An Array Section Analysis Package for the `D System`

Nathaniel McIntosh

October 23, 1995

Contents

Chapter 1

Description

1.1 Abstract

This document describes an implementation of a general-purpose interprocedural array section analysis package recently added to the `D System`. We describe some of the challenges involved with array section analysis, outline our implementation, and discuss some ways in which the code could be improved in the future.

1.2 Introduction

Interprocedural array section analysis seeks to determine the set of locations within a given array that are read or written at a given call site. It is analogous to scalar interprocedural MOD/REF analysis in that it summarizes effects of entire procedures. Section information can be used for a variety of applications, including dependence analysis (for parallelization) and communication analysis (in the context of an HPF compiler).

```
program root
   integer j, x(100,100), t, u, zz
   read *, u, zz
   do j = 1, 99
      x(1,j) = 0
      call leaf(x, j, u, 2, zz)
   enddo
end

subroutine leaf(lx, lj, lu, lm, lzz)
   integer i, lx(100,100), lj, lu, lzz
   do i = 3, lu
      lx(i,lj) = lx(i,lm) + lzz
   enddo
end
```

Figure 1.1 A motivating example

For example, in the program shown in Figure 1.1, in order to safely parallelize the `do j` loop, a compiler would need to have detailed information about the regions within the array `x` read and written by the call to `leaf` within the loop. Similarly, an HPF compiler examining the `do j` loop could benefit from array section information when placing communication, perhaps by hoisting communication interprocedurally from within the loop in `leaf` out into `root`.

Researchers have come up with various ways to do array section analysis (see [HK91] for a comprehensive bibliography). These techniques all use some form of summary data structure to describe the regions within an array that are accessed in some part of the program. Our work uses a section representation called the *Data Access Descriptor*, developed by Balasundaram in his thesis [Bal89]. We also draw on ideas from the work of Havlak and Kennedy [HK91].

1.2.1 Outline

In section 1.3, we describe some of the important goals and requirements we had for this project. In the rest of the sections in this chapter, we describe in general terms the components of the analysis and the various tasks that have to be performed. In chapter 2, we describe the actual implementation in the `D System`.

1.3 General goals and requirements

As is traditional for interprocedural analysis, providing a system that supports separate compilation is very important. We address this problem by using a split-phase design, composed of a local phase that is run once for every procedure, and an interprocedural phase that is run for the entire program [CKT86].

A second important requirement for our work was to provide good support for symbolic values whenever possible (as opposed to requiring that programs use compile-time constants for loop bounds, array dimensions, etc).

Finally, we decided to try to avoid using exponential algorithms. Some researchers have proposed using very general representations of array sections (ex: unrestricted systems of linear inequalities); these representations often require potentially exponential algorithms for operations like union or intersection).

1.4 Array section analysis overview

1.4.1 Local phase

The local phase for array section analysis gathers a number of important pieces of information. First, for each array access within the procedure in question, the local phase gathers summary information (i.e. a *Data Access Descriptor*) to represent the regions within the array that are accessed. Second, each call site within the routine is examined to locate cases where a slice or section of an array is passed to a called routine; in such cases, the local phase stores information about the "shape" of the actual. Finally, the local phase looks for call sites within loops, and stores context information about the loops in question.

1.4.2 Interprocedural phase

If the program being analyzed is non-recursive, then the interprocedural phase can be accomplished with a single sweep over the call graph from bottom to top. Prior to the propagation, the summary information at each node is initialized with the summary data collected during the local phase. Then for each node N we perform the following three steps:

1. **Edge translation**: for each edge $N \rightarrow M$ leaving N, we take the sections at M and translate them across the edge, applying reshapes if necessary and translating symbolics from the context of M to N. If the call site for edge $N \rightarrow M$ is contained in a loop, take this into account after M's section has been translated.
2. **Meet edge summaries**: for each array, union together all of the sections for all of the edges leaving X.
3. **Add local summaries**: for each array, union the result of the Meet operation with the local summaries at the node.

This process is applied to each node in the call graph, in post-order (i.e. depth-first search finish time), and the results are stored as annotations in the call graph.

