INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeib Road. Ann Arbor, MI 48106-1346 USA
313/761-4700  800/521-0600
Interprocedural array side effect analysis

Tsalapatas, Hariklia Aris, M.S.

Rice University, 1994
RICE UNIVERSITY

Interprocedural Array Side Effect Analysis

by

Hariklia Tsalapatas

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Ken Kennedy
Professor of Computer Science

Keith Cooper
Associate Professor of Computer Science

Richard Tapia
Professor of Mathematical Sciences

Houston, Texas

March, 1994
Interprocedural Array Side Effect Analysis

Hariklia Tsalapatas

Abstract

In the absence of better information worst case assumptions must be made for the side effects of calls on arrays. Interprocedural array side effect analysis computes the side effects of calls on array variables. The results enhance the precision of dependence testing and enable interprocedural transformations.

This thesis presents and evaluates an interprocedural array side effect analysis more precise than existing efficient solutions, yet not as expensive as the precise methods. Array shapes are described by simple sections. The diagonal information available in simple sections allows precise representation of array shapes commonly encountered in practice. Most operations on simple sections are quadratic in the number of dimensions of the sections. Given that the number of dimensions of arrays used in practice is small, operations on simple sections are very efficient. The thesis focuses on the MOD, REF, and KILL data flow problems for arrays. The results are used to perform array privatization.
Acknowledgments

I would like to thank my advisor Ken Kennedy for his support, guidance, and contribution to this research. Paul Havlak and Kathryn McKinley for all the discussions, their suggestions, and their feedback. The people working on the ParaScope programming environment for making the project possible and enjoyable. Ivy Yorgensen for proof reading my thesis. Keith Cooper and Richard Tapia for their input and their suggestions. Finally, I would like to thank my parents for being a source of inspiration in my life.
Contents

Abstract ii
Acknowledgments iii
List of Illustrations vi

1 Introduction 1
1.1 Motivation ........................................... 1
  1.1.1 Interprocedural analysis on arrays ............. 2
  1.1.2 Research goal .................................. 5
1.2 Overview ........................................... 6

2 Simple Sections 7
2.1 Description of simple sections .................... 7
2.2 Big and Small Union of Simple Sections ........... 12
2.3 Big and Small Intersection of Simple Sections ..... 17

3 Overview of Array Section Analysis 21
3.1 Interprocedural array section analysis ............ 21
3.2 Use of array section analysis in interprocedural transformations ... 24
3.3 Use of array section KILLS in the parallelization of loops ..... 24
3.4 Interprocedural strategy ................................. 25

4 Local Analysis 27
4.1 Goal of local analysis ................................. 27
4.2 Direct MOD and REF summaries .................... 29
4.3 Local KILL information ................................. 30
4.4 Preparation for the interprocedural phase ........... 33

5 Interprocedural Analysis 34
5.1 Algorithm for interprocedural array side effect analysis .......... 34
5.2 Translation into a call context .................................. 38
5.3 Accumulation of KILL information .......................... 40
5.4 Final local analysis ............................................ 43
5.5 Array privatization ............................................. 46

6 Experimental Validation ........................................... 48
  6.1 Implementation ............................................... 48
  6.2 Experimental results ..................................... 49

7 Conclusions ....................................................... 54
  7.1 Contribution .................................................. 54
  7.2 Future work .................................................. 55

Bibliography ......................................................... 56
Illustrations

2.1 Algorithm for building a simple section from the loop boundaries. 10
2.2 Test for redundant boundaries. 11
2.3 Examples of exact and inexact unions. 13
2.4 Algorithms for Big and Small Union. 14
2.5 Test for exact union. 15
2.6 Test for exact intersection. 16
2.7 The bounds of the x-y diagonal of the intersection must be tightened. 17
2.8 Algorithms for Big and Small Intersection. 18
2.9 Algorithm for tightening the redundant boundaries of a simple section. 19

3.1 Example control flow graphs. 22
3.2 Array sections enable interprocedural transformations. 23
3.3 Array section KILLS help in parallelization of loops. 26

4.1 Scalar interprocedural analysis is necessary to compute precise array section analysis solutions. 28
4.2 Outline of local analysis. 29
4.3 Algorithm for computing direct MOD and REF information. 30
4.4 Algorithm for computing direct KILL information. 32

5.1 Algorithm for interprocedural array section analysis. 35
5.2 Algorithm for accumulating direct MOD summaries. 37
5.3 Algorithm for translating a simple section into a call context. 39
5.4 Algorithm for accumulating the annotated control dependence graph. 41
5.5 Instance of the annotated control dependence graph. 42
5.6 Final phase. 44
5.7 Algorithm for computing simple sections privatizable with respect to each loop in a program 45
5.8 Algorithm for computing the cover of an SSA node. 46

6.1 Structure of ParaScope. 49
6.2 Results of experiments determining the effectiveness of the privatization algorithm. 50
6.3 Comments on the source programs used for array privatization. 50
6.4 Compilation time for local analysis and privatization. 51
6.5 Programs used to test the effectiveness of the interprocedural analysis. 52
Chapter 1

Introduction

Parallel computing has emerged as a low cost alternative to expensive hardware for increasing computation speed. To take full advantage of the potential of a parallel machine, an optimizing compiler must expose and exploit parallelism in the source programs. Parallel computing is particularly useful for processing the large amounts of data manipulated by scientific programs. Since a lot of the computation in these programs is performed in loops, it is important to determine whether the iterations of a loop can be executed in parallel without any synchronization. This is possible when the execution of an iteration of the loop does not depend on results produced by other iterations.

The use of procedure calls in loops is common practice, especially in scientific applications. Detecting parallelism in loops is difficult in the presence of procedure calls. To determine that no specific order of execution is required for the iterations of a loop, the compiler must prove that the side effects of the calls to their arguments in one iteration do not interfere with the side effects in another iteration. When the arguments of a procedure call are arrays, this task becomes even harder. This thesis presents and evaluates framework for performing side effect analysis on array variables across procedure boundaries.

1.1 Motivation

This thesis focuses on programs for scientific applications. The nature of these programs is highly compute-intensive, involving the manipulation of large amounts of data. Thus, computation speed is an important issue for the execution of scientific programs, making them good candidates for parallelization. These programs are usually several thousands of lines long. Their structure is complex, consisting of numerous nested loops and procedures. To model their input data, scientific programs use arrays extensively. Most of the computation is performed in loops, making loops
a significant source of parallelism. Optimizing compilers must take into account these characteristics of scientific programs to fully exploit the parallelism lying in them.

To detect and expose parallelism, optimizing compilers perform dependence analysis. Informally, there is a dependence between two statements, if one of them can use a value produced by the other at run time. Data dependencies impose a partial order on the statements of a program. Any execution order of the statements that preserves all dependencies preserves the semantics of the sequential program.

The iterations of a loop can be run in any order if no dependencies exist among two iterations (loop carried dependencies). Such a loop is said to be safely parallel, because the results of its execution are deterministic regardless of the order of execution of its iterations. If loop carried dependencies exist, the compiler usually rearranges the statements in the loop body in an attempt to disprove the dependencies. If some of the dependencies cannot be disproved, synchronization must be inserted to satisfy the execution order dictated by the dependencies.

To detect parallelism in the presence of procedure calls, an optimizing compiler must analyze the side effects of the calls on their arguments. That is, in the absence of better information, compilers must assume that any two calls can read and write the same memory locations, making the results of parallel execution of the loop nondeterministic. This complicates the task of parallelizing loops. However, since these loops occur commonly in practice, and are usually compute-intensive, it is desirable to parallelize them. The need to parallelize these loops has resulted in the development of algorithms that perform interprocedural analysis on the source programs. Interprocedural analysis summarizes, for each procedure call, the side effects occurring in the body of the invoked procedure.

1.1.1 Interprocedural analysis on arrays

Classical interprocedural side effect analysis only discovers the side effects of calls on scalar variables. It is common practice, however, to use array variables as arguments to procedure calls. In the absence of interprocedural analysis on array variables, compilers must be conservative about the side effects of a call to arrays. If a single element of an array is accessed in the body of the called procedure, the whole array must be assumed to be accessed. To safely exploit parallelism, it is important to determine the subarrays that are affected as a result of a procedure call.
A simple way to determine the side effects of a call is to perform inline expansion [24], that is, to substitute the call statement with the body of the called procedure. After inlining, dependence testing can be performed on a sequence of ordinary statements using existing dependence analyzers. Although simple and effective, inlining has limitations [12], [6]. The size of the loop body that contained the call may grow dramatically causing time and space explosion due to the non-linearity of array dependence analysis [25].

Since inline expansion is not always profitable, it is necessary to perform interprocedural array side effect analysis. The rest of this section presents a summary of work performed in this area.

Triolet regions. Triolet, Irigojn, and Feautrier proposed to represent the set of array locations accessed by a procedure call as a set of linear inequalities [21], [22]. This representation is precise for convex regions. Array accesses with non-unit stride or non-convex shapes are given convex approximations. The meet of two regions is computed by finding the convex hull of the combined set of inequalities. Intersection requires the use of a potentially exponential inequality solver. Analysis performed using Triolet regions is precise, but operations on the regions are expensive because of their complex representation.

Lists of array accesses. Two separate studies have been performed where the summary of the side effects of a call is represented as a list of all array accesses occurring in the body of the called procedure. The main disadvantage of these methods is that translation of a summary at a call site requires time proportional to the number of accesses in the body of the called procedure.

Burke and Cytron proposed representing each array access by linearizing its subscript expressions to a one dimensional address expression [15]. This method also maintains information about the bounds of loop induction variables occurring in the subscript expressions of an array access. The main advantage of this method is that it can handle arbitrary array reshaping. The main disadvantage is that if some subscript of the original array access is nonlinear the linearized array access is also nonlinear. This reduces the precision of dependence testing. Without linearization it might be possible to detect independence of the linear subscripts.

Li and Yew proposed describing the side effects of a procedure on an array using atom images [14]. Atom images represent precisely array accesses whose subscript
expressions are linear expressions of the loop induction variables. Atom images also provide bound information for the loops surrounding an array access. This representation is precise, but intersection of lists of atom images is expensive. Also, the computation of atom images for each access in the body of the called procedure is expensive.

Summary sections. Summary section methods describe the side effects of a call on an array using a single array section descriptor. The descriptor summarizes all accesses to locations of the array occurring in the body of the called procedure. The descriptor must be precise enough to describe the array shapes commonly encountered in practice, yet simple to allow efficient implementation of operations on sections. Several different array section representations have been proposed:

Callahan introduced restricted regular sections in his PhD dissertation [5]. This representation is limited to access patterns in which each subscript is a procedure invariant expression, unknown, or unknown but diagonal to one or more other subscripts.

This representation is efficient. The meet operation is quadratic to the number of dimensions of the array and the intersection operation is linear. However, restricted regular sections loose a lot of precision because they do not represent loop bound information, and therefore cannot be used for effective loop parallelization.

Anticipating the limitations of restricted regular sections Callahan and Kennedy introduced bounded regular sections [1], [5]. This representation includes bounds and stride information. The meet and intersection operations are linear to the number of dimensions of the arrays. Bounded regular sections were used by Havlak [6] to perform interprocedural array side effect analysis. Experiments showed that the precision of this analysis is comparable to the precision of atom images for arrays commonly encountered in practice [6].

Bounded regular sections do not include diagonal information. To overcome this limitation Balasundaram and Kennedy introduced Data Access Descriptors (DAD's) [2], [3], [4]. DAD's include both array shape and traversal order information. Array shapes are described by simple sections, a representation that includes lower and upper boundaries on each coordinate axis (array dimension), lower and upper boundaries on diagonals forming a 45° angle to an axis, and stride information. Simple sections describe precisely array shapes commonly encountered in practice, such as rectangular sections, triangular sections, rows, columns, diagonals, and trapezoidal
sections. Operations on simple sections are quadratic to the number of dimensions of the arrays.

