# Experience with Interprocedural Analysis of Array Side Effects

*Paul Havlak*
*Ken Kennedy*

**CRPC-TR90063**
**August, 1990**

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

# Experience with Interprocedural Analysis of Array Side Effects

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

## 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 is inconvenient for 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:** parallelization, data dependence, interprocedural analysis, array sections

## 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 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.

This is because, in the absence of better information, a compiler 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. For example, consider the following loop:

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

We can parallelize this loop if we discover that `SOURCE` can only modify locations in the `I`th column of `A`. In other words, classical interprocedural analysis, which discovers which variables are used and which are

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

defined as side effects of procedure calls, is not good enough. Instead, we must be able to determine the *subarrays* of program arrays that are affected by a call.

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: elements, rows, columns and diagonals [CK88a, Cal87]. This paper describes an implementation of regular section analysis in the Rice Parallel Fortran Converter (PFC) [AK82], an automatic parallelization system that also computes dependences for the ParaScope programming environment[BKK+89]. The overriding concern in the implementation is that it be efficient enough to be incorporated in a practical compilation system.

We also examine the performance of regular section analysis on two benchmarks: the LINPACK library of linear algebra subroutines and the Rice Compiler Evaluation Program Suite (RiCEPS), a set of complete application codes from a variety of scientific disciplines.

## 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 [AC72]. This precisely represents the effects of the procedure with 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.

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. More formally, given a call to procedure p at statement $S_1$ and an array global variable or parameter $A$, we wish to compute:

- the set $M_{S_1}^A$ of locations in $A$ that may be modified by p called at $S_1$ and
- the set $U_{S_1}^A$ of locations in $A$ that may be used by 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

$$M_{S_1}^A \cap U_{S_2}^A \neq \emptyset.$$

### 2.1 Evaluation of Other Methods

Following Callahan and Kennedy [CK88a], we use the following criteria to evaluate interprocedural algorithms for computing array access sets:

- the complexity of representing the sets $M_{S_1}^A$ and $U_{S_2}^A$,
- the cost of merging two representations to summarize the accesses of multiple statements (since we use a lattice of representations, we call this the *meet* operation),
- the cost of determining whether the *intersection* of two representations is empty (dependence testing),

- the cost of computing representations for all procedures in a program, and
- the precision of the representations, since these can only approximate the true access sets.

**Classical Methods.** The classical methods of interprocedural summary analysis compute, for each parameter and global variable, two bits of information which indicate whether or not that variable *may* be modified and *may* be used [Ban78, Bar77, CK85]. This simple representation has a very efficient meet and intersection, consisting of logical operations on single bits. Furthermore, there exist algorithms that compute complete solutions using a number of meets that is linear in the number of procedures and call sites in the program, even when recursion is permitted [CK88b].

Unfortunately, our experiences with PFC and PTOOL indicate that this summary information is too coarse for dependence testing and the effective detection of parallelism. The problem is that the only access sets representable in this method are "the whole array" and "none of the array" (see Figure 1). This 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 et al.** Triolet's implementation in Parafrase calculates linear inequalities bounding the set of array locations affected by a procedure call [TIF86]. While this representation can be precise, it is also complicated: the meet operation requires finding the convex hull of the combined set of inequalities, and intersection uses a potentially exponential linear inequality solver.

**Burke and Cytron.** Burke and Cytron [BC86] propose representing each multidimensional array reference by linearizing its subscript expressions to a one-dimensional address expression. They also retain bounds information for loop induction variables occurring in the expressions.