1.4.3 Localization

Prior to actually using section information within a given procedure, we have to "localize" it to the procedure. Primarily this means adjusting the section bounds to reflect the array bounds declarations of the local procedure and not those of a called procedure (which might have declared the array differently). For example, consider the program in Figure 1.2. Suppose that we are interested in getting section information for the variable "cx" at the call to "mid" within the top level program "example". This variable is declared as `cx(100,100)` within "example", but in the routine "low", it is declared as `cx(0:99, 0:99)` (i.e. the lower bounds are different). This has to be dealt with before we can easily use the section info.

```
program example
   common /foo/ cx, cy
   integer cx(100,100), cy(100,100), b(3)
   call writecol(cy(1,2), 1, 100, b(2))
   call mid(cy)
   end

subroutine mid(sy)
   integer sy
   call low(sy)
end

subroutine low(sy)
   integer sy, sz
   common /foo/ cx, cy
   integer cx(0:99, 0:99), cy(11:110, 11:110)
   do j = 10, 20
      call writecol(sy, 1, 100, sz)
   enddo
   call writecol(cx(2,3), 10, 20, sz)
end

subroutine writecol(x, l, u, s)
   integer x(0:*), l, u, s
   do i = l, u
      s = s + x(i)
   enddo
end
```

Figure 1.2 Example program

1.5 Challenges

There are a few aspects of array section analysis that are particularly difficult and deserve special mention. In this section we discuss a few of them briefly.

1.5.1 Call sites within loops

In general, building a section from a subscripted array reference within a loop is a fairly straightforward task – for us, it consists primarily of the application of Banerjee's inequality [Ban88], a relatively well-understood task (more or less equivalent to dependence testing). The infrastructure that supports this sort of thing has traditionally been used in an *intra*procedural setting, however.

Consider the call to **`writecol`** within the "do j" loop in the example program in Figure 1.2. Here we have a scenario where a section is initially built in the context of one function (i.e. when the local phase processes **`writecol`**), the section is then translated across the call site, at which point Banerjee's inequality must be applied again (in order to take into account the bounds of the "do j" loop). If separate compilation is to be preserved, the second application of the inequality can't take place until during the interprocedural phase. A consequence of this is that a fair amount of the section analysis machinery must be capable of operating during *either* the local phase or the interprocedural phase.

Scenarios like the one depicted above also place a lot of strain on the interprocedural scalar symbolic analysis – without precise translation of symbolic values across call sites, it's very difficult to get any meaningful information in such situations.

1.5.2 Reshapes

Consider the second call to **`writecol`** within the function `low` in the example program in Figure 1.2. Here we have a 2-dimensional array "sy" being passed to "writecol" and received as a 1-dimensional array. In addition, we pass a particular column within "sy", as opposed to the beginning of the array. We refer to this sort of thing as a reshape. The difficulty that it presents to the section analysis code is that a particular section may begin life with a given dimensionality and then acquire additional dimensions as it gets translated upwards across a call site. There is also a need for dimension conformability checks to make sure that the information being propagated is meaningful. See section 2.2.2 for a description of the specific types of reshapes handled by our implementation.

Chapter 2

Implementation

In this chapter we describe the actual implementation of the array section analysis package within the `D System`.

2.1 Local phase implementation

The class `ArraySectionModAttr` (see section 2.4.19) encapsulates the local phase for array sections. It contains the method `ArraySectionModAttr::ComputeUpCall`, which computes a `ProcArraySectionInfo` (see section 2.4.18) object for each procedure in the given Fortran source file. `ComputeUpCall` in turn calls the helper routine `ComputeProcArraySectionInfo`.

2.1.1 `ComputeProcArraySectionInfo`

This driver routine calls a series of helper functions to assemble the various components that go into a `ProcArraySectionInfo` object. It calls the helper routines `ComputeLoopContextTable`, `ComputeArraySectionSummary`, `ComputeFormalShapes`, and (indirectly) `CollectArrayShapeInformation`.

2.1.2 `ComputeLoopContextTable`

This helper routine looks for procedure invocations in loops. When it finds such an invocation, it adds a `LoopContext` (see section 2.4.12) object for the loop to the `LoopContextTable` (see section 2.4.13) object it is building.