1.1.2 Research goal

The studies described in the previous section concentrated on the MOD and REF data flow problems, that is, summaries of array sections that may be modified or may be referenced, respectively, as a side effect of a call. Another data flow problem that is very important in loop parallelization is KILL, that is, the summary of array sections that must be modified as a result of a procedure call. If an array section is killed before any use in the body of a loop, it can be privatized, that is, a private copy of the array can be assigned to each iteration, eliminating flow and output loop carried dependences on that array.

Work on interprocedural array KILL analysis has been performed by Itsuka [23]. This analysis uses rectangular array sections. If an array access cannot be precisely described by a section, minimum and maximum approximations are computed. Operations on sections are performed only when the result is precise. If the result is not precise, the minimum and maximum approximations of the input sections are kept in a list. Since multiple sections may be maintained for each array, this method requires a lot of space. Also, information about diagonals of array sections is not taken into account in the solution.

Studies on array privatization have been conducted by Maydan, Amarasinghe, and Lam [11], and by Li [20]. However, the analysis performed in these studies is confined to a single procedure and does not take into account the side effects of calls.

The goal of this thesis is to present and evaluate an interprocedural array side effect analysis that is more precise than existing efficient solutions, yet not as expensive as the precise methods. Array shapes are described by simple sections. The diagonal information available in simple sections allows precise representation of array shapes commonly encountered in practice. Most operations on simple sections are quadratic to the number of dimensions of the sections. Given that the number of dimensions of arrays used in practice is small, operations on simple sections are very efficient.

The analysis presented in the following chapters concentrates on the interprocedural MOD, REF, and KILL side effect problems for arrays. The solutions to those problems enhance the precision of dependence testing in the presence of procedure
calls. Particularly important is the interprocedural array KILL analysis. The results of this analysis are used to recognize privatizable arrays.

1.2 Overview

The analysis presented in this thesis is divided into a local phase, an interprocedural phase, and a dependence testing phase. This strategy has been chosen for two reasons. First, to minimize the number of passes over each procedure body. The necessity to merge the side effects of a procedure call with the rest of the accesses in a procedure might result in multiple passes over each procedure body. To avoid that, the analysis gathers information about direct accesses in each procedure in the local phase, and computes the side effects caused through procedure calls in the interprocedural phase, without direct examination of any procedure body. This reduces the number of passes over each procedure body to two, one in the local phase and one in the dependence testing phase. Second, by summarizing local information for each procedure in a separate phase, it is possible to avoid recomputation of all local information when some procedure is edited. Instead, only the local information for the edited procedure must be updated. The following chapters present a strategy for performing interprocedural array side effect analysis. The analysis has been implemented as part of the ParaScope programming environment developed at Rice University. ParaScope is an interactive tool that helps the user to detect and exploit parallelism in a sequential program.

The implementation is targeted to FORTRAN source code, since most scientific applications are written in FORTRAN. However, the described techniques are not restricted to FORTRAN programs. They are applicable to most block structured languages.

Chapter 2 describes in detail simple sections and operations performed on them. Chapter 3 describes the array side effect problems a solution to which is presented in later chapters. Also, examples are provided to illustrate the use of array side effect analysis for the parallelization of loops. Chapters 4 and 5 describe an algorithm for performing interprocedural array side effect analysis. Chapter 4 describes the local phase and chapter 5 the interprocedural phase of the analysis. Chapter 6 shows results from experiments performed to determine the precision of the analysis, and discusses its effectiveness.
Chapter 2

Simple Sections

To determine whether a loop can be safely parallelized, compilers use dependence analysis. In the presence of procedure calls dependence analysis becomes imprecise, unless information about the side effects of the calls on their arguments is available. To compute this information some compilers perform interprocedural data flow analysis. In particular, they compute the MOD, REF, and KILL sets for each procedure. These sets contain, respectively, the variables that may be modified, may be referenced, or must be modified as a result of a procedure call. Without some representation of array sections, the computation of the MOD, REF and KILL sets for array variables is imprecise and conservative: for example, when a single element of an array is referenced one must assume that the whole array is referenced. Array sections enable the computation of subarrays that are accessed due to a procedure call, resulting in more precise REF, MOD and KILL solutions.

Representations of array sections have been proposed by Triolet, Irigoin, and Feautrier [21], by Li and Yew [12], by Callahan and Kennedy [1], [5], by Burke and Cytron [15], and by Balasundaram and Kennedy [4], [2]. In the following, array sections are described by simple sections, a representation proposed by Balasundaram and Kennedy [2], [4]. The elegance of simple sections lies in their ability to compactly represent the array shapes commonly encountered in practice while having a low cost for the operations performed on array sections.

2.1 Description of simple sections

To achieve the properties of simple sections mentioned above, that is, compact representation of commonly encountered array shapes and easy and fast implementation of array section operations, it is necessary to impose some restrictions on the shapes that can be described by an array section. The most important restriction is that a simple section can only be bounded by simple boundaries.
Definition 2.1 Given an n dimensional space with coordinate axes $x_1, x_2, \ldots, x_n$, a simple boundary is a hyperplane of the form $x_i = c$ or $x_i \pm x_j = c$, where $x_i, x_j$ are two different coordinate axes and $c$ is a constant.

Thus, a simple boundary can either be parallel to some axis or it can form a 45° angle with a pair of coordinate axes. The constant $c$ can be either numeric or symbolic. A constant is symbolic if it is an arithmetic expression involving program variables.

The definition of a simple section is given by Balasundaram [2].

Definition 2.2 A simple section is a complex polytope with simple boundaries.

Some of the commonly occurring array shapes that can be precisely described by a simple section are the entire array, rectangular shapes, a row, a column, a triangular section, and a diagonal. Depending on the array side effect problem being solved an array section that cannot be precisely described by a simple section is represented by a simple section that contains it or a simple section that is contained in it.

Any simple section with $n$ dimensions can be described by at most $2n^2$ simple boundaries. It is possible to describe some simple sections with less than $2n^2$ boundaries. This happens when some boundaries do not contribute to the shape of the simple section. Such boundaries are called redundant boundaries. Although redundant boundaries do not contribute to the shape of the section, they are useful for computing the union and intersection of simple sections [2]. Thus, all $2n^2$ boundaries are used to represent a simple section. Boolean flags are used to distinguish between redundant and non-redundant boundaries. A flag has value true if the corresponding boundary is redundant, otherwise it has value false.

Lemma 2.1 The total number of flags needed to keep redundancy information for the boundaries of a simple section with $n$ dimensions is $4n(n-1)$.

Proof: Consider an $n$ dimensional space with $n$ coordinate axes $x_1, x_2, \ldots, x_n$. A rectangular boundary $x_i = c$, where $1 \leq i \leq n$ is redundant or not with respect
to some dimension $x_j$ where $1 \leq j \leq n$. That is, it can be redundant on some hyperplane and non-redundant on some other. Thus, the number of flags we need for each rectangular boundary is $n-1$. There are $2n$ rectangular boundaries, so the total number of flags we need for all rectangular boundaries is $2n(n-1)$. Each diagonal boundary lies on exactly one hyperplane, so we need one flag per diagonal bound. The number of diagonal boundaries is $2n(n-1)$, so the total number of flags we need for diagonal boundaries is $2n(n-1)$. Thus, the total number of flags we need for the simple section is $4n(n-1)$. □

A convenient way to represent simple sections is by specifying its simple boundary pairs.

**Definition 2.3** A simple boundary pair is an inequality of the form $\alpha \leq \psi(x_1, x_2, \ldots, x_n) \leq \beta$, where $\psi(x_1, x_2, \ldots, x_n)$ is a function over the axes such that $\alpha = \psi(x_1, x_2, \ldots, x_n)$ and $\beta = \psi(x_1, x_2, \ldots, x_n)$ are both simple boundaries of the section. $\alpha = \psi(x_1, x_2, \ldots, x_n)$ is called a lower boundary and $\beta = \psi(x_1, x_2, \ldots, x_n)$ is called an upper boundary.

**Note:** For each dimension $x_i$ where $1 \leq i \leq n$, $x_i.lo$ and $x_i.up$ denote the lower and upper boundaries, respectively, of the simple section for dimension $x_i$. For each pair of dimensions $(x_i, x_j)$, where $1 \leq i, j \leq n$, $[x_i \pm x_j].lo$ and $[x_i \pm x_j].up$ denote the lower and upper bound, respectively, of the $x_i \pm x_j$ diagonal.

Figure 2.1 shows the algorithm for constructing a simple section from the loop boundaries. This algorithm appears in Balasundaram's dissertation [2], and is included for completeness.

A new element added to the algorithm is procedure `set_redundant_flags` that sets the redundant flags of each boundary. For each rectangular boundary $x_i = c$, it determines whether the boundary is redundant with respect to dimension $x_j$, for $1 \leq j \leq n$. Also, it determines redundancy for each diagonal boundary $x_i \pm x_j = c$.

The idea behind procedure `set_redundant_flags` is illustrated in figure 2.2. Here $x$ and $y$ are dimension numbers. The figure displays the boundaries on the $xy$ hyperplane. A diagonal boundary is redundant if the point of intersection of the adjacent rectangular boundaries is tighter. A rectangular boundary is redundant if the point of intersection of some of the adjacent rectangular and diagonal boundaries is tighter.
Algorithm BUILD_SIMPLE_SECTION

Input: the dimension $n$ of the array, the nesting depth $m$ of the array reference, the boundaries $L_h, U_h, \ldots, L_m, U_m$ of the loop induction variables $I_h, \ldots, I_m$ and the subscript expressions $x_1, x_2, \ldots, x_n$ of the array reference. The loop boundaries of a loop at level $k$, where $h \leq k \leq m$ must be of the form $a_0 + \sum_{l=h}^{k-1} a_l I_l$, where $a_l$ is a signed constant for $h \leq l \leq k - 1$ and $a_0$ is a loop invariant expression.

Output: the simple section $S$ that approximates the portion of the array referenced in the region enclosed by the $I_h$ loop.

begin

1 The simple section consists of simple boundaries of the form
   \[ \alpha \leq \psi(x_1, x_2, \ldots, x_n) \leq \beta \]

2 for each possible expression $\psi$ do

3   express $\psi$ in the form $\psi = a_0 + a_h I_h + \ldots + a_m I_m$, where $a_k$ are signed
   constants and $a_0$ is a loop invariant expression.

4   define $a_k^+ = \max(a_k, 0)$ and $a_k^- = \max(-a_k, 0)$, $h \leq k \leq m$

5   add to $S$ the simple boundary pair
   \[ \text{LBOUND}(h, \psi) \leq \psi \leq \text{UBOUND}(h, \psi) \]

6 endfor

set_redundant_flags()

end

procedure LBOUND($h, \psi$)
begin

1 if $\psi$ does not contain any terms in $I_h, I_{h+1}, \ldots, I_m$
2   return $\psi$

3 else
4   replace each $I_k$, $h \leq k \leq m$, in $\psi$ by $(a_k^+ L_k - a_k^- U_k)$
5   let $\psi'$ be the resulting expression

6 return LBOUND($h$, $\psi'$)

end

procedure UBOUND($h, \psi$)
begin

1 if $\psi$ does not contain any terms in $I_h, I_{h+1}, \ldots, I_m$
2   return $\psi$

3 else
4   replace each $I_k$, $h \leq k \leq m$, in $\psi$ by $(a_k^+ U_k - a_k^- L_k)$
5   let $\psi'$ be the resulting expression

6 return UBOUND($h$, $\psi'$)

end

Figure 2.1 Algorithm for building a simple section from the loop boundaries.
Conditions for testing for redundant boundaries.