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

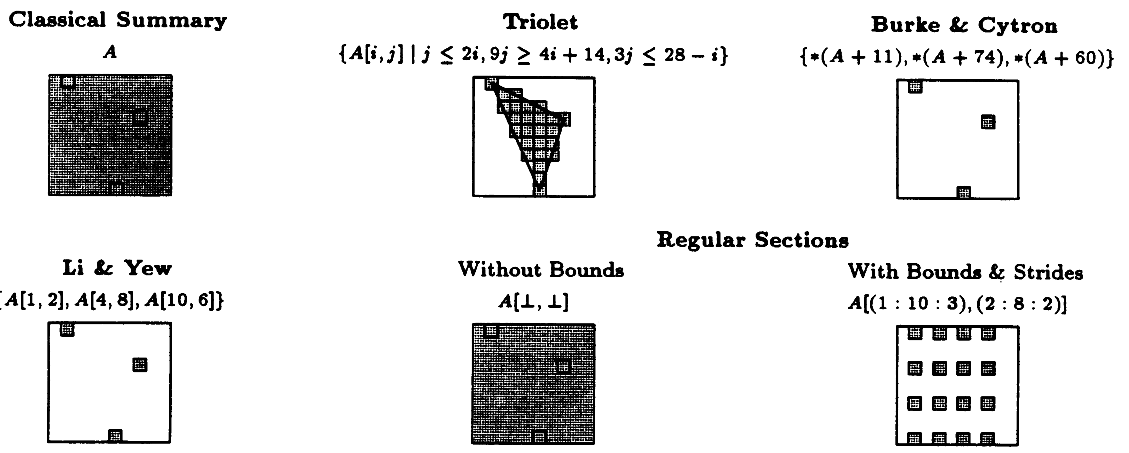

FIGURE 1: Summarizing the References $A[1,2]$, $A[4,8]$, and $A[10,6]$

loop induction variables. The first meet implementation is fast, and allows the use of standard dependence tests for intersection; however, testing for the independence of two lists of length $n$ requires $n^2$ invocations of the standard tests. The composite address expressions produced by the second meet implementation require an intersection test more complicated than standard dependence tests, which only handle linear expressions.

**Li and Yew.** Li and Yew implemented an algorithm in Parafrase using sets of *atom images* to describe the side effects of procedures [LY88a, LY88b]. Like the original version of regular sections described in Callahan's thesis [Cal87], 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.

Standard dependence tests are used to test for intersection of atom images, but they must be invoked $n^2$ times to test for the independence of two lists, as with Burke and Cytron's method. Nonetheless, Li and Yew state that their method is experimentally 2.6 times faster than Triolet's approach.

## 2.2 Regular Section Analysis

The above methods are expensive because, in pursuit of precision, they allow arbitrarily large representations of a procedure's access sets. Such precision may not be useful in practice; certain array access patterns are probably more common than others.

To avoid complicated meet and intersection operations, only simple access patterns should be represented. To avoid excessive intersection operations during dependence testing, only a limited number of access patterns should be used to summarize each procedure; for example, one pattern for the uses and one for the modifications to each array.

Researchers at Rice have defined several variants of *regular sections* to satisfy these design criteria [Cal87, CK88a, Bal89, BK89]. To avoid confusion, we will describe only the implemented version in detail.

Our implementation in PFC uses regular sections with bounds and stride information. These 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[1] of a variant subscript; and
- $\perp$, indicating no knowledge of the subscript value.

Bounded regular sections represent the same set of subarrays which can be written using triplet notation according to the proposed extensions to the ANSI Fortran standard [X3J89]. They allow both sparse regions such as stripes and grids and dense regions such as columns, rows, and blocks.[2]

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 can detect intersection of

[1] A stride of 1 is conservative and will often be omitted from examples.

[2] Triangular regions and others with diagonal bounds are not represented. Modifications to support such regions will be discussed in Section 6.