2.1.3 `ComputeArraySectionSummary`

This helper routine scans through the procedure looking for subscripted references; when it encounters the right sort of reference, it builds a DAD (see section 2.4.8) for the reference and adds it to an `ArraySectionSummary` (see section 2.4.11) object, which is then returned.

2.1.4 `ComputeFormalShapes`

This helper routine examines the array formals of the given subroutine and records their dimensions.

2.1.5 `CollectArrayShapeInformation`

This routine examines the actuals at a call site and records their shapes (in the form of `ArrayShapeInfo` (see section 2.4.14) objects).

2.1.6 A note on derived "Builder" classes

It's worth noting at this point that many of the internal components of the `ProcArraySectionInfo` object (example: `SimpleSection`) have derived classes with "`Builder`" suffixes (example: `SimpleSectionBuilder`). In all cases, the `XXXBuilder` class does not add any additional data to the class, it simply exports a `Create` routine that allows the object to be built from some other component (an AST, for example).

There is a very implementation-specific reason for this. In order for objects like `DataAccessDescriptor` to be used both by the interprocedural phase code and the local phase code, they must be located relatively late in the link order. As a result, the directory containing them is *earlier* in the link order than the various tools that need to be used to build them (ex: the CFG, the value numbering package, etc). As a result, we split them into two pieces: the main class (which lives in the `libs/ipAnalysis/ipInfo` directory and acts primarily as a container class) and a `Builder` subclass (which lives in the `libs/moduleAnalysis/sections` subdirectory).

2.2 Interprocedural phase implementation

The most important class in the interprocedural phase is `ArraySectionDFProblem` (see section 2.4.2). It contains the guts of the interprocedural phase, and it orchestrates all of the various tasks that have to be done. `ArraySectionDFProblem` inherits from the class `CallGraphFlowInsensitiveDFP`, so in fact much of the code for walking the call graph, storing intermediate results, etc., resides in the base class. The interesting routines in `ArraySectionDFProblem` are all methods of `CallGraphFlowInsensitiveDFP` that have been overridden. The most important methods in `ArraySectionDFProblem` are described in the following five sections.

2.2.1 `ArraySectionDFProblem::InitializeNode`

This routine initializes a `CallGraphNode`; it receives the `ProcArraySectionInfo` object computed during the local phase for the function in question and "unpacks" it in preparation for the interprocedural propagation. It creates a new `ValIPMgr` object for the call graph node. It takes the auxiliary information for the node (including things like the `LoopContextTable` and the `ArraySectionCallSiteContextTable` for the node) and stores them in a table for later use. As a final result it produces an `ArraySectionAnnot` for the node containing the local mod/ref summaries for the node.

2.2.2 `ArraySectionDFProblem::NodeToEdge`

This routine is passed a `CallGraphEdge` and a pointer to the completed `ArraySectionAnnot` for the node corresponding to the sink of the edge. It takes the `ArraySectionAnnot` and translates it back into the context of the calling function. This means mapping formals to actuals, handling reshapes, and handling call sites within loops.

A note on reshapes: in our current implementation, we handle the common cases of reshapes without handling the most general scenarios. In general, we handle situations where arrays are passed to scalars (and vice versa) and situations where the dimensions of the actual are conformable to the leading dimensions of the formal. Other cases (example: a 1-D array being passed to a 2-D array) are not handled – we instead simply set the entire section to "unknown".

More general algorithms for handling reshapes exist, however they tend to require more general representations for sections than DADs (i.e. unrestricted sets of linear inequalities) [CI95, HML+95].

2.2.3 `ArraySectionDFProblem::Meet`

Given a call graph node X with out-edges $X \rightarrow W$, $X \rightarrow Y$, ..., $X \rightarrow Z$, this routine operates as a means of collecting together all of the summaries of the called routines $(W, Y, \ldots, Z)$. It is called once for each edge. The end result of the `Meet` is the merge of all of the sections of the called routines.

2.2.4 `ArraySectionDFProblem::AtNode`

Here the result of the `Meet` is combined with the local effects (the summary computed by `InitializeNode`. The end result of `AtNode` corresponds to the final solution for the call graph node.