- For a rectangular boundary:
  - $x.\text{lo}$ is redundant with respect to $y$ iff
    \[
    ([x + y].\text{lo} \geq x.\text{lo} + y.\text{up}) \lor
    ([x - y].\text{lo} \geq x.\text{lo} - y.\text{lo}) \lor
    ([x + y].\text{lo} + [x - y].\text{lo} \geq 2 * x.\text{lo})
    \]
  - $x.\text{up}$ is redundant with respect to $y$ iff
    \[
    ([x + y].\text{up} \leq x.\text{up} + x.\text{lo}) \lor
    ([x - y].\text{up} \leq x.\text{up} - y.\text{up}) \lor
    ([x + y].\text{up} + [x - y].\text{up} \leq 2 * x.\text{up})
    \]
  - $y.\text{lo}$ is redundant with respect to $x$ iff
    \[
    ([x + y].\text{lo} \geq x.\text{up} + y.\text{lo}) \lor
    ([x - y].\text{up} \leq x.\text{lo} - y.\text{lo}) \lor
    ([x + y].\text{lo} - [x - y].\text{up} \geq 2 * x.\text{lo})
    \]
  - $y.\text{up}$ is redundant with respect to $x$ iff
    \[
    ([x + y].\text{up} \leq x.\text{lo} + y.\text{up}) \lor
    ([x - y].\text{lo} \geq x.\text{up} - y.\text{up}) \lor
    ([x + y].\text{up} - [x - y].\text{lo} \leq 2 * y.\text{up})
    \]

- For a diagonal boundary:
  - $[x + y].\text{lo}$ is redundant iff $[x + y].\text{lo} \leq x.\text{lo} + y.\text{lo}$
  - $[x + y].\text{up}$ is redundant iff $[x + y].\text{up} \geq x.\text{up} + y.\text{up}$
  - $[x - y].\text{lo}$ is redundant iff $[x - y].\text{lo} \leq x.\text{lo} - y.\text{up}$
  - $[x - y].\text{up}$ is redundant iff $[x - y].\text{up} \geq x.\text{up} - y.\text{lo}$

Figure 2.2 Test for redundant boundaries.
Special consideration must be given to degenerate cases, that is, when the upper and lower boundary of a boundary pair are equal.

2.2 Big and Small Union of Simple Sections

This section defines the union of two simple sections. It presents an algorithm for determining the exactness of the union. Also, it gives approximations for the union of two sections, useful when the union is not exact.

**Definition 2.4** Let \( a_1 \) and \( a_2 \) be two \( n \) dimensional array sections. The union of \( a_1 \) and \( a_2 \) is an \( n \) dimensional simple section \( U \) such that a point \( p \in U \) iff \( p \in a_1 \) or \( p \in a_2 \).

Simple sections are not closed under union, since the result of the union of two simple sections is not necessarily a simple section. This observation leads us to define approximations for the union of two simple sections. The **big** and **small union** of two simple sections are approximations to the precise union by Balasundaram [2].

**Definition 2.5** The union \( U \) of two simple sections is exact if \( U \) is a simple section.

Figure 2.3 shows examples of exact and inexact unions.

**Definition 2.6** Let \( s_1 \) and \( s_2 \) be two \( n \) dimensional simple sections. The big union of \( s_1 \) and \( s_2 \) is the smallest simple section that contains the union of \( s_1 \) and \( s_2 \).

**Definition 2.7** Let \( s_1 \) and \( s_2 \) be two \( n \) dimensional simple sections. The small union of \( s_1 \) and \( s_2 \) is the biggest simple section that is contained in the union of \( s_1 \) and \( s_2 \).

If the union \( U \) of two simple sections is exact, both the big and the small union of the two sections are equal to \( U \). Figure 2.4 shows algorithms for computing the big and small union of two simple sections \( s_1 \) and \( s_2 \).

The basic idea behind the algorithm for computing the big union, Big-\( U \), of \( s_1 \) and \( s_2 \) is that each boundary of Big-\( U \) is the outermost of the corresponding boundaries of \( s_1 \) and \( s_2 \). Thus, each boundary of Big-\( U \) is either a boundary of \( s_1 \) or a boundary of \( s_2 \). Balasundaram [2] showed that this algorithm correctly computes the big union of
$s_1$ and $s_2$. After computing the boundaries of $Big\_U$ the algorithm sets the redundant flags for each of the boundaries of $Big\_U$ using the inequalities shown in figure 2.2.

The algorithm for computing the small union, $Small\_U$, of $s_1$ and $s_2$ returns the big union of the two sections if the union is exact. Otherwise it returns the larger of $s_1$ and $s_2$. The larger of the two sections is not always equal to the small union of the two sections. However, this heuristic should perform well for the cases commonly encountered in practice. The key of this algorithm is procedure $union\_exact$ that returns $true$ if the union of the two sections is exact and $false$ otherwise.

Figure 2.3 displays the key observation for testing exact union: If a boundary is chosen for the union, its adjacent boundaries must also be chosen. The example displays two 2-dimensional simple sections $s$ and $s'$. In figure 2.3(a) boundary $b$ is chosen for the union since it is “more exterior” than $b'$. This imposes the constraint that boundaries $a$ and $c$ must be chosen to maintain an exact union. In figure 2.3(b) boundary $b$ is chosen over boundary $b'$ imposing the constraint that boundary $a$ must be chosen over boundary $a'$, and boundary $c$ must be chosen over $c'$. However, this constraint is violated because the algorithm for union chooses boundary $c'$, which is “more exterior” than $c$. So this union is inexact.
Algorithm BIG\_UNION

Input: Two \( n \) dimensional simple sections \( s_1 \) and \( s_2 \).
Output: The big union \( \text{Big}\_U \) of the two sections.

begin
1 for each simple boundary pair \( \alpha \leq \psi(x_1, x_2, \ldots, x_n) \leq \beta \) of \( s_1 \) do
2 pick the corresponding boundary pair \( \alpha' \leq \psi(x_1, x_2, \ldots, x_n) \leq \beta' \) of \( s_2 \)
3 add to \( \text{Big}\_U \) the boundary pair \( \min(\alpha, \alpha') \leq \psi(x_1, x_2, \ldots, x_n) \leq \max(\beta, \beta') \)
4 endfor
5 set\_redundant\_flags(\( \text{Big}\_U \))
6 return \( \text{Big}\_U \)
end

Algorithm SMALL\_UNION

Input: Two \( n \) dimensional simple sections \( s_1 \) and \( s_2 \).
Output: An approximation of the small union \( \text{Small}\_U \) of the two sections.

begin
1 if \( \text{exact}\_union(s_1, s_2) == \text{true} \)
2 return \( \text{Big}\_U(s_1, s_2) \)
3 else
4 return the larger of \( s_1 \) and \( s_2 \)
5 endif
end

Figure 2.4 Algorithms for Big and Small Union.

If two corresponding boundaries \( b \) and \( b' \) of the \( i \)th dimension are equal, we must verify that the union of the one dimensional simple sections formed by the boundaries of the \( i \)th dimension is exact. This ensures that there is no "gap" across the \( i \)th dimension of the union of the two sections \( s \) and \( s' \). If the simple sections have \( n \geq 2 \) dimensions, we must verify that the union for each pair of two dimensional sections formed by the boundaries of the \( i \)th and \( j \)th dimensions, where \( 1 \leq j \leq n \) and \( j \neq i \), is exact. This ensures that there is no "gap" on the \( ij \) hyperplane of the union of \( s \) and \( s' \).
Algorithm UNION EXACT

Input: Two simple sections $s_1$ and $s_2$ and their number of dimensions $n$.
Output: true if the union of $s_1$ and $s_2$ is exact; false otherwise.

begin
1 if not exact($s_1$) or not exact($s_2$) return FALSE
2 else if (n = 1) return overlap.or.abut($s_1$, $s_2$)
3 unmark($s_1$); unmark($s_2$)
4 U = BIG UNION($s_1$, $s_2$)
5 for each boundary position Loc such that redundant(Loc, U) is false
6 let B = boundary(Loc, U)
7 let $b_1$ = boundary(Loc, $s_1$)
8 let $b_2$ = boundary(Loc, $s_2$)
9 if $b_1$ = $b_2$
10 if $b_2$ = B
11 if marked(Loc, $s_1$) return FALSE
12 else mark_adjacent(Loc, $s_2$)
13 else if ($b_1$ = B)
14 if marked(Loc, $s_2$) return FALSE
15 else return mark_adjacent(Loc, $s_1$)
16 else ($b_1$ = $b_2$)
17 if B is a rectangular bound for dimension $x_i$
18 let $t_1$ = $s_1$ after setting $x_i$ = B tightening bounds and eliminating dimension $x_i$
19 let $t_2$ = $s_2$ after setting $x_i$ = B tightening bounds and eliminating dimension $x_i$
20 else B is a diagonal boundary of the form $x_i$ op $x_m$ = c where op is + or -.
21 let $t_1$ = $s_1$ after setting $x_i$ = B op ($-x_m$)
22 tightening bounds and eliminating dimension $x_i$
23 let $t_2$ = $s_2$ after setting $x_i$ = B op ($-x_m$)
24 tightening bounds and eliminating dimension $x_i$
25 for each dimension $x_j$, $j \neq i$
26 if $t_1$ and $t_2$ are one dimensional
27 if not UNION EXACT($t_1$, $t_2$, 1) return FALSE
28 else
29 for each dimension $x_k$, $k \neq i$
30 let $t'$ = $t_1$ boundaries on $x_j$, $x_k$
31 let $t''$ = $t_2$ boundaries on $x_j$, $x_k$
32 if not UNION EXACT($t'$, $t''$, 2) return FALSE
33 return TRUE
end

Figure 2.5 Test for exact union.
Algorithm INTERSECTION\_EXACT

*Input:* Two $n$ dimensional simple sections $s_1$ and $s_2$.
*Output:* true if the intersection of the $s_1$ and $s_2$ is exact; false otherwise.

begin
1 for each simple boundary $c = \psi(x_1, x_2, \ldots, x_n)$ of $s_1$ do
2 pick the corresponding boundary $c' = \psi(x_1, x_2, \ldots, x_n)$ of $s_2$
3 if $c$ or $c'$ is symbolic
4 if $c$ and $c'$ are not comparable
5 return false
6 endfor
7 return true
end

**Figure 2.6** Test for exact intersection.

Figure 2.5 shows the algorithm for testing for exact union. A similar algorithm was developed by Huelsbergen, Hahn, and Larus [7]. However, algorithm UNION\_EXACT appearing in figure 2.5 was developed independently. The algorithm takes as inputs two simple sections $s_1$ and $s_2$ and their dimension number $n$. It returns true if the union of $s_1$ and $s_2$ is exact and false otherwise. For each boundary of the two sections the algorithm maintains a marked flag that indicates whether the boundary must be chosen for the union or not.

If either section is inexact, exactness of the union cannot be determined, and the algorithm returns false. If the two sections are one dimensional, the algorithm ensures that they overlap or abut, that is, that they have a common point or the distance between them is equal to the the step of $s_2$.

Next the algorithm unmarks the boundaries of both sections. This indicates that no constraints have yet arisen requiring subsequent selection of some boundary of $s_1$ or $s_2$. The algorithm compares corresponding boundary pairs $b$ and $b'$ on each dimension $i$. Assume that $b$ is more exterior than $b'$. Then $b$ is chosen for the union. If $b'$ is marked, earlier in the algorithm a constraint arose that required $b'$ to be chosen for the union. So, the union is inexact and the algorithm returns false. Otherwise,
the adjacent boundaries of $b$ are marked and the algorithm continues to the next boundary pair.

