INLINE EXPANSION FOR THE POLARIS RESEARCH COMPILER

BY

JOHN ROBERT GROUT

B.S., Worcester Polytechnic Institute, 1981

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

ABSTRACT

This thesis describes a robust, high-performance general-purpose inliner for the Polaris research compiler. It also describes inline expansion and related interprocedural analysis techniques and transformations, and a series of experiments to measure the performance and effectiveness of complete inline expansion as performed by Polaris on the Sun SparcStation and SGI Challenge platforms and by the Power Fortran Accelerator (PFA) on the latter platform.

[1]Research supported in part by Army contract DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the Army or the Government.

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, David Padua, for his suggestions, his guidance, and, above all, his patience with the uneven pace of my research. I would also like to thank William Blume, Rudolf Eigenmann, Jay Hoeflinger, Paul Petersen, Stephen Weatherford, and the other members of the Polaris team, both past and present, for their support throughout my thesis research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

For many years, the Center for Supercomputing Research and Development (CSRD) has been working to make parallel computing more practical. After an effort to manually parallelize the Perfect Benchmarks® to identify effective program transformations [6, 7] for parallelization, CSRD began to implement the Polaris research compiler [2], which provides both a basic infrastructure for manipulating Fortran programs and many of the program transformations shown to be effective.

Though the Polaris compiler prototype implemented some new techniques to perform the transformations described in [7], it was decided that flow-sensitive interprocedural analysis would be performed initially by using source-to-source inline expansion to transform interprocedural data flow problems into intraprocedural form, and that more efficient interprocedural analysis techniques would be implemented later. This decision led to the implementation of a robust, high-performance, general-purpose inliner to minimize the number of call sites at which inline expansion could not be performed.

This thesis describes the design goals and overall design of the Polaris inliner driver and inliner, including discussions of the major issues involved in performing source-to-source inline expansion of Fortran programs. It also gives an overview of Polaris, and describes inline expansion and related interprocedural analysis techniques and transformations. It measures the performance and effectiveness of complete inline expansion on a subset of the Perfect Benchmarks® as performed by Polaris on the Sun SparcStation and SGI Challenge platforms and by the Power Fortran Accelerator (PFA), supplied by Kuck and Associates, on the latter platform. Finally, it discusses conclusions drawn by the implementation and use of the Polaris inliner, and plans for future work on the inliner and on the reuse of Polaris inliner technology.

CHAPTER 2

INLINE EXPANSION

Inline expansion replaces a procedure call site with a version of the associated procedure body which reflects information specific to the calling procedure's environment and to a particular call site. At the cost of increasing the size of the program, making a local copy of a procedure body may help a compiler perform site-specific optimizations. Complete inline expansion reduces a whole program to one program unit by performing inline expansion iteratively at each call site until a fixed point is reached. Though it may exponentially increase the size of the program, complete inline expansion removes any barriers which might prevent data-flow analysis of a whole program.

Inline expansion of Fortran code for the Parafrase compiler is described in [14], and a general-purpose inliner for C code is described in [13]. The limitations of source-to-source inline expansion, which are the inability to express some language constructs after expansion in source form, the potential for exponential growth in program size, and the inability of modern optimizing compilers to effectively compile large programs after complete inline expansion ([5]), sparked the development of interprocedural data flow analysis techniques, which are surveyed in [9], [12] and [19].

Most of the early interprocedural analysis techniques merge information into a flow independent summary form, which saves time or space but loses the precision needed to perform techniques such as array privatization ([20]). More recent interprocedural analysis techniques, such as atom image ([15]) and those developed for FIAT ([10]), can efficiently propagate precise array subscript information over procedure boundaries. However, demand-driven partial inline expansion is still useful as a supplement to such techniques to aid code scheduling ([18]) and automatic parallelization ([11]), and inliner technology can be reused when performing transformations such as loop embedding and loop extraction ([3, 4]).

CHAPTER 3

A POLARIS OVERVIEW

The Polaris research compiler ([2]) provides a basic infrastructure, the “Polaris base”, which includes a powerful internal representation, generalizable data structures and general-purpose utilities. It also provides implementations for many effective program transformations, including inline expansion, interprocedural constant propagation, induction variable substitution, propagation of constants, symbolic constants, and variables, dead-code and dead-store elimination, array privatization, reduction substitution and data dependence analysis. Finally, it provides a postpass to select loops for parallelization and output the transformed program in the Fortran dialects (including directives) of such systems as the SGI Challenge and Convex Exemplar.

Polaris's internal representation uses an abstract syntax tree to represent data (such as expressions, parameter lists, etc.) and more specialized constructs to represent program structures (such as a symbol table, or a list of a routine's statements). To catch programming errors as early as possible, assertions are used liberally throughout Polaris to enforce the semantic correctness of actions. For example, the statement list construct prohibits the insertion of statements (or groups of statements) which would leave a routine with unbalanced block structure[1].

It was chosen to implement Polaris using C ++ and represent most constructs using objects of classes which derive from broad categorical class hierarchies. For example, a Fortran DO statement is represented as an object of class DoStmt, which is derived from class Statement, the base of the statement class hierarchy. Most individual services for general class hierarchies (e.g., the statement and expression class hierarchies) are implemented using virtual member functions, which allows pointers and references to such constructs to be left in the most general form.

The Polaris collection class hierarchy provides a family of template classes which implement lists, sets, trees, and maps of class objects, and provides the concept of object ownership, iterators and mutators, and reference counting to maintain the semantic correctness of a collection. Polaris also contains many different general-purpose utilities, such as control flow and loop normalizers and expression search-and-replace routines.

A whole Fortran program is implemented in Polaris as a Program object containing one or more ProgramUnit objects, each of which represents an individual Fortran routine. Each ProgramUnit object contains a number of bookkeeping objects (e.g., a symbol table, DATA statements, FORMAT statements), and a StmtList object to represent the executable statements of a routine and their structure. A StmtList object owns various class objects derived from class Statement, the base of the statement class hierarchy, which represent the statements of the routine.

[1] Such as a DO without a matching ENDDO, or an IF without a matching ENDIF.

