IX correspond to different ranges for the subscript of DX: $[(1 + \texttt{INCX} * (1 - \texttt{N})) : 1 : \texttt{INCX}]$ and $[1 : (1 + \texttt{INCX} * (\texttt{N} - 1)) : \texttt{INCX}]$. It turns out that these can both be represented by $[1 : (1 + |\texttt{INCX}| * (\texttt{N} - 1)) : |\texttt{INCX}|]$. For the merge operation to produce this precise result requires that it have an understanding of the control conditions under which expressions are computed.

[7] In the inlining study at Rice, none of the commercial compilers was able to detect the parallel call in dogleg even after inlining, presumably due to complicated control flow [8].

7.2 Killed Regular Sections

We have already found programs (`scalgam` and `euler`) in which the ability to recognize and localize temporary arrays would cut the number of dependences dramatically, allowing some calls to be parallelized. We could recognize interprocedural temporary arrays by determining when an entire array is guaranteed to be modified before being used in a procedure. While this is a flow-sensitive problem, and therefore expensive to solve in all its generality, even a very limited implementation should be able to catch the initialization of many temporaries.

The subscript lattice for killed sections is the same one used for USE and MOD sections; however, since kill analysis must produce *underestimates* of the affected region in order to be conservative, the lattice needs to be inverted. In addition, this approach requires an *intra*procedural dependence analysis capable of using array kill information, such as those described by Rosene [37] and by Gross and Steenkiste [38].

8 Conclusion

Regular section analysis can be a practical addition to a production compiler. Its local analysis and interprocedural propagation can be integrated with those for other interprocedural techniques. The required changes to dependence analysis are trivial—the same ones needed to support Fortran 90 sections.

These experiments demonstrate that regular section analysis is an effective means of discovering parallelism, given programs written in an appropriately modular programming style. Such a style can benefit advanced analysis in other ways, for example, by keeping procedures small and simplifying their internal control flow. Our techniques will not do as well on programs written in a style that minimizes the use of procedure calls to compensate for the lack of interprocedural analysis in other compilers. Compilers must reward the modular programming style with fast execution time for it to take hold among the computation-intensive users of supercomputers. In the long run it should make programs easier for both their writers and automatic analyzers to understand.

Acknowledgements

We would like to thank our colleagues on the PFC and ParaScope projects, who made this research possible. We further thank David Callahan for his contributions to regular section analysis, and Kathryn McKinley, Mary Hall, and the reviewers for their critiques of this paper.

References

[1] D. Callahan and K. Kennedy, "Analysis of interprocedural side effects in a parallel programming environment," *Journal of Parallel and Distributed Computing*, vol. 5, pp. 517–550, 1988.

[2] D. Callahan, *A Global Approach to Detection of Parallelism*. PhD thesis, Rice University, Mar. 1987.

[3] J. R. Allen and K. Kennedy, "PFC: A program to convert Fortran to parallel form," in *Supercomputers: Design and Applications*, pp. 186–205, Silver Spring, MD: IEEE Computer Society Press, 1984.

[4] K. Kennedy, K. S. McKinley, and C. Tseng, "Interactive parallel programming using the ParaScope Editor," Tech. Rep. TR90-137, Dept. of Computer Science, Rice University, Oct. 1990. To appear in *IEEE Transactions on Parallel and Distributed Systems*.

[5] R. Allen, D. Callahan, and K. Kennedy, "An implementation of interprocedural analysis in a vectorizing Fortran compiler," Tech. Rep. TR86-38, Dept. of Computer Science, Rice University, May 1986.

[6] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," in *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, June 1986.

[7] F. Allen and J. Cocke, "A catalogue of optimizing transformations," in *Design and Optimization of Compilers* (J. Rustin, ed.), Prentice-Hall, 1972.

[8] K. Cooper, M. Hall, and L. Torczon, "An experiment with inline substitution," Tech. Rep. TR90-128, Dept. of Computer Science, Rice University, July 1990. To appear in *Software—Practice and Experience*.

[9] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth ACM Symposium on the Principles of Programming Languages*, (Los Angeles), pp. 238–252, Jan. 1977.

[10] P. Cousot, "Semantic foundations of program analysis," in *Program Flow Analysis: Theory and Applications* (S. S. Muchnick and M. D. Jones, eds.), pp. 303–342, Prentice-Hall,New Jersey, 1981.