If $b = b'$, the algorithm checks whether the unions of all pairs of two dimensional sections induced by the $i$th dimension are exact (if $n = 2$ the union of the one dimensional sections formed by the boundaries of the $i$th dimension is checked for exactness). If this does not hold, the algorithm returns $false$. Otherwise it continues to the next boundary pair.

When no constraints are violated in the above process the algorithm returns $true$.

2.3 Big and Small Intersection of Simple Sections

This section defines the intersection of two sections. Also, it gives approximations for the intersection, useful when the intersection is not exact.

**Definition 2.8** Let $a_1$ and $a_2$ be two $n$ dimensional array sections. The intersection of $a_1$ and $a_2$ is an $n$ dimensional simple section $I$ such that a point $p \in I$ iff $p \in a_1$ and $p \in a_2$.

**Definition 2.9** The big intersection of two simple sections $s_1$ and $s_2$ is the smallest simple section that contains the intersection of $s_1$ and $s_2$. 
Algorithm BIG\_INTERSECTION

Input: Two $n$ dimensional simple sections $s_1$ and $s_2$.
Output: The big intersection $\text{Big\_I}$ of $s_1$ and $s_2$.

begin
1 for each simple boundary pair $\alpha \leq \psi(x_1, x_2, \ldots, x_n) \leq \beta$ of $s_1$ do
2   pick the corresponding boundary pair $\alpha' \leq \psi(x_1, x_2, \ldots, x_n) \leq \beta'$ of $s_2$
3   add to $\text{Big\_I}$ the boundary pair $\max(\alpha, \alpha') \leq \psi(x_1, x_2, \ldots, x_n) \leq \min(\beta, \beta')$
4 endfor
5 set\_redundant\_flags($\text{Big\_I}$)
6 tighten\_redundant\_boundaries($\text{Big\_I}$)
7 return $\text{Big\_I}$
end

Algorithm SMALL\_INTERSECTION

Input: Two $n$ dimensional simple sections $s_1$ and $s_2$.
Output: The small intersection $\text{Big\_I}$ of $s_1$ and $s_2$.

begin
1 if intersection\_exact($s_1, s_2$) == TRUE
2   return big\_intersection($s_1, s_2$)
3 else return ⊥
end

Figure 2.8 Algorithms for Big and Small Intersection.

Definition 2.10 The small intersection of two simple sections $s_1$ and $s_2$
is the largest simple section that is contained in the intersection of $s_1$ and$s_2$.

Simple sections are closed under intersection. This holds because simple sections are
convex polytopes, which are closed under intersection. Also, since each boundary of theintersection is a boundary of one of the two sections, it is a simple boundary. Therefore, theintersection is always a convex polytope with simple boundaries, that is, a simple section.
Algorithm TIGHTEN REDUNDANT BOUNDARIES

Input: A n dimensional simple section s.  
Output: The simple section s with all redundant boundaries tightened.

begin  
    foreach rectangular boundary b do  
        for each dimension i  
            if b is redundant with respect to dimension i  
                insert (b, i) in the working list  
        endfor  
    while the working list is not empty  
        let (b, i) be the first element of the working list do  
            tighten b with respect to dimension i  
            for each dimension j, j \neq i do  
                if j.lo is redundant with respect to i  
                    add (j.lo, i) to the working list  
                if j.up is redundant with respect to i  
                    add (j.up, i) to the working list  
            endfor  
        endwhile  
    for each diagonal boundary b do  
        tighten b  
    endfor  
end

Figure 2.9 Algorithm for tightening the redundant boundaries of a simple section.

Since simple sections are closed under intersection, the intersection of two sections is exact if the corresponding boundaries are comparable. This observation is the base of algorithm intersection_exact, that is shown in figure 2.6. If the boundaries of the two simple sections are constant, then the intersection is exact. If they are symbolic, then the intersection is exact if the corresponding boundaries are comparable. Otherwise, the intersection is inexact.

The algorithm for computing the big intersection Big_I of two simple sections s_1 and s_2 are shown in figure 2.8. Each boundary of Big_I is the innermost of the corresponding boundaries of s_1 and s_2. The boundaries computed with this method are not always tight. An example illustrating this is shown in figure 2.7, where the
boundaries of the $x$-$y$ diagonal of the intersection must be tightened. An algorithm for tightening the redundant boundaries of a simple section is shown in figure 2.9.

The key idea of the algorithm is to first set the redundant flags of the section using the conditions in figure 2.2; then to set each redundant boundary to the “innermost” point of intersection of its adjacent diagonal and rectangular boundaries. This process makes some non-redundant boundaries become redundant. To overcome this problem the algorithm maintains a working list of pairs $(b, i)$, where $b$ is a rectangular boundary and $i$ is the dimension with respect to which $b$ must be tightened. First we insert in the working list all pairs $(b, i)$, such that $b$ is a rectangular boundary redundant with respect to dimension $i$. Let $(b, i)$ be the first element of the working list. The algorithm removes $(b, i)$ from the list and tightens $b$ with respect to dimension $i$. If some other rectangular boundary $b'$ becomes redundant with respect to $b$’s dimension, for example $k$, the pair $(b', k)$ is inserted in the list. When the working list is empty the algorithm tightens the diagonal boundaries of the section.

The algorithm for computing an approximation to the small intersection, $Small.I$, of two sections $s_1$ and $s_2$ is shown in figure 2.8. If the intersection of $s_1$ and $s_2$ is exact, the big and small intersection are equal, so the algorithm returns the big intersection. Otherwise there exists at least one boundary $b$ of $s_1$ that is not comparable to the corresponding boundary $b'$ of $s_2$. Since it is not known which of $b$ and $b'$ is a boundary of the intersection, one must be conservative and set the small intersection to $\perp$. 
Chapter 3

Overview of Array Section Analysis

Classical interprocedural analysis views arrays as units. Callahan showed that this is not sufficient for exposing parallelism in loops containing procedure calls, some of whose arguments are arrays [5]. Array section analysis helps expose parallelism in the presence of procedure calls and enables interprocedural transformations. This chapter describes a strategy for interprocedural array section analysis.

3.1 Interprocedural array section analysis

The goal of array section analysis is to determine the impact of a procedure call to the array variables that are visible at the call context. Following are the definitions of the array section analysis problems, a solution to which will be presented in this thesis.

**Definition 3.1** MOD(s) is a set of pairs \((a, R)\), where \(a\) is an array variable and \(R\) is an array section containing the elements of \(a\) that may be modified as a side effect of the invocation at call site \(s\).

**Definition 3.2** REF(s) is a set of pairs \((a, R)\), where \(a\) is an array variable and \(R\) is an array section containing the elements of \(a\) that may be referenced as a side effect of the invocation at call site \(s\).

**Definition 3.3** KILL(s) is a set of pairs \((a, R)\), where \(a\) is an array variable and \(R\) is an array section containing the elements of \(a\) that must be modified as a side effect of the invocation at call site \(s\).

MOD and REF are flow insensitive formulations [19], that is, knowledge about control flow is not required in their solution. A section \(R\) is in MOD(s), for a call site \(s\), if \(R\) may be modified through any statement in the body of the procedure called at \(s\). An analogous argument holds for REF(s). MOD and REF are may data flow
problems, since a solution to these problems determines what may be modified or referenced, respectively, as a side effect of a call.

KILL is a flow sensitive formulation [19]. Since a section $R$ is in $\text{KILL}(s)$, for a call site $s$, only if it must be modified on every control flow path in the body of the procedure called at $s$, control flow must be taken into account in the solution. KILL is a must data flow problem, since a solution to KILL determines what must be modified as a side effect of a call.

The solution to the MOD, REF, and KILL simple section side effect analysis is not always precise. This happens because operations on simple sections are not always exact, especially in the presence of symbolics. An approximation to the MOD and REF analysis must contain the precise solution to the problems, while an approximation to KILL must be contained in the precise solution.

There is a fundamental difference between the solutions to the MOD and REF problems and the solution to the KILL problem, caused by the fact that the first two are flow insensitive, while the latter is flow sensitive [19]. To illustrate the difference, figure 3.1 displays two example control flow graphs. The MOD solution for the graph
Example spec77 (from Perfect)

gwater
1. Parallel do k = 1, 7
2. call sumpls(\ldots, a[1,k], \ldots)
3. call sumpls(\ldots, b[1,k], \ldots)
4. enddo

sumpls(c)
dimension c(31)

5. Parallel do l = 1, 31
\ldots
6. c[l] = \ldots
\ldots
7. enddo

optimized gwater
8. Parallel do l = 1, 31
9. Parallel do k = 1, 7
10. call osumpls(\ldots, a[1,k], \ldots)
11. call osumpls(\ldots, b[1,k], \ldots)
12. enddo
13. enddo

Figure 3.2 Array sections enable interprocedural transformations.

on figure 3.1(a) is given by the union of sections $S_1$ and $S_2$. That is, a section is in MOD if it may be modified on any of the two paths. The same holds for REF. The KILL solution for this graph is given by the intersection of $S_1$ and $S_2$, since a section is in KILL only if it must be modified on both paths. The graph on figure 3.1(b) contains only one path, so the solution to both MOD and KILL is the union of $S_1$ and $S_2$.

Another difference between the solutions to MOD and REF and the solution to KILL is the use of different approximations for union and intersection. Since MOD represents the set of array sections that may be modified as a side effect of an
invocation, an approximate solution must contain the exact solution to the problem. The same holds for REF. This implies that big union must be used for solving MOD and REF. On the other side, KILL is a must problem. An approximate solution to KILL must be contained in the exact solution to the problem, and so small union and intersection must be used.

The following sections give examples that show the usefulness of array section analysis in the parallelization of loops and in performing interprocedural transformations.

3.2 Use of array section analysis in interprocedural transformations

Array section analysis can be used to disprove dependencies in the presence of procedure calls, enabling certain interprocedural transformations. This is illustrated in the example in figure 3.2, extracted from the spec77 benchmark, which is part of the Perfect suite.

Loop k (on line 1), in procedure gwater, contains two calls to procedure sumpls; procedure sumpls contains the parallel loop l (on line 2). Array section analysis shows that there are no dependencies between the calls on lines 2 and 3, since each call receives different arrays as arguments. Also, there are no dependencies from each call to itself, since a different column of the array is passed as an argument in each iteration. Since loop k carries no other dependencies, it can be run in parallel. To further exploit this parallelism, in the optimized version of gwater, loop l is extracted [16] from both the calls in lines 2 and 3. Since these calls are independent, the loops created in the body of gwater as a result of the extractions can be safely fused. Also, since none of the k and l loops carry dependencies, they can be safely interchanged. For this example, loop interchange would be profitable in the presence of more than seven processors.

3.3 Use of array section KILLS in the parallelization of loops

Array section KILLS are important for detecting parallelism. If all elements of an array are redefined before any use in each iteration of a loop, the array can be made private to each iteration. Privatization removes all loop carried dependencies involv-
ing elements of that array. If no other loop carried dependencies exist, the loop can be run in parallel. This is illustrated in the example of figure 3.3, which is part of the qcd benchmark of the Perfect suite.

Qcd is a stencil computation. Procedure update consists of a single loop that computes the value of each element of an array from the values of four of its neighbors. The addresses of the four neighbors, whose value will be used in the computation, are computed by procedure getpos and stored in the index array coord. The dir loop (on line 3) uses procedure hit to compute the value of the ith element. Classical interprocedural analysis must be conservative and assume that the i loop, on line 1, carries dependencies involving the elements of coord. Array section analysis shows that an invocation of procedure getpos kills the index array coord, that is, it redefines all of its elements. Since coord is killed before being used in each iteration of the i loop, it can be privatized. After the privatization of coord the i loop carries no other dependencies, and so it can be run in parallel.