A statement contains information about adjoining statements and sets representing references to variables, and owns a list of "top level" expressions which represent the top of an abstract syntax tree.[2] An expression is represented by an object of a class derived from class Expression, the base of the expression class hierarchy, and implements all or part of an abstract syntax tree representing an arbitrarily complex expression.

[2]Such as the left-hand side and right-hand side of an assignment statement.

CHAPTER 4

THE POLARIS INLINER

4.1 Goals

The effort to manually parallelize the Perfect Benchmarks® identified a number of new program transformations, such as array privatization ([20]), which required flow-sensitive interprocedural data flow analysis. To provide this analysis within the time and resource limits of the Polaris compiler prototype, it was chosen to combine intraprocedural data flow analysis and a high-performance, general-purpose inliner which would have enough functionality to perform complete inline expansion on a substantial portion of the Perfect Benchmarks® and enough flexibility to also support partial inline expansion on demand.

Another goal which evolved during the implementation of the Polaris prototype was to design Polaris inliner technology for reuse in new Polaris transformations, including run-time parallelization ([17]) and demand-driven flow-sensitive interprocedural analysis.

4.2 Design Overview

4.2.1 The InlineObject class

To minimize the cost of complete inline expansion, the transformations to inline a given subprogram into a designated program (e.g., for complete inline expansion) or subprogram (e.g., for selective inline expansion) are split into site-independent transformations and site-specific transformations.

To represent this transformation structure, inline expansion in Polaris uses instances of the InlineObject class (hereafter called InlineObjects). A newly-created InlineObject points to a subprogram which will be expanded into a "top level" routine (defined next, in Subsection 4.2.2). A template InlineObject contains a version of that subprogram with all site-independent transformations performed for expansion at any site in the "top level" routine. A work InlineObject contains a version of the subprogram with all site-specific transformations performed for expansion at a specific site in the "top level" routine.

4.2.2 The inliner driver

The Polaris inliner driver, whose algorithm is given in Figure 4.1, performs both selective and complete inline expansion and modifies a Program object passed to it in place. On input, the Program object contains ProgramUnit objects for each routine (program, subroutine, function or BLOCK DATA) within the original program. On output, the Program object contains ProgramUnit objects for the main program, any subprogram which was called without inline expansion somewhere in the original program, and any BLOCK DATA routines.

The inliner driver expands at a specific call site if the global default (set in switch "inline_site") is to expand at all call sites, or if an annotation requesting inline expansion is present at the site. The inliner driver does not expand a specific call site if the global default is to not expand at any call sites, or if an annotation preventing inline expansion is present at the site. Call site annotations are not path-sensitive: for example, if a specific call site to subroutine C from subroutine B is annotated to request inline expansion, the code of subroutine C will be expanded at that site wherever the code of subroutine B appears.

The inliner driver builds a directed acyclic call multigraph (call DAMG) which represents calls between routines of the program which are to be (or not to be) expanded. Between any two nodes, there can be at most two arcs: one to represent a call (or calls) to be expanded and one to represent a call (or calls) to not be expanded. A header node in the call DAMG represents a "top level" routine (a program, or a subprogram called somewhere without inline expansion) which will appear in the output Program object.

4.2.3 The inliner

The Polaris inliner, whose algorithm is given in Figure 4.2, reads an inline flow DAG, performs the inline expansions indicated by the graph, and outputs a "top level" ProgramUnit object. The header node of the inline flow DAG represents the top level routine (the target for expansion) and all the other nodes (and their corresponding template InlineObjects) represent routines which will be expanded into the top level routine. Inline expansion is done top down in depth-first order: if an expanded subprogram contains a call site to be expanded, the inliner will process the first such call site next.

```
Build the call DAMG
Prune the call DAMG by removing nodes representing “dead” subprograms
Build a list of header nodes to represent “top level” routines
FOR each header node $n$ which has edges representing inlined calls to $n$
    Split $n$, creating node $n'$ (see Figure 4.4)
    Reseat all edges representing inlined calls to $n$ to point to $n'$
END FOR
Collapse the call DAMG to a DAG (by removing all edges representing uninlined calls)
Partition the call DAG into inline flow DAGs (see Figure 4.3)
FOR each non-trivial inline flow DAG
    Call the inliner to create the corresponding “top level” routine (see Figure 4.2)
END FOR
```

Figure 4.1: Inliner Driver Algorithm

```
Find all entry points in the inline flow DAG
Substitute all integer symbolic constants (Fortran PARAMETERs) in “top level” routine
FOR each call site to be expanded (as encountered at “top level”)
    IF the called InlineObject is not a “template InlineObject”
        Create a “template InlineObject” by performing site-independent transformations
    ENDIF
    Clone the “template InlineObject” to create a “work InlineObject”
    Perform site-specific transformations
    Transform all references to subprogram variables to “top level” form
    Replace the calling statement in the “top level” routine by the transformed statements
END FOR
```

Figure 4.2: Inliner Algorithm

4.3 Design Considerations

4.3.1 Building and partitioning the DAMG

The directed acyclic call multigraph (call DAMG) represents the calling structure of the original program. Each node represents an executable routine in the original Program object, and takes over ownership of the corresponding ProgramUnit object (which is deleted if the node is deleted). Each edge is of a type (to be expanded or not), is from a node (representing a calling routine) to another node (representing a called routine) and represents all the calls of that particular type from the calling routine to the called routine (so there are at most two edges from one routine to another). The header nodes in the DAMG represent the routines which will appear at top level in the output Program object (either the main program or a subprogram called somewhere without expansion); after the header nodes are found, edges representing uninlined calls are removed, collapsing the call DAMG to a DAG.