2.2.5 `ArraySectionDFProblem::FinalizeNodeAnnotation`

This routine installs the annotation computed in `AtNode` as the finished result of the analysis.

2.3 Localization

The class `ArraySectionModAttr` provides a method, `ArraySectionModAttr::LocalizeAnnotation`, which will localize an `ArraySectionAnnot` to a particular procedure. This routine operates by iterating through the summary information in the annotation, localizing all internal value numbers, and applying offsets to section bounds if necessary.

2.4 Important C++ Classes

The following is a list of the important classes that are used during array section analysis. Most of these classes are primarily container classes; they hold the results from some pass of the analysis (local or interprocedural). In the following section, N refers to the number of dimensions in the array being summarized/described.

2.4.1 `ArraySectionAnnot`

This class inherits from `DataFlowSet`; it is an annotation that is placed on `CallGraphNode`s or `CallGraphEdge`s. It serves mainly as a container for an `ArraySectionSummary` object (see section 2.4.11).

2.4.2 `ArraySectionDFProblem`

This class inherits from `CallGraphFlowInsensitiveDFP`. Its job is to take a `CallGraph` and the local info for the array sections problem and run the interprocedural phase for array sections, and then annotate the call graph with `ArraySectionAnnot` annotations. It contains most of the driver code for the interprocedural phase of array sections.

2.4.3 `BoundaryPair`

The class `BoundaryPair` contains a pair of `ValNumber`s; they represent the upper and lower bound on a given dimension within the array (or a given diagonal constraint).

2.4.4 BoundaryPairMatrix

The class `BoundaryPairMatrix` is the chief data structure that makes up the representation of the class `SimpleSection`. It contains an NxN matrix of `BoundaryPairs`. The coordinates of a location within matrix determine the particular type of constraint (rectangular boundary vs. diagonal boundary) held by the location. The element $[K, K]$ within the matrix stores the constraints on dimension K. The element $[P, Q]$, where $P < Q$, stores the constraints on the diagonal $P + Q$. The element $[R, S]$, where $R > S$, holds the constraints on the diagonal $R - S$.

2.4.5 SimpleSection

The class `SimpleSection` implements the simple section abstraction described in Vas's thesis [Bal89]. It currently contains only a `BoundaryPairMatrix` object (along with the dimension of the array in question).

2.4.6 ReferenceTemplate

The class `ReferenceTemplate` implements the reference template abstraction described in Vas's thesis [Bal89]. It contains a vector of `ReferenceTemplateEntry` structure, each of which classifies the access in a particular dimension. The sizes of each dimension are also stored in this object.

2.4.7 DimensionTraversalOrder

The class `DimensionTraversalOrder` implements the traversal order abstraction described in Vas's thesis [Bal89].

2.4.8 DataAccessDescriptor

The class `DataAccessDescriptor` is the main representation for an array section used in our analysis. It consists of a `DimensionTraversalOrder`, a `ReferenceTemplate`, and a `SimpleSection`. See Vas's thesis for more info [Bal89].

2.4.9 DADList

The class `DADList` is simply a singly linked list of DAD's. If a given `DADList` contains multiple DADs, the implicit assumption is that these DADs represent successive accesses to a given array and need at some point to be unioned together to form a final result. Not performing the union operation immediately, however, can increase precision in some situations. As a result, a given `DADList` may contain multiple entries at various points in the analysis and a single entry at other points.

2.4.10 ArraySectionSummaryEntry

This class contains pointers to two `DADLists`, one holding the referenced region(s) of a given array and one holding the modified region(s). This class also provides mechanisms for controlling when the sections in its internal `DADLists` are unioned together.

2.4.11 ArraySectionSummary

The class `ArraySectionSummary` is intended to hold array section summary information for a set of variables. `ArraySectionSummary` inherits from the class `VariableSet`, which explicitly divides the variables it contains into three categories: formals, globals, and locals. Locals and globals are maintained in name/offset/length representation in order to allow for reshaping of common blocks and equivalencing. `ArraySectionSummary` provides all of the functions of `VariableSet`, but in addition associates an `ArraySectionSummaryEntry` with each variable in the set.