3.4 Interprocedural strategy

The following chapters present a solution to interprocedural array section analysis. The solution is broken down into a local and an interprocedural phase.

In the local phase we compute summaries for MOD, REF, and KILL for each procedure. The local summaries contain information about the sections that are directly modified, referenced, or killed, through assignment statements, in the procedure body. In this phase we also build a framework that enables translation of the indirect sections at the call sites in the interprocedural phase.

In the interprocedural phase we walk the call graph in reverse topological order. For each node: 1) we translate the local summaries of the callee to the framework of the caller, producing indirect sections, and 2) we accumulate the direct and indirect sections of the caller, producing cumulative sections.
Example qcd (from Perfect)

update()
1. do i = 1, nred
2. call getpos(..., coord, ...)
3. do dir = 0, 3
4. call hit(..., coord, ...)
5. enddo
6. enddo

getpos(..., coord, ...)
integer coord(4)

7. do i = 1, 4
8. coord(i) = ...
9. enddo

after privatization
10. Parallel do i = 1, nred
11. private coord
12. call getpos(..., coord, ...)
13. do dir = 0, 3
14. call hit(..., coord, ...)
15. enddo
16. enddo

Figure 3.3 Array section KILLS help in parallelization of loops.
Chapter 4

Local Analysis

Local analysis computes summaries of the sections that are directly modified, referenced, or killed in the body of each procedure. Information about interprocedural side effects is not available at this point. To increase the precision of the interprocedural solutions, this phase only records information about local side effects; the accumulation of the local information, that is, the unions and intersections of the sections in the local summaries, is performed in the interprocedural phase, when solutions of other interprocedural analyses is available.

For the MOD and REF problems the local phase computes, for each procedure, the shapes of the sections directly accessed (modified or referenced) in the procedure body. The local KILL information for each procedure consists of the shapes of the sections directly killed in the procedure body as well as control flow information. The local phase also gathers information that is necessary for computing indirect side effects in the interprocedural phase.

4.1 Goal of local analysis

To produce precise interprocedural solutions for array section analysis, results from scalar interprocedural analysis are required. This is illustrated by the example on figure 4.1. The section of array $A$ that is modified in the body of subroutine $S$ is $A(1:M:1)$. Since $M$ is an argument of the call to $clobber$ on line 3, to compute precise bounds for the section of $A$ that is modified, it is necessary to know the side effects of the call to $M$. The results of other scalar analyses, like constant propagation, can further increase the precision of interprocedural array section analysis.

Several phases may be required to produce precise array side effect solutions. To minimize the number of passes through each procedure body, the solution to interprocedural array section analysis presented here is split into a local and an interprocedural phase. By precomputing the local information, it is possible to construct an
1. SUBROUTINE S(A, N, M)
2.   DIMENSION A(N)
3.   CALL CLOBBER(M)
4.   DO i = 1, M, 1
5.       A(i) = 0.0
6.   ENDDO
7.   RETURN
8.   END

**Figure 4.1** Scalar interprocedural analysis is necessary to compute precise array section analysis solutions.

interprocedural phase that iterates on the call graph computing interprocedural side effects without direct examination of any procedure body.

The goal of the local phase is to record information about the sections that are directly modified, referenced, or killed in the body of each procedure. This information is not accumulated until the interprocedural phase, when the interprocedural scalar analysis solutions are available. An outline of the local analysis is shown on figure 4.2. In addition to the local summaries, local analysis computes the shapes of the array formal parameters of each procedure, and the shapes of the array actual parameters at each call site. These will be used in the interprocedural phase to compute indirect side effects on array sections.

In the implementation of this analysis, array sections are represented by simple sections. The bounds of the sections are value numbers. In this phase the value numbers are local: they are valid in the body of each procedure, but some of them are not valid interprocedurally. Before the interprocedural analysis each of the local value numbers is translated to an interprocedural value. The translation uses information from scalar interprocedural analysis, resulting in more exact bounds for the array sections. Scalar analyses that enhance the precision of the value numbers are interprocedural MOD analysis, constant propagation, etc.

Dividing array section analysis to a local and an interprocedural phase allows the computation of precise side effects on arrays using only two passes through the body
Local Array Section Analysis
for each procedure do
    compute MOD, REF, and KILL summaries
    for each array variable (formal, global, static) do
        save section describing shape
    for each call do
        for each array argument do
            save section describing shape

Figure 4.2 Outline of local analysis.

of each procedure: one during local analysis and one during dependence analysis and/or interprocedural transformations.

4.2 Direct MOD and REF summaries

The local phase records the array sections that are directly modified or referenced in the body of each procedure. Since MOD is a flow insensitive problem, to compute local MOD information it suffices to find all the definitions of array elements in the procedure body and merge the corresponding simple sections. Similarly, to compute local REF information, it suffices to merge the simple sections that correspond to array element references.

However, merging the MOD and REF summaries in the local phase may result in imprecise solutions. Since no interprocedural scalar information is available, the value numbers representing the bounds of the simple sections will be conservative. Higher precision can be achieved by postponing the merges of the corresponding simple sections until the interprocedural phase, when scalar interprocedural information is available. Therefore, instead of merging the corresponding sections, the local phase inserts them in a list. Two lists are used, one for the modified and one for the referenced simple sections.

Figure 4.3 displays algorithm MODREF_SUMMARIES, that computes the local MOD and REF summaries for a single procedure. The input to the algorithm is the
Algorithm MODREF_SUMMARIES

Input: the SSA graph, G, of a procedure body.
Output: lists of sections that are directly modified or referenced in the procedure body. The lists are called mod_list and ref_list, respectively, and are initially empty.

begin
1. for each node n in G do
2.     if n corresponds to an array access
3.         compute the corresponding array section S
4.         if n is a mod
5.             insert(mod_list, S)
6.         else if n is a ref
7.             insert(ref_list, S)
8.     endfor
end

Figure 4.3 Algorithm for computing direct MOD and REF information.

set of definitions and uses in the procedure body, represented by the static single assignment (SSA) graph. The result of procedure MODREF_SUMMARIES is a pair of lists that hold the simple sections that are directly modified or referenced in the procedure body. The procedure iterates through the nodes of the SSA graph. If a node corresponds to an array access, the algorithm computes the corresponding array section. If the node is a definition, it inserts it in the list of modified sections. If it is a reference, it inserts it in the list of referenced sections.

This algorithm visits each node of the SSA graph exactly once. Assuming that the algorithm inserts the simple sections in the beginning of the summary lists, the cost of an insertion if O(1). If the largest number of dimensions for any array in the procedure body is d, and the number of nodes in the SSA graph is N, the complexity of this algorithm is O(N ⋅ d^2).

4.3 Local KILL information

The fact that KILL is a flow sensitive problem makes the computation of local KILL information harder than that of the local MOD and REF summaries.
Since control flow is irrelevant for the computation of the local MOD and REF information, the local MOD and REF summaries can be stored as lists. Those lists can be accumulated either in the local or in the interprocedural phase. The difference is that accumulating the lists in the interprocedural phase produces more precise solutions.

Since control flow is important for the computation of KILL, the local phase must not only compute the directly killed sections, but must also gather and store control flow information. The algorithm presented in this section uses the control dependence graph as a representation of control flow in the body of a procedure. It annotates each node of the graph with information about the sections killed at that node. The annotated control dependence graph provides all the necessary information to compute the KILL set of the procedure in the interprocedural phase: the graph itself provides control flow information, and the annotations provide information about the sections killed at each node. The algorithm for computing the local KILL information for one procedure is shown on figure 4.4. In addition to the control dependence graph of a procedure, the algorithm takes as input the lower bound, the upper bound, and the induction variable of each loop in the procedure body. This information is needed to compute the killed simple sections that correspond to assignments surrounded by loop nests. If a node \( n \) of the control dependence graph corresponds to a direct kill, that is, an assignment to an array element, the algorithm computes the simple section \( R \) killed by the assignment and stores it as an annotation to \( n \). If the assignment is surrounded by a loop nest, the killed simple section is computed using algorithm BUILD_SIMPLE_SECTION, as described in chapter 2. If it is not in a loop nest, the killed simple section consists of the single element that corresponds to the assigned array element. If a node corresponds to an indirect kill, that is, an invocation, it is annotated with a place holder that identifies the array sections possibly killed by the invocation. The place holder is an interprocedural handle to each of the array arguments of the invocation. It provides the necessary information to identify the array variables in the interprocedural phase.

The information stored in the annotated control dependence graph cannot be fully accumulated in the local phase, since indirect KILL information is not yet available. After replacing the place holders with the killed sections they identify in the interprocedural phase, the information will be accumulated using the control flow information provided by the graph. The algorithm for accumulating the information on the graph is given in the next chapter.
Algorithm KILL_SUMMARIES

Input: The control dependence graph, CDG, of the procedure body. The lower and upper boundaries and the induction variables for each loop in the procedure body.

Output: The CDG, where each node is annotated with information about the sections that are killed at that node.

begin
1. for each node n in the CDG do
2.  if n is an assignment to an array element $A[s_1, s_2, \ldots, s_k]$
3.    let $l$ be the loop level of $A[s_1, s_2, \ldots, s_k]$
4.    if($l = 0$)
5.      $R = A[s_1, s_2, \ldots, s_k]$
6.    else
7.      let $L = \{L_1, U_1, \ldots, L_l, U_l\}$ be the set of lower and upper bounds of the loop nest surrounding $A[s_1, s_2, \ldots, s_k]$
8.      let $I = \{I_1, \ldots, I_l\}$ be the set of induction variables of the loop nest surrounding $A[s_1, s_2, \ldots, s_k]$
9.      $R = \text{BUILD_SIMPLE_SECTION}(A[s_1, s_2, \ldots, s_k], L, I)$
10.     annotate n with $R$
11.    else if n is an invocation
12.      annotate n with a place holder identifying the array actual parameters of the invocation
13. endfor
end

Figure 4.4 Algorithm for computing direct KILL information.

As an optimization, nodes containing direct KILL information can be accumulated and collapsed to a single node. Accumulation stops when it is inhibited by nodes carrying information about indirect KILLS. This reduces the size of the graph. Thus, the amount of information that must be passed to the interprocedural phase is reduced.

This algorithm visits each node of the control dependence graph exactly once. Assuming that the largest number of arguments passed to any procedure call is $k$, the cost for constructing a place holder for a single invocation is $O(k)$. If the largest
number of dimensions for any array in the procedure body is \( d \), and the number of nodes in the graph is \( N \), the complexity of this algorithm is \( O(N \cdot \max(d^2, k)) \).

4.4 Preparation for the interprocedural phase

The bounds of the sections computed in this phase may contain local information, that is, may be expressed in terms of local variables. A bound that contains local information is not valid interprocedurally. In the interprocedural phase the value of a local bound of a section in MOD or in REF is \( \perp \). Since the value of a local bound of a dimension \( i \) is unknown in the interprocedural phase, analysis must be conservative and assume that any element across dimension \( i \) can be accessed in the body of the procedure. Thus, before passing the local MOD and REF summaries to the interprocedural phase, bounds containing local information are set to the corresponding declared bound. If a bound of dimension \( i \) of a section in KILL contains local information, interprocedural analysis must be conservative and assume that no element of the section is killed across dimension \( i \); that is, it must assume that the section is empty. Thus, all sections in KILL that have at least one boundary containing local information must be set to \( \top \) before the KILL summaries are passed to the interprocedural phase.
Chapter 5

Interprocedural Analysis

Interprocedural analysis produces interprocedural array side effects by propagating on the call graph the information gathered in the local phase. At each call site, the local summaries of the callee are translated to the context of the caller, producing indirect array section side effects. The indirect side effects are accumulated with the local summaries, producing cumulative side effects for each procedure. To compute the side effects caused by a particular invocation, in a final pass over the call graph the cumulative side effects are translated to the local context of the calling procedure.