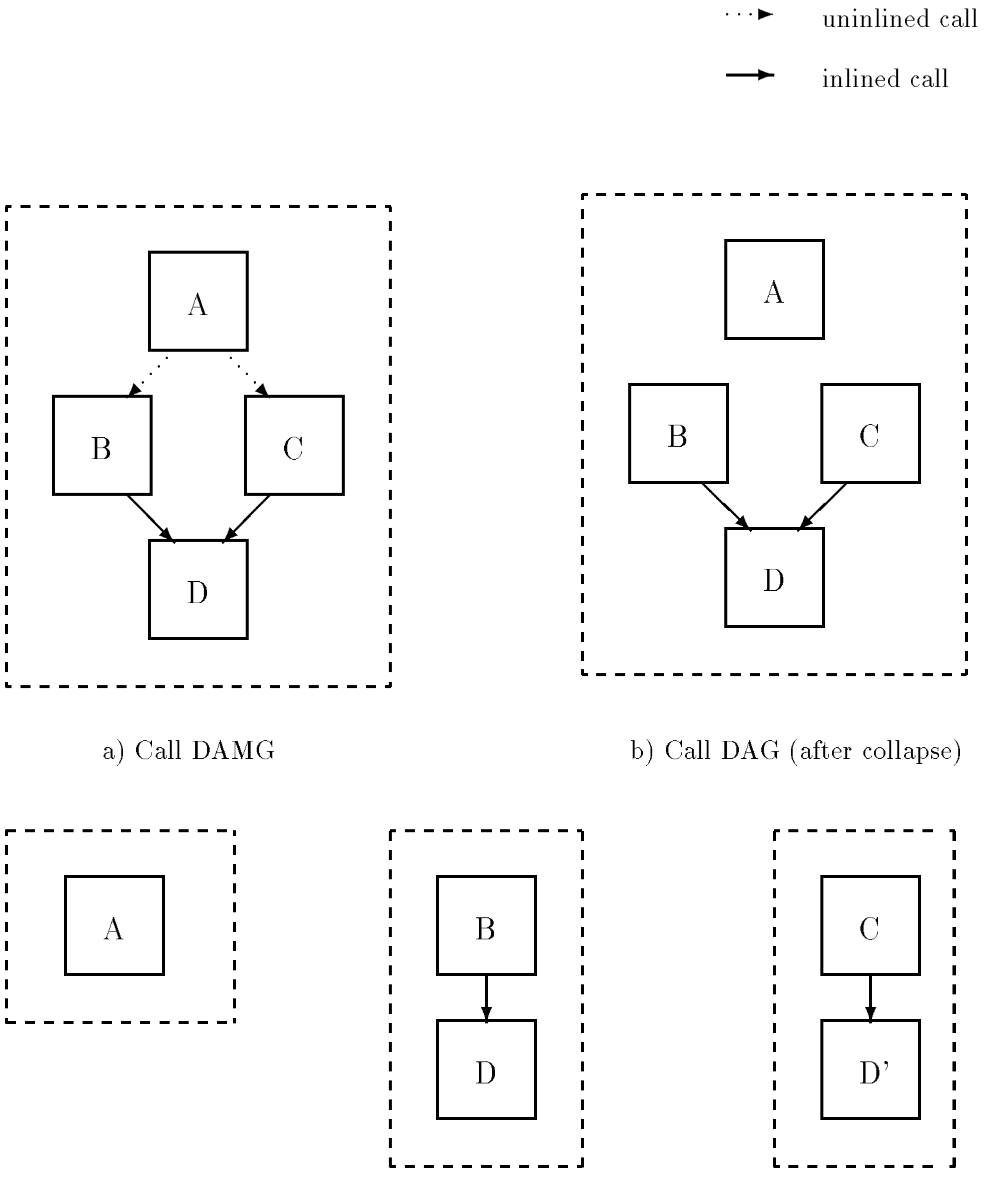
To allow site-independent transformations to be performed only once no matter how many times a given subprogram is to be expanded within a given top level routine, the inliner driver needs to create separate template InlineObjects to represent each combination of a called subprogram to be expanded and a top level routine into which it is expanded. This requirement is implemented using a variant of the standard flow graph algorithms for interval building and node splitting, given in Figure 4.3, to partition the call DAG into inline flow DAGs whose header nodes represent a top level routine before inline expansion, and whose other nodes (if any) represent template InlineObjects for routines which will be expanded into that top level routine. An example of building and partitioning the DAMG is given in Figure 4.5.

```
FOR each header node $h$ in the call DAG
    Initialize interval $i$ and new inline flow DAG $d$ to contain only $h$
    Initialize replication set $r$ to empty set
    REPEAT
        FOR each unvisited node $n$ in interval $i$
            Mark $n$ as visited
            FOR each successor $s$ of $n$ which is not in interval $i$
                IF each predecessor of $s$ is in interval $i$
                    Add $s$ to interval $i$ and inline flow DAG $d$
                    Move the edges which point at $s$ into $d$
                    Remove $s$ from $r$
                ELSE ($s$ must be replicated unless added to $i$ before a fixed point)
                    Add $s$ to $r$
                ENDIF
            END FOR
        END FOR
        FOR each node $n$ in replication set $r$
            Remove $n$ from $r$
            Split $n$, creating node $n'$ (see Figure 4.4)
            Add $n$ to interval $i$ and inline flow DAG $d$
            Reseat all edges from nodes outside $i$ to $n$ to point to $n'$
            Move the edges which point at $n$ into $d$
            FOR each edge $e$ originating from $n$
                Clone edge $e$, creating edge $e'$
                Reseat $e'$ to originate from $n'$
            END FOR
        END FOR
    UNTIL inline flow DAG $d$ reaches a fixed point
    Output inline flow DAG $d$
END FOR
```

Figure 4.3: Call DAG Partition Algorithm

```
IF $n$'s split flag is off
    Move any local saved variables to a new COMMON block
    Move corresponding DATA statements to a new BLOCK DATA routine
    Set $n$'s split flag on
ENDIF
Clone $n$ to create a new node $n'$
```

Figure 4.4: Node Splitting Algorithm



uninlined call
inlined call
A
B
C
D
a) Call DAMG
A
B
C
D
b) Call DAG (after collapse)
A
B
D
C
D'
c) Inline flow DAGs (after partition)

Figure 4.5: Call DAMG Transformation

4.3.2 Routine Cleaning

Each ProgramUnit object which is to have a subprogram call expanded or which is itself to be the target of expansion is "cleaned" to accurately represent the semantic requirements given in the Fortran 77 standard for use of intrinsic function names and for precalculation of actual parameter expressions. The algorithms to clean a whole routine and individual statements are given in Figures 4.6 and 4.7.