2.4.12 `LoopContext`

This class stores symbolic information about the loop or loops surrounding a given call site (or other program construct). For each loop surrounding the point of interest, it records `ValNumbers` for the induction variable, the upper and lower bounds, and the stride. It is used both during local analysis (to store the loop context surrounding a given array reference) and in the interprocedural phase (when dealing with call sites within loops).

2.4.13 `LoopContextTable`

The `LoopContextTable` class is a table of $< X, Y >$ tuples where X is the node number of a given loop (presumably obtained from the function `ft_NodeToNumber`), and Y is a `LoopContext` object for that loop (and the other loops that surround it). `LoopContextTable` inherits from `WordObjectTableIO`.

2.4.14 `ArrayShapeInfo`

The class `ArrayShapeInfo` exists primarily to store information about the size and shape of an array or array section passed at a particular actual position within a call site. It contains two N-long vectors of `ValNumbers`, where N is the dimension of the array in question. The first records the size of each array dimension. The second vector records the offset within the dimension passed in a given actual position. For example: for the first actual ("`cy`") in the call to `writecol` within the top level subroutine in Figure 1.2, the dimension size vector would be $[100, 100]$ and the dimension offset vector would be $[0, 1]$.

2.4.15 `CallSiteArrayShapes`

This class stores information about all of the actual positions within a given call site – it is essentially a sparse vector of `ArrayShapeInfo` objects (whose length is the number of actual positions in the call site).

2.4.16 `ArraySectionCallSiteContext`

This class stores information about a particular call site. It inherits from `WordObjectIO`. It contains three items: the node number of the call site in question (this is stored implicitly as `WordObjectIO::id`, the node number of the loop containing the call site (or -1 if the call site is not enclosed in a loop), and a `CallSiteArrayShapes` object.

During local analysis, this object is stored inside an `ArraySectionCallSiteLocalInfo` object. When local analysis is complete and the interprocedural phase is taking place, it is subsequently kept within a `ArraySectionCallSiteContextTable` that is part of `ArraySectionDFProblem`.

2.4.17 `ArraySectionCallSiteLocalInfo`

This class inherits from `CallSiteLocalInfo`. Its job is simply to contain an `ArraySectionCallSiteContext` object.

2.4.18 ProcArraySectionInfo

This abstraction stores all of the local information for a given function. It contains an `ArraySectionSummary` object, a `LoopContextTable` object, a vector of `ArrayShapeInfo` objects (storing dimension sizes for formals) and a `ValIPMgr` to store the value table for all of the value numbers contained in its sub-objects. It is computed by the routine `ComputeProcArraySectionInfo`.

2.4.19 ArraySectionModAttr

This class is the module attribute for the array sections problem. Its primary job is to provide demand-driven computation of `ProcArraySectionInfo` objects for a given Fortran source file (i.e. the array sections local phase). It inherits from the class `ModuleLocalInfo`.

2.4.20 SubsRef

This object is an abstract subscripted array reference. It contains a vector of `ValNumbers`, one for each subscript position. It exists only within the local phase – no `SubsRefs` are used during the interprocedural phase.

2.4.21 LoopContextBuilder

This class inherits from `LoopContext`. It contains no additional data; its only function in life is to build the guts of a `LoopContext` from an `AST_INDEX` for the loop in question. See section 2.1.6 for a discussion of why we use the derived "Builder" class technique for this class.

2.4.22 ReferenceTemplateBuilder

This class inherits from `ReferenceTemplate`. It contains no additional data; its only function in life is to build the guts of a `ReferenceTemplate` from locally available info (i.e. the AST, the symbol table, the CFG, etc). See section 2.1.6 for a discussion of why we use the derived "Builder" class technique for this class.

2.4.23 SimpleSectionBuilder

This class inherits from `SimpleSection`. It contains no additional data; its only function in life is to build the guts of a `SimpleSection` from locally available info (i.e. the AST, the symbol table, the CFG, etc). See section 2.1.6 for a discussion of why we use the derived "Builder" class technique for this class.