5.1 Algorithm for interprocedural array side effect analysis

The goal of the interprocedural phase is to produce, for each procedure, cumulative array section summaries, that is, accumulated direct and indirect array section side effects. The interprocedural phase uses the information gathered during local analysis, as well as interprocedural scalar analysis results.

The first step of the interprocedural phase is to map the value numbers representing the bounds of the simple sections in the local summaries to interprocedural value numbers. The value numbers used in the local summaries were computed in the context of a procedure body, and contain information local to the procedure. Since local variables are not visible in the interprocedural phase, value numbers involving local variables must be converted to a value that is interprocedurally valid. An interprocedural value number contains information that is visible interprocedurally, that is, formal parameters of a procedure or global variables. Scalar interprocedural analysis is folded into this process. The necessity of scalar interprocedural analysis in the translation of local to interprocedural value numbers is illustrated in the example in figure 4.1. The section of array $A$ modified in the body of procedure $S$ is $A(1:M:1)$. Since $M$ is an argument of the call to clobber, interprocedural scalar MOD analysis is necessary to determine the value of $M$ after the execution of the invocation. If no scalar MOD analysis is available the value of $M$ in the bounds of $A(1:M:1)$ must be
INTERPROCEDURAL ANALYSIS

Input: A program, $P$, and the corresponding call graph, $CG$.
Output: $CG$, where each node is annotated with cumulative MOD, REF, and KILL summaries.

begin
1. for each module in $P$ do
2. translate the local to interprocedural value numbers
3. read in the local information
4. endfor
5. for each node $n$ in CG in r.t.o. do
6. accumulate the MOD and REF lists of $n$
7. for each outgoing edge $e$ of $n$ do
8. translate the MOD, REF, and KILL summaries of the sink of $e$ to
   the context of $n$
9. accumulate the translated MOD and REF sections with the MOD and
   REF summaries of $n$
10. annotate $e$ with the translated KILL sections
11. endfor
12. accumulate the KILL structure of $n$
13. annotate $n$ with the produced MOD, REF, and KILL summaries
14. endfor
end

Figure 5.1 Algorithm for interprocedural array section analysis.

assumed to be $\bot$, resulting in loss of precision of the simple section side effects of

An algorithm for performing interprocedural array section analysis is shown in

an input a program and the corresponding call graph. It returns the call graph, where each node
is annotated with cumulative MOD, REF, and KILL summaries. The algorithm first
maps the local value numbers to interprocedural value numbers. Then it reads the in-
formation gathered in the local phase for each procedure; that is, the local summaries,

the shapes of the array variables that are visible interprocedurally (formal and global
variables array), and the shapes of the array variables passed as arguments to each invocation.

After reading the local information, the algorithm makes a single pass over the call graph in reverse topological order to produce cumulative side effects for each procedure. Reverse topological order is chosen because the computation of cumulative sections for a node in the graph requires the cumulative sections of its descendants. Recursion handling is discussed later in this section.

Before the interprocedural propagation of local summaries on the call graph begins scalar interprocedural information is folded into the bounds of simple sections in the local summaries of each procedure. The accumulation of the local MOD and REF summaries, which had been postponed until this information were available, can now be performed. This is the first action performed by the algorithm upon visiting a node $n$ of the call graph. Accumulation simply consists of a big union of the simple sections corresponding to the same array. An algorithm that accumulates the local MOD summaries is shown in figure 5.2. Direct REF summaries are accumulated similarly. Local KILL information is not accumulated until the side effects of calls in the body of the procedure corresponding to $n$ are known.

Next, the algorithm iterates over the outgoing edges of $n$. These edges represent calls in the body of the procedure corresponding to $n$. For each edge, the cumulative sections of the callee are translated to the context of the caller, producing indirect sections for that edge. A detailed description of the translation process is presented in section 5.2. Since MOD and REF are flow insensitive problems, the indirect sections can be merged with the accumulated local summaries, producing cumulative MOD and REF summaries for $n$. The cumulative summaries are stored on $n$ as an annotation. The translated KILL sections are stored on the call graph edge as an annotation. They will be used for accumulating the local KILL information of other nodes in the call graph. Finally the local KILL information of $n$ is accumulated. Accumulation is now possible because information about sections killed indirectly through invocations in the body of $n$ is available. Accumulation consists of a series of small unions and intersections of the sections stored in the local KILL summary of $n$. This process is described in more detail in section 5.3.

The algorithm described above does not handle recursion. Sections are translated from the callees to the callers in reverse invocation order, so no incomplete sections need to be translated at a call site. A simple iterative approach for propagating simple sections on the call graph in the presence of recursion may not terminate, since the
Algorithm ACCUMULATE_DIRECT_MODS

Input: List mod_list, that holds simple sections representing subarrays directly modified in the body of a procedure.
Output: A summary, S, of the accumulated directly modified simple sections. The summary has entries of the form \((n, A)\), where \(n\) is an interprocedural handle of an array variable and \(A\) is a simple section.

\[
\begin{align*}
\text{begin} \\
1. \quad & \text{for each simple section } A \text{ of an array } n \text{ in } \text{mod.list do} \\
2. \quad & \quad \text{if there is an entry } (n, A') \text{ in } S \\
3. \quad & \quad \quad \text{insert in } S \text{ the pair } (n, \text{big.union}(A, A')) \\
4. \quad & \quad \text{else} \\
5. \quad & \quad \quad \text{insert in } S \text{ the pair } (n, A) \\
6. \quad & \text{endfor} \\
\text{end}
\end{align*}
\]

Figure 5.2 Algorithm for accumulating direct MOD summaries.

The section lattice has unbounded depth. Efficient methods for propagating subarrays regardless of the depth of the lattice have been described by Li and Yew [12], [14], and Cooper and Kennedy [13]. A simple approach for dealing with recursion for the KILL problem would be to set the killed section of every array parameter to \(\bot\). Since an approximation to KILL must be contained in the exact solution, this is a safe answer. For MOD and REF, a simple iterative method can be used, where the number of times a section can be lowered in the lattice is bounded. A small counter is associated with each subscript of a summary simple section. The counter is increased every time the subscript range becomes larger (that is, is lowered in the lattice of the subscript). If the simple section has \(n\) dimensions and there is a limit \(k\) on the counter, the total number of times the simple section can be lowered in the lattice is \(kn + 1\). This method would cause variant subscripts to go to \(\bot\) quickly, while leaving the rest unaffected.
5.2 Translation into a call context

If the shape of the formal parameters of the callees is the same as the shape of the corresponding actual parameters of the caller at some call site, translation of the side effects of the callees to the context of the caller is easy. All that has to be done is replace the scalar formal parameters of the callees by the corresponding actual parameters of the caller in each bound of a summary section. However, FORTRAN allows array arguments of an invocation to be reshaped in the body of the called procedure. In this case, translation must take into account the shape of the actual parameter.

Figure 5.3 displays an algorithm for translating a summary section of a formal parameter \( F \) into a call context. This translation is valid for the MOD and REF problems only. The translation process for KILL is discussed below. The algorithm takes as inputs the shape of the formal parameter, \( \text{bounds}_F \), the shape of the referenced simple section, \( \text{ref}_F \), the shape of the corresponding actual parameter, \( \text{bounds}_A \), and the shape of the passed array section, \( \text{pass}_A \). It returns the translated simple section \( \text{ref}_A \). The algorithm first translates the rectangular boundaries of \( \text{ref}_F \). Translation is precise until a dimension is encountered where the formal and actual parameter are inconsistent, that is, they have different sizes or different starting addresses. The first inconsistent dimension can also be translated precisely if it is the last dimension of \( F \) and the referenced elements along that dimension are in the bounds of the corresponding dimension of \( A \). The range of the translated section for an inconsistent dimension is set to 1. This is safe because for MOD and REF an approximate translation of a simple section must contain the precise translation.

Next the algorithm translates the diagonal bounds of \( F \). For each diagonal \( x \pm y \), if some of \( x \) or \( y \) is inconsistent, the translated diagonal is set to 1. Otherwise, the adjustments made to the corresponding rectangular boundaries are folded into the bounds of the diagonal. This algorithm is an extension of the translation process for regular sections presented by Havlak [6].

The algorithm for translating killed simple sections looks very similar to the one described above. A fundamental difference is that once an inconsistent dimension is encountered, the translated killed section is set to 1. This is necessary because the translated killed section produced by the algorithm must be at most as big as the exact translation.

The cost for translating one boundary of a simple section is \( O(1) \). Assuming that the number of dimensions of a simple section is \( d \), the cost of the algorithm is \( O(d^2) \).
Algorithm Translate

Input: The bounds of the referenced simple section, \( \text{ref}_F \). The formal bounds of the referenced section, \( \text{bounds}_F \). The formal bounds of the actual parameter, \( \text{bounds}_A \). The section of passed array elements, \( \text{pass}_A \).

Output: The translated section, \( \text{ref}_A \).

begin
1. if \( \text{ref}_F = \top \)
2. return \( \top \)
3. consistent = true
4. for \( i = 1 \) to \( \text{rank}(A) \) do
5. if not consistent
6. \( \text{ref}_A[i] = \bot \)
7. else if \( i > \text{rank}(F) \)
8. \( \text{ref}_A[i] = \text{pass}_A[i] \)
9. else
10. replace scalar formal parameters in \( \text{bounds}_F \) and \( \text{ref}_F \) with the corresponding actual parameters
11. \( \text{bounds}_i = \text{bounds}_F[i] - \text{lo}((\text{bounds}_F[i]) + \text{pass}_A[i] \)
12. \( \text{ref}_i = \text{ref}_F[i] - \text{lo}((\text{bounds}_F[i]) + \text{pass}_A[i] \)
13. consistent = (\( \text{bounds}_i = \text{bounds}_A[i] \))
14. if consistent
15. \( \text{ref}_A[i] = \text{ref}_i \)
16. else if \( i = \text{rank}(F) \) and \( \text{ref}_i \) fits in \( \text{bounds}_A[i] \)
17. \( \text{ref}_A[i] = \text{ref}_i \)
18. else \( \text{ref}_A[i] = \bot \)
19. endfor
20. for each diagonal boundary \( d = x + y \) do
21. let \( t_x = x + c_x \) be the translated value of \( x \)
22. let \( t_y = y + c_y \) be the translated value of \( y \)
23. set \( t_x + t_y = d + c_x + c_y \)
24. endfor
25. for each diagonal boundary \( d = x - y \) do
26. let \( t_x = x + c_x \) be the translated value of \( x \)
27. let \( t_y = y + c_y \) be the translated value of \( y \)
28. set \( t_x - t_y = d + c_x - c_y \)
29. endfor
end

Figure 5.3 Algorithm for translating a simple section into a call context.
5.3 Accumulation of KILL information

As mentioned in chapter 3, the local KILL information of a procedure is stored in its control dependence graph. Each node of the graph is annotated with information about the section killed at the node. There are two types of annotations:

- Annotations representing sections directly killed through an assignment statement. These annotations hold simple sections.

- Annotations representing sections killed indirectly through an invocation. These annotations hold place holders identifying subarrays that might be killed as a result of the invocation.

The indirect simple sections represented by place holders in the graph must be available before accumulating the information in the graph. But since the interprocedural algorithm iterates through the call graph nodes in reverse topological order, the indirect sections have been computed in previous iterations and stored as annotations on the call graph edges.

Figure 5.4 illustrates algorithm ACCUMULATE.KILLS, that computes cumulative KILL summaries for a procedure. The algorithm takes as an input the annotated control dependence graph of the procedure $P$, and computes the sections that are killed on every control flow path in the body of $P$.