4.3.3 Site-independent transformations

The site-independent transformations, given in Figure 4.8, are performed to transform a newly-created InlineObject, which represents a subprogram to be expanded, and points at a designated "top level" routine, into a template InlineObject, which represents a version of that subprogram to be expanded into that top level routine. These transformations are done once per subprogram for each top level routine in which the subprogram's code is to appear.

```
FOR all symbols in routine to be cleaned
    IF symbol is used as a variable and is an intrinsic function name
        Rename the symbol to not conflict with any intrinsic function names
    ENDIF
END FOR
FOR each statement in routine to be cleaned
    Clean the statement (see Figure 4.7)
END FOR
```

Figure 4.6: Routine Cleaning Algorithm

```
IF a statement assigns the value of a function call to a scalar
    Exclude function call at top level of RHS from following search
ENDIF
FOR each function call $f$ (while searching statement in Fortran evaluation order)
    Precalculate $f$ into a scalar temporary variable
    Replace the call to $f$ with a reference to the scalar temporary
    Recursively clean the precalculation statement for potential expansion
END FOR
IF a statement represents a subprogram call
    FOR each actual parameter to the subprogram call
        SWITCH on type of actual parameter
        CASE Constant:
            Scalar Variable (or Array Variable without subscript):
            Array Reference (whose subscripts are constant or scalar):
            Substring Reference (whose arguments are constant or scalar):
            BREAK (actual parameter can be accessed by reference)
        DEFAULT:
            Precalculate expression(s) as needed to support access by reference
        END SWITCH
    END FOR
ENDIF
```

Figure 4.7: Statement Cleaning Algorithm

```
Move all entry points to the top of the subprogram (any GOTOs generated
        here will be removed after expansion)
Move all return points to the bottom of the subprogram
Substitute all subprogram integer symbolic constants (Fortran PARAMETERs)
Move all subprogram FORMAT statements into the top level routine
Precalculate all of the subprogram's adjustable array bounds
FOR each reference to a whole formal array in a subprogram I/O statement
    Build an implied DO-loop nest which references all formal array
            elements in column-major order
END FOR
Remap all local subprogram variables by creating new unconflicting top level variables
FOR each subprogram COMMON block cb
    IF cb is not already present in the top level routine (cb is unconflicting)
        Copy cb into top level routine
    ELSE (cb is conflicting)
        FOR each variable name in cb
            IF corresponding top level COMMON block variable is not of the same type
                Generate an equivalenced top level variable
            ENDIF
            Calculate top level reference corresponding to subprogram variable
            Make top level and subprogram COMMON block references conformable
                    (see Figure 4.10)
        END FOR
    ENDIF
END FOR
FOR each subprogram EQUIVALENCE block eb
    Scan eb for COMMON variables
    IF eb does not contain any COMMON variables
        Copy eb into top level routine (renaming variables to match)
    ELSEIF eb contains a variable from an unconflicting COMMON block
        Copy eb into top level routine (renaming variables to match)
    ELSE (eb contains a variable from a conflicting COMMON block)
        Add equivalence variables to corresponding top level EQUIVALENCE block
    ENDIF
END FOR
Move all subprogram DATA statements into the top level routine
```

Figure 4.8: Site-independent Transformation Algorithm

4.3.4 Site-specific transformations

The site-specific transformations, given in Figure 4.9, are those which need to be done each time inline expansion is performed, or which depend on the actual parameters passed at a call site. These transformations are done once per call site.

A database of statement label numbers is kept for each ProgramUnit, and this is used by the inliner in a fixed-point iteration routine to assign new statement label numbers to subprogram statements which conflict with those in the top level routine into which it is to be expanded.

```
FOR each subprogram statement label (in ascending order)
    Find a label number which does not conflict with top level routine
    Relabel the statement if necessary
END FOR
FOR each pair of actual and formal parameters
    Make the actual and formal parameters conform to each other (see Figure 4.10)
END FOR
Translate all subprogram references to top level references (see Figure 4.12)
```

Figure 4.9: Site-specific Transformation Algorithm

4.3.5 Handling conformability

When transforming subprogram references to top level form, the most difficult issues center around the conformability of actual and formal parameters and of subprogram and top level COMMON block variables. The three fundamental types of conformability problems are (a) different array bounds (including differences caused by mapping REAL to COMPLEX or COMPLEX to REAL); (b) formal parameters whose bounds are not known at compile time (e.g., adjustable array bounds); (c) COMMON block variables in the subprogram which partially overlap one or more corresponding COMMON block variables in the top level routine.

```
Test top level and subprogram references for conformability (see Figure 4.11)
IF conformability type is Non-Conformable
    Generate an equivalenced variable for the top level reference
    Test equivalenced and subprogram references for conformability (see Figure 4.11)
    Assert that the references are now conformable
ENDIF
SWITCH on conformability type
CASE Identical:
    Map subprogram reference to top level reference in the symbol map
    BREAK
CASE Transformable Constant:
    Map subprogram reference to top level constant in the constant map
    BREAK
CASE Transformable Array:
    Generate translation pattern for subprogram subscripts
    Map subprogram reference to top level reference in the symbol map
    Map subprogram reference to pattern in the array map
END SWITCH
```

Figure 4.10: Algorithm to Make References Conformable

```
SWITCH on type of subprogram and top level reference
CASE references are not of the same type:
    Return Non-Conformable (to force generation of a linearized equivalence variable)
CASE subprogram variable is a scalar:
CASE subprogram and top level variables have same rank, identical lower bounds throughout,
      and identical size in dimensions before the last dimension:
    Return Identical
CASE top level reference is a symbolic constant:
    Return Transformable Constant
CASE top level variable is not a formal variable and all bounds are known at compile time:
    Return Non-Conformable (to force generation of an equivalenced multi-dimensional variable
    to enhance dependence test accuracy)
CASE top level variable is linear and subprogram variable is multi-dimensional:
    Return Transformable Array (linearize subprogram subscript expressions)
DEFAULT:
    Return Non-Conformable (to force generation of a linearized equivalence variable)
END SWITCH
```

Figure 4.11: Algorithm to Test Reference Conformability

To support the generation of equivalenced variables for conformability, all of the inline expansion within a particular top level routine is done top down in depth-first order. This increases the number of calls to inline expansion, but, since many subprogram transformations are then site-independent (and are only done once per top level routine containing that subprogram), it also significantly reduces the amount (and complexity) of site-specific processing done for each call to inline expansion.