2.4.24 DimensionTraversalOrderBuilder

This class inherits from `DimensionTraversalOrder`. It contains no additional data; its only function in life is to build the guts of a `DimensionTraversalOrder` from locally available info (i.e. the AST, the symbol table, the CFG, etc). See section 2.1.6 for a discussion of why we use the derived "Builder" class technique for this class.

2.4.25 DataAccessDescriptorBuilder

This class inherits from `DataAccessDescriptor`. It contains no additional data; its only function in life is to build the guts of a `DataAccessDescriptor` from locally available info (i.e. the AST, the symbol table, the CFG, etc). See section 2.1.6 for a discussion of why we use the derived "Builder" class technique for this class.

2.5 Source files

2.5.1 Shared abstractions/classes

```
libs/ipAnalysis/ipInfo/ArraySectionCallSite.C
libs/ipAnalysis/ipInfo/ArraySectionCallSite.h
libs/ipAnalysis/ipInfo/ArraySectionSummary.C
libs/ipAnalysis/ipInfo/ArraySectionSummary.h
libs/ipAnalysis/ipInfo/ArrayShapeInfo.C
libs/ipAnalysis/ipInfo/ArrayShapeInfo.h
libs/ipAnalysis/ipInfo/CallSiteArrayShapes.C
libs/ipAnalysis/ipInfo/CallSiteArrayShapes.h
libs/ipAnalysis/ipInfo/CoVarExpr.C
libs/ipAnalysis/ipInfo/CoVarExpr.h
libs/ipAnalysis/ipInfo/DADList.C
libs/ipAnalysis/ipInfo/DADList.h
libs/ipAnalysis/ipInfo/DataAccessDescriptor.C
libs/ipAnalysis/ipInfo/DataAccessDescriptor.h
libs/ipAnalysis/ipInfo/DimensionTraversalOrder.C
libs/ipAnalysis/ipInfo/DimensionTraversalOrder.h
libs/ipAnalysis/ipInfo/LoopContext.C
libs/ipAnalysis/ipInfo/LoopContext.h
libs/ipAnalysis/ipInfo/LoopContextTable.C
libs/ipAnalysis/ipInfo/LoopContextTable.h
libs/ipAnalysis/ipInfo/ProcArraySectionInfo.C
libs/ipAnalysis/ipInfo/ProcArraySectionInfo.h
libs/ipAnalysis/ipInfo/ReferenceTemplate.C
libs/ipAnalysis/ipInfo/ReferenceTemplate.h
libs/ipAnalysis/ipInfo/SimpleSection.C
libs/ipAnalysis/ipInfo/SimpleSection.h
libs/ipAnalysis/ipInfo/SubsRef.C
libs/ipAnalysis/ipInfo/SubsRef.h
libs/ipAnalysis/ipInfo/VariableInfo.C
libs/ipAnalysis/ipInfo/VariableInfo.h
```

2.5.2 Local phase code

```
libs/fileAttrMgmt/analysisAttributes/ArraySectionModAttr.C
libs/fileAttrMgmt/analysisAttributes/ArraySectionModAttr.h
libs/fileAttrMgmt/analysisAttributes/ComputeArraySection.C
libs/fileAttrMgmt/analysisAttributes/ComputeArraySection.i
```

2.5.3 General-purpose analysis code

```
libs/moduleAnalysis/sections/ArrayDimensionUtils.C
libs/moduleAnalysis/sections/ArrayDimensionUtils.h
libs/moduleAnalysis/sections/DataAccessDescriptorBuilder.C
libs/moduleAnalysis/sections/DataAccessDescriptorBuilder.h
libs/moduleAnalysis/sections/DimensionTraversalOrderBuilder.C
libs/moduleAnalysis/sections/DimensionTraversalOrderBuilder.h
libs/moduleAnalysis/sections/LoopContextBuilder.C
libs/moduleAnalysis/sections/LoopContextBuilder.h
libs/moduleAnalysis/sections/ReferenceTemplateBuilder.C
libs/moduleAnalysis/sections/ReferenceTemplateBuilder.h
libs/moduleAnalysis/sections/SimpleSectionBuilder.C
libs/moduleAnalysis/sections/SimpleSectionBuilder.h
libs/moduleAnalysis/sections/SubsRefBuilder.C
libs/moduleAnalysis/sections/SubsRefBuilder.h
```