To illustrate the basic idea behind this algorithm, figure 5.5 displays an instance of a control dependence graph. The edges of the graph are annotated with labels. Edges with the same label lie on the same control flow path. Since edges $ab$ and $ac$ are annotated with the same label, nodes $b$ and $c$ lie on the same control flow path starting from $a$. Similarly, nodes $d$ and $e$ lie on the same path. Node $f$ lies on a separate path.

Each node is annotated with information about the section killed at that node and its descendants. Assume that the algorithm is at this point computing the summary of simple sections killed at node $a$ and its descendants. Also, assume that the summaries of sections killed at each of the nodes $b$, $c$, $d$, and $e$ consists of the simple sections $S_0$, $S_1$, $S_2$, $S_3$, and $S_4$ respectively. The place holder $\text{sub}(A, I)$ indicates that the only section killed at node $f$ and its descendants is the subsection of $A$ killed through an invocation of procedure $\text{sub}$ (the place holder is placed at node $f$ for illustrative purposes only. The actual algorithm processes the information at node $f$ before that
Algorithm ACCUMULATE_KILLS

Input: A node $n$ of the annotated control dependence graph, CDG, of a procedure, computed in the local phase. $D_n$ denotes the local KILL information annotated on $n$.

Output: The node $n$, annotated with the summary $S_n$ of the sections killed on $n$ and its descendants.

begin
1. Allocate a hash table $H_n$ with entries pairs $(l, S)$, where $l$ is a label and $S$ is a summary of the sections killed through edges with label $l$.
2. for each outgoing edge $e$ of $n$ with label $l$ do
3. \hspace{1em} $n' = sink(e)$
4. \hspace{2em} if not marked($n'$)
5. \hspace{3em} mark($n'$)
6. \hspace{2em} ACCUMULATE_KILLS($n'$)
7. \hspace{1em} $S_n' = \text{GetAnnotation}(n')$
8. \hspace{2em} if there is an entry $(l, S)$ in $H_n$
9. \hspace{3em} enter the pair $(l, \text{MergeSummaries}(S_n', S))$ in $H_n$
10. \hspace{2em} else
11. \hspace{3em} enter the pair $(l, S_n')$ in $H_n$
12. endfor
13. $S_n = \text{IntersectSummaries}(H_n)$
14. if $D_n$ holds a simple section $s$
15. \hspace{1em} if $s$ is included in summary $S_n$
16. \hspace{2em} union $s$ with the corresponding section in $S_n$
17. \hspace{1em} else
18. \hspace{2em} insert $s$ in $S_n$
19. end if $S_n$ holds a place holder corresponding to an invocation $c$
20. extract from the call graph the list of sections killed through $c$
21. for each simple section $s$ in the list
22. \hspace{1em} if $s$ is included in summary $S_n$
23. \hspace{2em} union $s$ with the corresponding section in $S_n$
24. \hspace{1em} else
25. \hspace{2em} insert $s$ in $S_n$
26. endfor
27. endfor
end

Figure 5.4 Algorithm for accumulating the annotated control dependence graph.
at node a. Thus, when computing the KILL set for node a, the place holder will already have been replaced by the indirect side effects of the call to sub on array A).

The algorithm must annotate node a with the summary of simple sections killed on every control flow path starting at a. Since nodes b and c lie on the same control flow path, the summary of the sections killed on both b and c is $S_1 \cup S_2$. Similarly, the summary of the sections killed on both c and d is $S_3 \cup S_4$. The summary of sections killed on all control flow paths rooted at a is:

$$\bigcup (\bigcap (\bigcup (S_1, S_2),
    \bigcup (S_3, S_4)),
    sub(A, 1)),
S_0)$$

Algorithm ACCUMULATE.KILLS takes as an input a node n the annotated control dependence graph. $D_n$ represents the local KILL information annotated on node n. The output of the algorithm is the node n annotated with the summary $S_n$ of the sections killed on n and its descendants. The algorithm performs a depth first search on the graph rooted at n and applies the operations described in the above example to each node it visits. To distinguish nodes that have already been visited, the algorithm marks n before performing any operation. Then it allocates a hash table $H_n$. Each entry $(l, S)$ of the hash table holds the union, $S$, of sections killed through edges with source n and label l. That is, it holds the summary of simple sections killed on the
same control flow path. For each edge \((n, n')\) with label \(l\), the algorithm computes
the summary \(S_{n'}\) of \(n'\). If \(n'\) has not been visited, \textsc{Accumulate.Kills} is called
on \(n'\). If it has been visited, then \((n, n')\) is either a cross edge or a back edge of the
depth first search tree. If \((n, n')\) is a cross edge, the summary \(S_{n'}\) has already been
computed. If \((n, n')\) is a back edge, \(S_{n'}\) is assumed to be empty. This is legal if \((n, n')\)
is a loop edge, because each node in the loop body has been annotated in the
local phase with the sections that are modified on that node in any iteration of the
loop. After computing \(S_{n'}\), the algorithm checks whether there is an entry \((l, S)\) in
\(H_n\). If this is the case, it merges \(S_{n'}\) and \(S\). Otherwise it inserts the pair \((l, S_{n'})\) in
\(H_n\).

After merging the sections killed through edges with the same label, the algorithm
intersects the summaries held in each entry of \(H_n\). That is, it intersects the sections
killed on different control flow paths. The resulting summary, \(S_n\), is merged with the
sections directly killed on node \(n\), held in annotation \(D_n\). If \(D_n\) holds a simple section,
\(s\), then \(s\) is merged with the summary \(S_n\). If \(D_n\) holds a place holder representing an
invocation \(c\), then the list of sections killed through \(c\) is extracted from the call graph
edge representing \(c\) and merged with the summary \(S_n\).

When the algorithm terminates, each node is annotated with a summary of the
sections killed at that node and its descendants. The cumulative \textsc{Kill} summary of
the procedure is the annotation of the root of the graph.

5.4 Final local analysis

The interprocedural propagation of array section side effects on the call graph
produces cumulative summaries of the side effects occurring in the body of each proce-
dure. The final local phase uses these summaries to compute the side effects of each
call in the context of the calling procedure. The results of this phase can be used to
perform dependence testing and optimizations in the presence of procedure calls.

An outline of the final local phase is shown on figure 5.6. For each call site in
the program, the algorithm translates the cumulative summaries of the callee in the
context of the caller. Since the cumulative side effects of each procedure have been
computed in the interprocedural phase, no particular order of processing is required
for the call sites.

There are two differences between the translation processes performed at each call
site during the interprocedural phase and the final local phase:
Algorithm FINAL LOCAL ANALYSIS

Input: The call graph, CG, of a procedure P.
Output: CG, where each edge is annotated with sections translated to the local context of P.

begin
1. read in the local value numbers of P
2. read in the cumulative summaries for each procedure
3. read in the shapes of the array actual parameters for each call site in P
4. read in the shapes of the formal parameters for each call site in P
5. for each node in CG do
6. for each outgoing edge do
7. translate the cumulative summaries of the callee to the context of P
8. endfor
9. endfor
end

Figure 5.6 Final phase.

In the final local phase the side effects occurring in the body of the called procedure are translated to the context of the call for each array actual parameter of the call. In the interprocedural phase only the side effects on arguments that are global variables or formal parameters of the caller are translated. This difference is due to the fact that during the interprocedural propagation only the side effects of calls to formal and global variables are significant, while in the final local phase the side effects to all variables visible in the context of a call are important.

Also, in the final local phase the caller's local value numbers are used, as opposed to the interprocedural value numbers used during the interprocedural propagation. This happens because this phase computes side effects of an invocation in the call context, where information about local scalar variables can enhance precision of the bounds of simple sections.

The translation of a simple section does not differ in the two phases. Thus, translation of simple sections in the final phase can be performed using the procedures discussed in section 5.2.
Algorithm PRIVATIZE ARRAYS

Input: The SSA graph of the program. For each loop $L$ in the program, a list $L_p$ of simple sections killed through a single iteration of $L$. The KILL and REF side effects for each call.

Output: For each loop $L$ in the program, the list $L_p$ containing the simple sections privatizable with respect to $L$, and a list $L_f$ of arrays that must be finalized after privatization with respect to $L$.

begin
1. for each loop $L$ do
2.    for each array variable $A$ defined in $L$ do
3.        let $s$ be the first SSA def of $A$ in $L$
4.        work_list = $\{ s \}$
5.        let $d_s$ be the nesting level of $s$
6.        while work_list is not empty
7.            $n$ = first(work_list)
8.            mark $n$ as visited
9.            let $d_n$ be the nesting level of $n$
10.       if $n$ is a use
11.          if $d_n < d_s$
12.              insert($L_f$, $A$)
13.          else
14.              let $U_n$ be the section referenced at $n$ in a single iteration of $L$
15.              for each def pred reaching of $n$ do
16.                  let $C_{pred}$ be the cover of pred
17.                  if $C_{pred}$ does not include $U_n$
18.                      delete($L_p$, $A$)
19.              else ($n$ is a def)
20.                  $C_n$ = COVER($n$)
21.              for each successor succ of $n$ do
22.                  if succ is not visited
23.                      insert(work_list, succ)
24.              endwhile
25.          endwhile
26.    endfor
end

Figure 5.7 Algorithm for computing simple sections privatizable with respect to each loop in a program
Algorithm COVER

Input: An SSA node \( n \).
Output: The cover simple section of \( n \).

\[
\begin{align*}
&\text{begin} \\
&1. \text{ if } (s = n) \text{ and this is the first visit to } s \\
&2. \quad \text{return empty\_section} \\
&3. \quad \text{if } n \text{ is a direct definition} \\
&4. \quad \text{let } kill_n \text{ be the section killed at } n \text{ in a single iteration of } L \\
&5. \quad \text{else if } n \text{ is an indirect definition} \\
&6. \quad \text{let } kill_n \text{ be the section killed through the call corresponding to } n \\
&7. \quad \text{else (this is a phi node)} \\
&8. \quad \quad kill_n = T \\
&9. \quad \quad intersect = T \\
&10. \quad \text{for each definition } pred \text{ reaching } n \text{ do} \\
&11. \quad \quad \text{let } C_{pred} \text{ be the cover of } pred \\
&12. \quad \quad \quad intersect = \text{SMALL\_INTERSECTION}(intersect, C_{pred}) \\
&13. \quad \text{endfor} \\
&14. \quad \text{return } \text{SMALL\_UNION}(intersect, kill_n) \\
&\text{end}
\end{align*}
\]

Figure 5.8 Algorithm for computing the cover of an SSA node.

5.5 Array privatization

This section discusses the use of array side effect analysis in array privatization, one of the most effective optimizations for the exploitation of parallelism.

Array privatization removes true and output dependences introduced when locations of an array are written in different iterations of a loop by allocating a private copy of the array to each iteration. An array section is privatizable with respect to a loop \( L \) if it is killed before any use in each iteration of \( L \).