If a real actual variable is passed to a complex formal variable, to generate the proper top level equivalence for the subprogram's references, the inliner must know if the offset within the actual variable is an even or odd multiple of the complex variable element size. Unfortunately, this may not be known at compile time.

Currently, a Polaris assertion is created which states that non-constant offsets are even multiples of the complex variable element size; this assertion helps later Polaris passes handle the associated transformed subscripts (which contain integer division operations). However, since this assertion is not tested, it could lead to an undiagnosed run-time error. Two more comprehensive solutions to this problem are: (a) if this assertion cannot be validated (or rejected) by another Polaris pass, generate code in the postpass which diagnoses it at run-time; (b) generate two-version code (and hope that, in practice, a later pass will eliminate one of the versions).

4.3.6 Handling reference transformations

The transformation of references to subprogram variables to the equivalent top level variables (given in Figure 4.12) is done using high-performance search and replace routines on trees of expressions (see Section 4.4.2) using a set of maps whose keys are local subprogram symbols and whose data reflects references to corresponding top level symbols (the symbol, constant and format maps) and patterns to translate subprogram subscript expressions (the array map).

```
FOR each statement to be translated (including newly generated ones)
    FOR each assertion on the current statement
        Translate assertion expression list (see Figure 4.13)
    END FOR
    IF statement is an I/O statement
        Translate I/O control expression list (see Figure 4.13)
    ENDIF
    Translate statement body expression list (see Figure 4.13)
    IF I/O control or statement body expressions were modified
        Perform a fixed-point iteration until all generated precalculation statements
              have been cleaned up (see Figure 4.7)
    ENDIF
    Rebuild input, output, and input-output statement references
END FOR
```

Figure 4.12: Algorithm to Translate All Statements

```
FOR each expression $e$ on the expression list
    Pull $e$ temporarily out of the list
    REPEAT
        Translate expression $e$ (see Figure 4.14)
        IF $e$ was just modified
            Replace any lambda call expressions (see Subsection 4.4.3)
        ENDIF
    UNTIL $e$ was not just modified
    IF $e$ was ever modified
        Simplify any array subscripts under $e$
    ENDIF
    Put $e$ back into the list
END FOR
```

Figure 4.13: Algorithm to Translate Expression List

```
SWITCH on type of expression
CASE array reference:
    Remove subscript expression and recursively translate it
    Look up subprogram symbol name in symbol and array maps
    IF it is in the array and symbol maps
        Define a new array reference on the top level symbol with a
              subscript expression of a lambda call expression with a
              pattern from the array map and an argument of the subprogram
              subscript expression (see Subsection 4.4.3)
        Replace the original expression with this array reference
    ELSE IF it is only in the symbol map
        Reseat the subprogram symbol reference in the original expression
              to point at the top level symbol
    ENDIF
    BREAK
CASE symbol reference:
    SWITCH on type of symbol
    CASE variable symbol:
        Look up subprogram symbol name in various maps
        IF it is in the symbol and array maps
            Define a new array reference on the top level symbol
                  with a constant subscript expression
            Replace the original expression with this array reference
        ELSE IF it is only in the symbol map
            Reseat the subprogram symbol reference in the original
                  expression to point at the top level symbol
        ELSE IF it is in the constant map
            Replace the original expression with a constant expression
        ENDIF
        BREAK
    CASE program, subprogram or symbolic constant symbol:
        Look up subprogram symbol name in symbol and array maps
        IF it is in the symbol map
            Reseat the subprogram symbol reference in the original
                  expression to point at the top level symbol
        ENDIF
    END SWITCH
END SWITCH
```

Figure 4.14: Algorithm to Translate Expression

4.3.7 Handling name conflicts

The site-independent transformations on a subprogram for local and COMMON variables and for EQUIVALENCE blocks resolve potential name conflicts. These include the use of the same variable name in the subprogram and top level routine, the use of the same EQUIVALENCE block name in the subprogram and top level routine and the use of intrinsic function names as variable names (which is legal Fortran but could inhibit future inline expansion).

During the reference translation process, it does not matter whether a subprogram symbol has the same name as the corresponding symbol in the top level routine: the symbol pointer in the subprogram variable reference must be changed regardless. So, for many variables, Polaris uses the standard base facilities to rename conflicting copied variable names when they are inserted into the top level routine.

Since a COMMON or EQUIVALENCE block refers to its associated variables, while the variables point to the block, attempts by a Polaris user to directly copy the block and its associated variables from one ProgramUnit to another fail because the user cannot fix all the pointers at once. If the block is copied first, Polaris rejects insertion of the corresponding variables as an attempt to break the encapsulation of the block; if the variables are copied first, Polaris rejects insertion of the corresponding block as an attempt to break the encapsulation of the variables.

Figure 4.15 presents two possible ways to copy a COMMON or EQUIVALENCE block, and its associated variables, from one ProgramUnit to another.

- Copy each variable as if it were not in the associated block.
- Rebuild the block item by item.

This approach does not require any support routines, but is slower and cumbersome to implement (e.g., EQUIVALENCE block offsets must be saved and restored), so it was not used.

- Directly copy each variable (iterate until no name conflict)
- Directly copy the block (for EQUIVALENCE blocks, iterate until no name conflict).
- Use the support routines for ProgramUnit cloning (discussed later in Subsection 4.4.1) to fix up the variable and block pointers.

Since fixed-point iteration routines for variable and block names would be useful in other Polaris passes, this approach was used.

Figure 4.15: Copying COMMON or EQUIVALENCE blocks

4.3.8 Handling variables with limited scope

When a subprogram is expanded into a top level routine, its unsaved (and, in some cases, saved) local variables, its precalculation variables, and any unconflicting COMMON block mappings are moved into the top level routine. These same top level variables (and COMMON block mappings) are used each time the subprogram is expanded within that top level routine. If a COMMON block mapped in the subprogram is not mapped in the top level routine, the mapping is moved into the top level routine; if it is mapped in both places, the original mapping in the top level routine is used (but it may be expanded with equivalences to handle subprogram variables which partially overlap one or more top level variables).