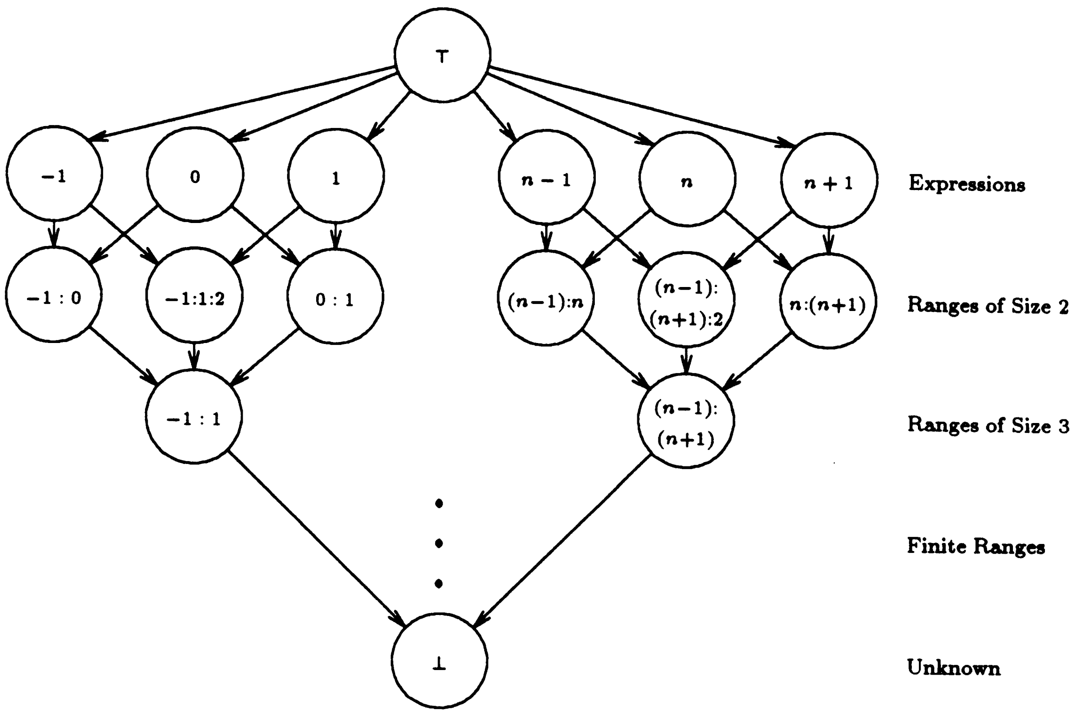

FIGURE 2: Lattice for Regular Section Subscripts

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.

Our implementation currently does not support recursion. Since our subscript lattice has infinite descending chains, some subtlety will be required to guarantee termination of the interprocedural propagation (see Section 4.2).

Regular section analysis is integrated into the three-phase interprocedural analysis and optimization structure of PFC [ACK86, CCKT86]. Algorithm 1 outlines the phases of regular section analysis. Sections 3 and 4 describe the construction of local sections and their propagation, respectively. Dependence analysis is an application of existing techniques in PFC.

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

### 3.1 Operations on Ranges

We represent a regular section as a vector of subscript values, which may be either expressions or ranges of expressions. Ranges are typically built to represent the values of loop induction variables, such as I in the

```
Local_Analysis:
    for each procedure
        for each array global/formal parameter
            save section describing shape
            for each reference
                build ranges for subscripts
                merge resulting section with summary MOD or USE section
        for each array actual parameter
            save section describing shape
        for each scalar global/actual parameter
            build 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 $\perp$
    iteratively translate summary sections into call sites
        merge translated sections into caller's summary

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

ALGORITHM 1: Overview of Regular Section Analysis

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 $S \geq 1$. Standardization sometimes causes loss of information; for example, we can reverse $[L, U, -2]$ with an accurate stride only if we can determine that $L - U$ is a multiple of 2. Therefore, we postpone standardization until it is forced by some operation, such as merging two sections.

Expressions in ranges are converted to ranges; for example, 2*I in the above example is represented as $[(2 * L + 1) : (2 * U + 1) : (2 * S)]$. Only adding or multiplying in an invariant expression can be handled precisely; approximations are built 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. This approach 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 values are those containing references to scalar parameters and global vari-

ables; therefore, our standardization and merge operations must be able to manipulate symbolic expressions and test them for equality.

## 3.2 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 [Kar76, RT81, RWZ88], their preliminary transformations and complexity make them difficult to integrate with PFC. Our implementation builds global value numbers using PFC's existing dataflow analysis machinery. Instead of propagating an arbitrary number of constraints per variable, we only track bounds and stride.

Primitive 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 $\perp$.

Value numbers for ranges arise by direct inspection of DO loops and by merging simpler value numbers whose constant difference can be determined.

For example, consider the following code fragment.

```
SUBROUTINE S1(A,N,M)
    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: $M * [1 : N : 1]$, which is converted to the range $[M : M * N : M]$.

## 3.3 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: 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 directly accessing 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 [Tor85, CCKT86].

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

## 3.4 Building Summary Regular Sections

With the above machinery in place, the USE and MOD regular sections for the local effects of a procedure can be constructed easily.

In one pass through the procedure, we examine each reference to an array that is a formal parameter or in a common block. 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 cumulative summary USE and/or MOD sections as appropriate, then discarded.

# 4 Interprocedural Propagation

Regular sections for formal parameters are translated into sections for actual parameters as we traverse edges in the call graph.[3] 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.

## 4.1 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

[3]For the experiments given in this paper, sections were propagated only for explicit array parameter passing. Local sections were constructed for common block arrays, but not propagated.

in subscript values of the formal section with their corresponding actual parameter values, then copy the resulting subscript values into the new section.