Figure 5.7 displays an algorithm that computes simple sections privatizable with respect to each loop in a program. Specifically, for each array variable \( A \) and each loop \( L \), the algorithm determines whether the simple section of \( A \) that is killed in a single iteration of \( L \) is privatizable with respect to \( L \). The definitions and uses in \( L \), as well as the def-use edges, are accessed through the SSA graph of the program. The
algorithm determines whether the simple section corresponding to a use in the body of $L$ is a subset of the section killed in each reaching definition in the body of $L$. To achieve that, it computes the cover section for each def node in $L$. For a def node $d$, $\text{cover}(d)$ is defined as:

$$\text{COVER}(d) = \text{KILL}(d) \cup \{\cap_{p \in \text{pred}(d)} \text{cover}(p)\}$$

That is, the cover section of $d$ represents the simple section that is killed on $d$ and its predecessors. An algorithm for computing the cover of a def node in the SSA graph is shown on figure 5.8. Based on the above definition, a simple section is privatizable with respect to $L$ if the simple section referenced through each use node $u$ in $L$ is a subset of the cover section of each of $u$'s predecessors. That is, if $S_u$ is the section referenced through $u$, the following must be true:

$$S_u \subseteq \bigcap_{p \in \text{pred}(d)} \text{cover}(p)$$

In addition to determining privatizability the algorithm determines whether a definition in the body of $L$ reaches a use outside the loop. If this happens, the privatized section must be finalized at the end of the loop, that is, the values generated by the last iteration of the loop must be restored to the original array.

This algorithm can be extended to determine when privatization of an array is profitable. A simple extension would be to mark simple sections whose bounds vary with the loop induction variable of a loop as not profitable to privatize with respect to $L$. Moreover it can be extended to automatically privatize simple sections, and update the dependence graph by removing dependences deleted through privatization.
Chapter 6

Experimental Validation

This chapter discusses the implementation of the interprocedural side effect analysis presented in this thesis, and presents results of experiments conducted to evaluate the precision of the analysis.

6.1 Implementation

The analysis described in the previous chapters has been implemented in the context of the ParaScope programming environment. ParaScope is an interactive parallel programming tool that assists knowledgeable users in developing scientific FORTRAN programs. It displays the results of sophisticated program analyses, provides a set of powerful program transformations, and supports program editing.

Figure 6.1 displays the structure of ParaScope. The source editor computes local information for each procedure body. The composition editor produces the call graph. The local information and the call graph are used by the program compiler to perform interprocedural data flow analysis. The interprocedural information is used by the module compiler to perform the final analysis of each module. In this phase no compilation order dependences exist.

This scheme allows incremental analysis of the source program. If some procedure is edited, its local information is updated and interprocedural data flow analysis is performed again by the program compiler. Local analysis is not necessary for procedures that are not modified.

The array side effect analysis presented in this thesis has been designed to naturally fit into ParaScope’s structure. The local analysis is part of the source editor, the interprocedural propagation is part of the program compiler, and the dependence testing phase is performed by the module compiler. This design allows use of other local and interprocedural analyses results in the interprocedural propagation of local array side effects. Currently results of scalar MOD analysis are used to enhance the precision of the bounds of simple sections. Results of other analyses can be used in the
future without modification of the array side effect analysis. It would be interesting to see how constant propagation affects the precision of simple section side effects.

6.2 Experimental results

Array privatization is a critical optimization for loop parallelization. The computation of privatizable arrays with respect to loops containing procedure calls requires interprocedural KILL and REF array side effect information. The importance of array KILL analysis for array privatization has been stressed in a study performed on the ParaScope programming environment [17], and by Havlak [6]. The precision of the interprocedural analysis presented in this thesis is evaluated by using the results to perform array privatization. This section presents the experiments conducted to evaluate the precision of the analysis and discusses the results.

The goal of the first experiment was to evaluate the precision of the local analysis and the privatization algorithm. Thus, in most cases calls were inlined. Worst case assumptions were made for non-inlined calls. Figure 6.2 presents the results of this experiment. For each source program the figure displays the total number of lines, the total number of loops, the number of calls in loops, the number of loops proven
<table>
<thead>
<tr>
<th>source</th>
<th>lines</th>
<th>loops</th>
<th>calls in loops</th>
<th>parallel loops</th>
<th>dependences</th>
<th>% red</th>
</tr>
</thead>
<tbody>
<tr>
<td>init (slab2d)</td>
<td>57</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>setup1 (slalom)</td>
<td>167</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>292</td>
</tr>
<tr>
<td>poteng (mdg)</td>
<td>119</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1395</td>
</tr>
<tr>
<td>filter3d (arc3d)</td>
<td>68</td>
<td>11</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>42</td>
</tr>
<tr>
<td>proj (qcd)</td>
<td>64</td>
<td>7</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>1247</td>
</tr>
<tr>
<td>pln2 (spec77)</td>
<td>33</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>qrsolv (minpack)</td>
<td>52</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>141</td>
</tr>
<tr>
<td>update (qcd)</td>
<td>57</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>30</td>
</tr>
</tbody>
</table>

**Figure 6.2** Results of experiments determining the effectiveness of the privatization algorithm.

<table>
<thead>
<tr>
<th>source</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter3d</td>
<td>Missed array <em>wr2</em> because of insufficient symbolics.</td>
</tr>
<tr>
<td>poteng</td>
<td>Inlined 3 calls to <em>cshift</em>. Array <em>rl</em> not killed in all branches.</td>
</tr>
<tr>
<td>pln2</td>
<td>Remaining dependences are on scalars and can be eliminated through auxiliary induction variable elimination.</td>
</tr>
<tr>
<td>qrsolv</td>
<td>Array <em>r</em> is not privatizable. Some dependences on scalars can be eliminated through scalar expansion.</td>
</tr>
<tr>
<td>proj</td>
<td>Passed commons as parameters. Inlined one call to <em>cfill</em>.</td>
</tr>
<tr>
<td>update</td>
<td>Inlined two calls to <em>getpos</em>. Remaining dependences are on scalars.</td>
</tr>
</tbody>
</table>

**Figure 6.3** Comments on the source programs used for array privatization.
<table>
<thead>
<tr>
<th>source</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>filt3d</td>
<td>14.232</td>
</tr>
<tr>
<td>poteng</td>
<td>9.332</td>
</tr>
<tr>
<td>init</td>
<td>13.199</td>
</tr>
<tr>
<td>setup1</td>
<td>130.811</td>
</tr>
<tr>
<td>pln2</td>
<td>0.783</td>
</tr>
<tr>
<td>qrsolv</td>
<td>4.099</td>
</tr>
<tr>
<td>projec</td>
<td>3.433</td>
</tr>
<tr>
<td>update</td>
<td>1.233</td>
</tr>
</tbody>
</table>

**Figure 6.4** Compilation time for local analysis and privatization.

safely parallel after scalar interprocedural analysis (IP) and after array privatization (PRIV), the total number of dependences after interprocedural analysis (IP) and after array privatization (PRIV), the number of dependences deleted though array privatization (deleted), and the percentage of dependence reduction (% red).

Since the implementation of the analysis currently does not support common variables, in some cases common variables were passed as parameters. Also, to overcome limitations of ParaScope in auxiliary induction variable recognition, auxiliary induction variables were in some cases eliminated. The transformations performed on each of the source programs appear in figure 6.3.

In all but one of the source programs all privatizable arrays were discovered. In procedure \textit{filter3d} one privatizable array was not recognized because of insufficient symbolic information. In procedure \textit{poteng} one array was not privatized because it was not killed on both branches of a conditional statement. This shows that branch prediction could improve the precision of the local array KILL analysis. Finally, a large number of the remaining dependences can be deleted through scalar expansion, exposing more parallel loops. These results show that the analysis in precise and useful in disproving dependences on arrays. Also, symbolic information on loop bounds and array indices is critical for the precision of the analysis.

Figure 6.4 displays compilation times in seconds for each of the source programs. The programs were compiled on a SPARC1 workstation with 13352K of RAM, 6016K of which is swap space. The times include local analysis and privatization. These numbers show that analysis is fast for all source programs except of \textit{setup1}. This program contains a lot of three dimensional array sections. Analysis is slow because
of the complexity of the exact union test. However, the analysis has disproved all dependences in this example. To speed up the analysis, the exact union test could be replaced with a less precise but faster heuristic.

A second experiment was conducted to determine the precision and efficiency of the interprocedural propagation. The experiment used the results of the interprocedural analysis to determine privatizable array sections. The source programs were kernels extracted from qcd, matrix300, mdg, and spec77. In particular, procedure choos from qcd, procedure sgemm from matrix300, the main loop of procedure poteng from mdg, and a kernel containing a call to sumpls that appears frequently in spec77. The latter was used in chapter 3 as an example of the use of array section analysis in dependence testing. Results of this experiment are shown in figure 6.5. The first two columns display characteristics of the source programs, that is, the number of lines and the number of calls in loops. The next column shows the number of opportunities for privatization. Since an array is privatizable with respect to a loop this column displays, for each program, the sum of the number of arrays that are privatizable with respect to each loop. The last two columns display compilation times in seconds measured on two machines. Column $T_1$ shows compilation times on a machine comparable to the one used in the first experiment. Those times are provided as a reference to the compilation times required for interprocedural array section analysis as compared to the ones measured in the first experiment, where array section analysis was performed locally on programs with inlined calls. Column $T_2$ displays total
compilation times on a SPARC1 workstation with 43304K of RAM, 3236K of which is swap space.

For choos, sgemm, and sumpls all privatizable arrays were recognized. Some of the privatizable arrays in poteng were not discovered. This is not due to lack of precision of the interprocedural analysis (which works as expected), but rather to the current implementation of the privatization algorithm. The algorithm rules an array not privatizable if, for some call site, sections that are referenced through the call are not covered by all reaching definitions in the body of the caller. Although this approximation works well in a lot of cases, it fails to recognize array sections that are privatizable because, for each call site, the sections referenced through the call are covered by all reaching definitions in the body of the callee. This is something that can be worked on as an enhancement to the privatization algorithm. In general, the results of the above experiments show that the interprocedural analysis presented in this thesis is stable, precise, and useful in loop parallelization.
Chapter 7

Conclusions

This chapter discusses the unique contributions of the analysis presented in this thesis and presents open questions.

7.1 Contribution

This thesis presents and evaluates an interprocedural array side effect analysis that is more precise than existing efficient solutions, yet not as expensive as the precise methods. It presents an efficient implementation of interprocedural array MOD and REF side effect analysis. Most importantly, it presents a framework for performing interprocedural array KILL analysis.

The most important use of array KILLS is in array privatization. This thesis presents an algorithm for array privatization that uses the results of interprocedural KILL and REF analysis. It evaluates the precision of the analysis by determining the number of arrays proven to be privatizable, and the number of dependences deleted through array privatization. The results show that the analysis is precise and useful for automatic loop parallelization.

The analysis is based on simple sections, a representation that allows precise description of the shapes commonly encountered in practice. Specifically, the information about diagonals available in simple sections allows precise description of triangular and trapezoidal sections, which is not possible with efficient rectangular representations. When operations on simple sections are not precise, approximations to the exact solutions are used. Since this results in loss of information, it is important to determine when operations on simple sections are precise. This thesis presents a precise algorithm for determining exactness of the union of simple sections.

Most operations on simple sections are quadratic to the number of dimensions of their operands. This contributes to the efficiency of the analysis. Another factor that contributes to the efficiency of the method is the division of the solution to a local and an interprocedural phase, which minimizes the number of passes over each procedure
body. Also, the description of the side effects on arrays occurring in each procedure body through a single simple section for each variable allows fast computation of indirect side effects at the call sites. The division of the solution to phases and the description of cumulative side effects on each array through single descriptor was also used by Havlak [8].

Finally, the implementation of this analysis provides the basis for the array side effect analysis that will be incorporated in the ParaScope programming environment.

7.2 Future work

Several optimizations can be performed on the analysis presented in the previous chapters. Precision of the bounds of simple sections can be enhanced through use of the results of more interprocedural scalar analyses, such as constant propagation. Also, precision of the local KILL analysis can be increased by using symbolic information to determine the targets of branches.

The final local phase can be extended to make the results of array KILL analysis useful for dependence testing. Finally, the results of the analysis presented in this thesis can be used in conjunction with other array side effect analyses, for example LIVE, to further improve the precision of dependence testing.
Bibliography