Moving a subprogram's unsaved local (and precalculation) variables into the top level routine is semantically correct but, by itself, could introduce spurious anti and output dependences on these variables (between invocations of the subprogram's code). To break spurious dependences on these variables, the inliner attaches LOCAL assertions naming these variables to each Polaris block containing the expanded code.[1] The previous values of variables declared local to a block are not used in the block, and their final values are not used outside the block. In addition, variable names which have been declared local within some block in a routine must not be used except within that (or some other block) to which it is local.

The values of a subprogram's saved local variables must be preserved between invocations of the subprogram's code. In practice, this is done by using the same variables in all the copies of its code. In some cases (e.g., complete inline expansion), all these copies will end up in one top level routine, and so the variables can be moved (still marked saved) into the top level routine. If one or more call sites of the subprogram are not expanded, or copies of its code will appear in more than one top level routine after inline expansion (which happens iff the call DAG partitioning algorithm splits the corresponding node), common storage must be used instead. So, before a node is to be split for the first time, the partitioning algorithm calls a Polaris base routine to move any saved local variables in the corresponding subprogram into a new saved COMMON block (and any associated DATA statements into a new BLOCK DATA subprogram).

Sections 8.9 and 9.3 of the Fortran 77 standard limit the use of unsaved COMMON blocks to passing values directly up and down a program's call tree and allow them to be dynamically allocated, reinitialized and deallocated in certain situations. Though most Fortran compilers place both unsaved and saved COMMON blocks into static storage with global scope, one that implemented overlay programming could allow unsaved COMMON blocks (and any associated named BLOCK DATA subprograms) to be placed into the highest level of an overlay structure

[1]Until support for LOCAL assertions is added to other Polaris transformations, the inliner generates the similar PRIVATE assertion instead.

in which they were used. Similarly, a parallelizing compiler could privatize these COMMON blocks (and BLOCK DATA subprograms) to the routine using them which is closest to top level. Since this kind of privatization was used to manually parallelize MG3D (one of the Perfect Benchmark codes), Polaris might eventually privatize unsaved COMMON blocks automatically.

4.3.9 Handling argument aliasing

To support the generation of efficient code for formal array references, Section 15.9.3.6 of the Fortran 77 standard prohibits the modification of variables (or sections of variables) which could be accessed by a subprogram either by (a) multiple formal arguments or (b) by a formal argument and a COMMON block name. So, no dependence arc involving a write can exist within a subprogram on such "aliased" variables (or sections of variables).

At the time the inliner expands a subprogram at a call site, it has information about the formal arguments, which arguments are aliased to one another (at the level of the call), their ranges (as declared in the subprogram) and which array references are made to each formal argument. If this information is discarded after inline expansion, dependence arcs on aliased variables (or sections of variables) are not automatically broken and may inhibit later parallelization of an enclosing loop.

A method to preserve some of this aliasing information for later Polaris passes will label each translated array reference with a tag of some sort which will reflect the name of (and possibly the declared range of) the corresponding formal array.

4.3.10 Handling Fortran's semantic quirks

Section 12.8.2 of the Fortran 77 standard states that if an array name appears as an item in an I/O list in a data transfer I/O statement, the compiler must implicitly generate an implied DO-list which references all the array elements in column-major order. If such a reference is made to a formal array within a subprogram to be expanded, the inliner must replace the formal array name with an explicit implied DO-list on it which reflects its bounds and layout so that, after inline expansion, the I/O statement will explicitly specify which elements of the corresponding actual array are to be transferred.

Namelist I/O is an extension of the Fortran 77 standard in which the names of the variables to be read (or which are written) appear in the input (or output). To preserve the semantics of a subprogram which contains namelist I/O statements, the existing variable names (and bounds) must be preserved upon execution of the program. If this is not possible, inline expansion will fail for the subprogram. Currently, the expansion of subprograms containing namelist I/O statements is not supported; to support it in the general case, the inliner could store the original names and bounds of variables used in namelist I/O statements with name conflicts, and the postpass could generate corresponding facade subprograms containing namelist I/O statements whose arguments were formal variables with correct names and bounds.

4.4 Performance Considerations

To minimize the costs of an individual inline expansion and of the whole process, the Polaris inliner uses a number of high-performance services provided by other Polaris components [2, 8].

4.4.1 ProgramUnit cloning

The Polaris base implements a set of routines which support "cloning": making a usable copy of an existing ProgramUnit object. Since cloning bypasses the assertions which protect the semantic integrity of a ProgramUnit, it also supports the copying of parts of existing ProgramUnit objects without breaking their encapsulation.

The Polaris inliner uses ProgramUnit cloning directly when making a work copy of a template InlineObject, and uses its support routines to copy symbols, COMMON blocks and EQUIVALENCE blocks from expanded subprograms into top level routines.

4.4.2 Expression translation

The Polaris base implements template classes which provide high-performance random access to data using red-black trees. The support routines for FORBOL, a high-level pattern matching language built using Polaris[22], provide high-performance search and replace routines for trees of expressions.

The Polaris inliner uses these FORBOL support routines and several random-access maps to find each expression within a subprogram which contains a pointer to a local symbol. A subprogram expression can be translated in place if the corresponding top level variable (or variables) have identical type (and, for an array reference, bounds). If a subprogram expression needs more complex transformations, the inliner replaces it with an equivalent expression either directly (e.g., replacing a scalar formal with a reference to an actual array) or indirectly (e.g., replacing a formal array reference with a lambda call expression which adjusts or linearizes its subscripts: see Subsection 4.4.3).

4.4.3 Other Polaris transformations

Other Polaris transformations used by the Polaris inliner are lambda call replacement, control flow normalization, constant propagation and dead store elimination.

A lambda call expression contains an argument which is a list of expressions to be evaluated (and possibly precalculated) and a pattern which is to be filled in by either the actual arguments or precalculation variables. Lambda call expressions are translated into standard Polaris expressions (optionally preceded by one or more precalculation statements) by lambda call replacement routines in the Polaris base.