But in Fortran, there is 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. 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 were ranges or non-linear expressions, linearization effectively contaminates the other subscripts, greatly reducing the precision of dependence analysis. For this reason, we intend to linearize only as a last resort.[4]

We successfully avoid linearization in many cases where the translation of the referenced regular section is still a regular section, by assuming that declared bounds of arrays are obeyed.[5] Translation proceeds one dimension at a time, and is trivial in those dimensions where the formal and actual parameter have consistent sizes.

**Definition** *Given an actual parameter array* `A` *bound to a formal parameter array* `F`*, the* $k^{th}$ *dimension of* `A` *and* `F` *is* consistent *if*

- `A` *and* `F` *both have rank* $\geq k$,
- *all existing dimensions less than* $k$ *(to the left of* $k$*) are consistent,*
- *the* $k^{th}$ *subscript of the passed location of* `A` *is the same as the lower bound of the* $k^{th}$ *dimension of* `A`*, and*
- *the* $k^{th}$ *dimension of* `A` *and the* $k^{th}$ *dimension of* `F` *are declared to have the same size.*

We accurately translate subscripts for consistent dimensions; this precision is not affected if we linearize some rightmost dimensions. The translated subscript for a consistent dimension $k$ is the $k^{th}$ subscript of the referenced section of `F`, with the lower bound for that dimension of `F` added and the corresponding lower bound for `A` subtracted.

If all the dimensions of `F` are consistent, we are almost done. For each additional dimension of `A`, the translated section has the same subscript as the passed location of `A`.

If, instead, `F`'s bounds are inconsistent in just the rightmost dimension, the *referenced section* of `F` might still be translated without linearization. To check, we translate the referenced section as if its rightmost dimension was also consistent. If the resulting section does not violate the declared bounds for `A`, then we need not linearize.[6]

[4]Currently, our implementation gives up on subscripts that would require linearization, marking them as $\perp$.

[5]For one notorious case where this assumption is usually false, a rightmost dimension of "1" for a formal parameter, we ignore the declared bounds.

[6]When, in the leftmost dimensions, `F`'s referenced section has a stride that is a multiple of `A`'s declared size, *de*-linearization is possible. We do not know if this ever occurs in practice

### 4.2 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 then built in order, so that we never need to translate an incomplete regular section into a call site.

However, the new Fortran standard will probably have recursion [X3J89], and we plan an extension or reimplementation 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 [LY88b] and Cooper and Kennedy [CK88b] both describe approaches for propagating array sections which 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 $\perp$. Since subscripts can go to $\perp$ independently, it is even more useful to keep one small counter (e.g., two bits) per subscript.[7] 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 approach allows us to keep the infinite-depth lattice, with useful range information, while maintaining a time bound comparable to a bounded lattice.

## 5 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 [DBMS79] and the Rice Compiler Evaluation Program Suite of numerical programs. 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, a interactive dependence browser and program transformer [BKK+89].

**LINPACK** Analysis of LINPACK provides a basis for comparison with other methods for analyzing interprocedural array side effects. Both Li and Yew [LY88a] and Triolet [TIF86] 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.

[7]By giving up one subscript at a time, we can can let some subscripts go to $\perp$ while maintaining precise information about the others.

**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, modeling cache performance using an adapted versions of PFC [Por89]. Some RiCEPS and RiCEPS candidate codes have also been examined in a study on the utility of inline expansion of procedure calls [CHT90]. The six programs studied here are two RiCEPS codes **linpackd** and **track**) and four of the codes described in the inlining study.

## 5.1 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 noticed that this programming style was rare outside of those programs (**dogleg** and **linpackd**) that made extensive use of LINPACK.