2.5.4 Interprocedural phase code

```
libs/ipAnalysis/problems/sections/ArraySectionAnnot.C
libs/ipAnalysis/problems/sections/ArraySectionAnnot.h
libs/ipAnalysis/problems/sections/ArraySectionAnnotMgrs.C
libs/ipAnalysis/problems/sections/ArraySectionDFProblem.C
libs/ipAnalysis/problems/sections/ArraySectionDFProblem.i
```

2.6 Bugs

This section describes some of the outstanding bugs in the implementation, along with a list of the major bugs that have been fixed.

2.6.1 Unfixed bugs

None known at the moment.

2.6.2 Fixed bugs

1. The IP solver now checks for the presence of cycles in the call graph (and aborts if it discovers any).
2. The bugs that caused ipdriver to crash when reading previously computed analysis results from the database have now all been fixed.
3. The local phase now allows a restricted form of statement function use.
4. `ArraySectionAnnot` annotations can now be demanded for call graph edges in addition to nodes.

2.7 Future improvements

There are numerous aspects of this implementation that are not as strong as they could be; as with any large, complicated component of a parallelizing compiler, there is always room for improvement. The following is partial list of areas of concern.

1. Our implementation records stride information in the `DimensionTraversalOrder` abstraction, but doesn't try to take advantage of it anywhere. In particular, it is not examined during the DAD intersection operation.

2. As mentioned previously, we handle only the most common reshapes. With a little imagination it should be possible to increase the number of cases handled for formal and global reshapes.

3. Currently there are a set of fixed places in the IP phase where annotations are coalesced (i.e. all of the DADs on the `DADLists` for a given variable are unioned together). A better scheme would be to write some code which detects when a union operation can be applied without loss of precision, and to apply unioning eagerly in situations where no loss of precision would arise. This would increase efficiency without losing any precision.

4. There is currently no way to geometrically classify the sections generated by the analysis (i.e. rectangular vs. triangular). This would not be too difficult to do, and would be helpful for certain applications. The primary tool needed would be a redundant boundary test [Tsa94].

5. The current value numbering package does not do a good job of handling min/max operations in loop bounds – it treats them as variants instead of handling them symbolically. As a result, the more complicated forms of triangular and skewed loop nests can't be analyzed very well.

6. Union and intersection operations are described thoroughly in Vas's thesis for `SimpleSections`, but there are no algorithms given for either `ReferenceTemplates` or `DimensionTraversalOrders`. The algorithms used in our current implementation are somewhat simplistic and could be improved quite a bit (if this is needed) [Bal89].

7. The local phase currently uses multiple pass over the `AST` to collect different pieces of information (loop contexts, call site info, etc). It would almost certainly be more efficient to combine the various passes into a single sweep over the tree.

8. Statement function containing array references are not allowed by the local phase; it would be nice to remove this restriction. *Note: this may require major surgery on the SSA package to fix.*

2.8 Acknowledgements

A large part of the credit for this work clearly goes to Vasanth Bala (formerly Vasanth Balasundaram) for his thesis work. We are also very grateful for the symbolic analysis package in the `D System` developed by Paul Havlak, and the interprocedural analysis framework developed by Mary Hall et al and later refined by John Mellor-Crummey.

Bibliography

[Bal89] V. Balasundaram. *Interactive Parallelization of Numerical Scientific Programs.* PhD thesis, Dept. of Computer Science, Rice University, May 1989.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, MA, 1988.

[CI95] B. Creusillet and F. Irigoin. Interprocedural array region analyses. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, Columbus, Ohio, August 1995.

[CKT86] 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*, 8(4):491–523, October 1986.

[HK91] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[HML+95] M. Hall, B. Murphy, S. Liao, S. Amarasinghe, and M. Lam. Interprocedural analysis for paralellization. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, Columbus, Ohio, August 1995.

[Tsa94] H. Tsalapatas. Interprocedural array side effect analysis. Master's thesis, Rice University, February 1994.