The control flow normalization routines used by the inliner float a subprogram's entry points to the top of the routine and sink a subprogram's return points to the bottom of the routine. Constant propagation and dead store elimination are used to eliminate conservatively generated precalculation variables and statements.

CHAPTER 5

EXPERIMENTS

A series of experiments were performed to measure the performance and effectiveness of complete inline expansion on a subset of the Perfect Benchmarks®. These experiments involved running unexpanded and completely expanded versions of these codes on a Sun SparcStation platform and on an SGI Challenge platform; Polaris was evaluated on each platform, and a SGI-distributed version of Kuck and Associates (KAI)'s KAP, the Power Fortran Accelerator (PFA), was evaluated on the SGI Challenge platform.

The execution time reported for Polaris and PFA on each code is an average of between three to five identical runs,[1] but, since the codes output by Polaris and PFA were run only once (to demonstrate validation), their reported execution times should be taken only as a rough approximation. When codes were small enough to be effectively optimized by Sun's f77 or SGI's Power Fortran, global optimization was used to compile Polaris and PFA output. When the use of global optimization was not possible, local optimization was performed instead.

[1]The number of runs was determined by the variability of the execution time and the size of the code.

The measurements on the number of routines, number of executable lines (except in two cases), and the number of original call sites to contained routines were computed exactly by an application program using Polaris facilities. The number of original call sites left to contained routines after inline expansion for PFA was estimated by using the output of this application program and the PFA listing file. Finally, the number of lines in QCD and SPEC77 after complete inline expansion by Polaris were approximated using the flint utility.[2]

No. of Routs	Number of routines (including BLOCK DATA)
No. of Lines	Number of executable lines (but see AX below)
Calls Orig.	Number of calls to contained routines before complete expansion
Calls Left	Number of original calls to contained routines left after complete expansion (ignoring any duplicates created by expansion)
Time to Expand	User CPU Time spent in Polaris
Time to Execute	User CPU Time spent executing code
-O1	Local optimization used
-O3	Global optimization used
-r8	Eight-byte reals needed for validation
AX	Approximate number of lines (of all non-comment statements)
BD	Program contained a BLOCK DATA routine
NX	Program did not successfully execute

Table 5.1: Key to Experiment Comments

These experiments demonstrated that both Polaris and PFA were accurate overall in performing the inline expansions which they attempted, and, given their different design criteria,[3] both do it relatively efficiently. However, there were significant differences between Polaris and PFA in inliner functionality; in five of the nine test cases, there were a substantial number of

[2]Due to limitations in the Polaris base routine to create ProgramUnit objects from FORTRAN input, the completely expanded versions of QCD and SPEC77 were too large to be read back into Polaris in a reasonable amount of execution time.

[3]In some areas, such as the handling of expressions, compilers must make tradeoffs between execution time, execution space, functionality and ease of system implementation and extension; since PFA is a commercial product, the first two criteria are more important for it: in Polaris, the last two criteria are more important.

Code	No. of Routs	No. of Lines	Calls Orig.	Time to Execute	Comments
ARC2D	42	1860	53	7:30	-O3
BDNA	46	3085	103	2:11	-O3 BD
FLO52	30	1699	46	0:58	-O3
MDG	16	775	28	6:37	-O3
OCEAN	39	1719	247	5:30	-r8 -O3 BD
QCD	35	1443	93	0:22	-O3
SPEC77	65	2613	183	32:36	-r8 -O3 BD
TRACK	32	1535	50	0:21	-r8 -O1
TRFD	7	415	5	0:49	-O3

Table 5.2: Unexpanded codes on the Sun SparcStation Platform

Code	No. of Routs	No. of Lines	Calls Orig.	Calls Left	Time to Expand	Time to Execute	Comments
ARC2D	1	2259	53	0	3:06	7:57	-O3
BDNA	2	4455	103	0	4:40	2:15	-O3 BD
FLO52	1	3316	46	0	2:50	1:00	-O3
MDG	1	1249	28	0	0:52	9:09	-O3
OCEAN	2	11195	247	0	4:28	15:56	-r8 -O1 BD
QCD	1	13039	93	0	7:42	1:05	-O3 AX
SPEC77	2	43112	183	0	13:31	15:23	-r8 -O1 AX BD
TRACK	1	2384	50	0	1:23	0:24	-r8 -O3
TRFD	1	279	5	0	0:14	0:52	-O3

Table 5.3: Codes expanded by Polaris on the Sun SparcStation Platform

Code	No. of Routs	No. of Lines	Calls Orig.	Time to Execute	Comments
ARC2D	42	1860	53	3:27	-O3
BDNA	46	3085	103	0:54	-O3 BD
FLO52	30	1699	46	1:23	-O1
MDG	16	775	28	4:18	-O1
OCEAN	39	1719	247	4:40	-r8 -O1 BD
QCD	35	1443	93	0:24	-O1
SPEC77	65	2613	183	4:02	-O1 BD
TRACK	32	1535	50	0:09	-r8 -O1
TRFD	7	415	5	1:19	-O1

Table 5.4: Unexpanded codes on the SGI Challenge Platform

Code	No. of Routs	No. of Lines	Calls Orig.	Calls Left	Time to Expand	Time to Execute	Comments
ARC2D	1	2259	53	0	1:45	7:32	-O3
BDNA	2	4455	103	0	2:37	1:43	-O3 BD
FLO52	1	3316	46	0	1:41	1:48	-O1
MDG	1	1249	28	0	0:26	4:29	-O1
OCEAN	2	11195	247	0	2:36	4:30	-r8 -O1 BD
QCD	1	13039	93	0	5:01	0:25	-O1 AX
SPEC77	2	43112	183	0	8:07	4:36	-O1 AX BD
TRACK	1	2384	50	0	0:39	0:09	-r8 -O1
TRFD	1	279	5	0	0:06	1:21	-O1

Table 5.5: Codes expanded by Polaris on the SGI Challenge Platform

Code	No. of Routs	No. of Lines	Calls Orig.	Calls Left	Time to Expand	Time to Execute	Comments
ARC2D	5	2350	53	4	0:22	4:06	-O3
BDNA	17	3888	103	19	0:25	0:55	-O3 BD
FLO52	5	3421	46	4	0:29	1:48	-O1
MDG	4	1164	28	7	0:08	4:37	-O1
OCEAN	27	1819	247	193	0:10	4:30	-r8 -O1 BD
QCD	12	4140	93	46	0:45	0:24	-O1
SPEC77	38	3716	183	142	0:38	NX	-O1 BD
TRACK	12	1738	50	20	0:09	0:09	-r8 -O1
TRFD	3	305	5	2	0:01	0:81	-O1

Table 5.6: Codes expanded by PFA on the SGI Challenge Platform

call sites to contained routines left unexpanded by PFA (and, in all nine cases, at least one call site to a contained routine was left unexpanded): in one case tested (SPEC77), PFA performed inline expansion incorrectly, leading to a run-time error. In all other cases, execution was successful and the programs validated (sometimes only with eight-byte real variables: see the tables for more information).