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.[8] 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
```

That such modifications were sufficient reflects favorably on regular sections as a summary form. That they were needed indicates the desirability of a clearer Fortran programming style or more sophisticated handling of control flow (described in Section 6).

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

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

[8] Triolet made similar changes to the BLAS; Li and Yew avoided them by first performing interprocedural constant propagation.

## 5.2 Efficiency

We measured the total time taken by PFC to analyze the complete programs in our set. Parsing, local analysis, interprocedural propagation, and dependence analysis were all included in the execution times. Table 1 compares the analysis time required for classical interprocedural summary analysis alone with that for summary analysis and regular section analysis combined.[9]

The additional analysis time is comparable to that required to analyze programs after heuristically-determined inline expansion in Cooper, Hall and Torczon's study [CHT90].

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 [TIF86, LY88a]. However, for reasons cited in Section 2, we believe our method to be more efficient, particularly in dependence analysis, where we compare summary regular sections rather than lists of atom images.

## 5.3 Utility

We chose three measures of utility:

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

Table 2 compares the dependence graphs produced using classical interprocedural summary analysis alone and summary analysis plus regular section analysis.[10] LINPACK was analyzed without interprocedural constant propagation, since library routines may be called with varying array sizes.

The first set of three columns gives sizes of each dependence graph, including scalar and array references not incident on 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.

Table 3 examines the number of calls in LINPACK which could be parallelized after regular section analysis and gives related results reported by Li and Yew (who parallelized more calls than Triolet).

|  | All Dependences |  |  | Array Dep. on Calls in Loops |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
|  |  |  |  | loop carried |  |  | loop independent |  |  |
| source | IP | RS | % ⇓ | IP | RS | % ⇓ | IP | RS | % ⇓ |
| LINPACK | 12336 | 11035 | 10.5 | 3071 | 2064 | 32.8 | 1348 | 1054 | 21.8 |
| **dogleg** | 1858 | 1675 | 9.8 | 226 | 168 | 25.7 | 80 | 59 | 26.2 |
| **efie** | 12338 | 12338 | 0.0 | 177 | 177 | 0.0 | 81 | 81 | 0.0 |
| **euler** | 1818 | 1818 | 0.0 | 70 | 70 | 0.0 | 30 | 30 | 0.0 |
| **linpackd** | 496 | 399 | 19.6 | 191 | 116 | 39.3 | 67 | 45 | 32.8 |
| **track** | 4737 | 4725 | 0.25 | 68 | 67 | 1.5 | 27 | 26 | 3.7 |
| **vortex** | 1966 | 1966 | 0.0 | 220 | 220 | 0.0 | 73 | 73 | 0.0 |
| RiCEPS | 23213 | 22921 | 1.25 | 952 | 818 | 14.1 | 358 | 314 | 12.3 |

TABLE 2: Effects of Regular Section Analysis on Dependences

[9] 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 10 percent or less.

[10] 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.

These results indicate that, at least for LINPACK, there is no benefit to the extra precision of Triolet's and Li and Yew's methods. We detected most, if not all, of the parallelizable calls in DO loops using regular section analysis.

Most (17) of the call sites were parallelized in ParaScope, based on PFC's dependence graph, with no transformations being necessary.

Bulleted entries (•) indicate calls which were recognized as parallel on the basis of classical interprocedural summary analysis alone. The parallelism was apparent in ParaScope, but the transformations required to achieve it are not yet supported.

Starred entries (⋆) indicate parallel calls which were correctly summarized by regular section analysis, but for which a bug in PFC's symbolic dependence analysis prevented parallelization. While PFC seems to have missed a parallel call in `QRDC`, we were unable to find a fifth parallel call by inspection.

Two calls in the RiCEPS programs were parallelized: one in `dogleg` and one in `linpackd`.[11] 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.

## 6 Future Work

Our experiments have shown the suitability of regular section analysis for discovering parallel calls in loops, given an appropriate Fortran coding style. Future work should determine the importance of those idioms which do not fit into this style and the extensions needed to handle them.

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

TABLE 3: Parallelization of LINPACK

[11] We were surprised to discover that, in the inlining study, no commercial compiler was able to detect the parallel loop in dogleg even after the call had been inlined.

## 6.1 Improved Scalar Analysis

Consider the following example:

```
SUBROUTINE DSCAL(N, DA, DX, INCX)
    DOUBLE PRECISION DA, DX
    IF (INCX .NE. 1) THEN
        DO I = 1, N*INCX, INCX
            DX(I) = DA * DX(I)
        ENDDO
    ELSE
        DO I = 1, N
            DX(I) = DA * DX(I)
        ENDDO
    RETURN
END
```

The two sections, DX[1 : N * INCX : INCX] and DX[1 : N : 1],cannot accurately be merged without knowing the value of INCX. Even interprocedural constant propagation will not help us to build an accurate section if different values are passed to INCX.

One solution would be to add control dependence analysis to the local phase and annotate regular section subscripts with control conditions based on formal parameters. The new merge operation for value numbers then would then maintain separate value numbers for disjoint control conditions. By attaching the condition on INCX to the value numbers it affects, we can postpone merging the sections until we have propagated both into a call site, where INCX may be constant (even if each call site has a different constant value). This technique is compatible with our strategy for efficient interprocedural analysis.

## 6.2 More General Access Sets

One can write loops with parallel calls whose data decomposition cannot be described with bounded regular sections. Without too much damage to the generally efficient framework of regular sections, some additional shapes can be represented. These should be added if such loops are an important source of parallelism and cannot easily be rewritten.

Both the regular sections originally described by Callahan and Kennedy [Cal87, CK88a] and the Data Access descriptors developed by Balasundaram and Kennedy [Bal89, BK89] allow diagonal constraints. Balasundaram provides efficient merge and intersection operations for the resulting diagonal and triangular regions.

In the event that tinkering with the shapes allowed is insufficient to represent the effects of some procedures, a last option is to keep a bounded-length list of distinct regular sections (e.g., with length less than or equal to the rank of the array). The key problem with such an extension is how to decide which sections to merge when the list would otherwise become too long. The goal would be to merge sections that already have a substantial overlap.

## 6.3 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.

We could use the same lattice of ranges for the subscripts of killed sections as we used for summarizing USE and MOD — however, since kill analysis must produce *underestimates* of the affected region in order to be conservative, the lattice would need to be inverted.

The success of this approach would also require an intraprocedural dependence analysis capable of using array kill information, such as those described by Rosene [Ros89] and by Gross and Steenkiste [GS90].

## 7 Conclusions

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 which 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.

## References

[AC72] F. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[ACK86] Randy Allen, David Callahan, and Ken Kennedy. An implementation of interprocedural analysis in a vectorizing fortran compiler. Technical Report TR86-38, Dept. of Computer Science, Rice University, May 1986.

[AK82] J. R. Allen and K. Kennedy. PFC: a program to convert fortran to parallel form. Technical Report MASC-TR 82-6, Dept. of Mathematical Sciences, Rice University, March 1982.

[Bal89] Vasanth Balasundaram. *Interactive Parallelization of Numerical Scientific Programs*. PhD thesis, Rice University, July 1989. Also available as Rice COMP TR89-95.

[Ban78] J. Banning. *A Method for Determining the Side Effects of Procedure Calls*. PhD thesis, Stanford University, August 1978.

[Ban86] 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, April 1986.

[Bar77] J. Barth. An interprocedural data flow analysis algorithm. In *Conference Record of the Fourth ACM Symposium on the Principles of Programming Languages*, Los Angeles, January 1977.

[BC86] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175, June 1986.

[BK89] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, June 1989.

[BKK+89] V. Balasundaram, K. Kennedy, U. Kremer, K. S. McKinley, and J. Subhlok. The ParaScope Editor: An interactive parallel programming tool. In *Supercomputing '89*, November 1989.

[Cal87] D. Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Rice University, March 1987.

[CCKT86] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, June 1986.

[CHT90] K. Cooper, M. Hall, and L. Torczon. An experiment with inline substitution. Rice COMP TR-90-128, Dept. of Computer Science, Rice University, August 1990.

[CK85] 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.

[CK88a] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[CK88b] 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.

[DBMS79] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK User's Guide.* SIAM Publications, Philadelphia, 1979.

[GS90] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software-Practice and Experience*, 20(2):133–155, February 1990.

[Har77] W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[LY88a] Z. Li and P.-C. Yew. Efficient interprocedural analysis for program parallelization and restructuring. In *ACM SIGPLAN PPEALS*, pages 85–99, 1988.

[LY88b] 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, January 1988.

[Por89] Allan Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, May 1989. Also available as Rice COMP TR88-93.

[Ros89] C. M. Rosene. *Dependence Analysis for a Multiprocessor Programming Environment.* PhD thesis, Rice University, 1989.

[RT81] J. H. Reif and R. E. Tarjan. Symbolic program analysis in almost-linear time. *SIAM Journal on Computing*, 11(1):81–93, February 1981.

[RWZ88] 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*, pages 12–27, San Diego, CA, January 1988.

[TIF86] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 176–185, June 1986.

[Tor85] L. Torczon. *Compilation Dependences in an Ambitious Optimizing Compiler.* PhD thesis, Dept. of Computer Science, Rice University, May 1985.

[X3J89] X3J3 Subcommittee. *American National Standard for Information Systems Programming Language Fortran: S8 (X3.9-198x) Draft S8, Version 111.* ANSI, March 1989.