[11] Z. Li and P.-C. Yew, "Interprocedural analysis and program restructuring for parallel programs," CSRD Rpt. No. 720, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Jan. 1988.

[12] Z. Li and P.-C. Yew, "Program parallelization with interprocedural analysis," *The Journal of Supercomputing*, vol. 2, pp. 225–244, 1988.

[13] J. Banning, *A Method for Determining the Side Effects of Procedure Calls*. PhD thesis, Stanford University, Aug. 1978.

[14] J. Barth, "An interprocedural data flow analysis algorithm," in *Conference Record of the Fourth ACM Symposium on the Principles of Programming Languages*, (Los Angeles), Jan. 1977.

[15] K. Cooper and K. Kennedy, "Efficient computation of flow insensitive interprocedural summary information," in *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices Vol. 19, No. 6*, July 1985.

[16] K. Cooper and K. Kennedy, "Interprocedural side-effect analysis in linear time," in *Proceedings of the ACM SIGPLAN 88 Conference on Program Language Design and Implementation*, (Atlanta, GA), June 1988.

[17] R. Triolet, F. Irigoin, and P. Feautrier, "Direct parallelization of CALL statements," in *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, (Palo Alto, CA), pp. 176–185, July 1986.

[18] R. Triolet, "Interprocedural analysis for program restructuring with Parafrase," CSRD Rpt. No. 538, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Dec. 1985.

[19] U. Banerjee, "A direct parallelization of CALL statements – a review," CSRD Rpt. 576, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Apr. 1986.

[20] M. Burke and R. Cytron, "Interprocedural dependence analysis and parallelization," in *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp. 162–175, June 1986.

[21] Z. Li and P.-C. Yew, "Efficient interprocedural analysis for program parallelization and restructuring," in *ACM SIGPLAN PPEALS*, pp. 85–99, 1988.

[22] V. Balasundaram, *Interactive Parallelization of Numerical Scientific Programs*. PhD thesis, Rice University, July 1989. Available as Rice COMP TR89-95.

[23] V. Balasundaram and K. Kennedy, "A technique for summarizing data access and its use in parallelism enhancing transformations," in *Proceedings of the ACM SIGPLAN '89 Conference on Program Language Design and Implementation*, (Portland, OR), June 1989.

[24] V. Balasundaram, "A mechanism for keeping useful internal information in parallel programming tools: the Data Access Descriptor," *Journal of Parallel and Distributed Computing*, vol. 9, pp. 154–170, 1990.

[25] X3J3 Subcommittee of ANSI, *American National Standard for Information Systems Programming Language Fortran: S8 (X3.9-198x)*. New York, NY: American National Standards Institute, 1989.

[26] M. Karr, "Affine relationships among variables of a program," *Acta Informatica*, vol. 6, pp. 133–151, 1976.

[27] J. H. Reif and R. E. Tarjan, "Symbolic program analysis in almost-linear time," *SIAM Journal on Computing*, vol. 11, pp. 81–93, Feb. 1981.

[28] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, (San Diego, CA), pp. 12–27, Jan. 1988.

[29] W. H. Harrison, "Compiler analysis of the value ranges for variables," *IEEE Transactions on Software Engineering*, vol. SE-3, pp. 243–250, May 1977.

[30] K. Cooper, K. Kennedy, and L. Torczon, "The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment," *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 491–523, Oct. 1986.

[31] L. Torczon, *Compilation Dependences in an Ambitious Optimizing Compiler*. PhD thesis,

Dept. of Computer Science, Rice University, May 1985.

[32] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK User's Guide*. Philadelphia: SIAM Publications, 1979.

[33] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck, "Supercomputer performance evaluation and the Perfect benchmarks," in *Proceedings of the 1990 ACM International Conference on Supercomputing*, (Amsterdam, The Netherlands), June 1990.

[34] A. Porterfield, *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989. Available as Rice COMP TR88-93.

[35] G. Goff, K. Kennedy, and C. Tseng, "Practical dependence testing," Tech. Rep. TR90-142, Dept. of Computer Science, Rice University, Nov. 1990. To appear in the *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[36] Z. Li, "Private communication," Oct. 1990.

[37] C. M. Rosene, *Incremental Dependence Analysis*. PhD thesis, Rice University, March 1990. Available as Rice COMP TR90-112.

[38] T. Gross and P. Steenkiste, "Structured dataflow analysis for arrays and its use in an optimizing compiler," *Software—Practice and Experience*, vol. 20, pp. 133–155, Feb. 1990.