Most of the differences in function between Polaris's and PFA's inliners involve design issues discussed in Chapter 4, such as I/O on whole subprogram formal arrays, mismatches in the layouts of COMMON blocks, and, especially, the inability to inline subprograms with saved variables (which was a problem with OCEAN) or variables initialized by DATA statements (which was a problem with SPEC77). One technique which PFA performs which Polaris does not is partial linearization of subprogram array references (e.g., mapping a three-dimension formal array reference to a two-dimensional actual array); however, in practice, the Polaris range test[1] has been able to recreate the dependence information lost by full linearization of subprogram array references.

CHAPTER 6

REFLECTIONS AND CONCLUSIONS

As the Polaris inliner grew, it became clear that source-to-source inline expansion of Fortran routines in many real programs was a very complicated problem, and so the Polaris inliner grew into a much more general-purpose entity than had been originally intended, which has already allowed reuse of its technology by several other Polaris transformations.[1]

As Polaris evolves, the technology implemented in the Polaris inliner, in combination with other Polaris transformations and the Polaris infrastructure, should allow the implementation of demand-driven flow-sensitive interprocedural analysis techniques which use existing facilities for gated SSA[21] and give the power of complete inline expansion without the cost of analyzing any extra code produced by inline expansion which was not needed for parallelization.

As discussed in [16], the Polaris prototype compiler has shown excellent results on a number of real Fortran codes, from many of the Perfect Benchmarks® to several "Grand Challenge" codes provided by the National Center for Supercomputing Applications (NCSA). A major reason for this success was the robust, high-performance, general-purpose Polaris inliner.

[1] Equivalence normalization and run-time parallelization.

REFERENCES

[1] William Blume and Rudolf Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. *Proceedings of Supercomputing '94, November 1994, Washington D.C.*, pages 528–537, November 1994.

[2] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, William Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the Effectiveness of Parallelizing Compilers. *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing, Ithaca, New York; also: Lecture Notes in Computer Science 892, Springer-Verlag*, pages 141–154, August 1994.

[3] K. D. Cooper, M. W. Hall, R. Hood, K. Kennedy, K. McKinley, J. Mellor-Crummey, L. Torczon, and S. Warren. The ParaScope Parallel Programming Environment. *Proceedings of the IEEE*, pages 244–263, February 1993.

[4] K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proc. IEEE International Conference on Computer Languages*, pages 1–11, April 1992.

[5] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software-Practice and Experience*, 21(6):581–601, June 1991.

[6] Rudolf Eigenmann, Jay Hoeflinger, G. Jaxon, and David Padua. The Cedar Fortran Project. Technical Report 1262, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res, & Dev., April 1992.

[7] Rudolf Eigenmann, Jay Hoeflinger, Zhiyuan Li, and David Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Lecture Notes in Computer Science 589. Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 65–83, August 1991.

[8] Keith A. Faigin, Jay P. Hoeflinger, David A. Padua, Paul M. Petersen, and Stephen A. Weatherford. The Polaris Internal Representation. *International Journal of Parallel Programming*, 22(5):553–286, October 1994.

[9] Mary W. Hall, Ken Kennedy, and Kathryn S. McKinley. Interprocedural transformations for parallel code generation. In *Proc. Supercomputing '91*, pages 424–434, 1991.

[10] Mary W. Hall, John M. Mellor-Cummey, Alan Carle, and René G. Rodríguez. Fiat: A framework for interprocedural analysis and transformation. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computers, vol. 2*, February 1995.

[11] Mary W. Hall, Brian R. Murphy, and Saman P. Amarasinghe. Interprocedural parallelization analysis: A case study. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[12] Michael Hind, Michael Burke, Paul Carini, and Sam Midkiff. A comparison of interprocedural array analysis methods. *Scientific Programming*, 3:255–271, 1994.

[13] Anne M. Holler. A study of the effects of subprogram inlining. Technical Report No. TR-91-06 (Computer Science), University of Virginia, March 20, 1991.

[14] Christopher Alan Huson. An In-Line Subroutine Expander for Parafrase. Master's thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Sci., Dec., 1982.

[15] Zhiyuan Li and Pen-Chung Yew. Efficient Interprocedural Analysis for Program Parallelization and Restructuring. *Proceedings of ACM/SIGPLAN PPEALS, New Haven, CT*, pages 85–99, July 1988.

[16] David A. Padua, Rudolf Eigenmann, and Jay P. Hoeflinger. Automatic Program Restructuring for Parallel Computing and the Polaris Fortran Translator. *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, CA, February 15-17, 1995*, pages 647–649, February 1995.

[17] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation*, June 1995.

[18] S. Richardson and M. Ganapathi. Code Optimization Across Procedures. *IEEE Computer*, February 1989.

[19] Dale Allan Schouten. An Overview of Interprocedural Analysis Techniques for High Performance Parallelizing Compilers. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1990.

[20] Peng Tu and David Padua. Automatic Array Privatization. In Utpal BanerjeeDavid GelernterAlex NicolauDavid Padua, editor, *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science.*, volume 768, pages 500–521, August 12-14, 1993.

[21] Peng Tu and David Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. Technical Report 1399, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., January 1995.

[22] Stephen Weatherford. High-Level Pattern-Matching Extensions to C++ for Fortran Program Manipulation in Polaris. Master's thesis, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., May 1994.